TABLE DES MATIERES

1. L' HARDWARE DE L'AMIGA

1.1 Introduction

L'AMIGA de COMMODORE propose des possibilités que personne n'aurait seulement imaginées, il y a quelques années, pour un ordinateur familial.

Pour rendre ce résultat possible, on trouve réunis sur l'AMIGA, d'une part un système d'exploitation performant et d'autre part une architecture évoluée.

L'un des buts des concepteurs de cet ordinateur était une grande facilité d'utilisation. Ils ont intégré dans le système d'exploitation un grand nombre de routines simplifiant la programmation du *HARDWARE*.

Cette simplicité n'est pourtant qu'apparente. Malgré de confortables routines, l'utilisateur pourra se faire piéger en programmation directe, notamment lors de l'emploi des routines système, dont les vitesses sont bien plus faibles que celles auxquelles on aurait pu s'attendre. Une des raisons principales est que le système d'exploitation est écrit dans sa plus grande partie, avec le langage de programmation C.

Quelle que soit l'utilisation que vous réservez à votre AMIGA, une bonne connaissance de cet ordinateur passera nécessairement par l'acquisition de notions sur la structure interne et la programmation de ses processeurs caractéristiques. C'est d'ailleurs dans cette optique que ce livre a été conçu.

1.2 Les composantes de l'Amiga

Le Hardware des différentes versions de l'AMIGA, A500, 1000, A2000, B2000 reste essentiellement identique :

- microprocesseur 68000 de Motorola,
- deux adaptateurs d'interface de commande, type 8520,
- trois circuits spécialisés, AGNUS, DENISE, PAULA.

En écartant la *RAM* (mémoire vive) et toute la construction logique, ce sont ces six circuits décrits plus haut qui assurent l'essentiel du travail de l'AMIGA.

Une des particularités de cet ordinateur est aussi la présence de nombreux connecteurs :

- port parallèle (centronics),
- connecteur série RS-232,
- connecteur vidéo RGB,
- connecteur modulateur TV,
- sortie audio stéréo,
- sortie vidéo composite (absente sur la version A2000 mais disponible sur la version B2000),
- connecteur clavier,
- connecteur disquette externe,
- 2 connecteurs identiques pour joystick, souris ou manette,

- connecteur d'extension mémoire de 256 Koctets (il ne sera fait aucune description de ce connecteur, étant donné qu'il ne concerne que l'AMIGA 1000, qui n'était doté que de 256 Koctets de RAM au départ ; l'adjonction d'une extension mémoire lui permet de passer à un total de 512 Koctets de RAM),
- connecteur d'extension système (ce connecteur se trouve sur les versions 500 et 1000, celui du 2000 se divisant en plusieurs connecteurs d'extension).

Afin de mieux comprendre le travail d'ensemble de tous ces éléments, nous devons tout d'abord examiner chaque composante à part.

1.2.1 LE PROCESSEUR 68000

Le Motorola 68000 est indiscutablement l'un des plus remarq processeurs 16 bits. Malgré sa présence sur le marché depuis 19 ne le trouve que depuis peu sur des ordinateurs de la clas l'AMIGA.

Vous ne trouverez pas dans ce volume une description détaillée 68000. Ceux qui veulent en savoir plus sur la programmation peu se référer à la littérature spécialisée que l'on peut trouver sur ce sujet. A la place, nous allons examiner exclusivement le brochage du circuit ainsi que les groupes de signaux isolés. Une connaissance fondamentale des signaux du 68000 est essentielle à la compréhension du Hardware de l'AMIGA.

Brochage du 68000

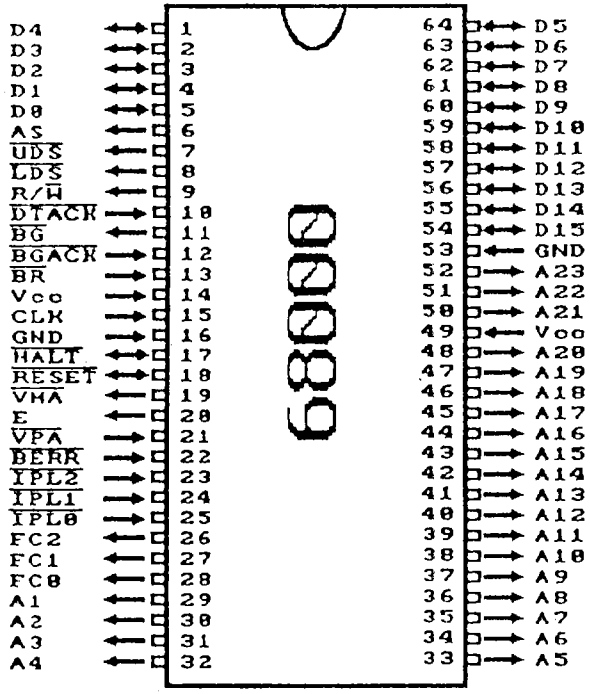


Figure 1.2.1.1

Remarques :

- les flèches indiquent le sens du signal,
- le trait au-dessus du nom du signal indique le double état que peut prendre ce signal (0 correspondant à actif).

On peut distinguer ces broches suivant plusieurs groupes :

- L'alimentation VCC et GND

Le processeur Motorola 68000 demande une alimentation de 5 volts. Les deux broches sont dédoublées et placées au centre du circuit, afin que les chutes de tension soient minimales.

- Broche d'horloge : CLK

La fréquence de l'horloge du processeur 68000 dépend de la version du processeur. Sur l'AMIGA, la fréquence est de 7.16 MHZ (millions de cycles par secondes).

- Bus de données D0-D15

Le bus de données est de type 16 bits et peut ainsi véhiculer un mot (16 bits) à la fois. Le processeur permet aussi le transfert d'octets (8 bits) soit supérieurs (D8-D15), soit inférieurs (D0-D7).

- Bus d'adresse A1-A23

Le bus d'adresses peut adresser avec ses 23 lignes unidirectionnelles huit mega mots ou 16 mega octets.

- Bus de contrôle asynchrone : AS, R/W, UDS, LDS, DTACK

Une des particularités du 68000 est son fonctionnement asynchrone. En mode asynchrone, le processeur signale avec AS (ADRESS STROBE/ adresse valide) qu'une adresse valide se trouve dans le bus d'adresses. Simultanément, on détermine à l'aide de R/W (READ-WRITE/lecture-écriture) le sens du transfert (1 pour la lecture et 0 pour l'écriture). Le 68000 adresse sa mémoire avec des mots de 16 bits. Les deux signaux de contrôle UDS (UPPER DATA STROBE) et LDS (LOWER DATA STROBE) permettent de faire la distinction entre les deux octets d'un même mot lors du transfert d'un octet (8 bits).

- Si UDS est à 1 et LDS à 0 : il s'agit de l'octet de poids faible (inférieur),
- Si UDS est à 0 et LDS à 1 : il s'agit de l'octet de poids fort (supérieur),
- Si UDS et LDS sont à 0 : il s'agit alors d'un mot entier.

Le signal DTACK (Data Transfer Acknowledge/R connaissance de transfert de données) indique que les donnée nt prêtes à être transférées lorsqu'il est positionné à 0.

En mode asynchrone, le processeur s'aligne donc toujours sur la vitesse de la mémoire.

Les mots et octets isolés s'organiseront de la manière suivante en mémoire :

Organisation des bus de données

Adresse		D8-D15	D0-D7
0	mot 0	octet 0	octet 1
2	mot 1	octet 2	octet 3
4	mot 2	octet 4	octet 5
6	mot 3	octet 6	octet 7

- Bus de contrôle en mode synchrone : E, VPA, VMA

Au moment où le Motorola 68000 a été commercialisé, il n'y avait pas encore de circuits périphériques disponibles. Les circuits existant alors avaient été conçus pour la série précédente (série 6800, d'où descendait d'ailleurs le 6502) et n'avaient pas la possibilité de s'interfacer avec le bus de contrôle asynchrone du 68000. Ainsi, ce dernier s'est vu attribuer des bus synchrones, afin de faciliter les échanges entre ce microprocesseur et ceux de la famille 6800.

Sur la broche E, on applique un signal d'horloge correspondant au dixième du signal CLK (Clock) (7,16 MHZ) qui peut servir à tous les circuits périphériques (il sera relié à l'entrée Phi-2 d'un circuit périphérique).

Le passage du mode synchrone en asynchrone se produit sur l'entrée VPA (VALID PERIPHAL ADRESS/adresse périphérique valide).

Cette entrée doit être mise à 0 par un décodeur externe d'adresse ; dès que ce dernier reconnaît une adresse de circuit périphérique, le processeur répond aussitôt en mettant également à 0 le signal VMA (VALID MEMORY ADRESS/adresse mémoire valide).

Le circuit périphérique concerné prend alors en charge les données (et par conséquent les met à disposition) pendant un cycle d'horloge de E. Après cela, le processeur 68000 abandonne le mode synchrone automatiquement jusqu'à ce que le signal VPA soit à nouveau actif.

Ceci signifie donc qu'un circuit périphérique est contraint de prendre en charge ou de mettre à disposition, des données en mode synchrone.

- Commandes système : RESET, HALT, BERR

Pour initialiser le système, il faut mettre à 0 le signal HALT et le signal RESET. Dès que ces signaux sont remis à 1, le 68000 commence à exécuter le programme qui a été mis au préalable dans l'adresse mémoire 4. Le signal RESET peut aussi être remis à 0 à partir du 68000, afin d'initialiser le système, sans modifier l'état du processeur.

Avec le signal BERR (BUS ERROR), un contrôle externe peut indiquer au processeur qu'une information erronée circule sur un bus, comme par exemple : *"Hardware défectueux"* ou *"Accès à une adresse inexistante"*.

Si un signal BERR se déclenche, le processeur 68000 renvoie à une routine spéciale du système d'exploitation qui prend en charge le traitement de l'erreur (salutations et méditations du Gourou !). Si pendant ce traitement de l'erreur, un nouveau signal BERR se déclenche, le 68000 stoppe toute l'exécution en cours et met le signal HALT à 0. Cette double erreur est le seul cas où le processeur, pardonnez l'expression, se plante. Pour d'autres types d'erreurs, le processeur accède à des vecteurs spéciaux dans les routines de programmation, qui peuvent effectuer le traitement de l'erreur et permettre ainsi au système la poursuite du travail (on remarque l'abondance des Guru-méditations qui se comportent chez l'AMIGA

suivant la loi de Murphy : un ordinateur se plante t ours lorsque l'on travaille sur des données importantes, de préf e lorsqu'elles ne sont pas encore sauvegardées).

Si le processeur arrête l'exécution d'un programme en raison d'une double erreur du bus, il ne pourra redémarrer qu'au moyen d'un RESET (HALT et RESET à 0).

Une fonction du signal HALT est aussi l'arrêt des processeurs. En mettant HALT à 0, le 68000 achève ses accès mémoire en cours, puis attend alors jusqu'à ce que le signal HALT soit remis à 1.

- Etat de fonctionnement du processeur : FC0, FC1, FC2

Le signaux FC0-FC2 traduisent l'état de marche du processeur.

Voici les différents états possibles :

FC2	FC1	FC0	ETAT
0	0	1	Accès aux données utilisateur
0	1	0	Accès au programme utilisateur
1	0	1	Accès aux données superviseur
1	1	0	Accès au programme superviseur
1	1	1	Reconnaissance d'interruption

Le processeur peut avoir deux modes différents de travail : le *mode utilisateur* et le *mode superviseur*. Un programme qui tourne en mode superviseur accède à la totalité du registre d'état (SR = Status Register) du processeur. Le système d'exploitation travaille toujours en mode superviseur.

En mode utilisateur, la partie superviseur de SR est verrouillée. Pour plus d'informations, reportez-vous à la littérature sur le 68000.

Les trois signaux FCx permettent ainsi au système de connaître le mode de fonctionnement actuel du processeur, et selon le cas, de réagir sur ce mode. En mode utilisateur, par exemple, lors de l'accès à une zone mémoire réservée au système d'exploitation, une erreur (BERR=0) peut être engendrée.

- Broches d'inte ption : IPL0, IPL1, IPL2

Les signaux des trois broches d'interruption (IPL = INTERRUPT PRIORITY LEVEL) permettent au processeur 68000 de différencier 8 signaux d'interruption (2^3) parmi lesquels 0 correspond à une absence d'interruption et 7 au plus haut degré d'interruption. Tous les niveaux se voient affecter un vecteur d'interruption, renfermant une adresse de routine d'interruption.

Si une interruption, conforme aux niveaux autorisés se déclare, le processeur met tous ses signaux FCx à 1, signalisant ainsi qu'il l'a reconnue et qu'il attend une confirmation de la part de l'auteur de cette interruption. La réponse pourra intervenir par le biais des signaux UPA et DTACK. Pour une confirmation par un signal UPA, il résulte une auto-interruption, le processeur sautant à une adresse se trouvant dans le vecteur correspondant à l'interruption. On peut donc accéder à 7 niveaux d'interruption ; le niveau 0 correspondant à l'état *"pas d'interruption"*.

A chaque niveau d'interruption correspond une source d'interruption et le processeur saute alors au programme conforme.

L'AMIGA utilise seulement ces vecteurs "auto-programmés" en cas d'interruption.

- Signaux d'attribution du bus : BR, BG, BGACK

Ces trois signaux autorisent l'allocation du bus par un autre processeur. Ceci peut être le cas du contrôleur d'un disque dur, par exemple, qui écrit les données du disque directement dans la mémoire (DMA, DIRECT MEMORY ACCESS/mémoire accès direct).

Brochage du 8520

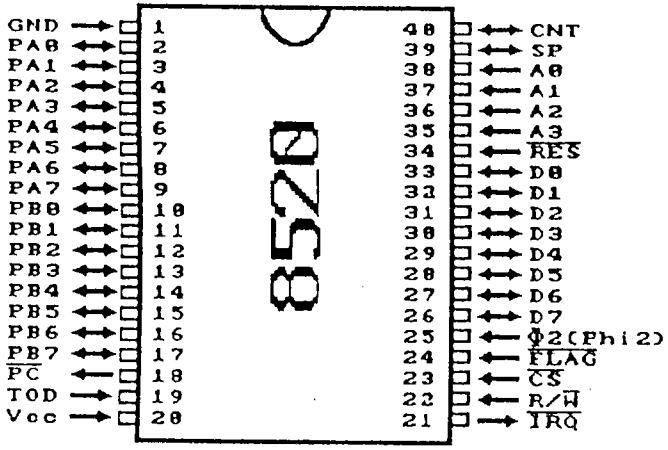


Figure 1.2.2.1

Remarques :

- les flèches indiquent le sens des signaux,
- le trait au-dessus des noms des signaux, traduit le double état que peut prendre le signal (0-actif).

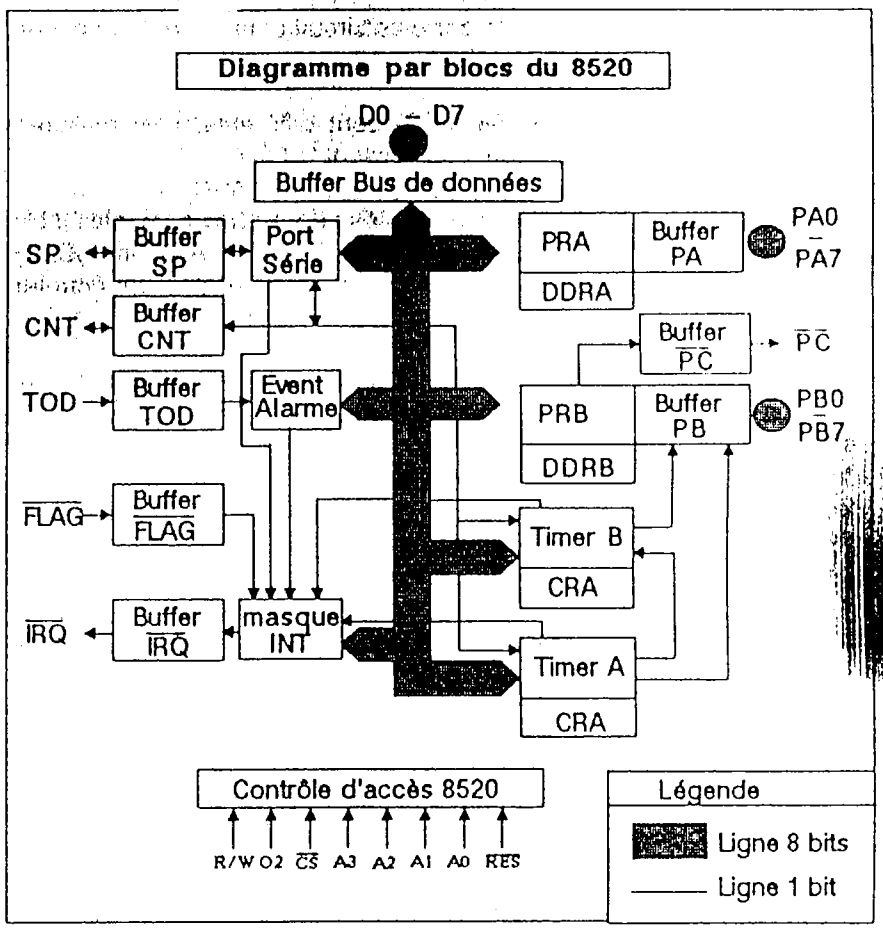


Figure 1.2.2.2

Le 8520 est un circuit périphérique du type *Complex Interface Adapters* (CIA), correspondant à un circuit à connecteurs multiples.

Ce sont en fait les concepteurs du 8520 qui ont cherché à introduire le maximum de fonctions au sein d'un même circuit. En examinant bien le 8520, on remarque la similitude entre ce circuit et un autre bien connu, le 6526 du C64.

Seuls les registres 8 à 11 ($8 à $B) sont différents, cela rassurera sûrement les programmeurs qui connaissent le 6526.

Le 8520 dispose des possibilités suivantes : deux ports parallèles 8 bits programmables (PA et PB), 2 minuteries 16 bits (Timer A et Timer B), un port série bidirectionnel (SDR) et un compteur 24 bits (Event counter) avec une fonction alarme lorsqu'il atteint une valeur fixée.

Certaines fonctions sont en mesure de libérer des interruptions.

Les fonctions du 8520 sont réparties en 16 registres. Pour le processeur, ils apparaissent comme des registres mémoires normaux, les circuits périphériques d'un système 68000 étant considérés comme faisant partie intégrante de la mémoire, ceux-ci supportant toutes les opérations de lecture et d'écriture du 68000.

Le 8520 a été développé pour le processeur 8 bits de la série 65xx, et ne peut donc communiquer avec le 68000 qu'en mode synchrone.

L'horloge E du 68000 est reliée à l'entrée Phi 2 du 8520. Les 16 registres internes sont accessibles au moyen de quatre entrées A0-A3 du 8520. Des détails plus précis sur les liens entre le système de l'AMIGA et le CIA se trouvent à la fin de ce chapitre.

Voici le détail des 16 registres du circuit (le registre 11 ($B) étant inutilisé).

Détail des registres du 8520

Registre		Nom	Description
0	0	PRA	Registre données port A
1	1	PRB	Registre données port B
2	2	DDRA	Registre direction des données port A
3	3	DDRB	Registre direction des données port B
4	4	TALO	Minuterie A (octet bas)
5	5	TAHI	Minuterie A (octet haut)
6	6	TBLO	Minuterie B (octet bas)
7	7	TBHI	Minuterie B (octet haut)
8	8	EVENT LO	Compteur (valeur événement octet bas) bits 0-7
9	9	E. 8-15	Compteur bits 8-15 (octet moyen)
10	A	EVENT HI	Compteur bits 16-23 (octet haut)
11	B		Inutilisé
12	C	SPR	Données série
13	D	ICR	Contrôle Interruption
14	E	CRA	Registre contrôle port A
15	F	CRB	Registre contrôle port B

- Les ports parallèles

Registre	Nom	D7	D6	D5	D4	D3	D2	D1	D0
0	PRA	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0
1	PRB	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0
2	DDRA	DPA7	DPA6	DPA5	DPA4	DPA3	DPA2	DPA1	DPA0
3	DDRB	DPB7	DPB6	DPB5	DPB4	DPB3	DPB2	DPB1	DPB0

Le 8520 dispose de 2 ports 8 bits parallèles PA et PB, chacun relié à un registre de données PRA et PRB. En toute conformité, le circuit dispose de 16 signaux de port, PA0-PA7 et PB0-PB7.

que *signal de sortie*. Ceci sera indiqué par le sens des données. Le 8520 autorise la modification du sens des données de chaque signal. Ainsi à chaque port correspond un registre direction de données, DDRA et DDRB (DATA DIRECTION REGISTER).

Chacun de ces registres contient 8 bits. Si l'un des bits des registres DDR est placé à 0, le bit correspondant du port (PR) est une entrée. Si ce bit est mis à 1, le bit correspondant dans PR est une sortie.

La lecture des bus de données (registre PR) donne l'état des broches d'E/S (entrée/sortie), quel que soit le sens du transfert.

Les transferts de données via le port parallèle peuvent être contrôlés au moyen des signaux PC et FLAG. Ces signaux indiquent un début de transfert de données (Handshaking). La broche PC se positionne à l'état bas au troisième cycle après tout accès au port B. Elle indique donc que des données ont été émises de ce port. La broche FLAG réagit quand le signal à son entrée passe de 1 à 0. Dans ce cas, le bit FLAG du registre de contrôle des interruptions sera positionné.

On peut réaliser un contrôle efficace en connectant la broche PC d'un 8520 sur la broche FLAG du deuxième.

Le circuit transmetteur écrit ses données sur le port du registre et doit attendre un signal FLAG pour pouvoir écrire l'octet suivant. Puisque ce signal peut libérer une interruption, le transmetteur a la possibilité, pendant une attente, de se consacrer à une autre occupation.

Les minuteries (Timer en anglais)

Accès lecture

Registre	Nom	D7	D6	D5	D4	D3	D2	D1	D0
4	TALO	TAL7	TAL6	TAL5	TAL4	TAL3	TAL2	TAL1	TAL0
5	TAHI	TAH7	TAH6	TAH5	TAH4	TAH3	TAH2	TAH1	TAH0
6	TBLO	TBL7	TBL6	TBL5	TBL4	TBL3	TBL2	TBL1	TBL0
7	TBHI	TBH7	TBH6	TBH5	TBH4	TBH3	TBH2	TBH1	TBH0

Accès écriture

Registre	Nom	D7	D6	D5	D4	D3	D2	D1	D0
4	PALO	PAL7	PAL6	PAL5	PAL4	PAL3	PAL2	PAL1	PAL0
5	PAHI	PAH7	PAH6	PAH5	PAH4	PAH3	PAH2	PAH1	PAH0
6	PBLO	PBL7	PBL6	PBL5	PBL4	PBL3	PBL2	PBL1	PBL0
7	PBHI	PBH7	PBH6	PBH5	PBH4	PBH3	PBH2	PBH1	PBH0

Le 8520 dispose de deux minuteries 16 bits. Elles sont en mesure de décompter à partir d'une valeur fixée au préalable jusqu'à 0. De plus, il existe un grand nombre de modes possibles qui sont sélectionnés avec un registre de contrôle (CONTROL REGISTER) spécifique à chaque minuterie (CRA et CRB).

Chaque minuterie est composée de 2 registres 16 bits (un pour la lecture et le deuxième pour l'écriture). Le registre d'écriture et celui de lecture ont la même adresse. Pratiquement, cela signifie que l'on ne peut pas savoir à quelle valeur la minuterie a été initialisée.

Lorsqu'un des registres de la minuterie est accédé en écriture, sa valeur est verrouillée en mémoire, chargée dans le registre du compteur puis décrémentée jusqu'à ce que le compteur atteigne une valeur négative.

compteur.

Pour obtenir une valeur correcte de l'état de la minuterie, il est nécessaire d'arrêter le compteur. Voici un exemple de ce qui peut se produire (cas typique d'un effet de bord) :

Etat du compteur à un moment donné : $0100.

Un accès lecture sur le registre nous donne d'abord l'octet haut de l'état actuel, soit $01.

Avant que l'octet bas ne puisse être lu, une impulsion peut être transmise au compteur, ce dernier étant alors décrémenté d'un pas. La valeur du compteur sera donc $00FF.

La lecture de l'octet bas donnera : $FF
On obtiendra donc pour le compteur : $01FF !

Au lieu d'arrêter le compteur, on peut employer la méthode suivante, beaucoup plus élégante : on lit l'octet haut, puis l'octet bas et à nouveau l'octet haut. Si la valeur de l'octet haut n'a pas changé, c'est que la valeur est correcte. Sinon, le processus doit être recommencé.

Le signal déterminant la décrémentation du compteur correspond, d'un part au bit 5 pour la minuterie A, et d'autre part aux bits 5 et 6 pour la minuterie B des registres de contrôle correspondants.

Il n'y a que deux sources possibles de signaux pour la minuterie A :

1) La minuterie A sera décrémentée à chaque cycle d'horloge (le CIA de l'AMIGA étant relié au signal horloge E du processeur, sa fréquence est donc de 716 KHertz) (INMODE=0).

2) Chaque impulsion sur le signal CNT décrémente le compteur (INMODE=1).

binaire, le premier ...re étant pour le bit 6, le deuxième pour le bit 5) :

1) Cycle d'horloge (INMODE = 00)
2) Impulsion CNT (INMODE = 01)
3) Combinaison avec la minuterie A (les deux minuteries correspondant à une seule minuterie 32 bits) (INMODE =10).
4) Combinaison avec la minuterie A lorsque le signal CNT est "haut" (on peut de toute façon mesurer la longueur d'une impulsion sur le signal CNT) (INMODE = 11).

Le passage à zéro d'un compteur sera indiqué dans le registre de contrôle d'interruption (ICR). Pour la minuterie A, ce sera le bit TA (bit 0) et pour la minuterie B le bit TB (bit 1). Ces bits restent actifs jusqu'à la lecture du registre ICR.

Toutefois, il reste la possibilité de diriger les minuteries vers le port B. Il faut pour cela activer le bit PB du registre de contrôle d'un des compteurs (CRA ou CRB). La minuterie A fonctionne avec le port PB6 et la minuterie B avec PB7.

Avec le bit *outmode*, on peut choisir entre deux types de sorties :

outmode = 0 Pulse-mode

Chaque décrémentation sera émise comme impulsion positive d'une durée d'un cycle d'horloge sur le port correspondant.

outmode = 1 Toggle-mode

A chaque décrémentation, le signal du port correspondant changera d'une valeur basse en valeur haute et vice-versa. A chaque départ de la minuterie, la sortie correspondra à une valeur haute.

La minuterie sera démarrée ou stoppée à parti- du bit START du registre de contrôle (0 = ARRET, 1 = DEMARRAGE).

Avec le bit RUNMODE, on peut choisir entre le *one-shot-mode* et le *continuous mode.* Le premier mode arrête la minuterie après chaque décompte et remet le bit START à 0. Avec le *continuous-mode*, le compteur recommence à la valeur de départ.

Comme cela a été précisé, l'écriture d'une valeur dans un registre minuterie ne sera pas directement exécutée, mais auparavant sauvegardée et verrouillée en mémoire (aussi appelée *Prescaler* ; le nombre de décrémentations par seconde est égal à la fréquence du compteur divisée par la valeur se trouvant dans le *Prescaler*).

Il existe plusieurs possibilités de transfert des valeurs verrouillées en mémoire vers le compteur :

1) Activer le bit LOAD du registre de contrôle. Ceci entraîne automatiquement un chargement (Force- Load). Indépendamment de l'état du compteur, la valeur verrouillée en mémoire y sera placée. Le bit LOAD est un bit STROBE, ce qui signifie que le bit ne sera pas mis en mémoire, mais permettra l'exécution d'un processus. Pour déclencher à nouveau un transfert prioritaire (*Force Load*), on doit remettre à 1 le bit LOAD.

2) A chaque fin de décrémentation des minuteries, la valeur verrouillée en mémoire est transférée dans le compteur.

3) Après un accès en écriture dans le registre minuterie octet haut, alors que le compteur est arrêté (STOP = 0), ce dernier sera chargé automatiquement avec la valeur verrouillée en mémoire. C'est pour cette raison que l'on doit retenir la suite dans l'ordre :

   - premièrement : l'octet haut (octet de poids fort)

   -deuxièmement : l'octet bas (octet de poids faible).

Détail des bits du registre de contrôle A

Registre N° 14 / $E Nom : CRA

D7	D6	D5	D4	D3	D2	D1	D0
TOD IN 0=60Hz 1=50Hz	SPMODE 0=entrée 1=sortie	INMODE 0=Horl. 1=CNT	LOAD 1=FORCE LOAD (strobe)	RUNMODE 0=cont. 1=one- shot	OUTMODE 0=pulse 1=toggle	PBON 0=PB6OFF 1=PB6ON	START 0=stop 1=start

Détail des bits du registre de contrôle B

Registre N° 15 / $F Nom : CRB

D7	D6+D5	D4	D3	D2	D1	D0
ALARM 0=TOD 1=Alarm	INMODE 00=Horl. 01=CNT 10=Timer A 11=CNT+Timer A	LOAD 1=FORCE LOAD (strobe)	RUNMODE 0=cont. 1=one- shot	OUTMODE 0=pulse 1=toggle	PBON 0=PB7OFF 1=PB7ON	START 0=stop 1=start

Le compteur (Event-counter)

Registre	Nom	D7	D6	D5	D4	D3	D2	D1	D0
8 $8	LSB Event	E7	E6	E5	E4	E3	E2	E1	E0
9 $9	Event 8-15	E15	E14	E13	E12	E11	E10	E9	E8
10 $A	MSB Event	E23	E22	E21	E20	E19	E18	E17	E16

Comme mentionné précédemment, le 8520 se différencie du 6526 par quelques modifications seulement, les seules différences se trouvant au niveau des fonctions des registres 8-11. Sur le 6526 on trouve une horloge temps réel (time of day, TOD), l'heure du jour étant sauvegardée sous forme d'heures, minutes et secondes dans des registres isolés. Dans le circuit 8520 on trouve cette horloge dans un compteur 24 bits, qui se nomme EVENT COUNTER.

On pourrait être induit en erreur, par le fait que COMMODORE utilise en partie, sur le 8520, les anciennes instructions TOD.

Le rôle de l'EVENT COUNTERS est simple : il reproduit un compteur 24 bits. Ceci signifie qu'il peut prendre les valeurs de 0 à 16 777 215 ($FFFFFF). A chaque impulsion positive (variable de la valeur basse vers la valeur haute), le signal TOD élève de 1 la valeur du compteur. Lorsque le compteur atteint $FFFFFF, il se réinitialise à 0 à la prochaine impulsion. Le compteur peut être fixé si l'on écrit la valeur désirée dans son registre.

Le registre 8 qui contient les bits 0-7 du compteur, est appelé LSB (Lowest Significant Byte = octet de plus faible poids), le registre 9 renfermant les bits 8-15, et le registre 10($A) les bits 16-23, le MSB (Most Significant- Byte = octet de plus fort poids).

A chaque accès en écriture, le compteur s'arrête, afin qu'aucune erreur ne se produise pendant les transferts de registres.

Après que la valeur ait été chargée dans le LSB, le compteur continue son travail. Dans l'ordre normal, le registre 10 (MSB), puis le registre 9 et enfin le registre 8 sont alors pris en compte.

S'il n'apparaît aucune erreur de transfert lors de la lecture de la valeur du compteur, cette dernière sera sauvegardée et verrouillée en mémoire pendant la lecture du MSB (registre 10). Chaque accès au registre du compteur fournit la valeur verrouillée pendant que le compteur interne tourne, la lecture pouvant se faire en toute tranquillité. La valeur sera à nouveau verrouillée lorsqu'on tentera de lire le LSB. Si l'on veut lire le compteur en entier, il faudra, comme en écriture, lire en premier le MSB, puis le registre 9 et enfin le LSB.

En outre, une fonction alarme est intégrée ; si l'on met dans le registre de contrôle B bit alarm (n°7) à 1, on peut écrire dans les registres 8-10, une valeur alarme.

Aussitôt que la valeur du compteur correspond à celle de l'alarme, le bit alarme du registre de contrôle des interruptions sera activé. La valeur de l'alarme ne pourra qu'être prise en compte. Un accès lecture sur les adresses 8-10 ne donne que l'état actuel du compteur, et ceci que le bit alarme du registre de contrôle soit fixé ou non.

Le port série

Registre	Nom	D7	D6	D5	D4	D3	D2	D1	D0
12 $C	SDR	S7	S6	S5	S4	S3	S2	S1	S0

Le port série se compose d'un registre de données séries (SDR, Serial Data Register) et d'un registre à décalage, sur lequel on n'a pas d'accès direct. Avec le bit SPMODE du registre de contrôle A, on peut le configurer soit en mode entrée (SPMODE = 0), soit en mode sortie (SPMODE = 1).

Dans le premier mode, les données séries sur le signal SP seront décalées dans le registre de même nom à chaque impulsion positive donnée par le signal CNT. Après 8 impulsions, le registre à décalage est plein et son contenu est transféré dans un registre de données. En même temps le bit SP du registre contrôle des interruptions est activé. Si une nouvelle impulsion CNT se présente, les données se décalent à nouveau dans le registre jusqu'à ce que celui-ci soit totalement renouvelé. Si entre temps, l'utilisateur a lu le registre de données séries (SDR), la nouvelle valeur sera copiée dans SDR, et les transferts se dérouleront continuellement suivant le même schéma.

Pour pouvoir utiliser le port série en tant que sortie, on met le bit SPMODE à 1.

La fréquence de la minuterie A détermine le débit (en bauds ou bits par seconde). Les données seront toujours extrait[illegible] du registre à décalage suivant la demi-fréquence de la minuterie, le taux de sortie maximum équivalant au quart de la fréquence d'horloge du 8520.

Le transfert commence après que le premier octet des données ait été inscrit dans SDR. Le CIA transfère l'octet dans le registre à décalage. Les bits des données apparaissent sur le signal SP suivant la fréquence réduite de moitié de la minuterie A, le signal d'horloge de cette minuterie s'appliquant sur le signal CNT.

Le transfert commence avec le MSB des octets des données. Lorsque les huit bits sont émis, CNT reste sur l'état actif et le signal SP, quant à lui, reste au niveau du dernier bit émis. De plus, le bit SP sera mis dans le registre de contrôle, afin de montrer que le registre de décalage peut être remplacé par de nouvelles données. Si l'octet suivant est inscrit dans le registre de données avant l'émission du dernier bit, l'émission des données se poursuivra sans interruption.

Si on veut un transfert continu, on doit alimenter opportunément le registre de données série en nouvelles données.

Les signaux SP et CNT sont utilisés comme des collecteurs entrée/ sortie. Ces signaux permettent aussi la direction de plusieurs circuits de type 8520.

Le registre de contrôle d'interruption (ICR)

Accès lecture (READ) = registre de données

Registre	Nom	D7	D6	D5	D4	D3	D2	D1	D0
13 $D	ICR	IR	0	0	FLAG	SP	Alarm	TB	TA

Accès écriture (WRITE) = registre masque

Registre	Nom	D7	D6	D5	D4	D3	D2	D1	D0
13 $D	ICR	S/C	x	x	FLAG	SP	Alarm	TB	TA

L'ICR se compose d'un registre de données et d'un registre masque. Chacune des 5 sources d'interruption trouve un bit correspondant dans le registre des données.

Voici un rappel des cinq sources d'interruptions possibles :

1 : passage à zéro de la minuterie A (TA, Bit 0).

2 : passage à zéro de la minuterie B (TB, Bit 1).

3 : concordance de la valeur de l'Event-Counter avec celle de l'alarme (alarm, bit 2).

4 : registre à décalage du port série plein (entrée) ou vide (sortie) (SP, bit 3).

5 : niveau négatif de l'entrée FLAG (FLAG, bit 4).

Lorsqu'on lit le registre ICR, on obtient toujours la valeur du registre de données, lequel sera effacé (tous les bits actifs IR inclu, sont remplacés). Si on a encore besoin de cette valeur, on devra la sauvegarder en mémoire (RAM) après la lecture.

Le registre masque ne supporte que le mode écriture. Sa valeur détermine si un bit dans le registre de données, libère ou non une interruption. Afin qu'une de ces dernières soit possible, le bit correspondant du registre masque doit être mis à 1. Le 8520 met le signal IRQ à 0 dès qu'un bit, dans un des registres, est activé, et met le bit 7, IR, dans le registre de données, afin qu'il signale une interruption au niveau du software.

Lorsque le registre ICR sera lu et que le registre des données sera effacé, le signal IRQ se remettra à nouveau à 1.

On peut écrire dans le registre masque de la même façon que dans un autre emplacement mémoire. Afin d'activer un bit du registre masque, on doit aussi activer le bit S/C (SET/CLEAR, bit n°7), les autres bits restant non influents. Pour effacer un bit, on doit activer le bit correspondant, le bit S/C devant être, lui, mis à 0. Le bit S/C détermine donc si les bits activés du masque effacent (S/C= 0) ou activent (S/C=1) les bits correspondants du registre masque.

Tous les bits effacés dans le masque n'ont aucune conséquence sur les bits du registre masque.

Exemple :

La valeur actuelle du registre masque est 0000 0011. On désire autoriser l'interruption du signal FLAG.

On écrit pour cela dans le registre masque la valeur 1001 0000 binaire (S/C = 1, FLAG = 1).

Le contenu du registre masque est maintenant : 0001 0011.

Si l'on veut interdire les deux interruptions minuterie, on écrit la valeur suivante : 0000 0011 (S/C = 0, TA = 1, TB = 1). Les bits TA et TB seront supprimés du registre masque.

Le registre masque renferme 0001 0000, et ainsi seule l'interruption FLAG est autorisée.

Le rôle du CIA dans le système AMIGA

Comme les entrées le précisent, l'AMIGA possède deux CIA du type 8520. L'adresse de base du premier 8520 (appelé 8520 A) est BFE001. Les écarts entre les registres sur l'espace adresse sont de 256 octets. Tous les registres du 8520 A se trouvent à des adresses impaires, ce dernier étant relié aux bus de données du processeur par les 8 signaux inférieurs (D0-7).

Les tableaux suivants donnent la liste des adresses des registres avec leur utilisation po. 'AMIGA.

CIA - A : adresses des registres

ADRESSE	NOM	D7	D6	D5	D4	D3	D2	D1	D0
$BFE001	PRA	/FIR1	/FIR0	/RDY	/TK0	/WPRO	/CHNG	/LED	/OUL
$BFE101	PRB	port parallèle							
$BFE201	DDRA	0	0	0	0	0	0	1	1
$BFE301	DDRB	utilisés pour marquer la direction des ports (entrées/sorties)							
$BFE401	TALO	minuterie A-utilisée en permanence pour le test							
$BFE501	TAHI	du clavier							
$BFE601	TBLO	minuterie B: nécessaire pour différentes							
$BF 701	TBHI	tâches							
$BFE801	E.LSB	compteur événement. Il compte les impulsions à							
$BFE901	E.8.15	50HZ du compteur d'alimentation, qui dérive							
$BFEA01	E.MSB	de la fréquence du réseau							
$BFEB01		inutilisé							
$BFEC01	SP	Registre de données séries (clavier)							
$BFED01	ICR	Registre de contrôle d'interruption							
$BFEE01	CRA	Registre de contrôle A							
SBFEF01	CRB	Registre de contrôle B							

Le CIA-B a son adresse de base à $BFD000. Ses registres se trouvent à des adresses paires. Les bus de données du CIA-B sont reliés par les signaux supérieurs aux bus de données du processeur.

CIA - B : adresses des registres

ADRESSE	NOM	ADRESSES DES REGISTRES
$BFD000	PRA	DTR /RST /CD /CTS /DER /SEL /POUT/BUSY
$BFD100	PRB	MTR /SEL3 /SEL2 /SEL1 /SEL0 /SIDE /DIR /STEP
$BFD200	DDRA	1 1 0 0 0 0 0 0
$BFD300	DDRB	1 1 1 1 1 1 1 1
$BFD400	TALO	La minuterie A n'est utilisée que pour le
$BFD500	TAHI	transfert de données séries
$BFD600	TBLO	La minuterie B est utilisée par le blitter en
$BFD700	TBHI	mode synchrone pour le transfert d'images
$BFD800	ELSB	L'event counter du CIA-B compte les impulsions
$BFD900	E.8.15	synchrones horizontales ; en temps normal elles
$BFDA00	EMSB	ont une fréquence de 15625 par seconde
$BFDB00		inutilisé
$BFDC00	SP	registre de données séries
$BFDD00	ICR	registre de contrôle d'interruption
$BFDE00	CRA	registre de contrôle A
$BFDF00	CRB	registre de contrôle B

Les deux adresses $BFD00 du CIA-B et $BFE001 du CIA-A sont données par COMMODORE comme étant les adresses de base du CIA. En examinant le schéma de montage, vous pouvez remarquer que les deux CIA sont entièrement adressés de A0XXXX à BFXXXX. Le choix entre ces deux circuits est déterminé par les signaux d'adresses A12 et A13. CIA-A sera sélectionné lorsque A12 sera égal à 0 et CIA-B, lorsque A13 sera égal à 0. Les adresses quant à elles, seront toujours prédéfinies et s'étaleront entre l'adresse A0XXXX et BFXXXX.

Du fait des liaisons des bus de données du CIA-A et du CIA-B avec les bus de données, respectivement, D0-7 et D8-15, on a la possibilité de transférer des mots (16 bits) lorsque A12 et A13 sont à 0.

MOVE.W $BF0000, D0 charge le registre PA des deux CIA dans D0, les 8 bits de poids faibles de D0, renfermant le contenu du registre PA du CIA-A, les bits 9-15, renfermant le contenu du registre CIA-B.

On peut remarquer la façon d'adresser le CIA :

Le CIA-A sera sélectionné par une adresse binaire de type :

```
101X XXXX XX01 RRRR XXXX XXX0
```

Le CIA-B :

```
101X XXXX XX10 RRRR XXXX XXX1
```

Les 4 bits décrits par R permettent la sélection d'un des 16 registres du CIA.

Ceci ne fonctionne qu'avec l'AMIGA 1000. Il est possible que vous ayez à le modifier sur les nouveaux modèles, et à utiliser les adresses recommandées par COMMODORE (CIA-A $BFE001 et CIA-B $BFD000).

La liste suivante donne les références des différents signaux des CIA de l'AMIGA.

CIA-A

IRQ	INT2 entrée de PAULA	
RES	broche de reset system	
D0-D7	bus de données du processeur bits 0-7	
A0-A3	bus d'adresse du processeur bits 8-11	
Phi 2	horloge E du processeur	
R/W	processeur R/W	
PA 7	port manette 1/broche 6 (bouton de tir)	
PA 6	port manette 0/broche 6 (bouton de tir)	
PA 5	/RDY	"disk ready", disque prêt
PA 4	/TKO	"disk track 0", disque piste 0
PA 3	/WPRO	"write protect", protection en écriture
PA 2	/CHING	"disk change", disque changé
PA 1	/LED	état de la diode LED (0= allumé)
PA 0	/OVL	"memory overlay bit" recouvrement mémoire

SP	KDAT	données série du clavier
CNT	KCLK	horloge clavier
PB0-PB7	signaux de données pour port parallèle (centronic)	
PC	/DRDY	données prêtes pour port parallèle
FLAG	/ACK	acquittement pour port parallèle

CIA-B

/IRQ	/INT 6-entrée de PAULA	
/RES	signaux de reset system	
D0-D7	bus de données du processeur bits 8-15	
A0-A3	bus d'adresse du processeur bits 8-11	
Phi 2	horloge E du processeur	
R/W	processeur R/W	
PA 7	/DTR	connecteur série sortie DTR
PA 6	/RTS	connecteur série sortie RTS
PA 5	/CD	connecteur série sortie CD (carrier detect)
PA 4	/CTS	connecteur série sortie CTS
PA 3	/DSR	connecteur série sortie DSR
PA 2	SEL	"SELECT" contrôle port parallèle
PA 1	POUT	"paper out", papier absent (imprimante sur port centronic)
PA 0	BUSY	"busy", imprimante occupée sur port parallèle
SP	BUSY	imprimante occupée port A bit 0
CNT	POUT	imprimante occupée port A bit 1
PB 7	/MTR	"motor" moteur lecteur de disquette
PB 6	/SEL 3	"drive select" sélection lecteur disquette n°3 (dF3:)
PB 5	/SEL 2	"drive select" sélection lecteur disquette n°2 (dF2:)
PB 4	/SEL 1	"drive select" sélection lecteur disquette n°1 (dF1:)
PB 3	/SEL 0	"drive select" sélection lecteur disquette interne (dF0:)
PB 2	/SIDE	"side select" sélection de la face de la disquette
PB1	DIR	"direction" direction d'avancement du moteur de lecteur de disquette
PB 0	STEP	"step", avance d'un pas du moteur du lecteur de disquette
FLAG	/INDEX	"index", début du cylindre (disquette)
PC	non utilisé.	

1.2.3 ROLE DES CIRCUITS SPECIALISES

DANS LE HARDWARE DE L'AMIGA

Les processeurs dont nous avons parlé jusqu'à présent, sont assez banals. Même le 68000 n'est qu'un circuit standard, qu'on peut trouver pour quelques centaines de francs dans n'importe quel magasin d'électronique.

Tout de même, les fans et les utilisateurs de l'Amiga ont d'autres aspects en vue.

Que l'ordinateur soit capable de calculer d'importantes quantités de feuilles de salaire par secondes, ou qu'il soit plus rapide qu'un vieux calculateur, ne sont pas les critères prépondérants pour l'achat d'un AMIGA.

Le fait d'avoir la possibilité d'afficher et de traiter des images de qualité télévisuelle, tout en écoutant la neuvième de Beethoven, une telle sonorité, que l'observateur cherche en vain une platine peut attirer l'attention.

Les concepteurs de l'AMIGA l'ont pourvu de capacités graphiques et sonores qui n'ont jamais été égalées par un ordinateur de cet ordre de prix.

Le but de ce chapitre est d'éclairer, d'expliquer le HARDWARE de l'AMIGA qui permet ces fantastiques possibilités graphiques et sonores, afin de donner au lecteur une base pour sa programmation.

Les éléments de base permettant les possibilités citées plus haut, se réduisent à 3 circuits.

Ils ont été conçus pour l'AMIGA et portent le nom de *circuits spécialisés* (on appelle circuit spécialisé un circuit intégré, conçu par une firme spécialisée dans les semi-conducteurs, pour une machine déterminée, dans un but précis).

Leur dénomination sont respectivement 8361, 8362, 8364.

Cette désignation étant trop lourde pour les concepteurs, ceux-ci les baptisèrent AGNUS, DENISE, PAULA (l'AMIGA f. çais comporte un circuit à la norme télévisuelle PAL, qui est en fait, une version ultérieure de AGNUS, sa dénomination étant 8367).

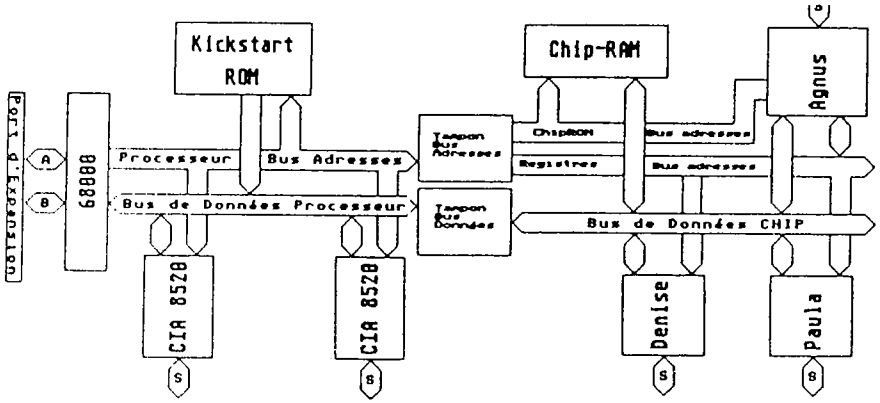
Ces puces prennent en charge la création des sons, la reproduction d'images et les accès disquette indépendamment du processeur, plus d'autres fonctions diverses. Ces différentes tâches ne sont pas séparées sur plusieurs circuits, afin que l'un s'occupe du son, l'autre du graphisme et le troisième des opérations de lecture sur disquette.

Au contraire, les tâches se répartissent sur plusieurs circuits à la fois, ainsi, par exemple, la reproduction du graphisme est assurée par deux des trois circuits intégrés.

Pour une telle mise en commun des tâches, on aurait pu réunir les 3 circuits au sein d'un seul circuit intégré. Mais la conception d'un tel circuit spécialisé serait revenue trop chère par rapport à la conception des trois séparés.

Avant de voir plus en détail les fonctions de AGNUS, DENISE et PAULA, voici une courte introduction à la structure de l'AMIGA.

1.2.3.1. L'architecture de l'Amiga



(S) sont des lignes d'E/S vers les différentes interfaces

Figure 1.2.3

La structure d'un ordinateur se compose simplement d'un processeur, d'une ROM (mémoire morte) contenant le système d'exploitation, d'une quantité de mémoire vive (RAM) connue, et d'au moins une structure périphérique assurant la gestion des entrées/sorties.

Il ne faut pas oublier les bus d'adresses et les bus de données. En effet, le processeur contrôle le système en envoyant ou en recevant les messages des composantes du système par le biais de ces bus.

Il contrôle aussi les signaux de direction du bus, comme par exemple, les signaux READ/WRITE.

Chaque système d'ordinateur comporte en outre des circuits de contrôle, comme un décodeur d'adresse, qui, pour une valeur déterminée sur le bus d'adresse, active un circuit affecté à cette adresse.

Comme on peut le remarquer sur le schéma 1.2.3, la structure de l'AMIGA diffère un peu de cette courte description.

On s'aperçoit que de nombreuses broches, présentes sur l'ancien circuit, n'apparaissent plus sur le schéma. Elles étaient en fait reliées à des circuits qui font maintenant partie intégrante de FAT-AGNUS, ce qui explique la présence de nouveaux signaux.

Sur la gauche on voit le processeur 68000 relié directement aux 2 CIA et à la ROM, par le biais des signaux de données et d'adresses. Cette partie de l'AMIGA est en fait conventionnelle, seul le processeur ayant accès à la ROM et aux deux CIA.

Sur la droite du schéma, on trouve les trois circuits spécialisés AGNUS, DENISE et PAULA, ainsi que la mémoire interne RAM, qui sont reliés directement entre eux, via le bus de données et séparés du processeur par un buffer (tampon). Le 68000 peut être relié ou séparé, suivant son choix, par le circuit bus de données.

Les trois circuits spécialisés peuvent être liés par le registre bus d'adresse, le processeur pouvant, selon son choix, être relié par le bus d'adresses processeur.

Etant donné que la CHIP-RAM possède une grande zone adressable et qu'elle nécessite un adressage multiplexé, ses accès transitent par un bus d'adresses particulier (bus CHIP-RAM).

Voici une petite explication de ce qu'est l'adressage multiplexé :

La CHIP-RAM de l'Amiga (A1000) est composée de 2^{16} adresses (65536). On a donc besoin de 16 signaux d'adresses pour pouvoir accéder à la totalité de la CHIP-RAM. Le boitier ne possède que 8 signaux d'adresse pour des raisons de proportions. C'est pour cela que l'on a introduit l'adressage multiplexé : les 8 bits de poids fort seront d'abord véhiculés puis les 8 bits de poids faible. L'octet de poids fort sera stocké en mémoire en attendant que l'octet le moins significatif soit obtenu.

La raison de la séparation du bus d'adresses processeur et du bus CHIP-RAM est que les différents signaux d'E/S nécessitent un approvisionnement continuel. Par exemple, les données se rapportant à un point de l'écran doivent être lues dans la RAM cinquante fois par seconde (norme PAL).

Un graphique haute résolution peut utiliser jusqu'à 64 Ko de mémoire écran. A chaque seconde, 50*64 Ko doivent être transférés de la mémoire vers l'écran, ce qui correspond à environ 1,5 millions d'accès mémoire par seconde. Le 68000 ne pouvant traiter autant de données par seconde, un tel travail serait au dessus de ses capacités.

L'AMIGA peut, en plus du graphisme, sortir des sons digitalisés tout en gérant des accès au lecteur de disquettes et ceci sans utiliser le 68000.

Il existe en fait un deuxième processeur, qui gère tout seul ces accès mémoire. Il s'agit du DMA-CONTROLER, inclus dans le circuit AGNUS. Pour cette raison, AGNUS est aussi reliée au bus d'adresses CHIP-RAM.

Les deux autres circuits spécialisés, DENISE et PAULA, ainsi que le reste d'AGNUS, sont structurés comme des circuits périphériques. Ils possèdent une grande quantité de registres qui peuvent être lus ou écrits par le processeur (ou le contrôleur DMA). Les registres sont sélectionnés sur le bus d'adresses registres.

Il possède 8 signaux et peut donc prendre 256 états différents. Il n'y a pas de sélection spéciale des circuits. Si le bus d'adresses véhicule la valeur 255 ($FF), les signaux sont tous activés (mis à 1) et aucun n'est sélectionné. La sélection est effectuée par le décodeur d'adresse registre.

Ainsi, puisque la sélection d'un registre ne dépend que de son adresse registre et non du circuit dans lequel il se trouve, il est possible de mettre la même valeur dans les registres de deux circuits différents, lorsqu'ils ont les mêmes adresses registres. Cette possibilité est utilisée par certains registres qui contiennent des données nécessaires à plusieurs circuits.

Tous les registres d'un circuit peuvent aussi bien être accédés en lecture qu'en écriture. La commutation entre lire et écrire se fait par moyen d'un signal R/W spécial (ceci, par exemple, n'existe pas d[illegible] 8520).

L'adresse registre détermine le mode d'accès (lecture ou écriture). registres qui acceptent ces deux modes sont réalisés de telle façon que les accès lecture et écriture ont lieu sur des adresses registres différentes.

Etant donné qu'AGNUS renferme le contrôleur DMA, il peut avoir lui-même accès à tous les registres des circuits spécialisés.

Si deux circuits délivrent en même temps des données sur le même bus, il en résultera une collision des données suivie d'une rupture du système. Les circuits doivent donc se partager le bus de façon alternée, ce qui a été réalisé sur AMIGA d'une élégante manière :

Premièrement, le bus d'adresses et le bus de données de l'AMIGA sont séparés en deux. La première partie relie tous les circuits qui peuvent dialoguer avec le processeur. Les liaisons entre les bus d'adresses et de données du processeur et ceux des circuits se faisant par l'intermédiaire des deux buffers (tampons).

Ainsi, le processeur et AGNUS peuvent avoir un accès protégé au système d'exploitation ou à une extension mémoire RAM.

Cette dernière est souvent appelée *FAST RAM,* é nt donné que le processeur peut y accéder sans perte de vitesse.

Deuxièmement, les accès au bus par le processeur et par AGNUS sont encastrés l'un dans l'autre, afin que les accès à la mémoire ou aux registres du 68000 ne soient pas freinés. Les tampons relient les deux systèmes de bus lors d'un tel accès.

Troisièmement, pour accéder aux bus, le processeur doit attendre qu'AGNUS les ait libérés. Ce cas de figure ne se présente que lorsque la haute résolution graphique est utilisée ou quand le blitter est mis à contribution.

L'AMIGA possède en fait trois types de RAM. La mémoire vidéo, que l'on appelle CHIP-RAM, la mémoire rafraîchie qui sert également à l'affichage vidéo et la FAST-RAM. Sur le modèle A1000, les 256 Ko de base sont de la CHIP-RAM et toute extension mémoire sera de la FAST-RAM. Les modèles A500 et B2000 possèdent 512 Ko de CHIP-RAM, les 512 Ko suivants forment la mémoire rafraîchie et au-dessus de 1 Mo on trouve de la FAST-RAM.

Brochage du 8361

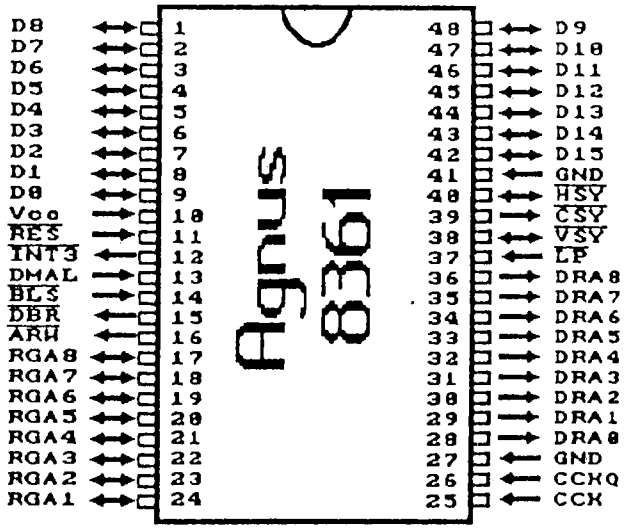


Figure 1.2.3.1

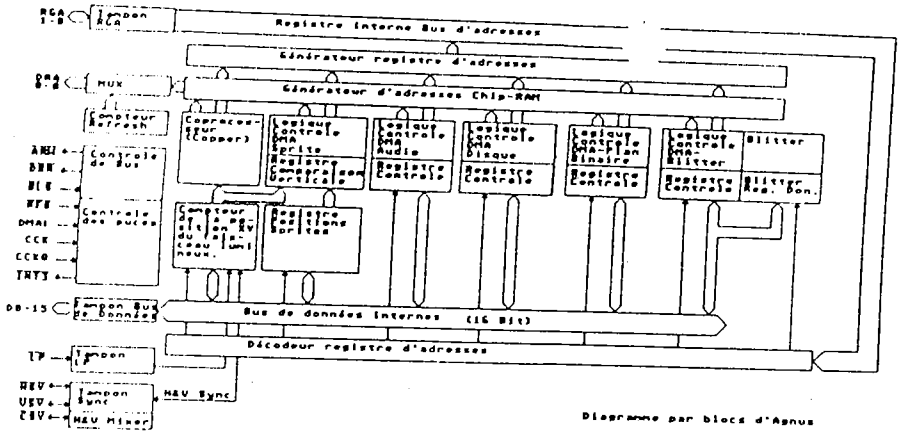


Figure 1.2.3.2

Comme mentionné plus haut, AGNUS renferme les canaux d'accès direct à la mémoire (DMA). Il existe pour chaque source DMA possible, un contrôle logique. Ces derniers sont reliés au générateur d'adresse mémoire, ainsi qu'au générateur d'adresse registre. Ces générateurs d'adresses produisent les adresses RAM des emplacements mémoire souhaités et les adresses registres des registres cibles. C'est pour cette raison que les canaux DMA ravitaillent les registres correspondants en données issues de la RAM ou écrivent le contenu de registres déterminés dans la RAM.

Il existe aussi un compteur dynamique lié au générateur d'adresse mémoire, qui produit un rafraîchissement du signal, permettant ainsi l'exploitation de la mémoire dynamique.

AGNUS dirige aussi le déroulement des accès DMA, par le biais d'un élément de base : une ligne écran. A chaque affichage d'une ligne écran, il y a 225 accès mémoire qui sont répartis par AGNUS entre les canaux DMA et le 68000. Comme AGNUS a besoin de connaître la position actuelle de la ligne et du point affiché, il mémorise aussi la trame désignée de l'écran et un compteur de colonnes.

Ce compteur de coordonnées est à l'origine des signaux de synchronisation horizontale et verticale, qui signalent à l'écran raccordé, le début d'une nouvelle ligne (H- SYNC) et d'une nouvelle image (V- SYNC).

Les signaux horizontaux et verticaux peuvent provenir d'une autre source que le circuit AGNUS, ce qui permet à l'image de l'AMIGA, d'être synchronisée avec une image extérieure, par exemple celle d'un magnétoscope. Ce système, aussi appelé *GENLOCK*, peut être facilement réalisé sur AMIGA.

Le *Blitter* et le coprocesseur *Copper* sont deux autres éléments importants du circuit AGNUS. Le *Blitter* est un circuit spécialisé, dont le rôle est de manipuler et de décaler les zones mémoire. Il peut ainsi soulager le 68000 de ces tâches et permettre un traitement plus rapide des données. Son rôle est aussi de tracer des lignes et de remplir des surfaces.

Le *Copper* est un coprocesseur, dont les programmes, aussi appelés *listes du Copper*, sont composés de trois instructions différentes (MOVE, WAIT et SKIP). Son rôle est de gérer les différents registres du circuit spécialisé. Des informations plus complètes se trouvent au chapitre : **Programmation des circuits intégrés**.

Voici une description détaillée des broches d'AGNUS :

Bus de données D0-15

Les 16 signaux de données sont directement reliés au circuit bus de données RAM, par le biais d'un tampon (buffer) interne.

Bus d'adresse registre RGA 0-8.

Les bus d'adresses registre sont bidirectionnels. Lors d'un accès DMA, le générateur d'adresses registre lâche l'adresse registre sur le bus. Lorsque le processeur intervient, ces signaux fonctionnent en mode entrée, et les adresses registre choisies par le processeur sont placées sur le décodeur d'adresses registre à l'intérieur d'AGNUS.

Les signaux d'adresses de la mémoire dynamique : DRA0-DRA8.

Ces signaux d'adresses sont reliés au bus d'adresses circuit RAM. Ce sont des signaux de sortie qui seront activés par AGNUS lors des accès DMA sur la CHIP-RAM. Les adresses, sur ces broches, sont multiplexées et peuvent relier directement 32 bits de mémoire dynamique avec les signaux d'adresses. C'est le cas pour les AMIGA 500 et 2000, le modèle précédent A 1000 ne possèdant que 8 signaux d'adresses. Le signal DRA8 d'AGNUS sera démultiplexé et employé pour la sélection entre les différentes banques RAM.

Les signaux d'horloge d'AGNUS CCK et CCKQ.

Ce sont les seuls signaux d'horloge de l'AMIGA. La fréquence des deux signaux s'élève à 3,58 MHertz, soit la demi fréquence du processeur. Le signal CCKQ est ralenti d'un quart de cycle d'horloge par rapport au signal CCK. Le timing de l'AGNUS est régi par ces deux signaux.

Les deux signaux sont reliés à la direction logique de l'AMIGA. Avec le signal DBR (Data Bus Request/demande de bus), AGNUS communique à cette direction logique qu'il prendra le bus en charge au prochain cycle de bus. Ce signal est toujours prioritaire pour une demande de bus du processeur. Si AGNUS a besoin du bus pendant plusieurs cycles, le 68000 devra patienter.

Le signal *ARW* (écriture mémoire par AGNUS) indique à la direction logique qu'AGNUS veut avoir un accès écriture sur la carte mémoire.

Le signal *BLS* (Blitter Slow Down/Ralentir Blitter) signale à AGNUS que le processeur attend pour un accès depuis trois cycles. Suivant l'état interne, AGNUS peut cèder le bus au processeur pendant un cycle.

Les signaux de direction : RES, INT3, DMAL

Le signal *RES* (RESET) est relié directement au signal RESET du processeur et réinitialise AGNUS.

Le signal *INT3* (Interruption #3) est une entrée qui est reliée au signal de même nom circuit PAULA. AGNUS signale à la logique d'interruption de PAULA, qu'une composante d'AGNUS a libéré une interruption.

Le signal *DMAL* (DMA Request Line) relie, dans tous les cas, PAULA à AGNUS. PAULA signale ainsi qu'AGNUS doit entamer un transfert DMA.

Les signaux : HSY, VSY, CSY et LP.

Les signaux de synchronisation du moniteur proviennent des signaux *HSY* (Horizontal SYnc/synchronisation horizontale) et *VSY* (Vertical SYnc/synchronisation verticale). Le signal *CSY* (Composite SYnc) est la somme des signaux *HSY* et *VSY*. Il sert à la connexion d'un moniteur, qui demande un signal synchronisé mélangé et est utilisé par le vidéomixer, qui génère le signal vidéo.

Le signal *LP* (Light Pen) permet la connexion d'un stylo lumineux ou photostyle. Un signal négatif à cette broche entraînera l'acquisition des coordonnées en mémoire (cf. 1.5.2).

L'AMIGA peut recevoir un signal vidéo. La synchronisation exte d'AGNUS (GENLOCK) utilise les signaux *HSY* et *VSY*.

Brochage du 8362

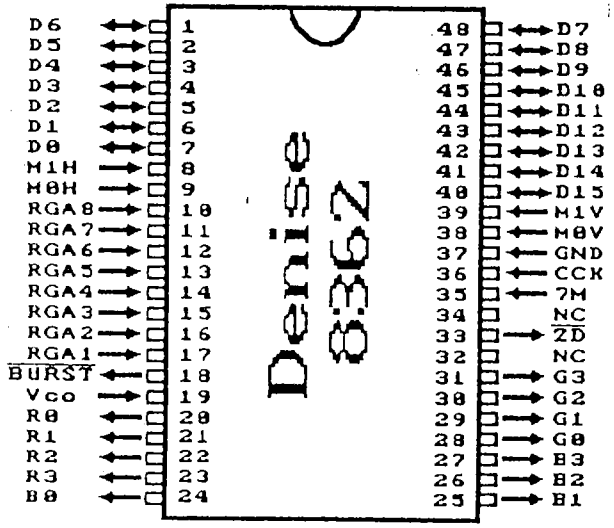


Figure 1.2.3.3

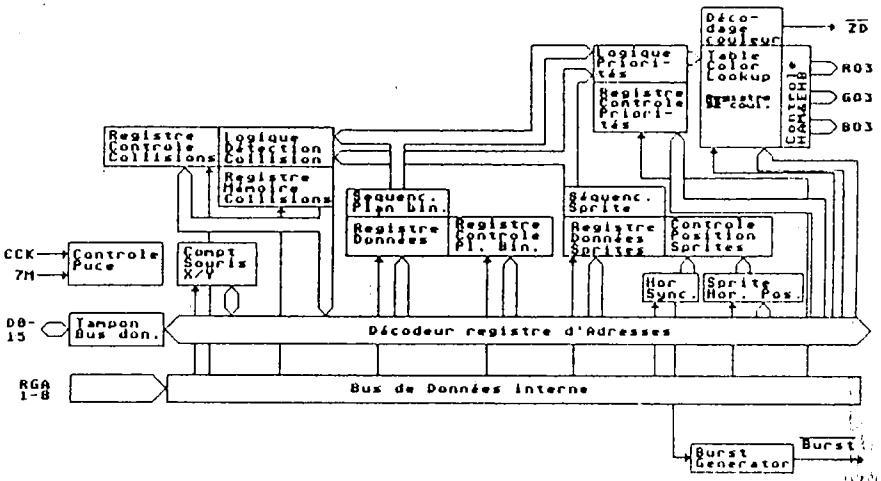


Figure 1.2.3.4

DENISE a pour rôle de générer les images. Les données graphiques en provenance de la CHIP-RAM sont transférées par AGNUS vers les registres BitMap (littéralement : Plan binaire) de DENISE. Le procédé est le même pour les sprites (lutins). La taille des registres de données de DENISE est de 16 bits. Chaque registre contiendra donc 16 points (un point par bit). DENISE va maintenant transformer ces données pour la visualisation RGB.

Un séquenceur va tout d'abord transformer les 16 points contenus dans le registre en signaux électriques séries. A chacun des 6 BitMaps correspond une transformation. Les données séries électriques en provenance du séquenceur BitMap formeront donc un signal d'une longueur de 6 bits.

Ensuite, le contrôle logique prioritaire, sur la base des priorités mises en place, va sélectionner les données valides du point parmi celles du séquenceur BitMap et celles du séquenceur Sprite.

A partir de ces données, le décodeur couleur va choisir un des 32 registres couleur.

La valeur de ce registre sera traduite sous la forme d'un signal digital *RGB* (Red, Green, Blue/Rouge, Vert, Bleu). Si l' des modes *HAM* (Hold and Modify) ou *EHB* (Extra Half Brite) est sélectionné, les données des registres couleur seront modifiées avant que le signal quitte le circuit. Les données du séquenceur passent aussi par le contrôle logique des collisions qui indiquera si un sprite occupe la même position sur l'écran qu'une image.

La dernière fonction de DENISE n'a rien à voir avec la représentation à l'écran. En effet ce circuit renferme également le compteur de la souris, ou autrement dit, les coordonnées x-y de la souris.

Voici la description des fonctions de chaque broche :

Bus de données : D0-D16

Les signaux bus de données sont reliés, comme AGNUS, au circuit bus de données.

Bus adresse registre : RGA1-RGA8

Le bus adresse registre se comporte seulement comme une entrée. Avec l'aide de la valeur du bus adresse registre, le décodeur d'adresse registre sélectionne le registre interne correspondant.

Les entrées Horloge : CCK et 7M

Le timing de DENISE s'aligne sur le signal CCK. La broche CCK est reliée à celle du circuit AGNUS. L'impulsion de l'Horloge du signal 7M a une fréquence de 7,15 MHertz. Le circuit nécessite une telle fréquence d'horloge, pour le traitement des points images. Un point en basse résolution (320 points/ligne) a exactement la durée d'un cycle d'horloge 7M.

En haute résolution (640 points/ligne), 2 points sont émis à chaque impulsion d'horloge 7M. Cette dernière correspond aussi à celle du processeur 68000, la broche 7M sera reliée avec son entrée CLK (entrée horloge).

Les signaux de sortie : R0-3, G0-3, B0-3,ZD et BURST.

Les signaux R0-3, G0-3 et B0-3 reproduisent le signal de sortie RGB de DENISE, ce dernier émettant la valeur sous forme digitale. Les trois composantes couleur seront reproduites à l'aide de 4 bits. Ceci fait en tout 16 valeurs par composante et totalise donc 16x16x16 (4 096) couleurs possibles. Les trois signaux couleur, après que DENISE les ait laché, passeront par un tampon (buffer) et seront transformés en signal analogique RGB au moyen de trois transformateurs digitaux/ analogiques, afin qu'ils puissent se tenir à disposition sur le port RGB.

Un mélangeur vidéo transforme alors ce signal analogique en signal vidéo pour le connecteur vidéo. Il nécessite pour cela le signal BURST de DENISE, qui est en fait une oscillation d'une fréquence de CCK, c'est-à-dire de 3,58 MHertz.

Pour plus de précision sur ce signal, reportez-vous à un manuel traitant de la technique de la télévision.

Le dernier signal de DENISE est le signal *ZD* (zero detect ou background indicator/indication de la couleur de fond). Il est toujours désactivé, lorsqu'un point de la couleur du fond est reproduit, couleur dérivant du registre couleur numéro 0.

Ce signal sera utilisé pour un adaptateur *GENLOCK* et servira dans ce cas de commutateur entre le signal vidéo externe (ZD=0) et le signal vidéo de l'amiga (ZD=1), le signal ZD s'applique sur le port RGB.

Les entrées souris-joystick : M0H, M1H, M0V, M1V.

Ces quatre signaux correspondent directement aux entrées souris ou joystick des deux connecteurs.

Etant donné que l'AMIGA possède deux connecteurs, il aurait fallu 8 entrées libres, alors que DENISE n'en possède que 4. Ce problème a été résolu par COMMODORE de la façon suivante :

Les huit signaux d'entrée des deux connecteurs accèdent à un commutateur, dont le rôle est de répartir les 4 signaux de chaque connecteur sur les 4 entrées de DENISE.

Ce commutateur fonctionne de façon synchrone avec l'horloge de DENISE, ceci permettant aux quatre signaux intern d'être partagés en deux registres, un pour chaque connecteur.

1.2.3.3 L'architecture de PAULA

Brochage du 8364

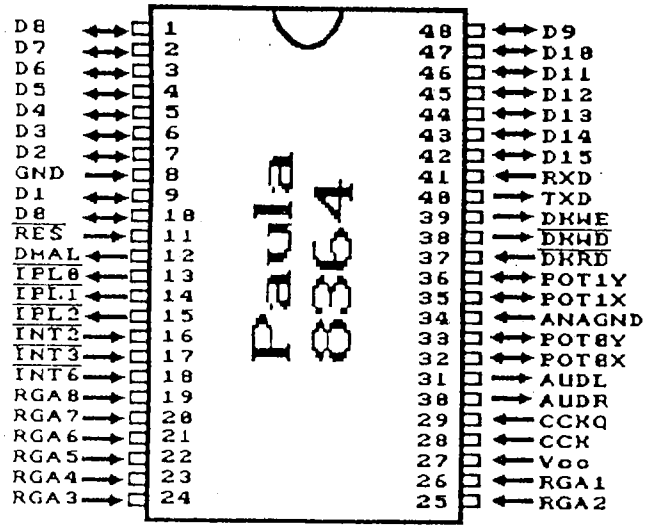


Figure 1.2.3.5

Diagramme par blocs de Paula

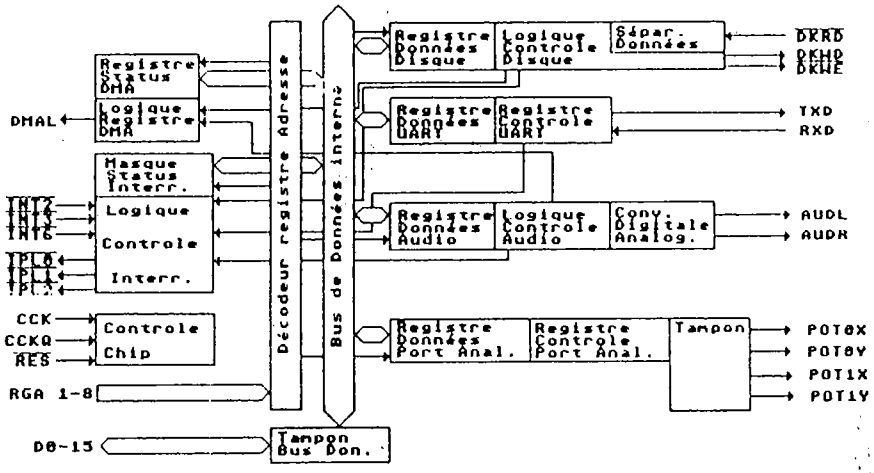


Figure 1.2.3.6

Le processeur spécialisé PAULA gère les traitements des entrées/sorties (Input/output - I/O), disquettes, séries, sons et l'entrée analogique.

De plus, Paula sert de support à la gestion des interruptions. Toutes les interruptions présentes dans le système passent par lui. A partir de 14 sources d'interruptions possibles, PAULA engendre les signaux d'interruption pour le 68000. Les signaux d'interruption des plans binaires seront générés sur les signaux IPL du 68000, par exemple. Paula offre au programmeur la possibilité d'autoriser ou d'interdire chacune des 14 sources d'interruption.

Le transfert de données à partir de la disquette et la sortie son se déroulent également via les canaux DMA. On ne peut pas toujours prévoir par les données d'une disquette, le moment où le mot sera prêt pour un transfert DMA d'AGNUS ; la raison essentielle étant l'inévitable variation du nombre de tours du moteur du lecteur de disquette.

De même pour la sortie son, AGNUS ne peut prévoir quand les données seront nécessaires.

Afin de rendre possible un transfert DMA sans accroc, PAULA utilise le signal DMAL qui indique à AGNUS le moment où un accès DMA est nécessaire.

Les communications séries sont prises en charge par un boîtier *UART* au sein de PAULA. *UART* signifie *Universal Asynchronous Receive Transmit* ou plus simplement : *Emetteur/Recepteur Asynchrone Universel.*

Les fonctions de l'UART seront décrites, comme les 4 canaux audio et le port analogique, dans le chapitre **Programmation des circuits intégrés.**

Description des broches

Bus de données : D0-15

Comme sur les autres circuits, ces broches sont reliées au circuit bus de données.

Register Adress Bus : RGA 1-8

Idem DENISE

Le signal d'horloge et Reset : CCK, CCKQ et RES

Paula renferme les mêmes signaux d'horloge qu'AGNUS. Le signal RES remet le circuit en état d'allumage.

DMA Request : DMAL

Paula signale à AGNUS, au moyen de ce signal, qu'il nécessite un transfert DMA.

Sorties Audio : AUDL et AUDR

Les sorties *AUDL* (LEFT AUDIO) et *AUDR* (RIGHT AUDIO) sont analogues et produisent les signaux sons, le signal *AUDL* correspond aux canaux audio internes 0 et 3 ; le signal *AUDR* correspond aux canaux 1 et 2.

Les signaux du connecteur série : TXD et RXD

RXD (RECEIVE DATA/Donnée série reçue) est l'entrée série de l'UART, *TXD* (TRANSMIT DATA/Donnée série envoyée) étant la sortie série. Ces signaux ont un niveau *TTL*, ce qui signifie, que leur tension d'entrée/sortie varie de 0 à 5 volts.

De plus un transformateur de niveau engendre des tensions de +12/-5 volts, nécessaires au connecteur série RS232 de l'AMIGA.

Les entrées analogiques : POT0X, POT0Y, POT1X, POT1Y

Les entrées *POT0X* et *POT0Y* sont reliées aux signaux correspondant du connecteur souris 0, *POT1X* et *POT1Y* étant reliées au connecteur 1. A ces entrées peuvent être raccordés, soit des manettes (*Paddles*), soit des joysticks. Ces appareils de commande contiennent des résistances variables, dénommées potentiomètres et branchées entre le +5V et l'entrée *POT*. PAULA peut mesurer la valeur de cette résistance et la mettre dans des registres internes. Les entrées *POT* peuvent être commutées en mode sortie par le biais du Software.

Les signaux d'accès disquette : DKRD, DRWD, DKWE

PAULA obtient la lecture des données d'une disquette par le signal *DKRD* (DISK READ/lecture données disquette). Le signal *DKWD* (DISK WRITE) permet l'écriture des données sur disquette. Le signal *DKWE*, quant à lui, sert de commutateur au lecteur de disquette pour passer du mode lecture en mode écriture (DISK WRITE ENABLE/autorisation d'écriture disque).

Les signaux d'interruption INT2, INT3, INT6, IPL0, IPL1, IPL2

Les trois signaux *IPL* permettent la création d'une interruption d'un niveau correspondant.

Le signal *INT2* est relié au boîtier 8520 (CIA-A). On retrouve ce signal sur le port d'expansion et sur le connecteur série. Lorsqu'il est en position *LOW*, PAULA engendre une interruption du niveau 2, à condition qu'une interruption de ce niveau soit autorisée. Le signal *INT3* est relié à la sortie correspondante d'AGNUS. Le signal *INT6* est relié au CIA-B et au port d'expansion. Toutes les autres interruptions se déclenchent suivant les composantes I/O de PAULA.

Les signaux *IPL0-IPL2* (ligne d'interruption du 68000) sont reliés directement aux signaux correspondants du processeur. PAULA engendre sur ces signaux, une interruption dont le niveau dépend du processeur.

1.2.3.4 Particularités de l'A500

La description du HARDWARE, dans ce chapitre concerne essentiellement l'AMIGA 1000. Elle est en grande partie valable pour l'A500. Son architecture principale n'est pas modifiée, les concepteurs n'ayant cherché qu'à présenter une version moins chère. Les plus grandes différences entre les deux modèles se trouvent dans la séparation des différents éléments du HARDWARE sur le même circuit.

On remarque, sur les circuits intégrés du modèle 1000, la présence d'un interrupteur logique de circuit, qui permet la création du signal d'horloge, la gestion des bus et le décodage d'adresse.

Ces fonctions logiques sont toutes réunies au sein d'un même circuit. On a ainsi rajouté au circuit AGNUS, une nouvelle partie du HARDWARE. Ce nouveau circuit portant le nom de *FAT-AGNUS* (modèle 8370).

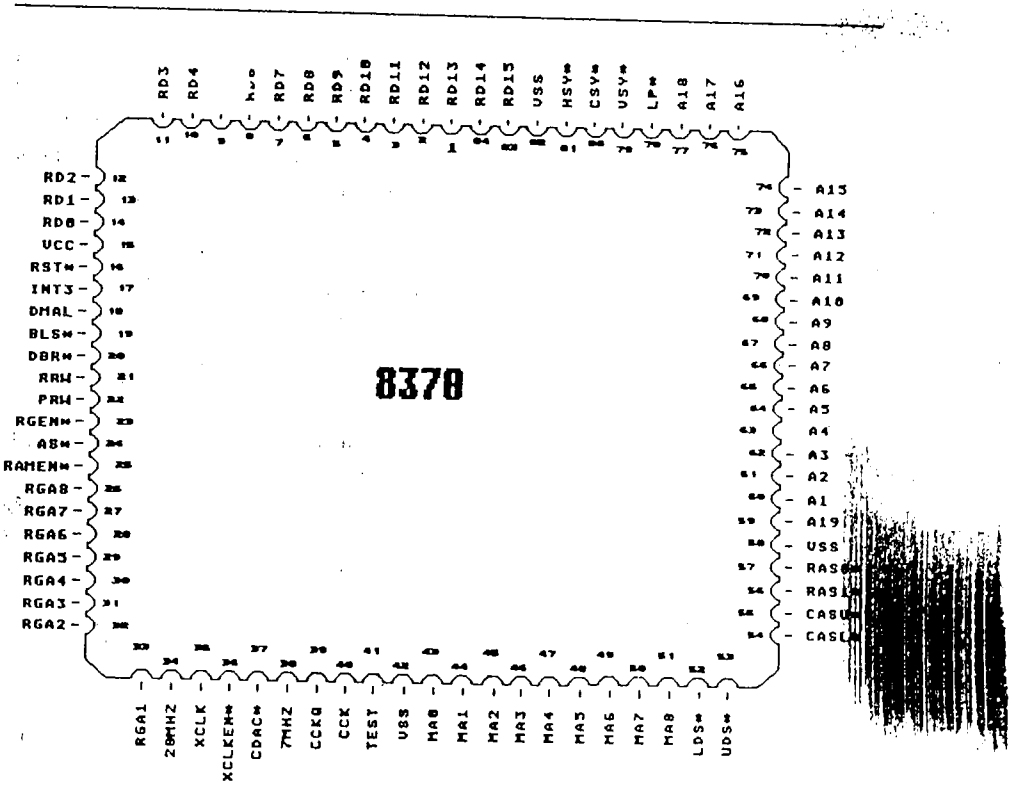


Figure 1.2.3.7

Le générateur d'horloge

Le générateur d'horloge du système de l'AMIGA est intégré dans AGNUS, seul le signal haute impulsion 28 MHERTZ doit être raccordé. Les signaux de cette fonction sont :

28MHERTZ, XCLK, XCLKEN,7MHERTZ, CCKQ,CCK,CDAC.

Le tampon bus d'adresse

Sur le schéma de l'architecture de l'AMIGA, on remarque un tampon reliant le bus d'adresse de l'AMIGA au bus d'adresse CHIP-RAM et au bus d'adresse registre.

De plus il multiplexe les adresses processeur correspondantes. Ce tampon est complètement intégré dans AGNUS le bus d'adresse processeur peut être raccordé directement aux signaux A1-18 de FAT-AGNUS. Le décodeur d'adresses signale que le processeur veut avoir accès à la RAM ou aux registres mémoire, au moyen des deux signaux *RAMEN* (RAM ENABLE) et *RGEN* (REGISTER ENABLE).

Toutefois, AGNUS est maintenant relié aux signaux UDS, LDS et PR/W du processeur.

La gestion de la Chip-Ram

La gestion de la chip-ram est complètement assurée par AGNUS. Celui-ci engendre les signaux nécessaires *RAS* et *CAS* avec les adresses RAM multiplexées. Ainsi AGNUS est capable de gérer 512 Koctets RAM de plus, soit un total de 1 mega. Les deux banques mémoires seront gérées au moyen des signaux de gestion de la RAM :

- *RAS0* et *CAS0* pour la chip-ram
- *RAS1* et *CAS1* pour l'extension mémoire.

Toutes les autres fonctions d'AGNUS décrites au début du chapitre ne sont pas modifiées.

En plus de FAT-AGNUS, un quatrième circuit spécialisé du nom de GARY a été réalisé. Il comprend les fonctions de décodeur d'adresses et de contrôleur de bus. Il crée les signaux de sélection des circuits spécialisés, tel *UPA* et *DTACK* du processeur. *GARY* renferme, enfin, la logique *RESET* et le *FLIP-FLOP* moteur pour le lecteur de disquette.

1.3 Les connecteurs de l'AMIGA

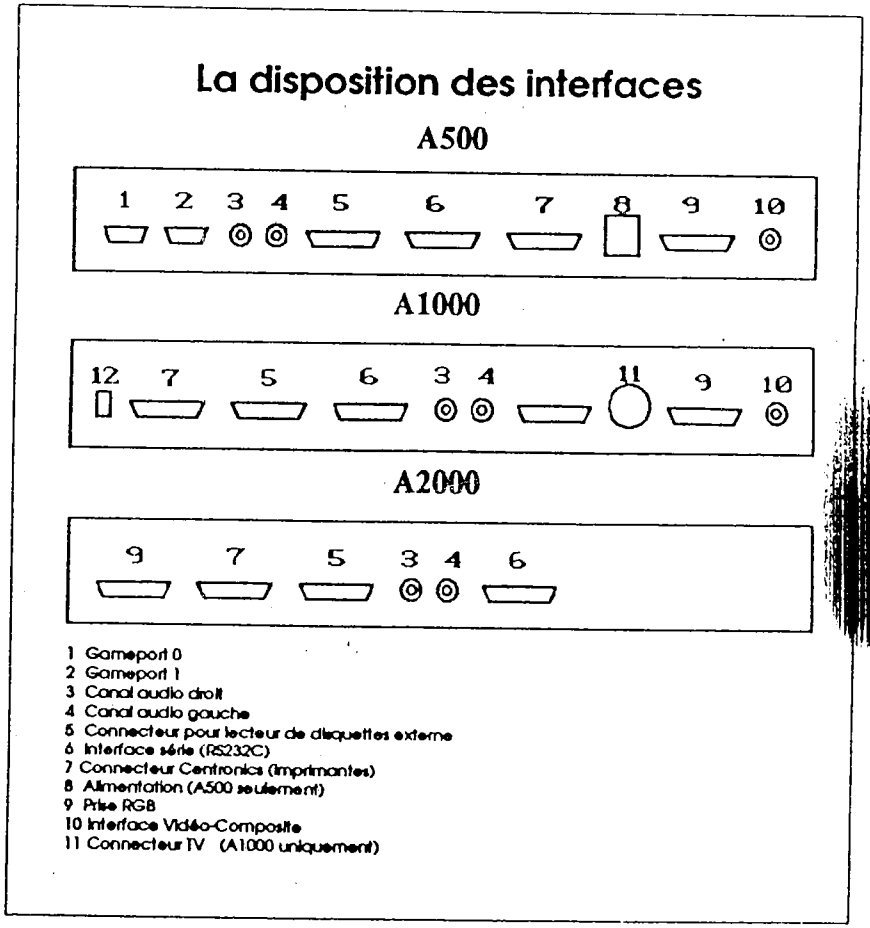


Figure 1.3

1.3.1 CONNECTEUR AUDIO/ VIDEO

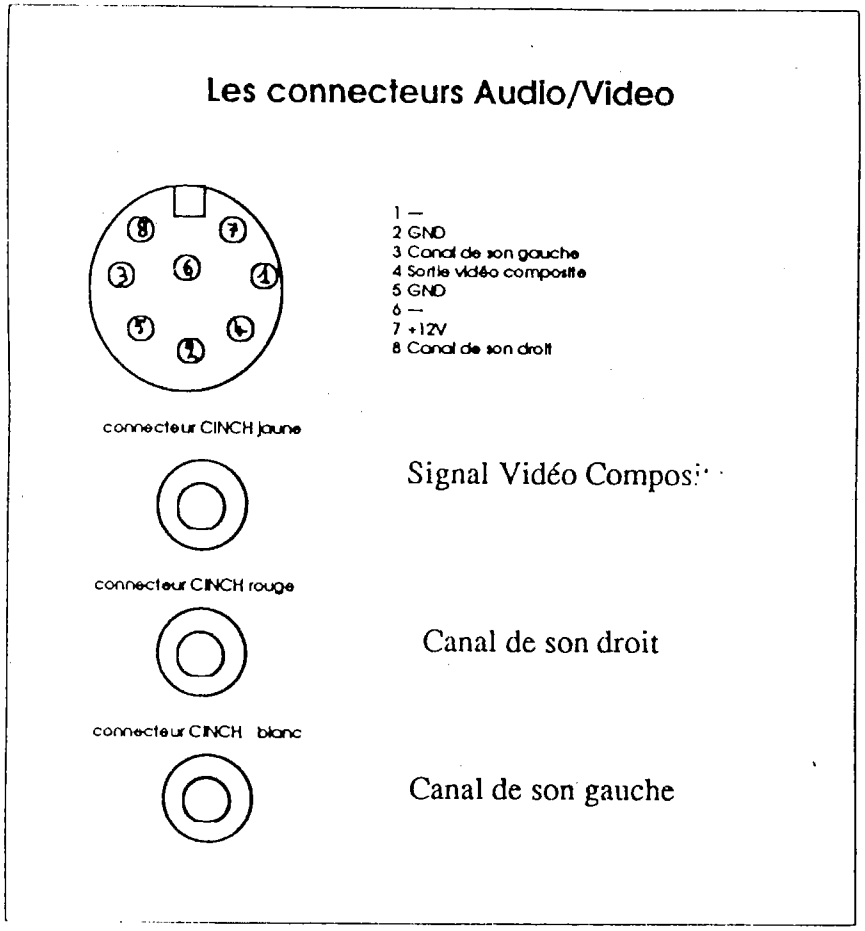


Figure 1.3.1

La description du connecteur vidéo diffère selon le type de l'Amiga. Le plus défavorisé l'Amiga A2000 qui ne dispose d'aucune sortie vidéo, juste un connecteur permettant l'adaptation d'un modulateur vidéo ou d'une interface *GENLOCK*. Seuls les AMIGA A500, A1000 et B2000 disposent d'une sortie vidéo sous la forme d'un connecteur *CINCH*. Le signal vidéo, sur le connecteur de l'A1000, est de type FBAS, permettant le branchement avec la plupart des moniteurs du marché. Sur l'A500, par souci d'économie, il ne livre qu'un signal *BAS* (signal vidéo noir et blanc). Un autre problème est que l'ancien modèle A1000 n'a pas été fourni avec un clavier français. Sur ces modèles, la sortie vidéo livre un signal *NTSC* (signal vidéo suivant la norme américaine).

Sur moniteur couleur à la norme *PAL* (comme le moniteur de l'AMIGA), l'image apparaît en noir et blanc avec des raies perpendiculaires. Une image couleur nette, à la norme PAL, ne peut être obtenue que sur les nouveaux modèles A1000, disponibles avec un clavier français.

Sur tous les modèles, le signal vidéo est véhiculé sur un tampon transistorisé avec une résistance de sortie de 75 ohms. Il est ainsi protégé contre les court-circuits de façon permanente.

Le signal audio est transmit par 2 connecteurs *CINCH* sur tous les modèles de l'AMIGA. Le canal stéréo droit correspond au connecteur *CINCH* rouge, celui de gauche au *CINCH* blanc. A l'aide de cables stéréo *CINCH*, on peut relier ces connecteurs à un amplificateur (entrée *AUX, TAPE* ou *CD*) stéréo. La résistance de sortie s'élève à 1Kohm (1000 ohm). Les sorties sont protégées contre les court-circuits et possèdent de façon interne une résistance de charge de 36 ohms.

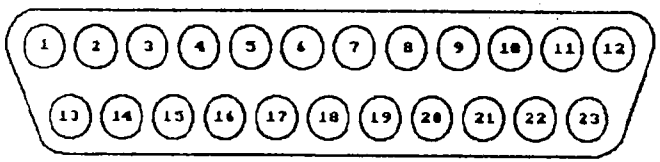
L'AMIGA 1000 dispose d'un autre connecteur audio-vidéo. Ce connecteur modulateur TV a été prévu pour le raccord avec un modulateur HF, permettant l'utilisation d'une télévision comme moniteur. Ce modulateur HF n'est plus compris dans la panoplie des connecteurs. Il véhicule aussi bien le signal vidéo que les signaux audio. On y trouve aussi une prise 12 volts, qui a été prévue pour l'alimentation du modulateur. La sortie vidéo dispose d'un tampon transistorisé, celui-ci n'étant donc pas seulement relié avec le connecteur *CINCH* vidéo. Les deux broches audio ont une résistance de sortie de 1 Kohm.

Etant donné qu'il n'a pas été prévu de résistance interne de charge, leurs broches se trouvent dans un état non c' gé, quatre fois aussi important que celui des connecteurs audio-CINCH.

Le connecteur modulateur *TV* est de type *DIN* à 8 broches. Les fiches convenant à ce type de connecteur sont difficiles à obtenir. Toutefois, il est bon de savoir que le connecteur modulateur TV est identique à celui du C64.

1.3.2 CONNECTEUR RGB

Le brochage de l'interface RGB



23 connecteurs D-SUB

Entrée	1	XCLK	Entrée pour un signal d'horloge externe
Entrée	2	XCLKEN	Agrément d'un signal d'horloge externe
Sortie	3	R	Signal Rouge analogique
Sortie	4	G	Signal Vert analogique
Sortie	5	B	Signal Bleu analogique
Sortie	6	DI	Signal digital lumineux
Sortie	7	DB	Signal Bleu digital
Sortie	8	DG	Signal Vert digital
Sortie	9	DR	Signal Rouge digital
Sortie	10	CCSY	Signal de synchronisation composite bufférisé
E/S	11	HSY	Signal de synchronisation horizontale
E/S	12	VSY	Signal de synchronisation verticale
	13	GND	
Sortie	14	ZD	Signal indicateur d'arrière-plan (background)
Sortie	15	CTU	Horloge C1U de chez l'Amiga (3.58 MHz)
	16	GND	
	17	GND	
	18	GND	
	19	GND	
	20	GND	
	21	-5 Volts	
	22	+12 Volts	
	23	+5 Volts	

Figure 1.3.2

Le connecteur RGB est identique sur les trois modèles d'AMIGA. Il rend possible le br hement de différents moniteurs RGB, mais aussi d'extensions spéciales, telles que l'adapteur *Genlock*. Lors d'un raccord avec un moniteur *RGB* analogique, tel le moniteur de l'AMIGA, les trois sorties *RGB* analogiques et la sortie composite *Sync.* sont utilisées. La transformation des signaux *RGB* digitaux issus de DENISE en signaux analogiques correspondants se fait au moyen d'un transformateur digital/analogique à 4 bits. Le signal de synchronisation composite provenant d'AGNUS est affiché après le mixage des signaux de synchronisation verticale et horizontale. Ces quatre fiches sont pourvues d'une résistance de sortie de 75 ohms et d'un tampon transistorisé, assurant une protection contre les court-circuits.

Les signaux DI, DB, DG et DR sont prévus pour les connexions des moniteurs *RGB* digitaux. La source des signaux *RGB* digitaux provient de la sortie digitale *RGB* de DENISE.

Les trois signaux de couleurs sont reliés avec ceux de DENISE (DB relié avec B3 par exemple). Toutefois on remarque un tampon de type 74HC 244 entre DENISE et les sorties. Les quatre signaux ont une résistance de sortie de 47 ohm et comme ils proviennent du tampon 74 HC 244, un niveau *TTL*.

On remarque la présence des broches *HSY* et *VSY*, celles-ci peuvent être nécessaires à certains moniteurs. Il faut cependant être attentif au fait que ces signaux ont une resistance de 47 ohms et sont directement reliés aux broches *HSY* et *VSY* d'AGNUS.

Si le bit *Genlock* d'AGNUS est activé, ces deux signaux fonctionnent en tant qu'entrée. L'AMIGA synchronise alors son signal vidéo d'après les signaux de synchronisation sur *HSY* et *VSY*. Aussi bien en mode entrée que sortie, ces signaux s'alignent au niveau *TTL*. Les signaux de synchronisation sont à usage *Low-active*, c'est-à-dire qu'à l'état normal, les signaux sont à 5 volts. Les signaux sont seulement à 0 volts lorsque les impulsions de synchronisation sont actives.

Lorsque l'adaptateur *Genlock* est branché, le signal *ZD* est activé (zéro detect). L'AMIGA met ce signal sur *LOW*, lorsque le point affiché reproduit un point de l'arrière plan de l'écran, autrement dit, lorsque cette couleur est contenue dans le registre de couleur 0.

Pendant le temps mort vertical (VSY=0), la fonction du signal ZD se modifie. On y trouve la valeur *GAUD-bits* (Genlock audio Enable) provenant du registre $100 d'AGNUS (BPLON0). Ce signal est utilisé comme interrupteur du signal son par l'interface *Genlock*.

Pour l'utilisateur normal, le signal ZD n'est pas intéressant puisqu'il n'est nécessaire qu'avec l'interface *Genlock*.

Les autres signaux du connecteur RGB n'ont plus aucun rapport avec le signal RGB.

Le signal C1U est un signal d'horloge à 3,58 MHZ et correspond au signal *CLK* des circuits spécialisés. Les signaux *XCLK* (External Clock) et *XCLKEN* (External Clock Enable) permettent d'alimenter l'AMIGA avec une fréquence d'horloge externe. Tous les signaux d'horloge de l'AMIGA dérivent d'une horloge de 28MHZ. Cette dernière peut être remplacée par une autre fréquence d'horloge par l'entrée *XCLV*, lorsque l'on met ce signal à 0. L'AMIGA peut donc être beaucoup plus rapide si on installe une horloge de 32MHZ à cette entrée, seul le HARDWARE de l'AMIGA peut répondre de la fiabilité d'une telle expérience. Lors de l'utilisation des signaux *XCLK* et *XCLKEN*, on utilise la broche de terre 13. Elle est directement reliée au signal de masse de l'horloge.

1.3.3 LE CONNECTEUR CENTRONICS

Le brochage du connecteur Centronics

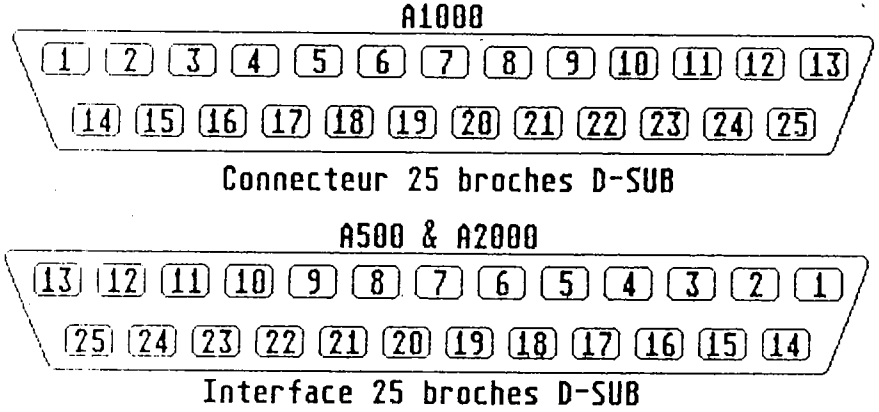


Figure 1.3.3

sortie		donnée valide
entrée/sortie	–	bit de donnée 0
entrée/sortie	3	bit de donnée 1
entrée/sortie	4	bit de donnée 2
entrée/sortie	5	bit de donnée 3
entrée/sortie	6	bit de donnée 4
entrée/sortie	7	bit de donnée 5
entrée/sortie	8	bit de donnée 6
entrée/sortie	9	bit de donnée 7
entrée	10	/acknowledge - acquittement
entrée/sortie	11	busy : imprimante occupée
entrée/sortie	12	paper out : papier manquant
entrée/sortie	13	on line : imprimante sélectionnée
	14	+ 5 volts
	15	inutilisé
sortie	16	reset
	17- 25	GND masse de référence

Chez l'AMIGA 1000, certaines broches sont disposées d'une autre façon :

14- 22	GND
23	+ 5 volts
24	inutilisé
25	reset

Le connecteur centronics de l'AMIGA est de nature à réjouir le coeur des utilisateurs. Il est compatible PC, les imprimantes à cette norme peuvent être directement raccordées. Ceci ne correspond en fait qu'aux modèles A500 et A2000, le port centronics de l'AMIGA 1000 n'obéissant pas à ce standard. En effet, à la place du connecteur *DSUB*, on trouve une prise mâle et la broche 23 est soumise à une tension de 5 volts. Ce signal serait relié à la masse (GND) si on branchait le câble d'une imprimante et entraînerait automatiquement un court-circuit.

Il est cependant possible de contourner ce problème en fabricant un câble approprié.

Les broches du port centronics sont tous reliés aux signaux de port CIA (et ce, jusqu'aux signaux RESET et + 5 volts).

Les attributions correctes sont les suivantes :

Centronics N° de broche	Fonction	CIA	Broche	Descriptif
1	Strobe	A	18	PC
2	bit de donnée 0	A	10	PB0
3	bit de donnée 1	A	11	PB1
4	bit de donnée 2	A	12	PB2
5	bit de donnée 3	A	13	PB3
6	bit de donnée 4	A	14	PB4
7	bit de donnée 5	A	15	PB5
8	bit de donnée 6	A	16	PB6
9	bit de donnée 7	A	17	PB7
10	Acknowledge	A	24	PB8
11	busy	B	2	PA0
			et 39	SP
12	paper out	B	3	PA1
			et 40	CNT
13	select	B	4	PA2

Le connecteur centronics est un connecteur parallèle. L'octet donnée se trouve sur les 8 signaux de donnée. Si un octet valide arrive sur le port, le signal *STROBE* est mis à 0 pendant 1,4 micro seconde, signalant ainsi à l'Imprimante qu'un octet valide est prêt à être transmis. L'imprimante signale alors qu'elle accepte les données en mettant le signal *Acknowledge* pendant 1 micro seconde à 0. Puis l'ordinateur remet le prochain octet sur le bus.

L'imprimante signale au moyen de *busy* qu'elle est occupée et qu'elle ne peut prendre en charge les données suivantes. Ceci peut arriver, lorsque, par exemple, le buffer de l'imprimante est plein. L'ordinateur attend alors, avant de reprendre le transfert, que le signal *busy* soit à nouveau actif. Avec le signal *paper out*, l'imprimante signale qu'elle n'est plus approvisionnée en papier. Le signal de sélection est activé à partir de l'imprimante. Il signale si l'imprimante est sélectionnée (on line/sel sur High) ou non (off line, sel sur low).

Le port centr[illegible]cs se prête parfaitement aux branchements d'extensions comme, par exemple, un digitaliseur de sons, les signaux se programmant facilement aussi bien en mode entrée qu'en mode sortie.

1.3.4 LE CONNECTEUR SERIE

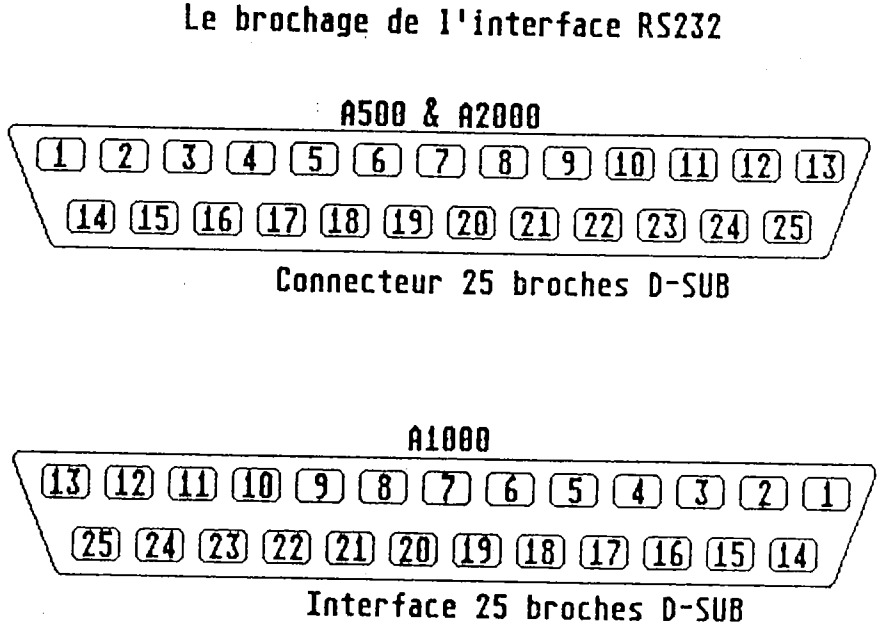


Figure 1.3.4

	1	GND	(FRAME GROUND) masse de protection
sortie	2	TXD	(TRANSMIT DATA) donnée transmise
entrée	3	RXD	(RECEIVE DATA) donnée reçue
sortie	4	RTS	(REQUEST TO SEND) demande d'émission
entrée	5	CTS	(CLEAR TO SEND) prêt à émettre
entrée	6	DSR	(DATA SET READY) donnée prête à envoyer
	7	GND	masse de référence
entrée	8	CD	(CARRIER DETECT) modem détecté
	9	+ 12 volts	
	10	- 12 volts	
sortie	11	AUDOUT	sortie audio

	12	inutilisé	
	13	inutilisé	
	14	inutilisé	
	15	inutilisé	
	16	inutilisé	
	17	inutilisé	
entrée	18	AUDIN	entrée audio
	19	inutilisé	
sortie	20	DTR	(DATA TERMINAL READY) terminal prêt à recevoir
entrée	21	RI	(RING INDICATOR)
	23	inutilisé	
	24	inutilisé	
	25	inutilisé	

Certains signaux sont disposés d'une autre façon chez l'AMIGA 1000 :

	9	inutilisé	
	10	inutilisé	
	11	inutilisé	
	12	inutilisé	
	13	inutilisé	
	14	- 5 volts	
sortie	15	AUDOUT	sortie audio
entrée	16	AUDIN	entrée audio
sortie	17	E	horloge tampon (716 KHZ)
entrée	18	/INT2	entrée d'interruption de niveau 2
	19	inutilisé	
sortie	20	DTR	(Data Terminal Ready) terminal prêt à recevoir
	21	+ 5 volts	
	22	inutilisé	
	23	+ 12 volts	
sortie	24	MCLK	horloge de transmission à 3,58 MHZ
sortie	25	/MERS	RESET

Le connecteur série possède tous les signaux RS232. De plus, sur ce connecteur, on trouve plusieurs signaux qui n'ont aucun rapport avec la communication série.

Seuls les signaux TXD, RXD, DSR, CTS, DTR, RTS et CD assurent la communication RS232. *TXD* et *RXD* sont les signaux de données séries, *TXD* étant la sortie série, *RXD* l'entrée série. Ils sont reliés aux signaux correspondant de PAULA. Le signal *DTR* indique aux périphériques raccordés, que le connecteur série de l'AMIGA est en activité. Le signal *DSR*, au contraire, signale aux périphériques de l'AMIGA, que le connecteur série est prêt à être activé. Le signal *RTS* indique aux périphériques que l'AMIGA veut envoyer des données séries sur le RS232 et qu'il est prêt à les transmettre (signal *CTS*). Le signal *CD* est seulement utilisé par un modem, celui-ci indique qu'il emploie une certaine fréquence de transfert. Les cinq signaux de gestion RS232 sont reliés avec le CIA- B(PA3- PA7) ; DSR-PA3 ; CTS-PA4 ; CD-PA5 ; RTS-PA6 ; DTR-PA7.

Le signal *RI* est relié au signal *SEL* du connecteur centronics, par l'intermédiaire d'un transistor.

Les signaux RS232 ne sont pas reliés directement aux circuits spécialisés, mais dirigés sur un pilote RS232. On pourra ainsi relier ce connecteur, par l'intermédiaire d'un câble approprié à des terminaux et modems. Le transformateur de niveau RS232 de type 1488 sera utilisé comme conducteur de sortie, avec une tension de +12 à -5 volts. Les circuits de type 1489 A seront utilisés comme tampon d'entrée, les entrées acceptant alors des tensions entre -12 et +0,5 volts (*état Low*) et des tensions de 3 à 25 volts (*état High*).

Les conventions retenues pour le connecteur RS232, sont que les signaux de gestion soient actifs à l'état *high*, alors qu'au contraire les signaux *RXD* et *TXD* sont mis à 1 suivant un niveau négatif. Etant donné que le conducteur de sortie est interverti, les bits de port correspondants du CIA-B seront *low-active*, c'est-à-dire qu'un bit du CIA-B d'une valeur 0 met le signal de gestion RS232 correspondant sur *high*. Cela se produit aussi en mode entrée.

Le signal *audout* est relié au canal audio gauche et pourvu d'une résistance de sortie de 1Kohm. Le signal *audin* possède une résistance de 47 ohm et est directement relié au signal *audr* de PAULA.

Amiga 3.5" Drive Interface

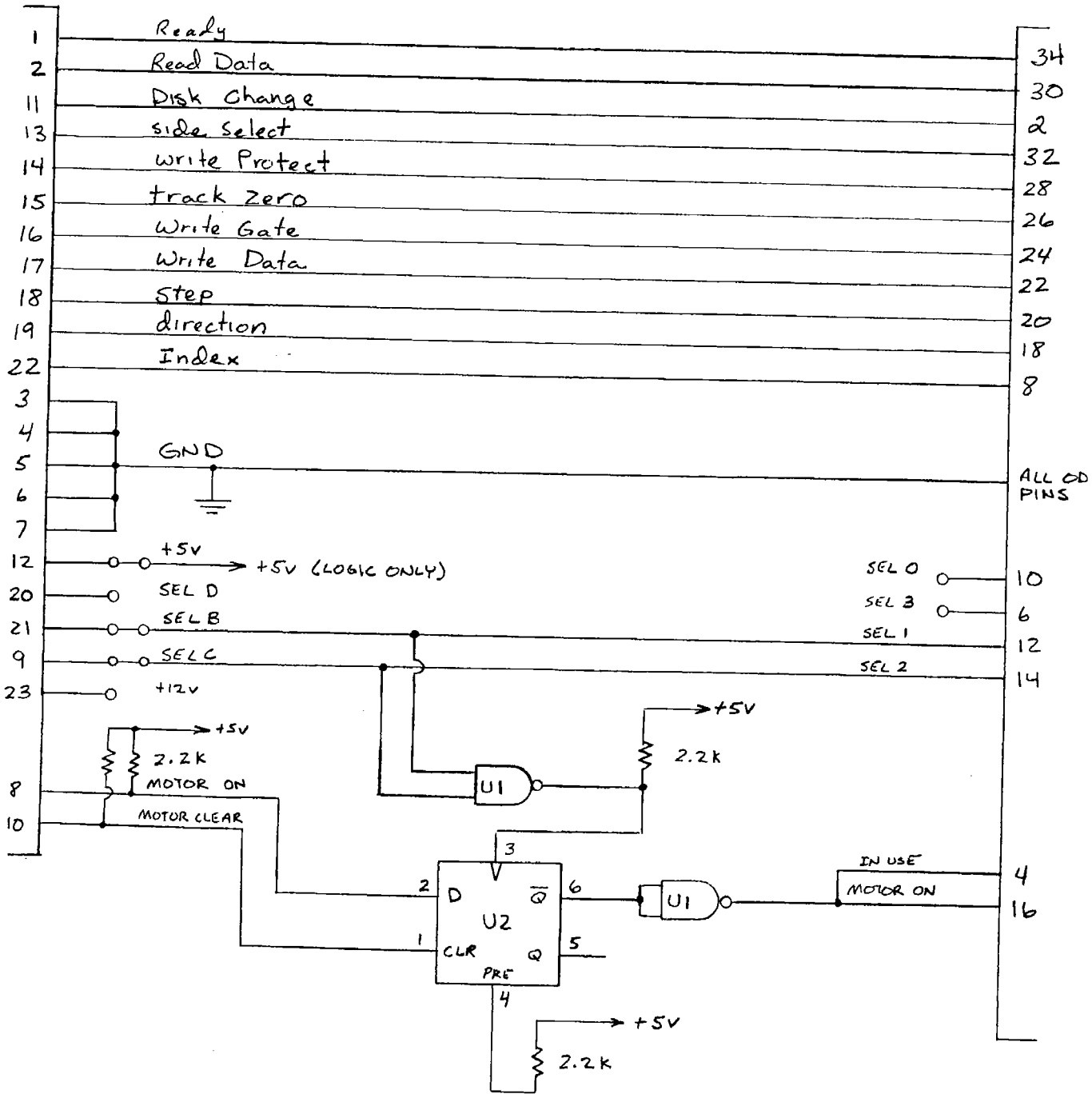
Jeff Easton 10-2-86

← Amiga Disk Port

23 PIN D connector

SONY
3.5" Drive

34 conduct
Header



U1 74LS38
U2 74LS74

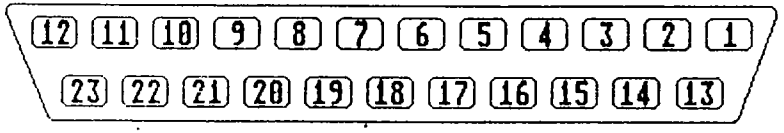
Le signal *audio*, qui passe dans l'AMIGA via la broche *audin*, accède, par le filtre *passe bas* et le canal *audio* droit de PAULA, à la sortie *audio* droite.

Le signal INT2 est directement relié à l'entrée INT2 de PAULA et peut libérer une interruption du processeur de niveau 2, lorsque le bit masque correspondant est activé. Le signal E est relié à l'horloge E du processeur via un tampon.

Une fréquence de 3,58 MHERTZ est appliqué au signal MCLK, ce signal d'horloge étant identique et en phase avec l'horloge du connecteur RGB et les deux fréquences des circuits spécialisés.

1.3.5 CONNECTEUR DISQUETTE EXTERNE

Le brochage de l'interface des lecteurs de disquettes externes.



Interface 23 broches D-SUB

Figure 1.3.5.1

Entrée	1	/RDY	disque prêt
entrée	2	/DKRD	lecture données disquette
	3	GND	
	4	GND	
	5	GND	
	6	GND	
	7	GND	
sortie	8	/MTRX	moteur on/off
sortie	9	/SEL	sélection disque off2
sortie	10	/DRES	floppyreset (arrêt moteur)
entrée	11	/CHNG	disque changé
	12	+5 volts	
sortie	13	/SIDE	sélection de face

entrée	14	/WPRO	protection en écriture
entrée	15	/TK0	piste 0
sortie	16	/DKWE	autorisation écriture
sortie	17	/DKWD	écriture donnée
sortie	18	/STEP	déplacement de la tête
sortie	19	/DIR	direction déplacement de la tête
sortie	20	/SEL3	sélection disque dF3
sortie	21	/SEL1	sélection disque dF1
entrée	22	/INDEX	Index début de cylindre
	23	+ 12 volts	

Le brochage du connecteur interne Floppy

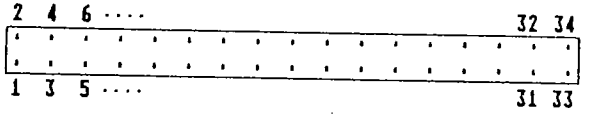


Figure 1.3.5.2

Toutes les broches impaires sont de type GND.

	2	/CHNG		4	/INUSE (RLI2 0 MTR0D)
	6	inutilisé	27	8	INDEX
23	10	SEL0		12	inutilisé
	14	inutilisé	19	16	/MTR0
17	18	DIR	15	20	/STEP
13	22	/DKWD	11	24	/DKWE
9	26	/TK0	7	28	/WPRD
5	30	/DKRD	3	32	/SIDE
1	34	/RDY			

Boîtier d'alimentation pour lecteur de disquette interne :

1	+ 5 volts
2	GND
3	GND
4	+ 12 volts

Le branchement du lecteur de disquettes de l'AMIGA est compatible avec le bus SHUGART. Il rend possible la connexion de 4 lecteurs de disquettes de ce type. La sélection du lecteur se fait au moyen des signaux *SEL x,x* désignant le numéro de l'unité. Etant donné que l'AMIGA possède déjà son propre lecteur, il n'autorise que le branchement de 3 unités externes sélectionnées grâce à SEL1, SEL2 et SEL3.

Le signal SEL0 est relié au connecteur interne du lecteur de disquette.

Voici les différentes fonctions des signaux du bus SHUGART de l'AMIGA :

SEL X

Avec ce signal on sélectionne une des quatre unités de disquette possibles. Lorsque ce signal est activé, tous les autres le sont aussi, à part MTRX et DRES.

MTRX

Ce signal active les moteurs de toutes les unités raccordées. La gestion de 4 unités n'est pas simple. C'est pourquoi on a prévu, pour chaque lecteur de l'AMIGA, un *Flip-Flop* (un Flip-Flop est un montage électronique, dont le rôle est de mettre en mémoire un bit de donnée). Lorsque le signal SEL du lecteur concerné se met à Low, le Flip-Flop de ce lecteur prend en charge la valeur du signal MTRX. La sortie du montage Flip-Flop est reliée au signal MTR du lecteur. Ainsi les lecteurs de disquettes peuvent être mis en marche ou arrêtés, indépendamment les uns des autres. Si on met, par exemple, le signal SEL0 sur Low pendant que le signal MTRX se trouve à 0, le moteur du lecteur interne s'enclenchera. Ce Flip-Flop se trouve sur le montage du lecteur interne. Pour chaque lecteur externe, il est nécessaire d'un rajouter un. Sur le deuxième lecteur on l'a installé sur un petit adaptateur.

RDY

Lorsque le signal MTR du lecteur correspondant est à 0, le signal RDY indique à l'AMIGA, que le moteur du lecteur a atteint son nombre de tour nominal, et que l'unité est prête pour les accès lecture comme pour les accès écriture. Si le signal MTR est à 1, le moteur du lecteur étant arrêté, il y a un mode d'identification spécial.

DRES

Le signal DRES (DRIVE RESET) est relié au signal RESET de l'AMIGA et remet dans sa position d'origine le Flip-Flop moteur, c'est-à-dire que tous les moteurs seront arrêtés.

DKRD

Les données disquette du lecteur sélectionné (SELX) accèdent à la broche DKRD (DISK READ DATA). Cette dernière est reliée à la broche DKRD de PAULA.

DKWD (DISK WRITE DATA)

Les données de PAULA sont transmises au lecteur de disquette, afin d'y être écrites par le biais du signal DKWD.

DKWE

Le signal DKWE (Disk Write Enable) permet de passer du mode lecture au mode écriture. Si le signal est High, les données seront lues sur la disquette, alors que si le signal est Low, les données seront écrites.

SIDE

Le signal SIDE permet la sélection de la face de disquette qui sera lue ou écrite.

S'il est high, ce sera la face 0, c'est-à-dire la tête de lecture sous le disque qui sera activée. A l'inverse, si le signal est low, c'est la face supérieure qui sera sélectionnée.

WPRO

Le signal WPRO (Write protect) indique à l'AMIGA si la disquette insérée est protégée (WPRO=0) ou non en écriture.

STEP

Un état positif du signal STEP (celui-ci variant de l'état high vers low), déplace la tête de lecture/écriture du lecteur d'une piste à l'autre, suivant l'état du signal DIR. Avant que le signal SELX du lecteur activé ne soit remis sur high, le signal STEP doit dans tous les cas être mis à 1, sinon il risque d'y avoir des problèmes, dans la reconnaissance de la disquette, lors d'un changement.

DIR

Le signal DIR (Direction) détermine la direction de la tête, qui sera déplacée après une impulsion du signal STEP. Low signifie qu'il y aura déplacement vers le centre de la disquette, alors que high engendre un déplacement vers le bord de la disquette, la piste 0 étant la première piste d'une disquette.

TK0

Le signal TK0 (TRACK 0) est sur Low lorsque la tête de lecture/écriture se trouve sur la piste 0. Il y a alors la possibilité de placer la tête à un endroit déterminé.

INDEX

Le signal INDEX est une impulsion courte qui est fournie par le lecteur à chaque début de cylindre (chaque "tour" de disquette) et toujours entre le début et la fin d'une piste.

CHWG

Le lecteur de disquette indique, par le biais de ce signal, un changement de disquette. Aussitôt que la disquette sera sortie du lecteur, le signal CHWG sera mis à 0. Ce dernier restera en cet état tant que l'ordinateur n'aura pas libéré une impulsion STEP. Si à ce moment on introduit une disquette, le signal CHWG se remettra à 1. Sinon il restera sur 0 et l'ordinateur sera obligé de libérer des impulsions périodiquement afin de savoir si une disquette se trouve ou non dans le lecteur. Ces impulsions périodiques sont à l'origine du bruit désagréable que l'on peut entendre lorsque l'unité est vide.

INUSE

Ce signal n'existe que pour le lecteur interne. Lorsqu'on le met à 0, le lecteur allume une diode lumineuse. En temps normal, ce signal est relié au signal MTR.

Afin de savoir si le lecteur est connecté au bus, il existe un mode spécial d'identification. Le lecteur lira un long mot de données série de 32 bits.

Afin de démarrer cette identification, le signal MTR du lecteur concerné devra être rapidement activé, puis désactivé. De cette façon, le registre à décalage du lecteur sera remis dans son état de base. On pourra, après cela lire chaque bit de donnée, tout en mettant le signal SELX sur Low, la valeur du signal RDY étant lue comme bit de données, puis remettre le signal SELX sur high. On répètera ce processus 32 fois. Le premier bit sera le MSB (*Most Signifiant Bit*/bit de poids fort) du mot de donnée. Etant donné que le signal est Low-Active, les bits de donnée doivent être intervertis.

Voici les définitions établies pour le lecteur de disquettes externe :

$0000 0000	pas de lecteur raccordé (00)
$FFFF FFFF	lecteur standard format 3 pouces 1/2 (11)
$5555 5555	lecteur format 5 pouces 1/4, 2*40 pistes (01)

Comme on peut le remarquer, il n'y a que peu d'identifications différentes et il suffit de lire les deux premiers bits. Les valeurs entre parenthèses donnent la combinaison des deux bits.

Comme cela a été mentionné plus haut, tous les signaux sauf DRES n'auraient qu'une influence sur le lecteur sélectionné par SELX. Initialement le signal MTRX oeuvrait indépendamment du signal SELX, mais les concepteurs de l'AMIGA l'ont modifié en rajoutant le Flip-Flop moteur.

Tous les signaux du bus SHUGART sont Low-actives, étant donné que les sorties de l'AMIGA, comme le lecteur de disquette, sont pourvues de conducteurs OPEN-COLLECTOR de type 7407.

Les quatre entrées CHWG, WPRO, TK0 et RDY sont reliées directement aux signaux PA4-PA7 du CIA-A. Les huit sorties STEP, DIR, SIDE, SEL0, SEL1, SEL2, SEL3, MTR proviennent du CIA-B, PB0-7 et sont reliées aux connecteurs disquette interne et externe sur le conducteur mentionné plus haut. Etant donné que ce dernier n'intervertit rien, les bits du CIA sont toujours intervertis. Les signaux DKRD, DKWD et DKWE proviennent de PAULA. Les branchements entre le lecteur interne et le lecteur externe sont identiques, la seule différence concerne les signaux MTRX et SELX.

Le lecteur interne est relié a SEL0. Son signal MTR passe par le Flip-Flop moteur.

Sur le connecteur externe on trouve le signal MTRX directement relié au CIA et aux 3 signaux SEL.

Branchement lecteur externe de l'AMIGA

L'utilisation d'un seul lecteur est possible pendant un bon moment. Mais il vient un temps où une deuxième unité devient indispensable.

Le lecteur A10[illegible] double face, au format 3 ½ pouces, utilisé par l'AMIGA, nous est propo[illegible]é à un tel prix, qu'il est préférable, actuellement, d'en "bidouiller" un soi-même. Que faut-il faire ?

La fiche de branchement d'un lecteur standard 3 ½ pouces, comme par exemple le NEC FD 1035 ou FD 1036, est identique à celle du lecteur interne à 34 broches, y compris les fiches d'alimentation. Afin de raccorder un lecteur du type FD1035, il suffit de rajouter un Flip-Flop moteur. La figure suivante reproduit le schéma de montage.

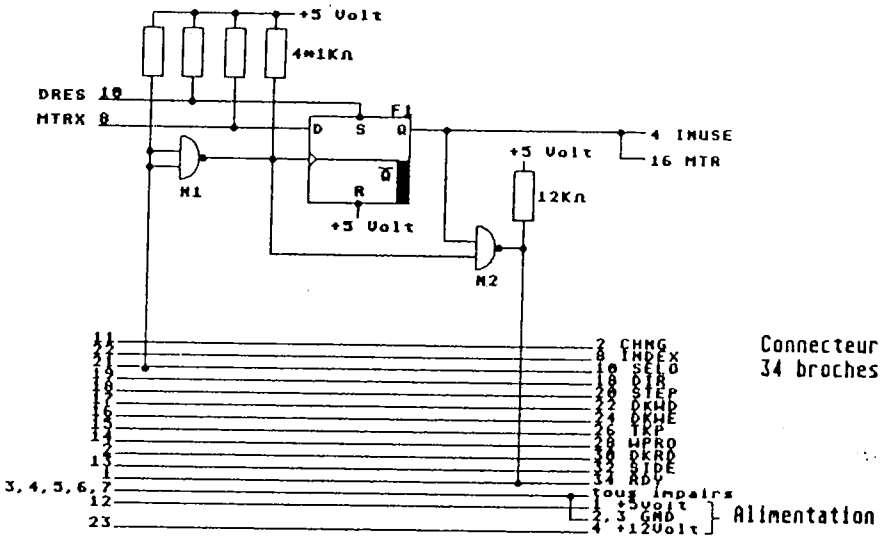
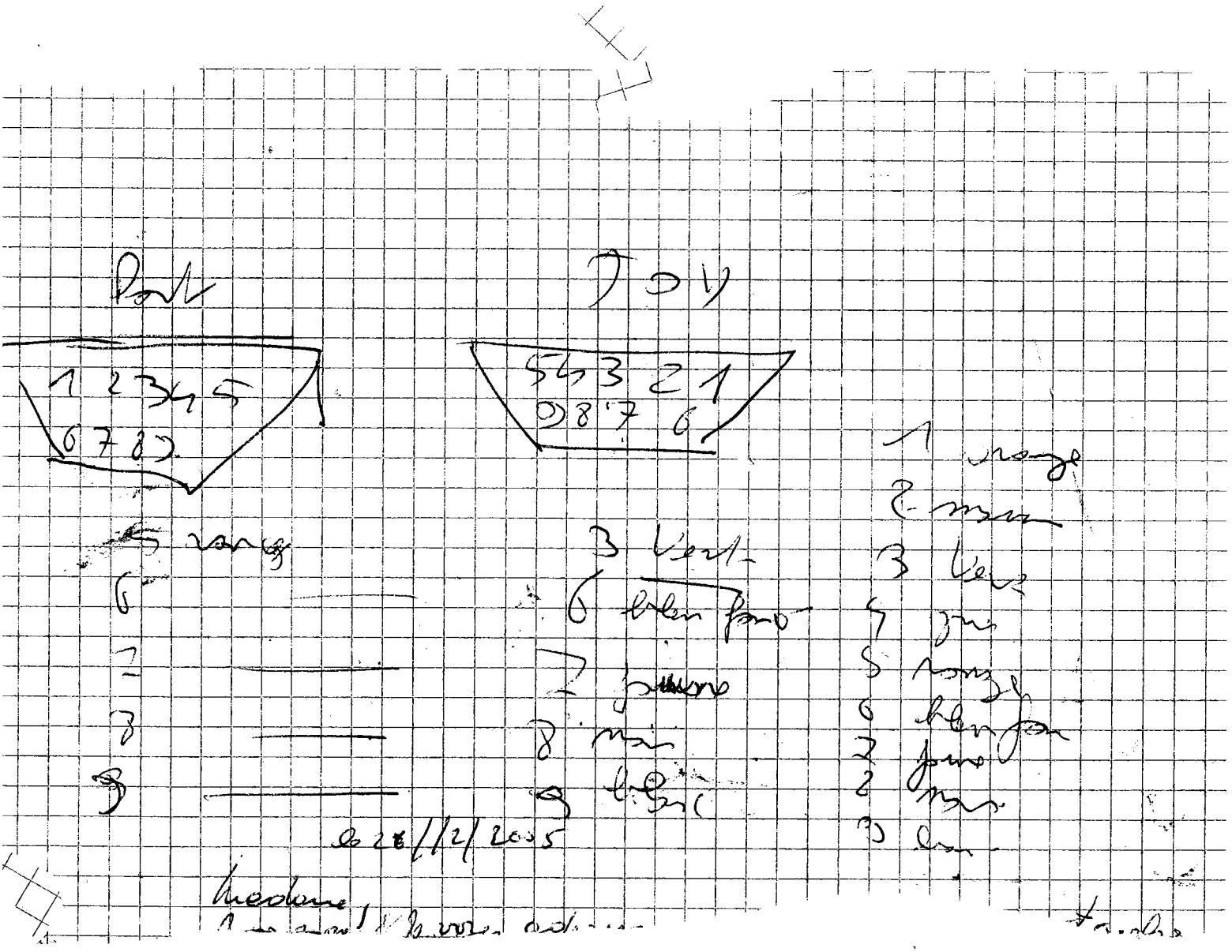
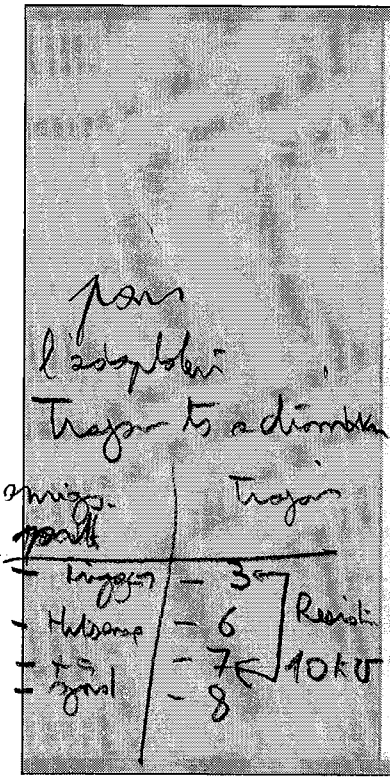


Figure 1.3.5.3

On remarque que le Flip-Flop caractéristique met en mémoire le signal sur MTRX au moyen de F1, lorsque le signal passe de High à Low. Etant donné que le Flip-Flop met en mémoire la valeur à son entrée donnée qu'avec le niveau positif de l'impulsion d'horloge, on doit inter[illegible] SEL1. C'est le rôle de la porte NAND N1. La sortie Q est reliée directement à l'entrée MTR du deuxième lecteur de disquette.





A pullup resistor must be used to interface the sega master system phaser to the amiga. The conversion is as follows:

To Amiga Joystick port 2:_____To Sega 3050:
_____9 pin female_____9 pin male

_____Trigger___5 ----------6----
_____HitSense__6 ----------7___ || 10k pullup res.
_____+5_____7 ----------5----__from pin 5 to 6
_____Gnd_____8 ----------8_____mounted in connector.

(The _ characters were used instead of spacebar characters because this site automatically removes extra spacebar characters and the formating is important.)
There is a 10k pullup resistor from +5 to the trigger (between pins 7 and 5 on the amiga joystick port). I found this on Usenet, but I don't know who designed it. It works very well for converting the sega 3050 to an actionware phaser.

Le deuxième port NAND n'a aucun rapport avec la gestion du moteur. Il sert au mode d'identification, la majorité des :teurs du commerce n'ayant besoin d'aucune aide. Lorsque le moteur est coupé, le signal SEL est actif, c'est-à-dire à 0, et le port met le signal RDY sur Low. L'AMIGA reconnaîtra ainsi la deuxième unité avec le terme DF1.

Etant donné que seul la moitié des deux circuits est nécessaire, cela suffit pour un lecteur supplémentaire. Les entrées de N1 doivent être reliées à SEL2 (broche 9 du lecteur externe).

Afin que le signal CHWG puisse fonctionner sans problème, certains lecteurs doivent être munis d'un JUMPER. Par exemple le lecteur FD 1035 de NEC doit être court-circuité par un JUMPER J1 caractéristique. Pour en savoir davantage, référez-vous au manuel du lecteur de disquette.

1.3.6 LE CONNECTEUR SOURIS-JOYSTICK

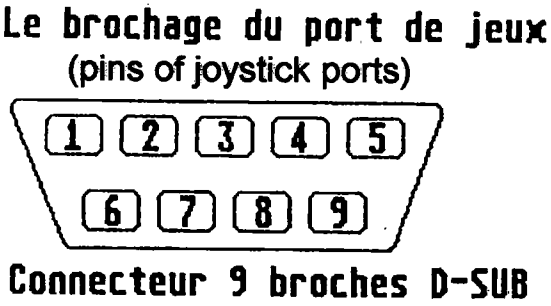


Figure 1.3.6

Utilisation		Souris	Joystick	Paddle	Actionware Light gun or Light pen	Trojan Light gun
entrée	1	impulsion V	avant	inutilisé	Unused	Unused
entrée	2	impulsion H	arrière	inutilisé	Unused	Unused
entrée	3	impulsion VQ	gauche	bouton gauche	Unused	Trigger
entrée	4	impulsion HQ	droite	bouton droit	Unused	Unused

		Souris (mouse)	Joystick	Paddle	Actionware Lightgun or Light pen	Trojan Light gun
E/S	5	(bouton 3)	inutilisé	potentiomètre droit	bouton (trigger)	Unused
E/S	6	bouton 1	bouton feu	inutilisé	signal LP	≃ HitSense
	7	+ 5 volts	+ 5 volts	+ 5 volts	+ 5 volts	+ 5 Volts
	8	GND	GND	GND	GND	GRN
E/S	9	bouton 2	inutilisé	potentiomètre gauche	inutilisé	Unused

Ces connecteurs sont des entrées d'outils de commande, tels la souris, le joystick, le trackball, la manette ou le stylo lumineux. On en trouve deux :

Le connecteur gauche (gameport 0) et *le connecteur droit* (gameport 1). La structure des deux connecteurs est identique, la seule différence réside dans l'utilisation du signal LP par le gameport 0. Ces connecteurs sont reliés, de façon interne, au CIA, à AGNUS, à DENISE et à PAULA.

Voici la répartition des liaisons entre les connecteurs et les circuits spécialisés.

GAMEPORT 0

Broche N°	Circuit	Broche
1	Denise	M0V (par multiplexeur)
2	Denise	M0H (par multiplexeur)
3	Denise	M1V (par multiplexeur)
4	Denise	M1H (par multiplexeur)
5	Paula	90Y
6	CIA-A	PA-6
	Comme Agnus	LP
9	Paula	P0X

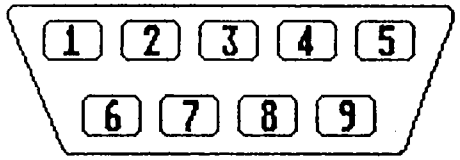
Le deuxième port NAND n'a aucun rapport avec la gestion du moteur. Il sert au mode d'identification, la majorité des teurs du commerce n'ayant besoin d'aucune aide. Lorsque le moteur est coupé, le signal SEL est actif, c'est-à-dire à 0, et le port met le signal RDY sur Low. L'AMIGA reconnaîtra ainsi la deuxième unité avec le terme DF1.

Etant donné que seul la moitié des deux circuits est nécessaire, cela suffit pour un lecteur supplémentaire. Les entrées de N1 doivent être reliées à SEL2 (broche 9 du lecteur externe).

Afin que le signal CHWG puisse fonctionner sans problème, certains lecteurs doivent être munis d'un JUMPER. Par exemple le lecteur FD 1035 de NEC doit être court-circuité par un JUMPER J1 caractéristique. Pour en savoir davantage, référez-vous au manuel du lecteur de disquette.

1.3.6 LE CONNECTEUR SOURIS-JOYSTICK

Le brochage du port de jeux



Connecteur 9 broches D-SUB

Figure 1.3.6

Utilisation

		Souris	Joystick	Manette	Light Pen
entrée	1	impulsion V	avant	inutilisé	inutilisé
entrée	2	impulsion H	arrière	inutilisé	inutilisé
entrée	3	impulsion VQ	gauche	bouton gauche	inutilisé
entrée	4	impulsion HQ	droite	bouton droit	inutilisé

E/S	5	(bouton 3)	inutilisé	potentiomètre droit	bouton
E/S	6	bouton 1	bouton feu	inutilisé	signal LP
	7	+ 5 volts	+ 5 volts	+ 5 volts	+ 5 volts
	8	GND	GND	GND	GND
E/S	9	bouton 2	inutilisé	potentiomètre gauche	inutilisé

Ces connecteurs sont des entrées d'outils de commande, tels la souris, le joystick, le trackball, la manette ou le stylo lumineux. On en trouve deux :

Le connecteur gauche (gameport 0) et *le connecteur droit* (gameport 1). La structure des deux connecteurs est identique, la seule différence réside dans l'utilisation du signal LP par le gameport 0. Ces connecteurs sont reliés, de façon interne, au CIA, à AGNUS, à DENISE et à PAULA.

Voici la répartition des liaisons entre les connecteurs et les circuits spécialisés.

GAMEPORT 0

Broche N°	Circuit	Broche
1	Denise	M0V (par multiplexeur)
2	Denise	M0H (par multiplexeur)
3	Denise	M1V (par multiplexeur)
4	Denise	M1H (par multiplexeur)
5	Paula	90Y
6	CIA-A	PA-6
	Comme Agnus	LP
9	Paula	P0X

GAMEPORT 1

1	Denise	M0V (par multiplexeur)
2	Denise	M0H (par multiplexeur)
3	Denise	M1V (par multiplexeur)
4	Denise	M1H (par multiplexeur)
5	Paula	P1Y
6	CIA-A	PA7
9	Paula	P1X

La structure des différents outils de commande a été choisie d'une telle façon, que tous les joysticks, souris, stylo lumineux du marché puissent convenir. Il est ainsi possible d'employer, par exemple, le stylo lumineux utilisé sur le C64. Le signal LP (light pen) est engendré par le stylo, lorsque sa pointe est au contact de l'écran.

Tous les signaux caractérisés par bouton, ainsi que ceux des quatre directions du joystick, sont *Low-actives*. Dans les différents outils de commande, se trouvent des interrupteurs qui sont reliés aux entrées correspondantes et à la masse (GND). Un signal *high* sur l'entrée signifie interrupteur ouvert, le signal *Low* correspondant à l'interrupteur fermé.

Les entrées analogiques raccordées à P0X, P0Y, P1X et P1Y acceptent des résistances variables (potentiomètre) d'une valeur de 470 Kohm. Elles sont installées entre le +5 volts et l'entrée correspondante.

Les deux boutons de tir, qui sont reliés au CIA-A, peuvent être programmés en mode sortie. Il faudra toutefois être attentif lors d'un accès écriture sur le registre port, à ne pas surcharger le dernier bit de port, afin de ne pas "planter" le système (PA0 : OVL).

La question des signaux des gameports sera vue plus en détail dans le chapitre **Programmation des circuits spécialisés.**

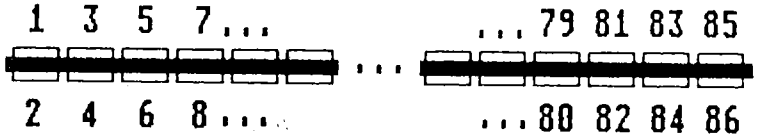
L'alimentation 5 volts des deux connecteurs n'est pas reliée directement à la tension de service de l'AMIGA. On a instauré un interrupteur frontière d'alimentation entre les deux.

Il sépare en fait le courant permanent de court-circuit de 400 mA des pointes de ter n de 700 mA. Cette entrée est donc complètement protégée des court-circuits. Comme la tension aux deux broches +5 volts ne doit pas tomber brusquement, la consommation de courant ne doit pas dépasser 250 mA à l'ensemble des deux connecteurs.

Malheureusement on a supprimé ces interrupteurs de protection sur les modèles A 500 et A 2000.

1.3.7 CONNECTEUR D'EXTENSION

Le brochage du port d'Expansion



Connecteur de platine 86 broches A500 & A1000

Figure 1.3.7

1	GND	2	GND
3	GND	4	GND
5	+5 Volts	6	+5 Volts
7	extension	8	-5 Volts
9	extension (28 M chez A 2000)	10	+12 Volts
11	extension	12	GND
13	GND	14	/C3
15	CDAC	16	/C1
17	/OVR	18	XRDY
19	/INT2	20	/PALOPE (inutilisé chez A 2000)
21	A5	22	/INT6
23	A6	24	A4
25	GND	26	A3
27	A2	28	A7
29	A1	30	A8

31	FC0	32	A9
33	FC1	34	A10
35	FC2	36	A11
37	GND	38	A12
39	A13	40	/IPL0
41	A14	42	/IPL1
43	A15	44	/IPL2
45	A16	46	/BERR
47	A17	48	/VPA
49	GND	50	E
51	/VMA	52	A18
53	/RES	54	A19
55	/HLT	56	A20
57	A22	58	A21
59	A23	60	/BR
61	GND	62	/BGACK
63	PD15	64	/BG
65	PD14	66	/DTACK
67	PD13	68	/PRW
69	PD12	70	/LDS
71	PD11	72	/UDS
73	GND	74	/AS
75	PD0	76	PD10
77	PD1	78	PD9
79	PD2	80	PD8
81	PD3	82	PD7
83	PD4	84	PD6
85	GND	86	PD5

On reconnaît tous les signaux de gestion importants et de bus du système de l'AMIGA sur le connecteur d'extension. On peut y raccorder des extensions mémoires, de nouveaux processeurs et autres. Ce connecteur, pour le modèle 1000, est placé du côté des deux gameports et caché sous un capot en plastique. En ce qui concerne l'AMIGA 500, ce connecteur est placé dans un boîtier sous l'appareil. Il se présente sous la forme d'un montage de 86 broches. L'écart entre les broches est de 0,1 pouce, ce qui correspond à 2,54 mm. Une fiche pouvant s'y raccorder peut se trouver assez facilement dans le commerce.

Dans l'AMIGA 2000, on observe 2 branchements de bus différents, composés d'un nnecteur MMU, correspondant à ce qui a été vu plus haut, et de 5 connecteurs à 100 broches, aussi appelés *Bus Zorro*. Ces six emplacements se trouvent sur le montage général, dans le boîtier de l'A 2000. Ces connecteurs de 86 et 100 broches sont conçus pour des cartes d'extension spéciales.

La majorité des signaux du connecteur d'extension sont reliés directement aux signaux correspondants du 68000. Les fonctions des signaux suivants sont décrites au chapitre 1.2.1.

A0-A23 : bus d'adresse

PD0-PD15 : bus de données processeur

IPL0-IPL2 : signaux d'interruption processeur

FC0-FC2 : signaux des codes fonction du 68000

AS, UDS, LDS, PRW, DYACK, VMA, VPA : signaux de gestion des bus

RES, HLT, BERR, BG, BGACK, BR, E : signaux de gestion du 68000

Les autres signaux ont les fonctions suivantes :

INT2 et INT 6

Ces deux signaux sont reliés aux broches du même nom de PAULA. Par leur entremise, une interruption de niveau 2 ou de niveau 6 peut être appelée.

CDAC, C1, C3 et 28M (A2000)

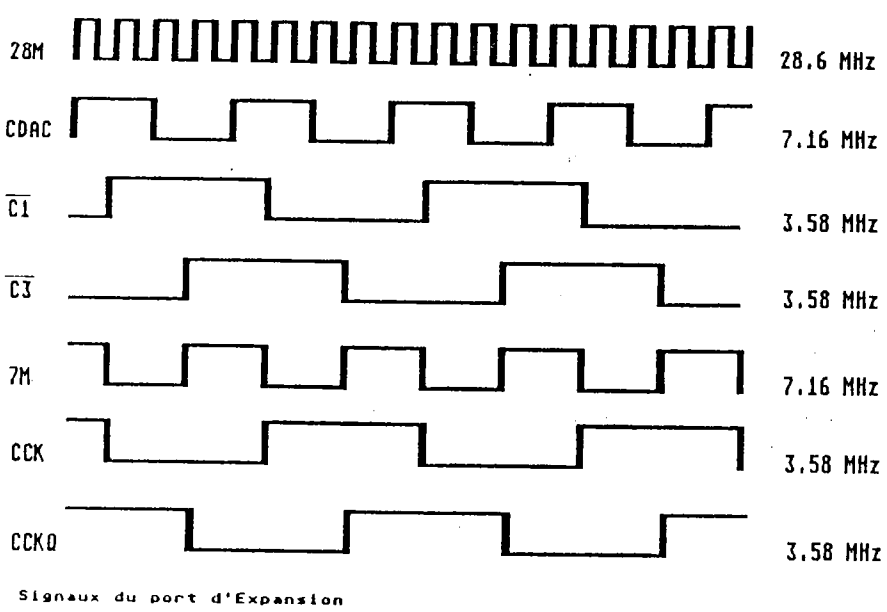


Figure 1.3.7.1

La fréquence et la position de phase des différents signaux d'horloge se déduisent facilement du schéma. Sur le modèle A2000, la haute impulsion d'horloge, d'une fréquence de 28,64 MHZ, s'applique aussi au connecteur d'extension. Les signaux 7M, CCK et CCKQ ne se trouvent évidemment pas sur le connecteur, 7M étant l'horloge du 68000, CCK et CCKQ étant reliés aux circuits spécialisés.

XRDY, OVR et PALOPE

Ces signaux servent à la configuration automatique des cartes d'extension. Leur fonction réelle n'a pu, jusqu'ici, être expérimentée. Les signaux caractérisés par *"extension"* n'ont jamais été référencés. Les futures extensions de l'AMIGA sont tous droits réservés. Avec l'AMIGA 2000, ils ont déjà été, en partie, utilisés (cf. signal d'horloge 28 MHZ).

1.3.8 ALIMENTATION DES CONNECTEURS

A chaque connecteur de l'AMIGA s'applique une ou plusieurs des trois tensions de service. Il est aussi possible d'alimenter plusieurs périphériques au travers des connecteurs correspondants.

On devra toutefois être attentif à la charge maximale admise par le branchement.

Voici le tableau des charges maximales recommandées par Commodore pour l'AMIGA 1000.

Connecteur	+ 5 volts	+ 12 volts	- 5 volts
Modulateur TV	-	60 mA	-
RGB	300 mA	175 mA	50 mA
RS232	100 mA	50 mA	50 mA
Disk ext.	270 mA	160 mA	-
Centronics	100 mA	-	-
Extension	1 000 mA	50 mA	50 mA
Gameport 0	125 mA	-	-
Gameport 1	125 m	-	-

Ce tableau n'est à prendre en compte que dans ses grandes lignes. Ces valeurs ne sont valables que lorsque tous les connecteurs sont en charge. Si, par exemple, le connecteur d'extension est inoccupé, les 1 000 mA du + 5 volts sont disponibles pour un autre connecteur. Si certains connecteurs sont libres dans une configuration système déterminée, les autres sorties se verront attribuer une charge maximale plus importante. Evidemment on peut aussi employer la méthode de forcer, tant qu'une carte d'extension est connectée, jusqu'à ce que le bloc d'alimentation saute.

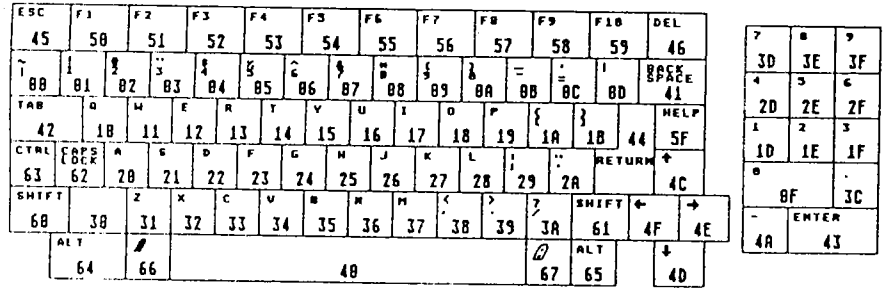
Cette façon de court-circuiter le système ne cause pas trop de dommages, mais les expériences sont à mener avec précaution. [illegible] d'un court-circuit, il peut s'écouler un courant de plus de 8 ampè[illegible]

Ce tableau n'est valable que pour l'AMIGA 1000 On ne peut pas s'y reporter pour l'AMIGA 500 ou 2000, étant donné que l'alimentation de ces ordinateurs est dimensionnée d'une autre façon. Pour l'A500, la charge admise de l'alimentation est plus faible. On est obligé d'utiliser un bloc d'alimentation supplémentaire pour toutes extensions un peu gourmandes en électricité.

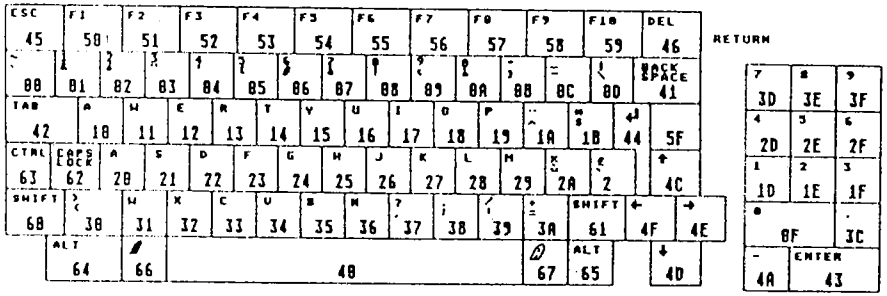
L'alimentation de l'A2000 est plus importante que celle de l'AMIGA 1000. En effet cet ordinateur doit être capable de supporter plusieurs cartes d'extension, dont celles d'émulation IBM.

Supplément : L'AMIGA 500 diffère aussi du modèle 1000 par sa tension négative, celle-ci étant de - 12 volts sur l'AMIGA 500 et de - 5 volts sur le modèle 1000.

1.4 Le clavier



Clavier ASCII Américain



Clavier français AZERTY

Figure 1.4.1

Le clavier de l'AMIGA est dit intelligent, du fait de la présence d'un micro-processeur qui prend en charge le traitement des touches, ne délivrant à l'AMIGA que les codes claviers effectifs. Il existe plusieurs réalisations et versions de clavier, mais elles ne diffèrent entre elles que par le rajout de certaines touches, donc de certains codes. Les schémas montrent la version américaine et la version française, avec leurs codes touches correspondants. Comme on peut le remarquer, les codes ne correspondent pas au standard ASCII. Le clavier délivre exclusivement des codes clés *RAW*, qui seront traduits en codes ASCII, par le système d'exploitation, à l'aide d'une table de transcription.

On remarque toutefois une organisation des codes clés RAW :

$00-$3F ce sont les codes des lettres de l'alphabet, des nombres et autres touches à caractères. Leur répartition correspond à l'ordre des touches du clavier.

$40-$4F codes des touches RETURN, TAB, BACKSPACE.

$50-$5F codes des touches de fonctions et HELP.

$60-$67 codes des touches d'option des différents niveaux du clavier (Shift, Amiga, Alternate et Control).

Le processeur clavier peut aussi différencier l'état *touche enfoncée* de l'état *touche relâchée*. Les codes clavier s'étalent sur 7 bits (valeur de $00 à $7F), le huitième bit étant le drapeau *KEY up/down*. C'est ce bit qui sera utilisé par l'ordinateur pour connaître l'état de la touche. Si le bit 8 vaut 0, c'est que la touche est enfoncée (*Key down*). Si ce même bit est à 1, c'est que la touche a été relâchée (*Key up*). On peut aussi utiliser le clavier en appuyant sur plusieurs touches en même temps, certains logiciels de musique notamment, exploitent cette possibilité afin de réaliser des accords.

La touche *Caps Lock* est une exception. En effet le clavier simule un interrupteur. Si on appuie sur cette touche, elle reste activée et la diode s'allume. Lorsqu'on appuie une deuxième fois, une fois la touche relâchée, la diode s'éteint. Cette touche retenue porte l'état du drapeau *Key up/down*.

Si on active *Caps-Lock*, la LED s'allume et le cc touche (8 bits) est envoyé à l'ordinateur, afin de signaler que la touche est enfoncée. Si cette touche est laissée tranquille, aucun code *Key up* ne sera envoyé à l'ordinateur. Lorsqu'on enfoncera la touche à nouveau, le code *Key up* sera envoyé (8ème bit activé) et la LED s'éteindra.

1.4.1 SCHEMA ELECTRONIQUE DU CLAVIER

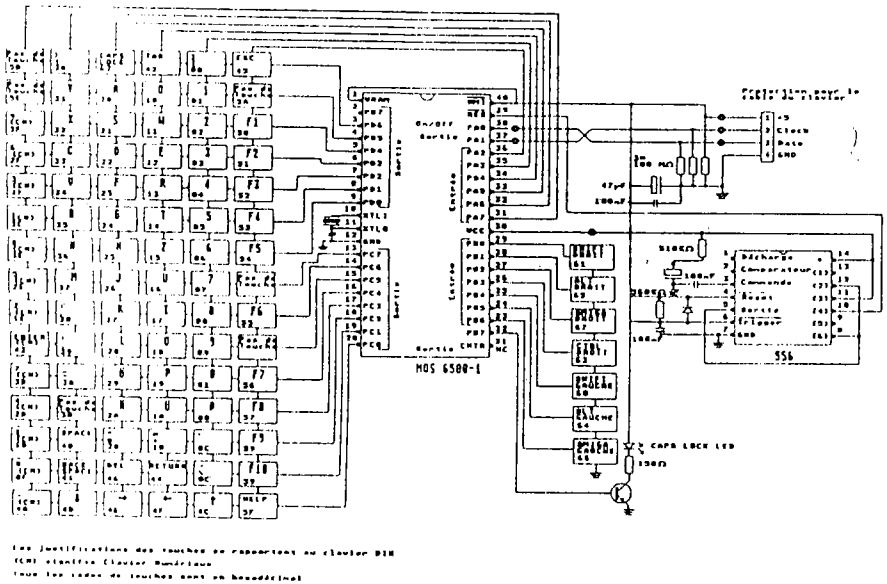


Figure 1.4.2

Le microprocesseur 6500/1 est le maître du montage clavier. Le 6500/1 est en fait un micro-ordinateur à lui tout seul et possède toutes les composantes nécessaires au travail d'un ordinateur. Le coeur du 6500/1 est en fait le microprocesseur de type 6502 (aha!). De plus, il possède 2 KO de ROM, contenant le programme directeur, 64 octets de RAM statique, 4 ports 8 bits directionnels, un compteur 16 bits avec une entrée directrice et, enfin, un générateur d'horloge.

Pour fonctionner, le 6500/1 n'a besoin que d'une alimentation de 5 volts et d'un quartz pour le signal d'horloge. Le clavier de l'AMIGA exploite le 6500/1 avec un quartz à 3 MHZ. Etant donné que la fréquence interne est divisée par 2, la fréquence d'horloge est de 1,5 MHZ.

Le deuxième cir it du montage clavier est une minuterie de précision du type 556. En .ait, cette minuterie en comporte deux dans le même boîtier. Ce circuit engendre le signal RESET du 6500/1.

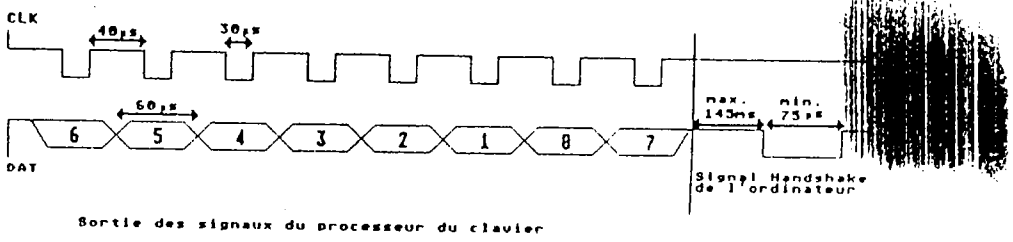
Les touches sont séparées en deux groupes. Les sept touches, *Shift* (gauche et droite), *Alt* (gauche et droite), *Amiga* (gauche et droite) et *Control* sont reliées directement aux premiers signaux de port PB.

Les autres touches sont ordonnées suivant une matrice de six colonnes sur 15 lignes. Les colonnes sont reliées du signal PA2 au signal PA7 du port A. Ces six signaux de port sont commutés en mode entrée. Les 15 lignes sont dirigées vers le Port C et D. La broche PD7, correspondant à une 16ème ligne, n'est plus connectée dans les dernières versions.

Lorsque le 6500/1 interroge le clavier, il met la rangée de chaque ligne à 0. Etant donné que les sorties des ports C et D sont de type *OPEN COLLECTOR*, sans résistance *PULLUP* intégrée, les sorties mises à 1 sont retenues comme étant inactives. Lorsque le processeur a mis une ligne à 0, il teste les 6 colonnes. Les six entrées colonnes sont pourvues de résistances *Pullup* internes, afin que les touches enfoncées ne soient pas interprétées en tant que signal *high*.

Chaque touche enfoncée relie une colonne à une ligne. Si la touche, dans la rangée activée par le processeur, est enfoncée, les entrées de colonnes correspondantes seront mises à 0. Une fois que toutes les rangées seront activées, et que la colonne correspondante sera lue, le processeur connaîtra l'état de toutes les touches. Si celui-ci s'est modifié depuis le dernier test, il enverra les codes claviers correspondant à l'ordinateur.

1.4.2 TRANSFERT DES DONNEES



Sortie des signaux du processeur du clavier

Figure 1.4.3

Le clavier est relié à l'Amiga au moyen d'un câb à quatre faisceaux. Deux des lignes ne servent qu'à l'alimentation 5 volts de l'électronique du clavier. Le transfert des données s'effectue sur les deux lignes restantes. L'une de ces lignes sert au signal des données *KDAT*, l'autre au signal d'horloge *KCLK*. Dans l'Amiga, les signaux *KDAT* et *KCLK* sont reliés respectivement au signal d'entrée *SP* et à la broche *CNT* du CIA-A.

Le transfert des données est unidirectionnel, du clavier vers l'Amiga. Le 6500/1 lâche les bits de données sur le signal *KDAT*, et les accompagne d'une longue impulsion *LOW* d'horloge (KCLK), d'une durée de 20 microsecondes. La durée entre chaque impulsion d'horloge est de 40 microsecondes. Le temps de transfert est donc de 60 microsecondes par caractère (40+20). Ceci correspond à 1 octet (8 bits) toutes les 480 microsecondes ou un débit de 16666 BAUD (bits/seconde).

Après l'émission du dernier bit, le clavier attend une impulsion *HANDSHAKE* de l'ordinateur. Dans ce but, l'Amiga met le signal *KDAT* sur *LOW*, pendant environ 75 microsecondes.

Le schéma explique bien le déroulement du transfert. On remarque que les données ne sont pas envoyées suivant l'ordre décroissant 7-6-5-4-3-2-1-0, mais après rotation vers la gauche d'une position de bit : 6-5-4-3-2-1-0-7. Par exemple, le code clavier de la lettre 'J', 10100110, sera envoyé, après rotation, sous la forme : 01001101. Le drapeau *Key up/down* sera toujours transmis en dernier.

Les signaux de données sont de type *LOW-ACTIVE*, c'est-à-dire que 0 reproduit l'état *LOW*, 1 reproduisant l'état *HIGH*.

Le registre à décalage du CIA de l'Amiga, prend en charge à chaque impulsion d'horloge, le bit se trouvant sur le signal SP. Après huit impulsions, le CIA possède un octet complet de données. Le CIA engendre alors une interruption de niveau 2, permettant au système d'exploitation de réaliser les opérations suivantes :

- lecture des registres de données séries,
- inversion et rotation des octets vers la droite, afin d'obtenir le code clavier valide,

- émissi de l'impulsion HANDSHAKE,
- tâche interne suivant les codes reçus.

Synchronisation

Afin qu'un transfert de données puisse aboutir sans problème et sans erreur, le timing du transmetteur et celui du recepteur doivent concorder. La position des bits doit être la même, lors du transfert. Si une erreur se déclare, le clavier peut très bien envoyer une série de huit bits, alors que le port série du CIA n'est qu'au milieu du traitement de l'octet précédent. Une perte de synchronisation peut être consécutive à un arrêt de l'Amiga, ou à un branchement intempestif du clavier alors que l'Amiga est allumé. L'ordinateur n'a aucune possibilité de reconnaître une erreur de synchronisation. Cette tâche incombe, en fait, au clavier.

Après chaque transfert d'octets, le clavier attend au maximum 145 millisecondes le signal HANDSHAKE. S'il ne se passe rien durant ce cours moment, le processeur clavier décide qu'il y a erreur de transfert et introduit un mode spécial, avec lequel il cherchera à retrouver la synchronisation perdue. Cela consiste à émettre le signal *KDAT* à 1 en même temps qu'une impulsion d'horloge et d'attendre à nouveau pendant 145 millisecondes, le signal de synchronisation. Ceci sera répété jusqu'à ce que le signal HANDSHAKE soit reçu, signifiant que la synchronisation est à nouveau effective.

Lorsque l'octet donnée reçu par l'Amiga est incorrect, l'état des sept premiers bits est ignoré. Seul le dernier bit est reconnu comme étant à 1, le processeur clavier n'émettant que des 1 comme cela a été décrit plus haut. Etant donné que le dernier bit est un drapeau *Key up/down*, le code clavier est toujours un code *Key up* erroné, c'est-à-dire, un code touche relâchée. Si l'émission du bit incorrect avait été du type *Key down*, les possibilités de trouble de programmation auraient été fréquentes. C'est pourquoi il y a, avant chaque transfert, rotation d'u vers la gauche, afin que le drapeau *Key up/down* soit toujours en en dernier.

Les codes spéciaux

Il existe encore quelques indications spéciales, qui, pendant le transfert, sont communiquées par le clavier de l'Amiga au moyen de codes spéciaux. En voici la liste :

Code	Signification
$F9	le dernier code clavier était incorrect
$FA	tampon clavier saturé
$FC	test propre du clavier était incorrect
$FD	début des touches enfoncées après mise en route
$FE	fin des touches enfoncées après mise en route

$F9

Le code $F9 sera toujours envoyé par le clavier lors d'une perte de synchronisation, et lorsque celle-ci est rétablie. L'Amiga reconnaît alors qu'il y a eu émission incorrecte. Après ce code spécial, le clavier transmettra le code précédemment perdu.

$FA

Le clavier dispose d'un tampon de 10 caractères. Lorsque ce dernier est saturé, il envoie le code $FA, afin de signaler à l'ordinateur que des codes clavier risquent d'être perdus s'il ne le vide pas.

$FC

Après la mise en route de l'ordinateur, le processeur clavier commence une série de tests propres. On le remarque à la courte activation de la *LED* de la touche *CAPS LOCK*. Si ces tests mettent à jour une erreur, le clavier, tout en se bloquant, envoie un code *$FC* à l'Amiga, la *LED* restant allumée.

$FD et $FE

Lorsque le test de mise en route est terminé, le clavier commence le transfert de tous les codes des touches qui ont été activées depuis la mise en route de l'ordinateur. Avant le début de l'émission, le clavier place le code *$FD*, le code *$FE* étant placé en fin de transfert.

Si aucune touche n'a été activée, *$FD* et *$FE* seront envoyés l'un derrière l'autre.

Reset par le clavier

Le clavier de l'Amiga a la possibilité de libérer un signal *RESET*. Si on appuie simultanément sur les 2 touches AMIGA, ainsi que sur la touche *CONTROL*, le processeur clavier met le signal *KCLK* sur *LOW*, pendant environ 0,5 secondes. Ceci provoque un signal *RESET*, libérant une remise à zéro du processeur. Ce dernier se déclenche dès qu'une des trois touches est relâchée. On le remarque aussi au voyant clignotant de la touche *CAPS LOCK* (un fait intéressant est que le signal *KCLK*, qui libère le *RESET*, est relié au signal *CNT* du CIA. On pourra donc, avec la programmation appropriée, déclencher une remise à zéro).

1.5 La programmation du hardware

Dans le chapitre précédent, c'est la structure du hardware de l'Amiga qui vous a été présentée. Maintenant que les profondeurs de votre ordinateur préféré n'ont plus aucun secret pour vous, nous allons découvrir les charmes de la programmation, qui permet, notamment, la création des sons et des graphismes.

Les bases d'une programmation sans problème, à tous les niveaux de la machine, passe par une bonne connaissance, d'une part, de l'organisation de la mémoire, et d'autre part, des registres de chaque circuit spécialisé.

1.5.1 ORGANISATION DE LA MEMOIRE

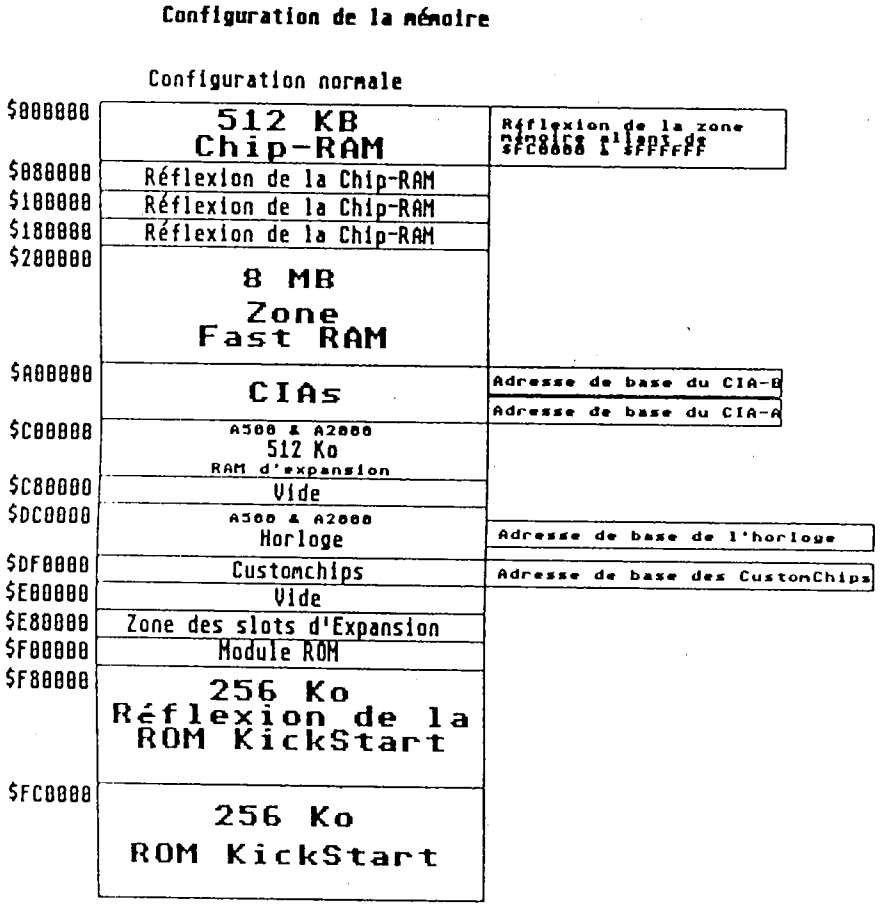


Figure 1.5.1

Ce graphique nous montre la configuration mémoire, telle qu'elle se présente au programmeur au démarrage. La zone adressable du 68000 se monte à 16 megaoctets (adresse 0 à $FFFFFF). La raison de cette taille n'est pas une surprise, étant donné qu'une grande zone est inexploitée et que certains circuits apparaissent à différentes adresses. Mais il n'y a pas de raison d'être avare avec la mémoire utilisable, l'époque où les switchings entre les zones étaient nécessaires étant, avec le 68000, heureusement dépassée.

La RAM

La zone mémoire représentée par CHIP-RAM correspond à la mémoire accessible normalement sur Amiga. Si l'extension mémoire de l'A1000 est absente, elle n'atteindra que $3FFFF. Etant donné que c'est la seule zone mémoire accessible par les trois circuits spécialisés, ces 512 Ko sont appelés *CHIP-RAM*.

Il est possible que le processeur soit ralenti par l'activité des trois circuits spécialisés, lors d'un accès mémoire à la *CHIP-RAM*. Si on veut l'éviter, on peut étendre la mémoire par adjonction de *FAST-RAM*. Celle-ci se placera à partir de l'adresse $200000, et pourra atteindre 8 mégaoctets. Comme cette zone est exclusivement réservée au 68000, ce dernier pourra y accéder avec toute sa vitesse, d'où le nom *FAST*. Dans sa version de base, l'Amiga ne comprend pas de *FAST-RAM*.

La carte d'extension des modèles 500 et 2000 a comme zone d'adresse $C00000 à $C7FFFF. Elle peut se comporter aussi bien en tant que *CHIP-RAM*, qu'en tant que *FAST-RAM*. En effet, l'accès peut être aussi bien interdit aux circuits spécialisés pendant un moment, que les accès processeurs peuvent y être ralentis par ces mêmes circuits. Cet état n'est pas dû à la malice d'un des concepteurs, mais à la simplicité de la conception qui est en fait une extension très bon marché.

Le CIA

Les différents registres du CIA apparaissent à l'adresse $A00000 et se terminent à l'adresse $BFFFFF. Les précisions sur l'adressage du CIA se trouvent au chapitre 1.2.

Voici à nouveau le détail des registres et leur adre › respective :

CIA-A	CIA-B	NOM	FONCTION
$BFE001	$BFD000	PA	données port A
$BFE101	$BFD100	PB	données port B
$BFE201	$BFD200	DDRA	direction port A
$BFE301	$BFD300	DDRB	direction port B
$BFE401	$BFD400	TALO	minuterie A (octet bas)
$BFE501	$BFD500	TAHI	minuterie A (octet haut)
$BFE601	$BFD600	TBLO	minuterie B (octet bas)
$BFE701	$BFD700	TBHI	minuterie B (octet haut)
$BFE801	$BFD800	E.LSB	eventcounter (bits 0-7)
$BFE901	$BFD900	E.MID	eventcounter (bits 8-15)
$BFEA01	$BFDA00	E.MSB	eventcounter (bits 16-24)
$BFEB01	$BFDB00		inutilisé
$BFEC01	$BFDC00	SP	données séries
$BFED01	$BFDD00	ICR	registre contôle interruption
$BFEE01	$BFDE00	CRA	registre contrôle port A
$BFEF01	$BFDF00	CRB	registre contrôle port B

Les circuits spécialisés

Les différents registres des circuits spécialisés se trouvent dans une zone de 512 octets. Chaque registre ayant une largeur d'un mot, on les trouve donc à toutes les adresses paires.

L'adresses de base des zones registres se trouve à $DFF000. L'adresse effective d'un registre se tient donc à $DFF000 + adresse registre. La liste qui suit présente les noms et fonctions de chaque registre des circuits. Il est clair que la plupart des descriptions de registres ne sont pas connues, étant donné qu'il n'a jamais rien été dit sur la plupart des registres. Cette liste procure une vue d'ensemble et peut servir de compléments aux adresses registres.

Il existe 4 type: › registres différents.

R (read)

Ces registres ne sont accessibles qu'en mode lecture.

W (write)

Ces registres ne sont accessibles qu'en mode écriture.

S (strobe)

Un accès à un tel registre libère une routine particulière du circuit correspondant. Ainsi, la valeur (le mot) se trouvant sur le bus de données et qui sera écrite dans un de ces registres n'est pas importante. Ce type de registre n'intéresse qu'AGNUS.

ER (early read)

Un registre signalé par *EARLY READ* est un registre d'émission *DMA*. Il contient des données qui seront inscrites dans la *CHIP-RAM* via les canaux DMA. Il n'y a que deux types de registres de ce genre (*DSKDATR* et *BLTDDAT* - registres d'émission du blitter et lecteur de disquette). L'écriture dans la *CHIP-RAM* se fera sous surveillance du contrôleur DMA d'AGNUS. Le processeur ne peut pas avoir accès à de tels fichiers.

A.D.P

Ces trois caractères correspondent aux trois circuits différents AGNUS, DENISE et PAULA. Ils permettent d'indiquer l'origine des registres correspondants. Il est possible qu'un registre soit issue de plusieurs circuits à la fois. Lors d'un accès écriture, la valeur pourra être mise en mémoire dans les deux ou même les trois circuits, en même temps.

Pour le programmeur, il n'est pas fondamental de savoir dans quel circuit un registre particulier se trouve. On peut considérer la zone des circuits spécialisés comme n'en faisant qu'une, et ne prendre en compte que l'adresse et la fonction du registre désiré.

p.d.

Un petit *d* signifie que le registre est assisté par le contrôleur DMA. La lettre *P* signifie que le registre n'est utilisé que par le processeur ou le *COPPER*. La combinaison des deux est possible, signifiant que le registre est accessible via DMA, mais aussi par le processeur.

Nombre de registres : 197

Nombre de registres accessibles via les canaux DMA : 54

Adresse de base de la zone registre : $DFF000

NOM	Adresse registre	CHIP	R/W	p/d	Fonction
BLTDDAT	000	A	er	d	destination BLITTER
DMACONR	002	AP	r	p	registre de contrôle DMA
VPOSR	004	A	r	p	position verticale ; bit de poids fort
VHPOSR	006	A	r	p	position horizontale et verticale
DSDATR	008	P	er	d	lecture données disque
JOY0DAT	00A	D	r	p	position souris/joystick gameport 0
JOY1DAT	00C	D	r	p	position souris/joystick gameport 1
CLXDAT	00E	D	r	p	registre des collisions
ADCONR	010	P	r	p	contrôle AUDIO/DISQUE
POT0DAT	012	P	r	p	potentiomètre gameport 0
POT1DAT	014	P	r	p	potentiomètre gameport 1
POTGOR	016	P	r	p	lecture données sur POT

SERDATR	0	P	r	p	status du port série
DSKBYTR	01A	P	r	p	status du port disque
INTENAR	01C	P	r	p	autorisation interrup.
INTREQR	01E	P	r	p	demande interruption
DSKPTH	020	A	w	p	adresse DMA disque bits 16-18
DSKPTL	022	A	w	p	adresse DMA disque bits 1-15
DSKLEN	024	P	w	p	longueur données disque
DSKDAT	026	P	w	d	données disque
REFPTR	028	A	w	d	compteur rafraichis.
VPOSW	02A	A	w	p	MSB de la position vert.
VHPOSW	02C	A	w	p	position verticale et horizontale
COPCON	02E	A	w	p	registre de contrôle COPPER
SERDAT	030	P	w	p	données série et bit de stop
SERPER	032	P	w	p	période et contrôle du port série
POTGOT	034	P	w	p	démarrage compteur POT
JOYTEST	036	D	w	p	écriture dans 2 compteurs souris/joystick
STREQU	038	D	s	d	synchronisation vert.
STRVBL	03A	D	s	d	synchronisation horiz.
STRHOR	03C	DP	s	d	signal synchronisation horizontale
STRLONG	03E	D	s	d	identification longueur horizontale

Seuls les registres suivants peuvent être accessibles au COPPER lorsque, COPCON = 1.

BLTCON0	040	A	w	p	contrôle Blitter
BLTCON1	042	A	w	p	contrôle Blitter
BLTAFWM	044	A	w	p	masque premier mot
BLTALWM	046	A	w	p	masque dernier mot
BLTCPTH	048	A	w	p	source C (bits 16-18)
BLTCPTL	04A	A	w	p	source C (bits 1-15)
BLTBPTH	04C	A	w	p	source B (bits 16-18)

BLTBPTL	04E	A	W	p	so : B (bits 1-15)
BLTAPTH	050	A	W	p	source A (bits 16-18)
BLTAPTL	052	A	W	p	source A (bits 1-15)
BLTDPTH	054	A	W	p	destination D (bits 16-18)
BLTDPTL	056	A	W	p	destination D (bits 1-15)
BLTSIZE	058	A	W	p	démarrage Blitter et dimensionnement taille fenêtre
	05A				inutilisé
	05C				inutilisé
	05E				inutilisé
BLTCMOD	060	A	W	p	modulo source C
BLTBMOD	062	A	W	p	modulo source B
BLTAMOD	064	A	W	p	modulo source A
BLTDMOD	066	A	W	p	modulo destination D
	068				inutilisé
	06A				inutilisé
	06C				inutilisé
	06E				inutilisé
BLTCDAT	070	A	W	d	données source C
BLTBDAT	072	A	W	d	données source B
BLTADAT	074	A	W	d	données source A
	076				inutilisé
	078				inutilisé
	07A				inutilisé
	07C				inutilisé
DSKSYNC	07E	P	W	p	remplissage sync. disque

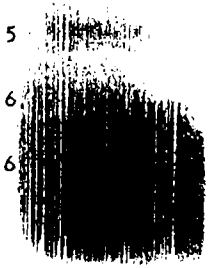
Seuls les registres suivants peuvent être, dans tous les cas, accessibles en mode écriture par le COPPER.

COP1LCH	080	A	W	p	1ère adresse COPPER (bits 16-18)
COP1LCL	082	A	W	p	1ère adresse COPPER (bits 1-15)
COP2LCH	084	A	W	p	2ème adresse COPPER (bits 16-18)

COP2LCL	16	A	W	p	2ème adresse COPPER (bits 1-15)
COPJMP1	088	A	s	p	redémarrage COPPER 1ère adresse
COPJMP2	08A	A	s	p	redémarrage COPPER 2ème adresse
COPINS	08C	A	W	d	identification instruc.
DIWSTRT	08E	A	W	p	coin sup. gauche fenêtre
DIWSTOP	090	A	W	p	coin inf. droit fenêtre
DDFSTRT	092	A	W	p	début bitplane (pos.hor)
DDFSTOP	094	A	W	p	fin bitplane (pos.hor)
DMACON	096	ADP	W	p	registre contrôle DMA
CLXCON	098	D	W	p	contrôle collision
INTENA	09A	P	W	p	autorisation interrup.
INTREQ	09C	P	W	p	demande d'interruption
ADKCON	09E	P	W	p	contrôle audio,disk,UART
AUD0LCH	0A0	A	W	p	canal audio 0 (bits 16-18)
AUD0LCL	0A2	A	W	p	canal audio 0 (bits 1-15)
AUD0LEN	0A4	P	W	p	longueur données audio
AUD0PER	0A6	P	W	p	période canal audio 0
AUD0VOL	0A8	P	W	p	volume canal audio 0
AUD0DAT	0AA	P	W	d	canal audio 0 données
	0AC				inutilisé
	0AE				inutilisé
AUD1LCH	0B0	A	W	p	canal audio 1 (bits 16-18)
AUD1LCL	0B2	A	W	p	canal audio 1 (bits 1-15)
AUD1LEN	0B4	P	W	p	longueur données audio
AUD1PER	0B6	P	W	p	période canal audio 1
AUD1VOL	0B8	P	W	p	volume canal audio 1
AUD1DAT	0BA	P	W	d	canal audio 1 données
	0BC				inutilisé
	0BE				inutilisé
AUD2LCH	0C0	A	W	p	canal audio 2 (bits 16-18)
AUD2LCL	0C2	A	W	p	canal audio 2 (bits 1-15)

AUD2LEN	0C4	P	W	p	lor. ur données audio
AUD2PER	0C6	P	W	p	période canal audio 2
AUD2VOL	0C8	P	W	p	volume canal audio 2
AUD2DAT	0CA	P	W	d	canal audio 2 données
	0CC				inutilisé
	0CE				inutilisé
AUD3LCH	0D0	A	W	p	canal audio 3 (bits 16-18)
AUD3LCL	0D2	A	W	p	canal audio 3 (bits 1-15)
AUD3LEN	0D4	P	W	p	longueur données audio
AUD3PER	0D6	P	W	p	période canal audio 3
AUD3VOL	0D8	P	W	p	volume canal audio 3
AUD3DAT	0DA	P	W	d	canal audio 0 données
	0DC				inutilisé
	0DE				inutilisé
BPL1PTH	0E0	A	W	p	bitplane 1 (bits 16-18)
BPL1PTL	0E2	A	W	p	bitplane 1 (bits 1-15)
BPL2PTH	0E4	A	W	p	bitplane 2 (bits 16-18)
BPL2PTL	0E6	A	W	p	bitplane 2 (bits 1-15)
BPL3PTH	0E8	A	W	p	bitplane 3 (bits 16-18)
BPL3PTL	0EA	A	W	p	bitplane 3 (bits 1-15)
BPL4PTH	0EC	A	W	p	bitplane 4 (bits 16-18)
BPL4PTL	0EE	A	W	p	bitplane 4 (bits 1-15)
BPL5PTH	0F0	A	W	p	bitplane 5 (bits 16-18)
BPL5PTL	0F2	A	W	p	bitplane 5 (bits 1-15)
BPL6PTH	0F4	A	W	p	bitplane 6 (bits 16-18)
BPL6PTL	0F6	A	W	p	bitplane 6 (bits 1-15)
	0F8				inutilisé
	0FA				inutilisé
	0FC				inutilisé
	0FE				inutilisé
BPLCON0	100	AD	W	p	reg. contrôle 0 bitplane
BPLCON1	102	D	W	p	reg. contrôle 1 bitplane
BPLCON2	104	D	W	p	reg. contrôle 2 bitplane
	106				inutilisé
BPL1MOD	108	A	W	p	contr. bitplane modulo plan impair

BPL2MOD	10A	A	W	p	contr. bitplane modulo plan pair
	10C				inutilisé
	10E				inutilisé
BPL1DAT	110	D	W	d	données bitplane 1 (RGB)
BPL2DAT	112	D	W	d	données bitplane 2 (RGB)
BPL3DAT	114	D	W	d	données bitplane 3 (RGB)
BPL4DAT	116	D	W	d	données bitplane 4 (RGB)
BPL5DAT	118	D	W	d	données bitplane 5 (RGB)
BPL6DAT	11A	D	W	d	données bitplane 6 (RGB)
	11C				inutilisé
	11E				inutilisé
SPR0PTH	120	A	W	p	données sprite 0 (bits 16-18)
SPR0PTL	122	A	W	p	données sprite 0 (bits 1-15)
SPR1PTH	124	A	W	p	données sprite 1 (bits 16-18)
SPR1PTL	126	A	W	p	données sprite 1 (bits 1-15)
SPR2PTH	128	A	W	p	données sprite 2 (bits 16-18)
SPR2PTL	12A	A	W	p	données sprite 2 (bits 1-15)
SPR3PTH	12C	A	W	p	données sprite 3 (bits 16-18)
SPR3PTL	12E	A	W	p	données sprite 3 (bits 1-15)
SPR4PTH	130	A	W	p	données sprite 4 (bits 16-18)
SPR4PTL	132	A	W	p	données sprite 4 (bits 1-15)
SPR5PTH	134	A	W	p	données sprite 5 (bits 16-18)
SPR5PTL	136	A	W	p	données sprite 5 (bits 1-15)
SPR6PTH	138	A	W	p	données sprite 6 (bits 16-18)
SPR6PTL	13A	A	W	p	données sprite 6 (bits 1-15)



SPR7PTH	13C	A	W	p	donnée sprite 7 (bits 16-18)
SPR7PTL	13E	A	W	p	données sprite 7 (bits 1-15)
SPR0POS	140	AD	W	dp	position départ sprite 0
SPR0CTL	142	AD	W	dp	contrôle sprite 0
SPR0DATA	144	D	W	dp	données A sprite 0 (RGB)
SPR0DATB	146	D	W	dp	données B sprite 0 (RGB)
SPR1POS	148	AD	W	dp	position départ sprite 1
SPR1CTL	14A	AD	W	dp	contrôle sprite 1
SPR1DATA	14C	D	W	dp	données A sprite 1 (RGB)
SPR1DATB	14E	D	W	dp	données B sprite 1 (RGB)
SPR2POS	150	AD	W	dp	position départ sprite 2
SPR2CTL	152	AD	W	dp	contrôle sprite 2
SPR2DATA	154	D	W	dp	données A sprite 2 (RGB)
SPR2DATB	156	D	W	dp	données B sprite 2 (RGB)
SPR3POS	158	AD	W	dp	position départ sprite 3
SPR3CTL	15A	AD	W	dp	contrôle sprite 3
SPR3DATA	15C	D	W	dp	données A sprite 3 (RGB)
SPR3DATB	15E	D	W	dp	données B sprite 3 (RGB)
SPR4POS	160	AD	W	dp	position départ sprite 4
SPR4CTL	162	AD	W	dp	contrôle sprite 4
SPR4DATA	164	D	W	dp	données A sprite 4 (RGB)
SPR4DATB	166	D	W	dp	données B sprite 4 (RGB)
SPR5POS	168	AD	W	dp	position départ sprite 5
SPR5CTL	16A	AD	W	dp	contrôle sprite 5
SPR5DATA	16C	D	W	dp	données A sprite 5 (RGB)
SPR5DATB	16E	D	W	dp	données B sprite 5 (RGB)
SPR6POS	170	AD	W	dp	position départ sprite 6
SPR6CTL	172	AD	W	dp	contrôle sprite 6
SPR6DATA	174	D	W	dp	données A sprite 6 (RGB)
SPR6DATB	176	D	W	dp	données B sprite 6 (RGB)
SPR7POS	178	AD	W	dp	position départ sprite 7
SPR7CTL	17A	AD	W	dp	contrôle sprite 7
SPR7DATA	17C	D	W	dp	données A sprite 7 (RGB)
SPR7DATB	17E	D	W	dp	données B sprite 7 (RGB)
COLOR00	180	D	W	p	valeur de la couleur 0
COLOR01	182	D	W	p	valeur de la couleur 1
COLOR02	184	D	W	p	valeur de la couleur 2
COLOR03	186	D	W	p	valeur de la couleur 3

COLOR04	188	D	W	p	valeur de la couleur 4
COLOR05	18A	D	W	p	valeur de la couleur 5
COLOR06	18C	D	W	p	valeur de la couleur 6
COLOR07	18E	D	W	p	valeur de la couleur 7
COLOR08	190	D	W	p	valeur de la couleur 8
COLOR09	192	D	W	p	valeur de la couleur 9
COLOR10	194	D	W	p	valeur de la couleur 10
COLOR11	196	D	W	p	valeur de la couleur 11
COLOR12	198	D	W	p	valeur de la couleur 12
COLOR13	19A	D	W	p	valeur de la couleur 13
COLOR14	19C	D	W	p	valeur de la couleur 14
COLOR15	19E	D	W	p	valeur de la couleur 15
COLOR16	1A0	D	W	p	valeur de la couleur 16
COLOR17	1A2	D	W	p	valeur de la couleur 17
COLOR18	1A4	D	W	p	valeur de la couleur 18
COLOR19	1A6	D	W	p	valeur de la couleur 19
COLOR20	1A8	D	W	p	valeur de la couleur 20
COLOR21	1AA	D	W	p	valeur de la couleur 21
COLOR22	1AC	D	W	p	valeur de la couleur 22
COLOR23	1AE	D	W	p	valeur de la couleur 23
COLOR24	1B0	D	W	p	valeur de la couleur 24
COLOR25	1B2	D	W	p	valeur de la couleur 25
COLOR26	1B4	D	W	p	valeur de la couleur 26
COLOR27	1B6	D	W	p	valeur de la couleur 27
COLOR28	1B8	D	W	p	valeur de la couleur 28
COLOR29	1BA	D	W	p	valeur de la couleur 29
COLOR30	1BC	D	W	p	valeur de la couleur 30
COLOR31	1BE	D	W	p	valeur de la couleur 31

Les registres de 1C0 à 1FC sont inutilisés.

Un accès à l'adresse registre *1FE* ne déclenche aucune fonction. Les circuits n'y ont d'ailleurs pas accès (cf : chapitre 1.2.3).

La ROM

La zone ROM se tient après la routine de démarrage. A partir de $FC0000, les 256 Ko de ROM renferment le kickstart de l'Amiga.

La zone mémoire de $F80000 à $FBFFFF est identiq à celle se tenant entre $FC0000 et $FFFFFF. On y retrouve encore une fois la ROM *KICKSTART*. Evidemment, cette configuration peut être modifiée. Après un RESET, le 68000 cherche l'adresse du premier ordre à exécuter à l'emplacement mémoire 4, que l'on appelle aussi vecteur RESET. Si la configuration mémoire n'est pas modifiée, le 68000 recherche le vecteur RESET dans la CHIP-RAM, qui se tient à l'adresse 4. Après le démarrage, ce contenu n'étant pas déterminé, le processeur sautera à une adresse arbitraire et le système se plantera dès le démarrage. La solution est la suivante : la CHIP-RAM, qui est si importante pour la configuration mémoire, a une entrée reliée au signal de port PA0 du CIA-A. Cette liaison, aussi appelée signal *OVL* (Memory Overlay), reste à l'état normal à 0, et la configuration mémoire correspond au schéma. Après un RESET, le signal de port se met automatiquement à 1. La zone mémoire, s'étalant de $F80000 à $FFFFFF, sera alors masquée par la zone d'adresse de 0 à $7FFFF (l'adresse 4 correspondra ainsi à $F80004). Le 68000 trouvera donc une adresse RESET valide, qui lui laissera l'accès au KICKSTART. Lors du déroulement de la routine RESET, le signal OVL sera remis à 0, ramenant ainsi la mémoire à un état normal.

On doit être très attentif lorsqu'on veut expérimenter ces signaux. Le fait que le programme, qui doit mettre à 1 le signal *OVL*, tourne dans la CHIP-RAM, peut avoir pour fâcheuse conséquence que ce programme s'élimine tout seul de la mémoire, et que le processeur aborde n'importe où à l'intérieur du KICKSTART, celui-ci se trouvant à la place de la CHIP-RAM après la commutation.

La WOM de L'A1000

On trouve d'autres particularités sur le modèle 1000. Les possesseurs de ce type d'Amiga se sont déjà rendus compte qu'on parle continuellement d'une *ROM KICKSTART* alors qu'au départ, ce dernier était chargé à partir d'une disquette. En fait la situation de l'A1000 était la suivante : le hardware était terminé, l'ordinateur prêt à être vendu, mais le software, c'est-à-dire le système d'exploitation KICKSTART, était incomplet et truffé d'erreurs. Pour résoudre ce problème, Amiga se vit inclure une RAM spéciale, où le système d'exploitation était chargé après le démarrage, l'accès à cette RAM étant verrouillé par la suite.

Celle-ci se cor orte en fait comme une ROM de 256 Ko et a été appelée par Co. .nodore : *WOM* (write once memory). Ce modèle a été commercialisé avec la version 1.0 du KICKSTART, les nouvelles versions corrigées étant à présent disponibles.

La mémoire morte des A500 et A2000 est de type ROM, cette dernière étant bien moins chère que sa précédente, la *WOM*. De plus la version 1.2 du KICKSTART était achevée.

La présence de cette *WOM* a tout de même soulevé des problèmes :

- *Après le démarrage, où se trouve le programme qui permet le chargement du KICKSTART ?*
- *Comment modifier ce KICKSTART s'il se trouve en RAM ?*

La zone du système d'exploitation de l'A1000 est identique à celle des modèles ultérieurs. En effet, on la retrouve aux adresses $FC0000 à $FFFFFF, même la zone $F80000 est présente. Il ne se passe rien si on cherche à écrire dans le KICKSTART. Il n'y a pas d'accès écriture possible. La routine de chargement (*BOOT ROM*) n'est pas non plus masquée.

En fait, l'ensemble est dirigé par le signal *RESET*. La configuration mémoire est modifiée après un signal *RESET*, celui-ci pouvant être d'origine différente (démarrage, touches *Amiga* & *Control*, ordre RESET du 68000).

Ainsi la *BOOT-ROM* se tient à partir de $F80000 (comme un reset a pour conséquence l'activation du signal *OVL*, le vecteur *RESET* est aussi issu de la *BOOT-ROM*) et il est alors possible d'écrire dans le KICKSTART et le modifier à son gré. Cet état ne dure que jusqu'au moment où l'on cherche à écrire dans la zone de la *BOOT-ROM* ($F80000 à $FBFFFF). La *BOOT-ROM* sera alors prise en charge et l'accès écriture sera interdit.

En d'autres termes :

- RESET permet l'écriture dans le KICKSTART et masque la BOOT-ROM.

- Un accès écriture à une adresse se trouvant entre $F80000 et $FBFFFF, interdit l'écriture et déconnecte la BOOT-ROM.

1.5.2 ELEMENTS DE BASE

Comme cela a été vu dans le chapitre précédent, il existe des registres qui peuvent être accessibles à partir du processeur, et d'autres qui peuvent être lus et écrits via les canaux DMA. Nous allons tout d'abord examiner le premier cas.

Programmation des registres processeurs

Les registres processeurs peuvent être adressés directement. Par exemple : la valeur du registre couleur arrière-plan doit être modifiée. Le registre a pour nom *COLOR00.* Lorsqu'on se reporte au tableau du chapitre 1.5.1, on trouve une adresse registre $180. A ceci, on rajoute l'adresse de base de la zone registre, c'est-à-dire l'adresse du premier registre du 68000. Celle-ci correspond à $DFF000. La somme des deux adresses donne $DFF180. L'instruction *MOVE.W* suffit à initialiser le registre :

```
MOVE.W #valeur,$DFF180     ; valeur dans COLOR00
```

Si on désire accéder à plusieurs registres, il suffit de mettre l'adresse de base dans un registre adresse et d'utiliser l'adressage indirect avec déplacement (la valeur de déplacement correspondant à l'offset) :

```
LEA $DFF000,A5             ; mise en mémoire de l'adresse de base dans A5
MOVE.W #valeur,$180(A5)    ; valeur 1 dans COLOR00
MOVE.W #valeur,$182(A5)    ; valeur 2 dans COLOR01
```


En temps normal l'accès à un registre processeur se fait de cette manière. On peut aussi y accéder avec un mot long, et dans ce cas, 2 registres correspondront à un seul. Ceci a un sens dans le cas d'un registre adresse. Celui-ci se trouve dans une paire de registres contenant une adresse 19 bits, avec laquelle toute la zone CHIP-RAM de 512 Ko est accessible. Tout ce qui, en tant que données, a un rapport avec les circuits spécialisés doit être mis dans la CHIP-RAM. Le registre *adresse* ne montre que des adresses paires. Comme un registre processeur ne peut contenir qu'un mot, deux registres à adresse contigue servent d'accueil à l'adresse mémoire 19 bits. Ainsi le premier contient les trois bits de plus fort poids (bits 16-18), le deuxième contenant les 16 bits inférieurs (bits 0-15). Il est ainsi possible d'initialiser deux registres lors d'un accès *mot long*.

Par exemple : le pointeur du premier bitplane doit être mis à l'adresse $40000. *BPL1PTH* est le nom du premier registre (bits 16-18) et *BPL1PTL* (bits 0-15) celui du deuxième. Les adresses registres sont respectivement *$0E0* et *$0E2* et *A5* contient l'adresse de base *$DFF000.*

```
MOVE.L #$40000,$0E0(A5)     ;initialise les registres BPL1PTL et
                            ;BPL1PTH avec la valeur correcte.
```

Il faut bien insister sur le fait qu'on ne peut écrire et lire un registre sur une seule et unique adresse registre. La plupart des registres ne sont d'ailleurs accessibles qu'en écriture et ne peuvent donc pas être lus. C'est le cas des registres utilisés plus haut. D'autres ne peuvent qu'être lus. Il est assez rare que les deux modes soient possibles. Ces registres possèdent alors deux adresses différentes, l'une permettant la lecture, l'autre l'écriture. On peut prendre comme exemple le registre de controle DMA, sur lequel on reviendra par la suite, où l'écriture est possible à l'adresse *$096(DMACON)* et où la lecture est possible à l'adresse *$002 DMACONR.*

Les accès DMA

On entend par accès *DMA,* comme cela a déjà été énoncé au chapitre 1.2.3, l'accès direct d'un circuit périphérique, le contrôleur DMA, à la mémoire système. Dans le cas de l'Amiga, le contrôleur DMA est inclus dans Agnus.

Il reproduit la liaison des différentes unités d'entrée/sortie (*input/output* ou *I/O*) du circuit spécialisé avec la CHIP-RAM. Qu'il s'agisse de données audio, écran ou disquette, le processus DMA se déroule toujours de la même manière. N'importe quelle unité, par exemple le contrôleur disque, nécessite de nouvelles données de la mémoire ou possède de nouvelles données prêtes pour la mémoire. Le contrôleur DMA attend alors le moment où le canal DMA sera libre, et transfère les données de ou vers la mémoire.

Pour simplifier les choses, il n'y a pas de transfert spécial des données d'unité d'entrée/sortie vers le contrôleur DMA. Cela se déroule normalement par registre. Chacune de ces unités d'entrée/sortie possède deux types de registres différents. Le premier est un registre normal, qui est accessible à partir du processeur et dans lequel les différents paramètres d'une tâche sont mis en mémoire. Le deuxième est un registre de données qui contient celles du contrôleur DMA. Celui-ci accède simultanément, lors d'un transfert DMA, au registre de données correspondant ainsi qu'à son emplacement mémoire RAM. Suivant le sens du transfert DMA, ceci correspond soit à un registre à lire et un accès écriture CHIP-RAM, soit à un registre à écrire et un accès lecture CHIP-RAM. Comme ces deux sont reliés sur le bus de données, ces dernières cheminent automatiquement du registre de données vers la RAM ou l'inverse. Les données ne sont pas mises en mémoire dans n'importe quel registre interne.

Au travers du transfert DMA se rajoute un troisième type de registre : le registre adresse DMA qui, suivant les besoins de l'unité d'entrée/sortie, contient l'adresse (ou les adresses) des données dans la RAM.

De plus, il existe des registres de contrôle centraux qui ne sont pas attribués à une unité spéciale d'entrée/sortie, mais préposés à des fonctions de gestion. C'est à cette catégorie qu'appartient le registre *DMACON*, vu plus haut.

Les registres de données sont accessibles en *mode écriture* par le processeur, mais ceci n'a que peu de sens, étant donné que le contrôleur DMA réalise cette opération d'une manière plus élégante et plus rapide.

Quelques unités d'I/O ne possèdent pas de canaux DMA propres. Le 68000 sera alors obligé de lire et écrire les données lui-même. Ce sont des exceptions à qui, par nature, il n'échoit que peu de données et où un DMA n'est pas nécessaire, comme par exemple l'entrée joystick/souris.

Voici les canaux DMA existants.

DMA bitplane : Par ces canaux DMA, les données image écran peuvent être lues dans la mémoire et écrites dans les registres de données de chaque bitplane, d'où ils atteignent les séquenceurs bitplane dont le rôle est la sortie de l'image sur l'écran.

DMA sprite : Transfert des données sprite de la RAM vers le registre de données sprite.

DMA disque : Transfert des données disquette vers la RAM ou inverse.

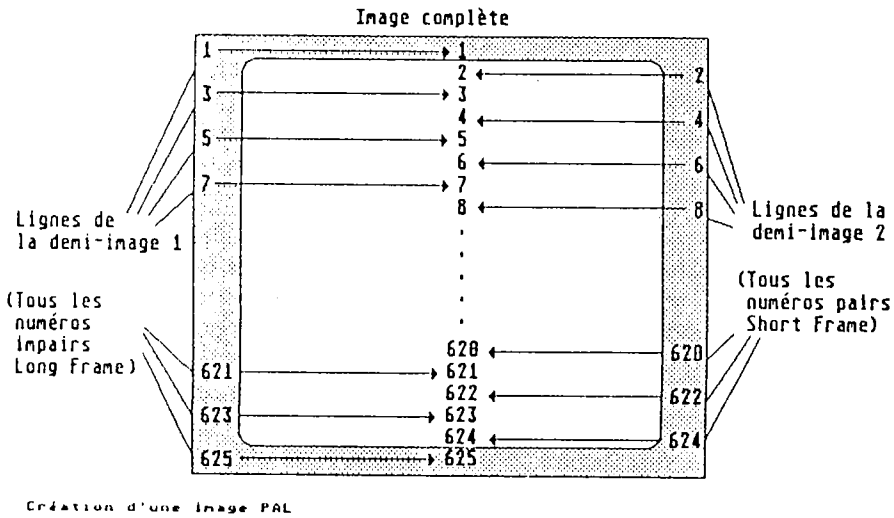
DMA audio : Lit les données sons digitalisés de la RAM et les dans le registre de données audio correspondant.

DMA copper : Par lui, le coprocesseur copper obtient son mot d'ordre.

DMA blitter : Transfert des données du et vers le blitter.

Il y a donc six canaux DMA qui ont tous accès à la mémoire, en plus du processeur qui, naturellement, en a aussi la possibilité. Pour résoudre les difficultés qui en résultent, on a conçu un temporisateur compliqué, dans lequel on attribue une position définie à chaque canal. Comme ce temporisateur s'oriente suivant l'image écran, nous devons d'abord étudier la structure de l'écran. Cet intermède sera aussi peu technique que possible car ce chapitre est consacré à la programmation des circuits spécialisés et non au hardware.

La structure de l'écran



Création d'une image PAL

Figure 1.5.2.1

La fonction de transfert d'une sortie vidéo d'un Amiga français correspond exactement à la norme PAL. Une telle image est composée de 625 lignes horizontales. Ces dernières sont affichées de gauche à droite. Chaque fin de ligne est suivi d'une pause, afin de permettre au *raster* de revenir à gauche. Le *raster* est le mouvement continu des électrons qui permettent l'affichage. Pendant cette phase, le rayonnement électronique n'est pas lumineux, permettant au retour du *raster*, de ne pas engendrer de scintillements parasitaires. Puis le processus est à nouveau répété pour l'affichage d'une nouvelle ligne.

Afin que l'image soit exempte de tout scintillement, on doit réafficher une nouvelle image assez rapidement. Comme notre oeil n'observe plus un changement d'image à une certaine fréquence, on a fixé un nombre d'images affiché par seconde supérieure à cette limite. Pour la norme PAL ceci correspond à 50 images par secondes. Evidemment il existe un fait qui complique la chose.

Si on veut afficher 625 lignes 50 fois par secondes, on arrive à un total de 31250 lignes par seconde. Lorsque l'élément de base de ce système vidéo a été mis au point, il n'aurait pas été possible de fabriquer un moniteur abordable et bon marché avec une telle fréquence d'affichage de ligne. Il a fallu trouver un arrangement. D'une part, le nombre d'images par seconde ne devrait jamais tomber en dessous de 50, pour raison de forts scintillements, et d'autre part, le nombre de lignes de l'écran ne pouvait être diminué. La solution fut la suivante : répartition des 625 lignes en deux images. La première image est alors affichée avec les lignes impaires (1, 3, 5, ..., 625) et la deuxième avec les lignes paires (2, 4, 6, ..., 624). Ainsi a-t-on à la suite 50 demi-images qui renferment toujours la moitié des lignes. Deux images d'un tel type correspondent à une image complète, renfermant 625 lignes. Le nombre d'images complètes par seconde se réduit à 25. La fréquence des lignes se monte à 15625 Hz (25*625 ou 50*312.5).

Malgré la haute résolution de 625 lignes, il apparaît un scintillement lorsqu'un contour est circonscrit par une ligne. Cette dernière n'est reproduite qu'une seule fois tous les 25ème de seconde, ceci étant visible à l'oeil nu. Cet effet peut être observé sur les bords horizontaux des surfaces, l'origine résidant dans l'affichage de la ligne horizontale.

Le terme anglais correspondant à cette technique d'affichage des lignes paires et impaires est *interlace*. Les deux termes suivants servent à différencier les deux types de 'demi-image'. Avec *LONG FRAME* on caractérise celle reproduite à l'aide des lignes paires et l'autre avec *SHORT FRAME*. Cette dénomination est conséquente à la différence du nombre de lignes entre les 2 demi-images, 313 pour les lignes impaires, 312 pour les lignes paires. L'image *LONG FRAME* sera donc affichée plus longtemps.

Chaque affichage d'une demi-image est suivi d'une pause. Cette phase noire entre deux images correspond au retour vertical du faisceau d'électrons.

L'image générée sur l'Amiga obéit aux règles énoncées plus haut, mais avec quelques particularités. Normalement, la deuxième demi-image (*short frame*) s'affiche légèrement différée, afin que les lignes paires puissent s'afficher exactement entre les lignes impaires pour former une image complète.

Sur l'Amiga, les deux demi-images sont identiques : la fréquence reste à 50 hertz. A la suite de quoi le nombre de lignes a été défini à 313. On reconnaît aussi la séparation entre deux lignes sur le moniteur, étant donné que les demi-images ne sont plus affichées, déplacées l'une vers l'autre.

Afin d'augmenter le nombre de lignes, l'Amiga a la possibilité de générer une image en mode *interlace*, 625 lignes étant alors possibles.

La structure de la sortie vidéo de l'Amiga

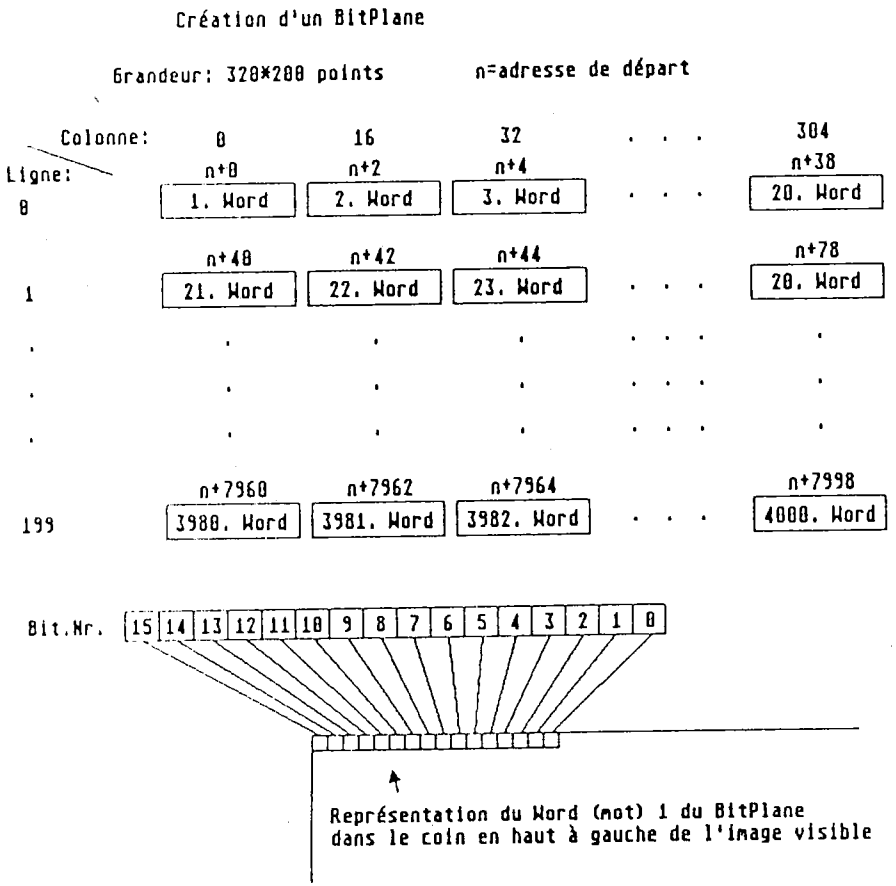


Figure 1.5.2.2

Bitplanes

L'amiga reproduit toujours son image dans une sorte de mode graphique, c'est-à-dire que chaque point de l'image possède un correspondant en mémoire. Plus simplement, un bit activé de la RAM correspond à un point sur le moniteur. Cette simple structure d'images en mémoire se nomme sur l'Amiga : *bitplane*.

C'est l'élément de base de toute reproduction d'image sur l'Amiga et il se trouve dans une zone mémoire cohérente. Un nombre fixé de mots d'une ligne d'écran donne la largeur d'une image. Un mot correspond à 16 points, étant donné que chaque bit représente un point. Une image de 320 points par ligne nécessite 320/16=20 mots par ligne. Etant donné qu'un *bitplane* ne peut différencier que deux états (point allumé/point éteint), plusieurs *bitplanes* peuvent être combinés. Dans ce cas, les bits auront toujours la même position dans tous les plans. Le premier point de l'image devient, par combinaison des plans, le bit de plus fort poids du premier mot. La valeur résultante de ces bits détermine la couleur du point à l'écran. L'obtention des couleurs par combinaison des bits d'un point sera plus détaillée dans le chapitre 1.5.5.

Les différentes résolutions graphiques

L'Amiga connaît deux résolutions horizontales différentes. Le mode *haute résolution* possède 640 points par lignes, la *basse résolution* étant de 320. Il est préférable de définir les deux différentes résolutions suivant le temps d'affichage d'un point d'écran. Un *pixel* (point d'écran) en mode haute résolution est affiché durant 70 nanosecondes, celui en mode basse résolution étant affiché durant 140 nanosecondes. Dans cette dernière résolution, le point étant deux fois plus gros, le faisceau électronique accomplira le double de trajet.

Le plus important pour le programmeur est de savoir qu'en mode *HR*, seul 4 bitplanes sont accessibles, alors qu'en mode *BR*, 6 bitplanes sont disponibles.

Structure d'une ligne Raster horizontale

Par la notion de *raster*, on entend une ligne complète horizontale, engendrée par le mouvement des électrons. Le *raster* sert de mesure du temps pour tous les processus DMA. Afin de mieux comprendre le partage du *raster*, on doit savoir, comment se répartissent les accès mémoire sur la CHIP-RAM et sur les registres des circuits spécialisés entre le contrôleur DMA et le processeur. Les accès à ces deux zones mémoire s'alignent sur les cycles de bus de même nom. Les cycles de bus déterminent le *timing* de la CHIP-RAM. A chaque cycle, un accès mémoire peut se déclarer, mais il n'est pas évident que les données soient lues ou écrites. Si le processeur veut, par exemple, accéder à un bus, on le lui attribue pendant un cycle. Le contrôleur DMA peut alors à nouveau accéder à la RAM au cycle suivant. Un cycle de bus dure environ 280 nanosecondes, 4 accès mémoire étant donc possibles lors d'une microseconde.

Mais le 68000 ne peut accéder à la CHIP-RAM aussi facilement, il n'est pas assez rapide. Suivant la fréquence d'horloge qui est exploitée dans l'Amiga, il ne peut réaliser un accès que tous les 560 nanosecondes. Dans le même temps se déroulent deux cycles de bus. Le 68000 ne peut donc s'imposer que tous les deux cycles, ou cycles mémoire pairs (*even cycles*). Les autres cycles, impairs, (*Odd cycles*) sont réservés au contrôleur DMA.

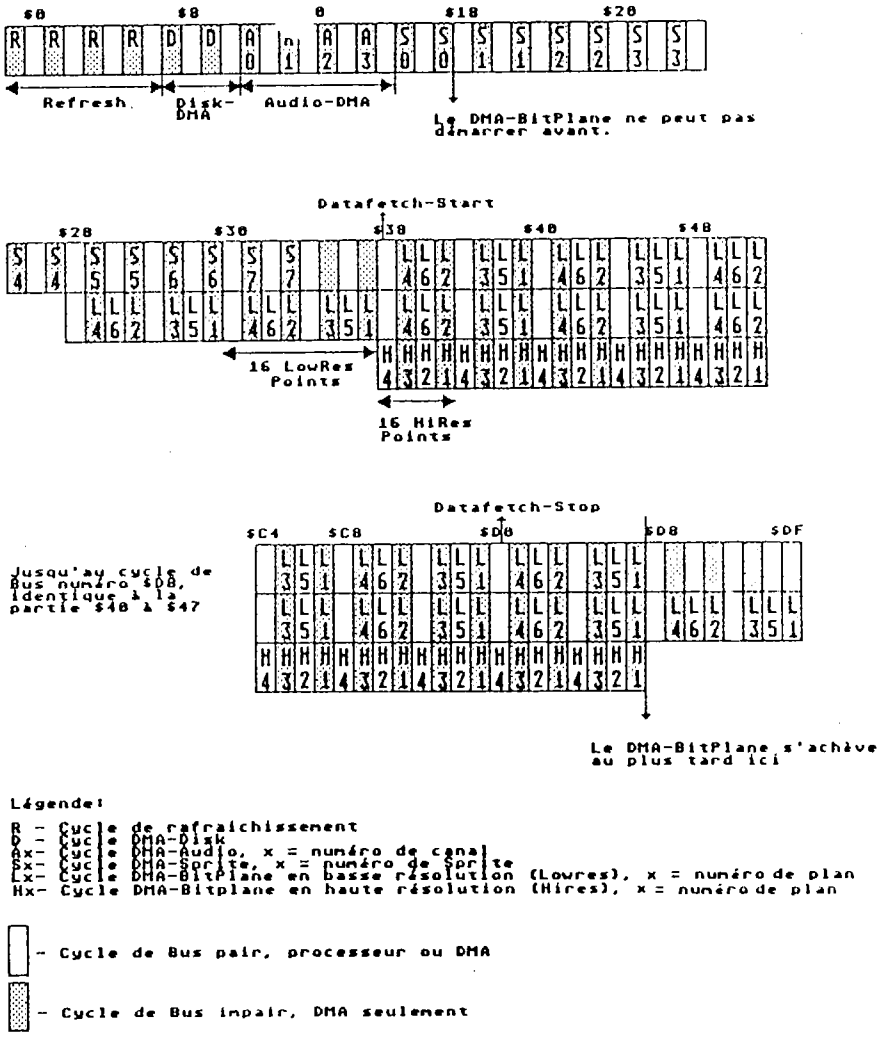


Figure 1.5.2.3

Le schéma nous montre le déroulement chronologique du raster sur une ligne. D'une durée de 63.5 microsecondes, ceci correspond à 227.5 cycles de bus par ligne, qui couvrent les 225 premiers du contrôleur DMA.

Ce qui se passe nous est indiqué par le schéma. ' s caractères à l'intérieur de chaque cycle représentent les canaux DMA correspondant. Alors que le contrôleur DMA n'utilise que les cycles impairs, le processeur doit avoir accès aux cycles pairs. Les accès DMA sont toujours prioritaires. Les accès *DMA blitter* et *DMA copper* ont lieu uniquement pendant les cycles pairs, mais n'ont aucune durée déterminée. Le *DMA copper* occupe tous les cycles pairs, jusqu'à ce qu'il ait fini sa tâche. Il est d'ailleurs prioritaire sur le *BLITTER*. Ce dernier fonctionne de la même manière, à ceci près qu'il peut allouer quelques cycles libres au 68000.

Comme cela apparaît sur le schéma, les accès *DMA sprite, audio* et *disquette* occupent des cycles impairs, n'influençant pas la vitesse du processeur. Les 4 cycles de bus indiqués par la lettre R correspondent aux cycles de rafraîchissement. Ils servent à raviver le contenu mémoire de la CHIP-RAM.

La répartition du *DMA bitplane* est quelque peu complexe. Pour reproduire les 16 premiers points sur l'écran, tous les bitplanes doivent être lus. Pendant que ces 16 points s'affichent, les bitplanes pour les 16 prochains points doivent être lus à leur tour. En basse résolution, 2 points sont émis à chaque cycle, signifiant ainsi que tous les 8 cycles, un bitplane est lu.

Les cycles impairs suffisent tant que 4 bitplanes et moins sont activés. Lorsque 5 ou 6 bitplanes sont utilisés, 2 cycles pairs doivent être alors employés, afin que toutes les données puissent être lues sur une durée de 8 cycles. L'utilisation est plus restreinte en mode haute résolution, seuls 4 points étant reproduits par cycles. Deux bitplanes *hires* peuvent donc être activés lors de l'utilisation des cycles impairs, l'utilisation des cycles pairs ammenant à un maximum de 4 bitplanes actifs. Le processeur perd ici plus de la moitié de ses accès au bus. Sa vitesse diminuera d'un même facteur, en partant du principe que le programme se trouve dans la CHIP-RAM, car ses accès à une éventuelle *FAST RAM* ou à la *ROM KICKSTART* ne sont pas diminués pour autant.

Les points indiqués *Datafetch-start* et *Datafetch-stop* correspondent au début et à la fin des accès DMA des bitplanes. Ils déterminent la longueur et la position horizontale de l'image reproduite.

Plus le DMA bitplan' 'ébute tôt et finit tard, plus le nombre de mots de données lus et donc le nombre de points émis, seront importants. Les résolutions normales de 320 et 640 points par lignes se laissent modifier. Si on met à $30 la valeur du *datafetch-start,* le canal DMA bitplane utilise normalement le cycle réservé au *DMA sprite.* De ce fait, suivant la valeur du *datafetch-start,* il peut disparaître jusqu'à sept sprites. Seul le sprite 0 ne se laisse pas éliminer de cette façon, celui-ci étant utilisé en tant que pointeur souris.

La ligne supérieure sur le schéma reproduit la répartition des cycles DMA, sur une largeur normale de 320 points en basse résolution. Le début du DMA bitplane, le *datafetch-start,* se trouve à $38 et la fin, le *datafetch-stop,* se trouve à $D0. A l'intérieur, le cycle marqué par L1 correspond à la lecture des données du bitplane 1, L2 à la lecture du bitplane 2 etc...

Si les bitplanes correspondants ne sont pas activés, leur cycle DMA sera éliminé.

La deuxième ligne reproduit le déroulement du *Raster* sur une ligne, dans laquelle le point *datafetch* a été déplacé vers l'extérieur. Jusqu'au *datafetch-start,* le déroulement est le même que celui de la ligne supérieure, le DMA bitplane débutant à $28. Ceci a pour conséquence l'élimination des sprites 5 à 7. La position du *datafetch-stop* ne pourra être décalée que jusqu'à la valeur maximale $D8.

La troisième ligne montre la répartition des cycles DMA en haute résolution, où les valeurs *datafetch* correspondent à celles de la première ligne.

Lors du retour vertical du faisceau d'électrons, les accès DMA bitplane n'ont pas lieu.

Les registres de contrôle DMA

Les canaux DMA sont activés ou désactivés suivant le registre cent[illegible] de contrôle.

Adresses du registre DMACON : écriture $096 ; lecture 02

Bit	Nom	Fonction
15	SET/CLR	Bits allumer/éteindre
14	BBUSY	Blitter travaille (seulement en lecture)
13	BZERO	résultat de toutes les opérations du Blitter est 0 (seulement en lecture)
12 et 11		inutilisé
10	BLTPRI	DMA blitter est prioritaire sur le processeur
9	DMAEN	activer DMA complet (bits 0 à 8)
8	BPELN	activer DMA bitplane
7	COPEN	activer DMA copper
6	BLTEN	activer DMA blitter
5	SPREN	activer DMA sprite
4	DSKEN	activer DMA disque
3-0	AUDxEN	activer DMA audio par canal son (le numéro de bit correspond au numéro de canal)

Le registre *DMACON* n'est pas utilisé comme un registre normal. On ne peut qu'activer des bits ou les effacer. Ceci sera établi par le bit 15 du mot de donnée écrit dans le registre *DMACON*. Si ce bit est à 1, tous les bits activés dans le mot de donnée le seront aussi dans le registre *DMACON*. Si le bit 15 est à 0, tous les bits activés du registre *DMACON* seront effacés. Les autres bits de ce registre ne sont pas influencés.

Le bit 9, *DMAEN*, est utilisé comme interrupteur général. S'il est à 0, tous les canaux DMA sont inactifs, et ceci malgré les bits 0 à 8. Un canal DMA est sélectionné seulement si le canal correspondant et le bit *DMAEN* sont activés.

Exemple :

Le DMA bitplane est activé (*BPLEN=1*), mais sans bit *DMAEN*. La valeur du registre *DMACON* est alors $0100. Le DMA disque est alors sélectionné. *DSKEN* et *DMAEN* sont alors activés et *BPLEN* est effacé.


```
MOVE.W #$0100,$DFF096   ; bit BPLEN est effacé (SET/CLR = 0)
MOVE.W #$0210,$DFF096   ; DSKEN et DMAEN sont activés (SET/CLR = 1)
```

Le registre *DMACON* contient alors la valeur voulue *$0210*.

Les bits 13 et 14 ne peuvent qu'être lus. Ils donnent des renseignements sur l'état du *Blitter* (plus de précision au chapitre sur le *Blitter*).

Le bit 10 gère la priorité du *Blitter* sur le processeur. S'il est activé, le *Blitter* a la priorité absolue sur le processeur. Ceci signifie que ce dernier n'a aucune possibilité d'accès à un registre d'un circuit ou à la *CHIP-RAM* durant toutes les opérations du *Blitter*. Lorsqu'il est désactivé, on alloue au processeur un cycle tous les 4 cycles de bus pairs. Ceci empêche que le processeur soit arrêté trop longtemps, surtout si un accès à une routine du système d'exploitation ou à un programme de la *FAST RAM* est absolument nécessaire (structure de données du système d'exploitation ou vecteur d'exception du 68000).

La position actuelle du faisceau d'électron

Etant donné que le timing du DMA est orienté suivant la position du *Raster*, on doit savoir à quel endroit de la ligne se trouve le faisceau d'électrons. AGNUS possède pour ceci un compteur interne, qui contient la position horizontale et verticale du faisceau sur l'écran suivant laquelle le système s'aligne. Le processeur a la possibilité d'accès à ce compteur grâce à deux registres :

VHPOS $006 (lecture,VHPOSR) et $02C (écriture,VHPOSW)

Bit n°	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Fonction	V7	V6	V5	V4	V3	V2	V1	V0	H8	H7	H6	H5	H4	H3	H2	H1

VPOS $004 (lecture,VPOSR) et $02A (écriture,VPOSW)

Bit n°	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Fonction	LOF	..	..	..	..	..	..	..	..	..	..	..	..	..	..	V8

Les bits H1 à H8 reproduisent la position horizontale du faisceau correspondant directement à chaque cycle de bus (schema précédent), et possédant les coordonnées de deux ou quatre points, respectivement, en basse et en haute résolution. La valeur d'une position horizontale peut varier de $0 à $E3 (0 à 227). Le temps mort horizontal correspond à la zone $F-$35.

Les bits caractérisant la position verticale, soit la ligne actuelle, sont répartis en deux registres. Les bits inférieurs V0 à V7 se trouvent dans le registre *VHPOS*, le bit V8 de plus fort poids se trouvant dans le registre *VPOS*. Ensemble, ils donnent le numéro de la ligne actuelle sur l'écran.

Les lignes de 0 à 312 sont possibles. Le temps mort vertical atteindra alors la ligne 25.

Le bit *LOF* (long frame) indique si le bit reproduit est du type *LONG FRAME* (demi-image composée des lignes impaires) ou du type *SHORT FRAME* (demi-image composée des lignes paires). Ce bit n'est nécessaire qu'en mode *interlace*, en temps normal, celui-ci est à 1.

Le registre *POS* est utilisé par le crayon lumineux (*lightpen*). Il retient la position de ce dernier, lorsque l'entrée *lightpen* d'AGNUS est activée et lorsque le crayon est maintenu contre l'écran. Son contenu sera bloqué tant qu'un faisceau d'électrons ne sera pas passé devant la pointe du crayon.

A la fin du temps mort vertical, c'est-à-dire à la ligne 26, le compteur sera débloqué. Si on veut lire la position du crayon, on doit suivre les indications suivantes :

- Attendre à la ligne 0 (début du temps mort vertical). Ceci peut se faire au moyen d'une interruption verticale *blanking* (cf. chapitre suivant).

- Lire les deux registres compteurs.

Si la position verticale se trouve entre 0 et 25, aucun signal *lightpen* ne sera réceptionné. Si la valeur est en dehors de ce temps mort, la position du crayon est reproduite.

Pour finir ce chapitre voici quelques détails sur les cycles de rafraîchissement.

Agnus possède un compteur 8 bits de rafraîchissement intégré. On peut y accéder par l'adresse registre $28 (attention ! le contenu mémoire peut être perdu). Au début de chaque ligne, Agnus procède à un rafraîchissement du contenu mémoire des lignes toutes les 4 millisecondes, en ravivant 4 adresses sur le bus d'adresse CHIP-RAM.

Pendant que l'adresse de la ligne sera émise sur le bus d'adresse *CHIP-RAM*, Agnus met l'adresse du registre *strobe* déterminé sur le bus d'adresse registre. Ce signal *strobe* sert à communiquer aux autres circuits, DENISE et PAULA, le moment où une ligne ou une image débute. Ceci est nécessaire, étant donné que le compteur de position à l'écran se trouve dans Agnus et qu'il n'existe aucun signal de transfert de synchronisation sur les autres circuits. Il existe 4 adresses *strobe* différentes.

Adr.	Circuit	Fonction
$38	D	Temps mort vertical d'une short frame
$3A	D	Temps mort vertical
$3C	D P	Cette adresse strobe est générée à chaque ligne du RASTER en dehors du temps mort vertical
$3E	D	Indicateur d'une longue ligne Raster (228 cycles)

Pendant le premier cycle de rafraîchissement, on accède toujours à l'une des trois adresses *strobe* ci-dessus. En temps normal, ce sera $3C, qui se trouve à l'intérieur du temps mort $38 ou $3A, qu'il s'agisse d'une *short* ou *long frame*.

La quatrième adresse nous indique un nouveau caractère du rafraîchissement. En effet une ligne du *Raster* a une longueur calculée de 227.5 cycles de bus. Comme on ne peut découper un cycle en deux, les lignes alternent entre 227 et 228 cycles de bus. L'adresse *strobe* $3E signale les 228 cycles d'une ligne et sera générée pendant les deux cycles de rafraîchissement.

1.5.3 LES INTERRUPTIONS

Toutes les unités d'entrée/sortie ainsi que les deux CIA ont la possibilité de libérer une interruption. Un circuit spécial, à l'intérieur de PAULA, prend en charge la gestion de chaque source d'interruption et engendre ainsi les signaux d'interruption du 68000. Le vecteur d'interruption du processeur sera alors utilisé, ces dernières pouvant être d'un niveau 0 à 6. L'interruption non masquable (*NMI*), de niveau 7, n'est pas prévue. Les deux registres correspondent, l'un au registre demande d'interruption (*INTREQ interrupt-request*), et l'autre au registre masque d'interruption (*INTENA interrupt-enable*). La répartition des bits est la même dans les deux registres.

Voici la répartition des bits des registres *interrupt-request* et *interrupt-enable* :

Adresse registre : INTREQ = $09C (écriture)
INTREQR = $01E (lecture)
INTENA = $09A (écriture)
INTENAR = $01C (lecture)

Bit	Nom	Niveau	Fonction
15	SET/CLR		Ecriture/lecture (cf. registre DMACON)
14	INTEN	(6)	Interruption autorisée
13	EXTER	6	Interruption du CIA-B ou port d'extension
12	DSKSYN	5	Valeur connue de synchronisation disque
11	RBF	5	Buffer d'entrée du port série est plein
10	AUD3	4	Données audio du canal 3 émises
9	AUD2	4	Données audio du canal 2 émises
8	AUD1	4	Données audio du canal 1 émises
7	AUD0	4	Données audio du canal 0 émises
6	BLIT	3	Blitter terminé
5	VERTB	3	Début du temps mort
4	COPER	3	Réservé aux interruptions du COPPER
3	PORTS	2	Interruption du CIA-A ou port d'extension
2	SOFT	1	Réservé aux interruptions du software
1	DSKBLK	1	Transfert DMA disque terminé
0	TBE	1	Buffer de sortie du port d'extension vide

Les 13 bits inférieurs caractérisent toutes les sources d'interruption. Les interruptions CIA sont rassemblées sous une seule interruption. Les bits du registre *DMAREQ* nous informent, dans ce cas, de quel type d'interruption il s'agit. Pour libérer une interruption processeur, il faut que les bits correspondants du registre *DMAENA* soit activés en même temps que le bit *INTEN*. Ce dernier agit comme un interrupteur général pour les 14 autres sources d'interruption, celles-ci pouvant être dissociées de chaque bit du registre *INTENA*. Les interruptions ne peuvent être libérées que lorsque le bit *INTEN* est à 1.

Si les bits *INTEN* des registres *INTENA* et *INTREQ* sont activés ensemble, une interruption processeur est libérée. Les numéros des vecteurs d'interruption se trouvent dans la colonne niveau du tableau. Volci encore pour rappel, les adresses des 7 niveaux de vecteurs :

Vecteur n°	Adresse DEC/HEX	Niveau d'interruption
25	100/$64	vecteur niveau 1
26	104/$68	vecteur niveau 2
27	108/$6C	vecteur niveau 3
28	112/$70	vecteur niveau 4
29	116/$74	vecteur niveau 5
30	120/$78	vecteur niveau 6
(31	124/$7C	vecteur niveau 7)

Les interruptions qui requièrent un traitement rapide sont de haut niveau.

Pour modifier les bits des deux registres, on doit accéder au registre *DMACON* et travailler avec le bit *SET/CLR*.

Après le traitement d'une interruption, les bits libérés du registre *INTREQ* doivent être remis par le processeur. Au contraire, ces mêmes bits ne sont pas désactivés automatiquement par lecture du registre de contrôle des interruptions du CIA.

Lorsqu'on active un bit du registre *INTREQ* au moy de l'instruction *MOVE*, le résultat est le même que si l'interruption correspondante avait été déclenchée. C'est de cette façon qu'on génère une interruption par le software (*SOFT*, bit 2). Même le *COPPER* ne peut qu'engendrer des interruptions par écriture sur ce registre.

Le bit 14 du registre *INTREQ* est particulier, en ceci qu'il n'a aucune fonction particulière comme dans le registre *INTENA*. Mais lorsqu'on l'active par écriture dans le registre *INTREQ* et que le bit *INTEN* du registre *INTENA* est à l'état *HIGH*, une interruption de niveau 6 est engendrée.

A chaque interruption du CIA-A, le bit 3 du registre *DMAREQ* est activé (bit 13 pour le CIA-B). La source d'interruption du CIA correspondant est communiquée par lecture du registre de contrôle des interruptions du CIA.

Les interruptions 3 et 13 peuvent être libérées par des cartes d'extension se trouvant sur le port.

Le bit n°5 correspond à l'interruption *transparent vertical*. Celle-ci se déclenche au début de chaque demi-image, lors du temps mort vertical (ligne 0) et ceci 50 fois par seconde.

Les autres types d'interruptions seront étudiés dans les chapitres correspondants.

1.5.4 LE COPROCESSEUR COPPER

Le *COPPER* est un simple coprocesseur dont le rôle est la surveillance des différents registres des circuits spécialisés et l'écriture de leur contenu avec des valeurs déterminées. Le *COPPER* a la possibilité, par exemple, de modifier le contenu de registres à n'importe quelle position du *raster* de l'écran. Il peut aussi diviser ce dernier en multiples zones de résolution et de couleur différentes. Cette possibilité est employée lorsque plusieurs écrans sont utilisés. Le *COPPER* est décrit comme un coprocesseur, puisqu'il dispose d'un programme qui se trouve en mémoire, où des instructions y sont traitées, comme cela se passe pour un vrai processeur.

Cependant le *COPP* ne reconnaît que 3 instructions différentes avec lesquelles on peut faire quantité de choses.

MOVE

L'instruction *MOVE* écrit une valeur immédiate dans n'importe quel registre d'un circuit spécialisé.

WAIT

L'instruction *WAIT* permet d'attendre que le faisceau d'électrons atteigne une position déterminée à l'écran.

SKIP

L'instruction *SKIP* permet de sauter l'instruction suivante, lorsque le faisceau d'électrons a atteint une position déterminée à l'écran. Avec cette instruction, on peut programmer des branchements particuliers.

Les programmes du *COPPER* sont appelés *listes du COPPER*. Les instructions s'y suivent les unes après les autres et se décomposent toujours en 2 termes.

Exemples :

```
WAIT (x1,y1) ; attend jusqu'à ce que la position écran x1,y1 soit atteinte.
MOVE #0,$180 ; écrit la valeur 0 dans le registre couleur d'arrière plan.
MOVE #1,$181 ; écrit la valeur 1 dans le registre couleur 1.
WAIT (x2,y2) ; attend jusqu'à ce que la position écran x2,y2 soit atteinte.

                    etc ...
```

La *liste COPPER* ne suffit pas seulement. En effet, il existe quelques registres importants, qui contiennent des paramètres nécessaires au *COPPER*.

Les registres du COPPER

Adresse	Nom	Fonction
$080	COP1LCH	Ces deux registres contiennent à la suite
$082	COP1LCL	l'adresse 18 bits de la première liste du COPPER
$084	COP2LCH	Ces deux registres contiennent à la suite
$086	COP2LCL	l'adresse 18 bits de la deuxième liste du COPPER
$088	COPJMP1	Charge l'adresse de la première liste dans le compteur du COPPER
$08A	COPJMP2	Charge l'adresse de la deuxième liste dans le compteur du COPPER
$02E	COPCON	Ce registre ne contient qu'un bit (BIT 0). Si ce dernier est activé, le COPPER peut accéder aux adresses de registre de $040 à $7E (registres appartenant au Blitter).

Seul l'accès écriture est permis sur les registres du *COPPER*.

Les deux registres *COPxLC* renferment chacun l'adresse d'une liste du *COPPER*. Etant donné que cette dernière est du type 18 bits, deux registres sont nécessaires. L'accès se fait avec l'instruction *MOVE.L*. La liste du *COPPER* doit se trouver, comme toutes les données propres aux circuits spécialisés, dans les 512 Ko de la *CHIP-RAM*.

Le *COPPER* utilise un compteur programme interne comme pointeur de l'instruction actuelle. Il traite les données sous forme de deux termes ou mots. Afin que le *COPPER* puisse débuter à une adresse déterminée, l'adresse de départ de la liste doit être transférée dans le compteur programme. C'est l'utilisation réservée aux deux registres *COPJMPx*. Ils mettent en place des registres *strobe*, c'est-à-dire une valeur, qui ne sera pas écrite dans un registre, mais qui servira exclusivement à libérer une action particulière. Cette valeur est constante et ne dépend que de l'accès d'un tel registre. Ces deux registres servent donc, dans le *COPPER*, a transférer le contenu des registres correspondants *COPxLC* dans le compteur programme.

Si on écrit dans registre *COPJMP1*, l'adresse contenue dans *COP1LC* sera transférée dans le compteur. En conséquence, l'exécution du programme *COPPER* pourra continuer. Le processus est le même pour les registres *COPJMP2* et *COP2LC*.

Au début du temps mort vertical, à la ligne 0, le compteur programme sera chargé automatiquement avec la valeur de *COP1LC*, permettant au *COPPER* d'exécuter le même programme à chaque image.

Structure des instructions

	MOVE		WAIT		SKIP	
Bit	BW1	BW2	BW1	BW2	BW1	BW2
15	x	DW15	VP7	BFD	VP7	BFD
14	x	DW14	VP6	VM6	VP6	VM6
13	x	DW13	VP5	VM5	VP5	VM5
12	x	DW12	VP4	VM4	VP4	VM4
11	x	DW11	VP3	VM3	VP3	VM3
10	x	DW10	VP2	VM2	VP2	VM2
9	x	DW9	VP1	VM1	VP1	VM1
8	RA8	DW8	VP0	VM0	VP0	VM0
7	RA7	DW7	HP8	HM8	HP8	HM8
6	RA6	DW6	HP7	HM7	HP7	HM7
5	RA5	DW5	HP6	HM6	HP6	HM6
4	RA4	DW4	HP5	HM5	HP5	HM5
3	RA3	DW3	HP4	HM4	HP4	HM4
2	RA2	DW2	HP3	HM3	HP3	HM3
1	RA1	DW1	HP2	HM2	HP2	HM2
0	0	DW0	1	0	1	1

Légende :

x : Ce bit est inutilisé. Il doit être initialisé avec 0
RA : Adresse registre
DW : Mot de donnée

VP : Position verticale du faisceau d'éle ns
VM : Bit masque vertical
HP : Position horizontale du faisceau d'électrons
HM : Bit masque horizontal
BFD : Blitter Finish Disable

L'instruction MOVE

L'instruction *MOVE* est caractérisée par la mise à 0 du bit 0 du premier mot. Au moyen de cette instruction, il est possible d'écrire une valeur immédiate dans un registre d'un circuit spécialisé. L'adresse registre de ce dernier est représentée par les 9 premiers bits du premier mot. C'est pour cette raison que le bit 0 doit rester à 0. Le deuxième mot de l'instruction contient l'octet de donnée, qui sera écrit dans le registre de même nom.

Il existe plusieurs restrictions sur l'adresse registre. En temps normal, le *Blitter* n'influence pas les registres se trouvant dans la zone $000-$07F. Si on initialise le bit 0 du registre *COPCON*, le *COPPER* a la possibilité d'accéder aux registres se trouvant dans la zone d'adresse de $040 à $07F. Le *COPPER* pourra ainsi utiliser le *Blitter*. Un accès aux registres inférieurs est de toute évidence interdit ($000 à $03F).

L'instruction WAIT

L'instruction *WAIT* est caractérisée par la mise à 1 du bit 0 du premier mot et la mise à 0 du bit 0 du deuxième mot. Elle oblige le *Blitter* à attendre l'exécution de la prochaine instruction, jusqu'à ce que la position du faisceau d'électrons souhaitée soit atteinte. Si cette dernière est déjà dépassée alors que l'instruction apparaît, le *COPPER* sautera de suite à la prochaine instruction.

Cette position est déterminée suivant les lignes verticales et les colonnes horizontales. La résolution verticale correspond à une ligne du *Raster*. Etant donné qu'il n'est prévu que 8 bits pour la position verticale et qu'il existe 313 lignes, l'instruction *WAIT* ne peut différencier les 256 premières lignes des 57 restantes.

Si on veut accèd à une de ces lignes, on doit s'aider de deux instructions WAIT.

1) *WAIT* sur la ligne 255.

2) *WAIT* sur la ligne souhaitée, tout en négligant le bit n°9.

Les positions horizontales sont au nombre de 112, étant donné que les deux bits inférieurs, *HP0* et *HP1*, ne peuvent être utilisés. Le mot de l'instruction *MOVE* ne contient que les bits *HP2* à *HP8*, c'est-à-dire que les coordonnées horizontales d'une instruction *WAIT* correspondent à un point sur 4 en basse résolution.

Le deuxième terme renferme le masque bit. On peut, grâce à lui, déterminer la position horizontale et verticale du bit amené à être comparé avec le faisceau d'électrons. Seuls les bits de positions, c'est-à-dire activés dans le masque de bits, seront pris en compte. Ceci permet de nombreuses possibilités :

```
WAIT position verticale $0F et masque vertical $0F
```

indique que toutes les 16 lignes, la condition *WAIT* sera exécutée, toujours lorsque les 4 derniers bits sont à 1, étant donné que les bits 4 à 6 ne rentrent plus dans la comparaison (masque bits 4 à 6 sont à 0). Le 7ème bit de la position verticale ne se laisse pas masquer. Pour cette raison, l'exemple précédent ne fonctionne que dans la zone des lignes 0 à 127 et dans la zone des lignes 256 à 313.

Le bit *BFD (Blitter Finish Disable)* a la fonction suivante : si le *COPPER* veut utiliser une opération *Blitter*, il doit savoir si ce dernier a fini son travail. Si le bit *BFD* est désactivé, le *COPPER* attend à chaque instruction *WAIT* que le *Blitter* ait terminé son opération en cours, puis examine les autres instructions *WAIT*.

On peut l'éviter en activant le bit *BFD*, le *COPPER* ignorant alors le statut actuel du *Blitter*. Si on désire que le *COPPER* n'influence pas les registres du *Blitter*, on met ce bit à 1.

L'instruction SKIP

L'instruction *SKIP* a la même structure que l'instruction *WAIT*. Seul le bit 0 du deuxième terme est activé, différenciant les deux instructions. Il teste si la position du faisceau d'électrons est plus grande ou égale à celle se trouvant dans le terme de l'instruction. Si cette comparaison est positive, le *COPPER* saute la prochaine instruction. Sinon, il continuera le traitement du programme avec l'instruction suivante. Cette instruction permet donc la création de branchements. L'instruction suivante peut être un accès à un registre *COMJMP*, par lequel un saut sera libéré.

La structure d'une liste du COPPER

Une *liste du COPPER* est composée d'une suite d'instructions *MOVE*, *WAIT* et *SKIP*. Son adresse de départ se trouve dans le registre *COPLC1*. Pour arrêter une liste, la dernière instruction doit être suivie d'une instruction *WAIT*, avec comme paramètre, une position du faisceau impossible. Le traitement de la liste se terminera jusqu'à ce qu'une image charge à nouveau l'adresse du registre *COPLC1* dans le compteur programme du *COPPER*. *WAIT ($0,$FE)* est un exemple d'arrêt de liste, étant donné qu'une position horizontale supérieure à $E4 n'est pas possible.

L'interruption COPPER

Comme cela a déjà été précisé, il existe un bit dans le registre d'interruption, réservé à l'interruption du *COPPER*. Celle-ci peut être libérée par une instruction *MOVE* dans le registre *INTREQ* :

```
MOVE #$8010,INTREQ      ;SET/CLR et COPPER activé
```

On peut modifier tous les autres bits de ce registre de la même manière, mais le bit 4 a été prévu spécialement pour le *COPPER*.

Une interruption du *COPPER* peut servir à communiquer au processeur, qu'une position image déterminée est atteinte. Ce type d'interruption *RASTER* peut donc être facilement programmée.

Le DMA COPPER

Le *COPPER* prend ses instructions de la mémoire via un canal DMA. Il occupe les cycles pairs et est prioritaire sur le *BLITTER* et le 68000. Chaque instruction demande deux cycles, étant donné qu'une instruction est composée de deux termes. *WAIT* nécessite un cycle de plus pour attendre que la position souhaitée soit atteinte par le faisceau. Pendant cette phase d'attente, le *COPPER* libère le bus.

Le bit *COPEN* du registre *DMACON* autorise les accès *DMA COPPER*. Si ce bit est désactivé, le *COPPER* libère le bus et n'exécute plus d'instructions. Si on active ce bit, l'exécution du programme débute à l'adresse se trouvant dans le compteur programme. Il est d'ailleurs absolument nécessaire, avant l'accès *DMA COPPER*, de s'occuper de cette adresse, source d'erreur possible. L'exécution d'une zone mémoire quelconque par le *COPPER* peut planter le système. La séquence d'initialisation du *COPPER* est la suivante :

```
LEA $DFF00,A5                   ; adresse de base du registre dans A5
MOVE.W #$0080,DMACON(A5)        ; DMA COPPER désactivé
MOVE.L #COPPERLIST,COP1LCH(A5)  ; adresse de la liste COPPER activé
CLR.W COPJMP1(A5)               ; cette adresse est transférée dans le
                                ; compteur du COPPER
MOVE.W #$8080,DMACON(A5)        ; DMA COPPER activé
```

Programme d'exemple

En conclusion, voici un programme d'exemple. Il affiche des bandes de couleur à l'aide des instructions *WAIT* et *MOVE*. Ce programme a été écrit avec l'assembleur PROFIMAT pour Amiga.

```
;*** exemple d'utilisation d'une liste de COPPER ***

;registres circuits spécialisés

INTENA = $9A    ; registre interrupt enable (écriture)
DMACON = $96    ; registre de contrôle DMA (écriture)
COLOR00 = $180 ; registre de couleur 0
```

```
;registre COPPER

COP1LC  = $80   ; adresse de la 1ère liste
COP2LC  = $84   ; adresse de la 2ème liste
COPJMP1 = $88   ; saut à la liste 1
COPJMP2 = $8A   ; saut la liste 2

;CIA-A registre port A (utilisé pour le bouton gauche de la souris)

CIAAPRA = $BFE001

;Exec library base offsets

OpenLibrary = -30-522 ; nom de la librairie, version / a1, d0
Forbid      = -30-102 ; suppression du multitasking
Permit      = -30-108 ; autorisation du multitasking
AllocMem    = -30-168 ; nombre d'octets, spécifications / d0, d1
FreeMem     = -30-180 ; pointeur mémoire, nombre d'octets / a1, d0

;graphicbase

StartList = 38

;autres labels

Execbase = 4
Chip     = 2  ; demande de CHIP-RAM (cf. AllocMem)

;initialisations

Start :

    move.l Execbase,a6
    moveq #CLsize,d0    ; demande d'allocation de CLsize octets
    moveq #Chip,d1      ; CHIP-RAM sollicitée
    jsr AllocMem(a6)

    move.l d0,CLadr     ; adresse de la CHIP-RAM récupérée par AllocMem
    beq.s fin           ; erreur d'allocation  -> fin !
```



```
;charger la liste :    OPPER à l'adresse CLadr

        lea CLstart,a0          ; adresse de la liste (source)
        move.l CLadr,a1         ; adresse de la liste (destination)
        moveq #CLsize-1,d0      ; taille de la liste - 1
CLcopy :                        ; boucle de chargement
        move.b (a0)+,(a1)+      ; transfert par octet
        dbf d0,CLcopy           ; fin de boucle

;*** programme principal ***

        jsr Forbid(a6)          ; Task switching bloqué
        lea $DFF000,a5          ; adresse de base dans a5
        move.w #$03A0,DMACON(a5) ; DMA bloqué
        move.l CLadr,COP1LC(a5) ; adresse de la liste COPPER dans COP1LC
        clr.w COPJMP1(a5)       ; chargement dans le compteur programme
                                ; du COPPER

;DMA COPPER activé

        move.w #$8280,DMACON(a5)

;attente bouton gauche souris enfoncé

Wait :
        btst #6,CIAAPRA     ; bit testé
        bne.s Wait          ; non actif ? alors attendre

;*** fin du programme ***

;liste du COPPER à nouveau activée

        move.l #GRname,a1               ; nom de la librairie dans a1
        clr.l d0                        ; version 0 (la dernière)
        jsr OpenLibrary(a6)             ; librairie graphique ouverte
        move.l d0,a4                    ; adresse de graphicbase dans a4
        move.l StartList(a4),COP1LC(a5) ; chargement de l'adresse de
                                        ; StartList
        clr.w  COPJMP1(a5)
```

```
        move.w #$83E0,DMACON(a5)        ; activation des   aux DMA
                                        ; nécessaires
        jsr Permit(a6)                  ; Task Switching autorisé

;mémoire libre pour liste du COPPER

        move.l CLadr,a1   ; libérer CLsize octets à partir de l'adresse CLadr
        moveq #CLsize,d0
        jsr FreeMem(a6)   ; mémoire à nouveau libre
fin :
        clr.l d0          ; drapeau d'erreur désactivé
        rts               ; quitter le programme

;************************
;       Données
;************************

;variable

CLadr :
        DC.L 0

;constante

GRname :
        DC.B "graphics.library",0
        EVEN
;liste du COPPER

CLstart :
        DC.W COLOR00,$000F
        DC.W $780F,$FFFE
        DC.W COLOR00,$00F0
        DC.W $D70F,$FFFE
        DC.W COLOR00,$0F00
        DC.W $FFFF,$FFFE
CLend :
CLsize = CLend - CLstart

        END
```


Ce programme insta la *liste du COPPER* et attend jusqu'à ce que le bouton gauche de la souris soit pressé. Il n'est pas évident, sur l'Amiga, de quitter un programme sans un *reset*.

Premièrement, il est nécessaire d'avoir de la mémoire. On y place les *listes du COPPER*, toutes les données ayant trait aux circuits spécialisés qui doivent se trouver dans la *CHIP-RAM*. Si on n'est pas sûr que le programme s'y trouve, il est important d'y copier les listes.

Lorsqu'on possède un système d'exploitation multi-tâches, on ne peut pas écrire n'importe où dans la mémoire. Il faut d'abord la solliciter, ce qui sera fait au moyen de la routine *AllocMem*. Celle-ci met dans d0 l'adresse de la zone *CHIP-RAM* sollicitée, zone où la *liste du COPPER* sera copiée.

Puis l'appel de *'Forbid'* désactive le *Task-Switching*, c'est-à-dire que l'Amiga ne traite plus que notre programme. Cette précaution empêche tout autre programme de perturber le bon déroulement du programme présent.

Ensuite, le *COPPER* est initialisé et démarré.

Le programme teste alors le bouton gauche de la souris, en questionnant le bit de port correspondant du CIA-A.

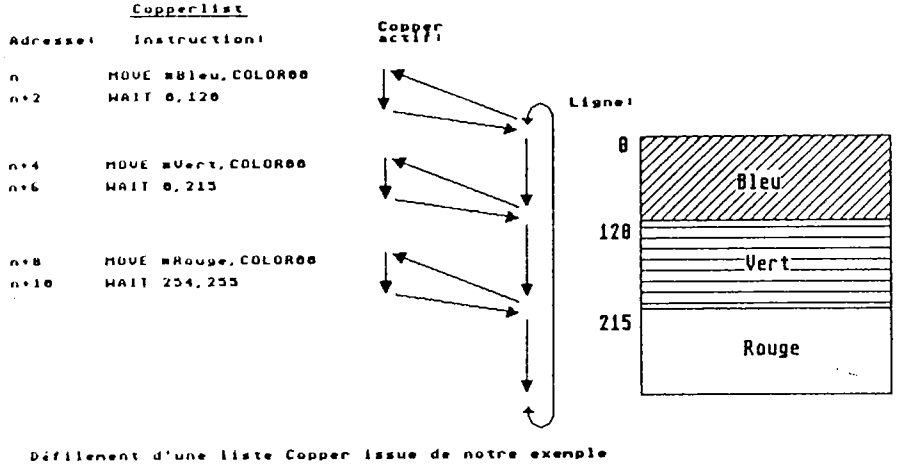
Si le bouton est enfoncé, le processeur quitte la boucle d'attente.

Pour revenir à nouveau à l'état de départ, une liste spéciale sera chargée et démarrée dans le *COPPER*. Cette liste se nomme *STARTUP-COPPERLIST* et initialise l'écran. Son adresse se trouve dans une zone de variables du système d'exploitation réservée aux fonctions graphiques. En fin de programme, le *Task-Switching* sera à nouveau activé au moyen de *'Permit'* et la mémoire occupée sera vidée avec *'FreeMem'*.

Ce programme possède un grand nombre de fonctions peu évidentes du système d'exploitation, qui se laissent facilement éluder, quand le programme tourne régulièrement. Ce qu'il faut bien comprendre, c'est la partie concernant le *COPPER*. Les routines du système seront quant à elles expliquées dans les prochains chapitres.

Tapez ce programme et expérimentez-le avec la [illegible] *du COPPER*. Modifiez l'instruction *WAIT*, ou joignez-y une nouvelle à votre gré, et si le coeur vous en dit, essayez d'y rajouter une instruction *SKIP*.

Encore une précision sur la liste : les deux instructions *WAIT* renferment *$E* comme position horizontale. Ceci correspond au début du temps mort horizontal, le *COPPER* l'utilisant pour modifier la couleur en dehors de la zone visible. Si on met 0 pour la position horizontale, la commutation des couleurs sera visible sur la bordure droite de l'écran.



Défilement d'une liste Copper issue de notre exemple

Figure 1.5.4

1.5.5 PLAYFIELDS

La sortie écran de l'Amiga possède deux éléments de base : les *SPRITES* (lutins) et les *PLAYFIELDS* (champs de jeu). Ce chapitre concerne essentiellement la structure et la programmation de toutes les possibilités des playfields. Les sprites seront étudiés dans le chapitre suivant.

Le playfield est l'él[illegible]ent de base d'une représentation à l'écran. Il peut comprendre au mi[illegible]num 1 et au maximum 6 bitplanes (la structure du bitplane est expliqué au chapitre 1.5.2). Un playfield reproduit donc une image graphique, constituée d'un nombre variable de zone mémoire ou bitplane. L'Amiga présente donc un nombre important de possibilités différentes de playfields :

- entre 2 et 4096 couleurs à l'écran.
- résolution allant de 16 sur 1 à 704 sur 625 points.
- possibilité de deux playfields entièrement indépendant l'un de l'autre.
- scrolling libre dans les deux directions (Smooth-scrolling).

Toutes ces possibilités peuvent se répartir en deux groupes.

1) La combinaison des bitplanes pour le calcul de chaque points d'image (la reproduction des modèles de bit des bitplanes de l'image).

2) Détermination de la forme, de la taille et de la position du ou des playfields (structure du playfield).

Les différentes possibilités de reproduction

Suivant l'utilisation d'un ou de plusieurs bitplanes, chaque point sera représenté par plus ou moins de bits. Cette valeur sera modifiée par la possibilité des 4096 couleurs, chaque pixel de l'image n'ayant qu'une couleur propre.

L'Amiga génère ses couleurs en mixant trois couleurs de base, le rouge, le vert et le bleu. Chacune de ces trois composantes pouvant prendre 16 degrés d'intensité différents, on obtient 4096 nuances différentes (16*16*16=4096). Chaque mémorisation de couleur nécessite 4 bits par composante, autrement dit, un total de 12 bits par nuance.

Si on veut attribuer une des 4096 couleurs à chaque point, ' ›its sont nécessaires. Etant donné que seuls 6 bits sont possibles, il est impératif de les convertir afin que le point visible puisse renfermer une couleur.

La palette couleur

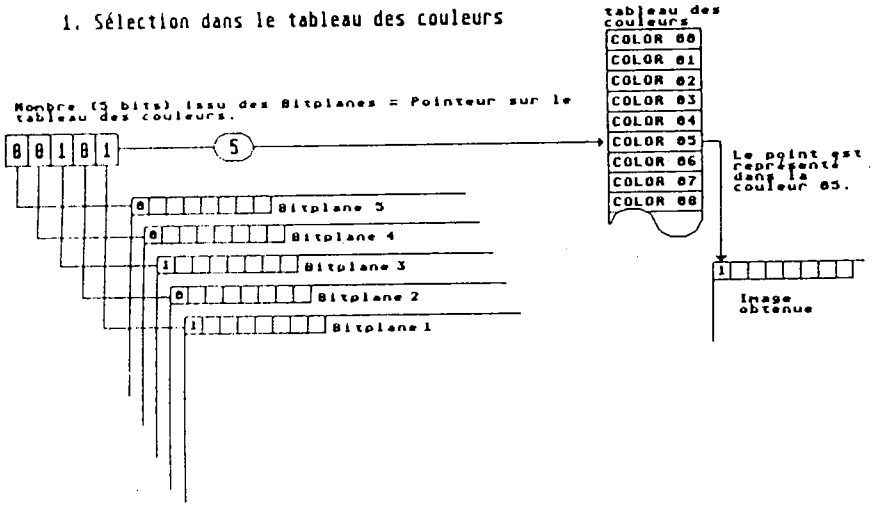


Figure 1.5.5.1

On emploie ainsi une palette ou tableau des couleurs. Celle-ci contient, sur l'Amiga, 32 entrées qui admettent chacune une valeur couleur 12 bits. La valeur du premier registre couleur *COLOR00* correspond à la couleur de l'arrière plan ou fond et à la couleur du cadre.

Les registres couleur 0 à 31 ne sont accessibles qu'en mode écriture.

Adresse	Registre palette couleur
$180	COLOR00
$182	COLOR01
...	...
$1BE	COLOR31

Structure d'un élément du tableau :

```
Bit    : 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
COLORxx :  x  x  x  x R3 R2 R1 R0 G3 G2 G1 G0 B3 B2 B1 B0
```

R0-R3 valeur 4 bits pour la composante rouge

G0-G3 valeur 4 bits pour la composante verte

B0-B3 valeur 4 bits pour la composante bleu

Les 4 bits supérieurs indiqués par x sont inutilisés.

La valeur issue du bitplane n'est utilisée qu'en tant que pointeur. Etant donné qu'il existe 32 registres couleur, seuls 5 bitplanes peuvent être combinés dans ce mode. Le bit inférieur issu du bitplane est appelé *LSB*, le bit supérieur étant le *MSB*.

Cette façon de sélectionner les couleurs sur un tableau ne permet la présence que de 32 couleurs choisies parmi 4096 sur la même image. En mode haute résolution, seuls 4 bitplanes peuvent être activés ensemble. La limite est ici 16 couleurs. Certains registres couleur resteront alors inutilisés.

Nbre de bitplane	Nbre de couleur	Registres couleurs utilisés
1	2	COLOR00-COLOR01
2	4	COLOR00-COLOR03
3	8	COLOR00-COLOR07
4	16	COLOR00-COLOR15
5	32	COLOR00-COLOR31

Le mode Extra-Half-Bright

En mode basse résolution, 6 bitplanes peuvent être utilisés. Ceci correspond à un domaine de valeurs de 2^6 ou 0 à 63. Or, seul 32 registres couleurs sont disponibles. On emploie dans ce cas un mode spécial appelé *Extra-Half-bright.*

Les bits inférieurs (bits 0 à 4 des plans 1 à 5) servent de pointeur sur les registres couleurs. Le contenu de ces derniers est aussitôt versé à l'image si le bit 5 (du plan 6) est à 0. Si ce bit est à 1, la valeur de la couleur sera divisée par deux, avant d'être communiquée à l'image.

La division par deux correspond à un décalage vers la droite d'un bit de la valeur couleur des trois composantes. Etant donné que chaque composante sera moitié moins importante, la couleur reproduite sera à peu près identique, seule la luminosité étant affectée. La correspondance du nom anglais de ce mode est en fait : *mode particulier à luminosité réduite de moitié.*

Exemple :

```
bit n°                : 5 4 3 2 1 0
valeur des bitplanes  : 1 0 0 1 0 0
```

Ceci correspond à l'entrée palette n°4 (00100 correspond à 8 en binaire).

COLOR04 doit contenir la valeur suivante (couleur orange) :

R3	R2	R1	R0	G3	G2	G1	G0	B3	B2	B1	B0
1	1	1	0	0	1	1	0	0	0	0	0

Comme le bit 5 est à 1, cette valeur est décalée d'un bit vers la droite :

R3	R2	R1	R0	G3	G2	G1	G0	B3	B2	B1	B0
0	1	1	1	0	0	1	1	0	0	0	0

Cette valeur correspond encore à la couleur orange, mais moitié moins claire.

A travers le choix des valeurs correspondantes dans les registres couleur, il est simple de caractériser la couleur d'un point parmi les 64 possibles, en mode *Extra-Half-Bright.* Dans le registre couleur, on trouve les nuances claires et si le bit 5 est activé, on initialisera des teintes plus foncées.

Le mode Hold-And-Modify

Ce mode permet l'affichage des 4096 couleurs simultanément. Il n'est possible qu'en mode basse résolution, étant donné qu'il nécessite 6 bitplanes. Ce mode profite du fait que les changements de couleur d'une image sont toujours constants. Certains nécessitent des passages graduels de couleurs claires vers de plus foncées ou Inversement.

En mode *Hold-and-Modify,* aussi nommé *HAM,* la couleur des premiers points sera modifiée progressivement. On peut donc favoriser des dégradés de couleurs en augmentant, par exemple, la composante bleue d'un pas à chaque pixel. On est évidemment limité dans le fait qu'on ne peut accéder qu'à une composante à la fois, et modifier d'un point à l'autre soit la valeur rouge, verte ou bleue et jamais plus en même temps. Pour obtenir un passage graduel du sombre au clair, on doit modifier les trois composantes en mixant de nombreuses fois les couleurs. Ceci ne peut se faire en mode *HAM,* qu'en mettant la valeur souhaitée d'une composante à un point. Trois points sont alors nécessaires.

On a toujours le compromis de modifier directement la couleur d'un pixel, en pointant une des 16 couleurs de la palette.

Comment interpréter la valeur issue des bitplanes en mode HAM ?

Les deux bits supérieurs (bits 4 et 5 des bitplanes 5 et 6) déterminent l'utilisation des 4 bits inférieurs (bitplanes 1 à 4). Si les bits 4 et 5 sont à 0, les 4 bits restants seront employés comme pointeurs sur la palette couleur, 16 couleurs pouvant être directement choisies. Si lors d'une combinaison des bits 4 et 5, le résultat diffère de 0, la valeur couleur du dernier point sera prise (à gauche du pixel actuel), deux des composantes restants inchangées, la troisième étant modifiée par les 4 bits inférieurs. Le choix entre les trois composantes dépend exclusivement des deux bits supérieurs.

Cette explication peut sembler compliquée. Voici donc un tableau qui explique l'utilisation des différentes combinaisons de bits.

Bit N° :

5	4	3	2	1	0	**Fonction**
0	0	C3	C2	C1	C0	Les bits C0 à C3 seront utilisés comme pointeur d'un registre couleur dans la zone COLOR00 à COLOR15, d'une manière identique à celle d'un choix de couleur normal.
0	1	B3	B2	B1	B0	Les composantes rouge et verte du dernier pixel restent inchangées. La valeur bleue précédente sera modifiée par la nouvelle valeur B3-B0.

1	0	R3	R2	R1	R0	Les composantes bleue et verte du dernier pixel restent inchangées. La valeur rouge précédente sera modifiée par la nouvelle valeur R3-R0.
1	1	G3	G2	G1	G0	Les composantes rouge et bleue du dernier pixel restent inchangées. La valeur verte précédente sera modifiée par la nouvelle valeur G3-G0.

Pour le premier point d'une ligne, la couleur cadre (COLOR00) sera utilisée comme couleur des premiers points.

Le mode Dual-Playfield

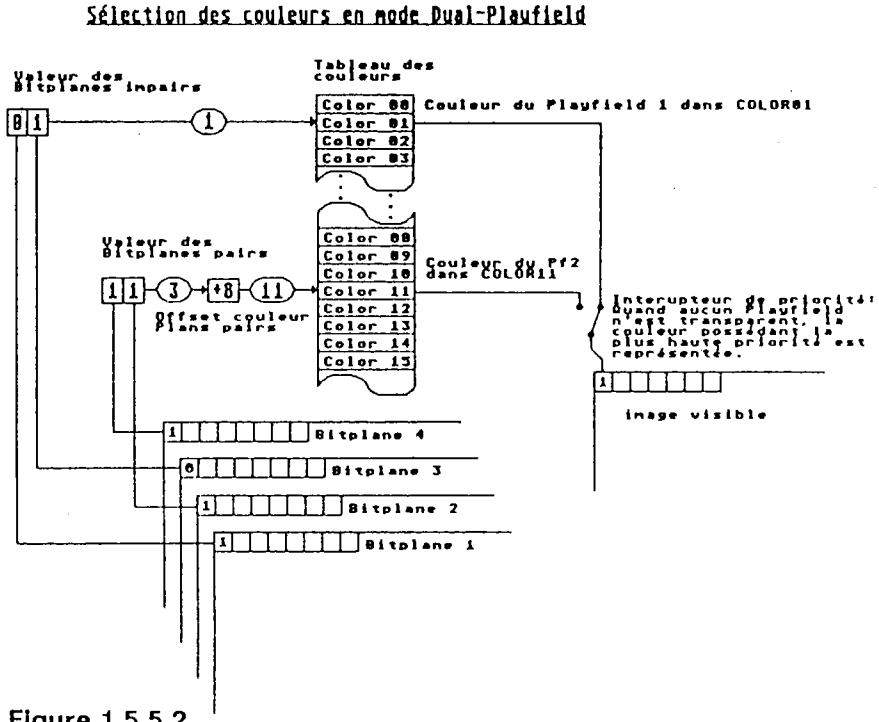


Figure 1.5.5.2

Les modes décrits jusqu'ici n'utilisaient, exclusivement, qu'un seul playfield. Le mode *dual-playfield* permet la reproduction de deux playfields, entièrement indépendants l'un de l'autre. Il en résulte l'existence de deux images sur le même moniteur et en même temps. Celles-ci pourront être utilisées facilement, rapidement et séparément.

Ceci peut être intéressant, notamment pour les jeux où, par exemple, un effet jumelle peut être facilement créé. Le playfield, en avant plan, sera de couleur noire, et s'il reste un espace au milieu, une partie du playfield formant l'arrière plan pourra être observable. Chacun des deux playfields est reproduit avec la moitié des bitplanes activés.

Le playfield n°1 sera formé des bitplanes impairs (*odd planes*), le playfield n°2 étant représenté par les bitplanes pairs (*even planes*). Si un nombre impair de bitplanes est exploité, le playfield 1 aura à sa disposition un bitplane de plus.

La sélection des couleurs se fait de la manière suivante :

La valeur des bits d'un point, des plans impairs (playfield 1) ou des plans pairs (playfield 2), correspond au pointeur de registre couleur. Etant donné que chaque playfield comprend au maximum 3 bitplanes, seules 8 couleurs sont disponibles. Ces dernières seront accessibles, pour le champ 1, aux huit premières entrées de la palette (COLOR00-COLOR07). Un offset de 8 sera rajouté à la valeur issue des bitplanes du playfield 2, étant donné que les registres couleur accessibles se trouvent dans les positions 8 à 15.

Si un point a la valeur 0, sa couleur ne sera pas issue du registre COLOR00 (ou COLOR08), mais sera représentée avec la couleur transparente. Ceci signifie que les éléments de l'arrière plan seront directement visibles. Ces derniers pourront être soit des sprites, soit la couleur (COLOR00) de l'arrière plan.

Le mode *Dual-Playfield* est utilisable en mode haute résolution. Chaque champ ne possède plus alors que 4 couleurs. La répartition des registres couleur n'est pas pour autant modifiée, les 4 registres couleur supérieurs n'étant tout simplement pas utilisés (COLOR04 à 07 et COLOR12 à 15).

Répartition des bitplanes avec le mode Dual-Playfield :

Bitplanes	Plans dans le playfield 1	Plans dans le playfield 2
1	PLAN 1	aucun
2	PLAN 1	PLAN 2
3	PLAN 1 et 3	PLAN 2
4	PLAN 1 et 3	PLAN 2 et 4
5	PLAN 1,3 et 5	PLAN 2 et 4
6	PLAN 1,3 et 5	PLAN 2, 4 et 6

Sélection des couleurs avec le mode Dual-Playfield

	Playfield 1		Playfield 2	
Plans	5 3 1	Registre couleur	5 3 1	Registre couleur
	0 0 0	Transparent	0 0 0	Transparent
	0 0 1	COLOR01	0 0 1	COLOR09
	0 1 0	COLOR02	0 1 0	COLOR10
	0 1 1	COLOR03	0 1 1	COLOR11
	1 0 0	COLOR04	1 0 0	COLOR12
	1 0 1	COLOR05	1 0 1	COLOR13
	1 1 0	COLOR06	1 1 0	COLOR14
	1 1 1	COLOR07	1 1 1	COLOR15

Structure d'un playfield

Comme cela a déjà été vu, un playfield est composé d'un nombre déterminé de bitplanes. La description de ces derniers (cf. Chapitre 1.5.2) montre qu'ils sont conçus comme des zones mémoires continues où la largeur d'écran d'une ligne est représentée par un nombre de mots. Ce nombre est de 20 en plus basse résolution (320 points répartis en 20 mots de 16 points) et de 40 en plus haute (640/16). Afin d'établir la structure exacte d'un playfield, les indications suivantes sont nécessaires :

- définition de la taille souhaitée de l'image
- mise en place de la taille du bitplane
- choisir le nombre de bitplanes
- initialiser la palette couleur
- déterminer le mode (Hires, Lores, HAM, etc...)
- structurer les listes du COPPER
- initialiser le COPPER
- activer le COPPER et le DMA bitplane

Détermination de la taille de l'image

L'Amiga autorise une position libre des coins supérieur gauche et inférieur droit du playfield. La position et la taille de l'image restent donc variables. La résolution correspond verticalement à une ligne du *Raster* et horizontalement, à un pixel basse résolution. Deux registres contiennent cette valeur.

DIWSTRT (Display Window Start)
contient la position verticale et horizontale de départ de la fenêtre d'écran, c'est-à-dire la ligne et la colonne où la reproduction du playfield débute.

DIWSTOP (Display window STOP)
renferme la position de fin + 1. Ceci correspond à la première ligne et colonne après le playfield.

En dehors de la zone visible, la couleur du cadre sera effective (elle correspond à la couleur d'arrière plan et est issue du registre COLOR00).

DIWSTRT $08E (écriture seulement)

Bit n° :	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	V7	V6	V5	V4	V3	V2	V1	V0	H7	H6	H5	H4	H3	H2	H1	H0

DIWSTOP $90 (écriture seulement)

Bit n° :	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	V7	V6	V5	V4	V3	V2	V1	V0	H7	H6	H5	H4	H3	H2	H1	H0

La position de départ déterminée dans *DIWSTRT* est limitée au premier quart de l'écran, c'est-à-dire aux lignes et colonnes comprises entre 0 et 255, étant donné que les *MSB* manquants, V8 et H8, sont initialisés à 0. Le cas est le même pour la position de fin, à part qu'ici H8 est mis à 1, permettant un accès à la zone comprise entre 256 et 458. Le cas de la position de fin verticale est différent. Celle-ci peut être inférieure ou supérieure à la position 256. Pour cette raison, le *MSB* V8 est généré suivant l'inversion du bit V7. Ainsi une position de fin comprise entre les valeurs 128 et 312 sera possible.

Si celle-ci est comprise entre 256 et 312, V7 sera mis 0 et V8 à 1. Si V7 est à 1 et V8 à 0, la position sera comprise entre 128 et 255.

La fenêtre d'écran normale possède un coin supérieur gauche de position horizontale 129 et verticale 41. Le coin inférieur droit correspond à la position 448/296, c'est-à-dire que *DIWSTOP* doit être initialisé avec les coordonnées 449 et 297. Les valeurs hexadécimales correspondantes de ces deux registres sont $2981 et $29C1. Avec ces dernières, la page de l'amiga de 640 points sur 256 (idem 320 par 256) est centrée exactement au milieu de l'écran.

Pourquoi n'utilise-t-on pas toute l'étendue possible de l'écran ?

Il y a en fait plusieurs raisons. Premièrement, un moniteur normal n'exploite pas toute l'image, la zone visible débutant après un certain nombre de lignes et de colonnes, ceci correspondant aux différents temps mort. De plus un écran cathodique n'a pas de coins carrés. Si on voulait faire correspondre la taille de l'image avec celle de l'écran, une partie de l'image se perdrait dans les coins.

Une autre limite concerne les valeurs *DIWSTRT* et *DIWSTOP* à travers le temps mort. Celui-ci correspond exactement à la zone verticale des lignes 0 à 25, la partie visible de l'écran se réduisant aux lignes 26 à 312 ($1A à $138). Il en est de même avec les colonnes, l'espace 30 à 106 ($1E à $6A) n'étant pas visible du fait du temps mort. Seules sont donc possibles les positions horizontales après 107 ($6B).

Après avoir déterminé la position de la fenêtre d'écran, il faut établir le début et la fin des accès DMA bitplane. Afin qu'un point s'affiche à un moment désiré sur l'écran, les données des bitplanes doivent être lues

de façon opportune. Verticalement, ce n'est pas un problème, étant donné que le DMA écran commence et s'arrête aux mêmes lignes que celles précisées dans les registres *DIWSTRT* et *DIWSTOP*, c'est-à-dire aux limites de la fenêtre d'écran.

Horizontalement, c'est plus compliqué. Pour qu'un point soit représenté sur l'écran, l'électronique nécessite le mot actuel de chaque bitplane. Avec 6 bitplanes en basse résolution, 8 cycles de bus sont nécessaires pour lire tous les plans.

La haute résolution ne nécessite que 4 cycles (*rappel* : pendant un cycle de bus, 2 points en basse résolution ou 4 points en haute résolution seront reproduits).

De plus, le hardware a encore besoin d'un demi cycle, avant que les données ne soient affichées à l'écran. L'accès DMA doit donc commencer 8.5 cycles (17 points) avant le début de l'affichage (4.5 cycles ou 9 points en haute résolution).

Le cycle de bus du premier accès DMA bitplane d'une ligne sera mis dans le registre *DDFSTRT (Display Data Fetch Start)*, le dernier sera dans *DDFSTOP (Display Data Fetch Stop)* :

DDFSTRT $092 (écriture seulement)
DDFSTOP $094 (écriture seulement)

Bit n°	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Fonction	x	x	x	x	x	x	x	x	H8	H7	H6	H5	H4	H3	x	x

La résolution demande 8 cycles de bus en mode *Lores*, où H3 sera toujours à 0 et 4 en mode *Hires*. Dans ce dernier cas, H3 sert de bit de plus faible poids. La raison de la limitation de la résolution réside dans la distribution des accès DMA bitplane. En mode *Lores*, chaque bitplane sera lu tous les 8 cycles de bus. Pour cette raison, la valeur *DDFSTRT* doit être un entier multiple de 8 (H1 à H3=0).

C'est la même chose en mode *Hires*, mais avec, cette fois-ci, 4 cycles de bus (H1 à H2=0). La différence entre *DIWSTRT* et *DIWSTOP* doit toujours être divisible par 8, étant donné que le hardware partage les lignes en zone de 8 cycles, et ceci indépendamment de la résolution.

En mode *Hires*, les accès DMA bitplane se déroulent encore 8 cycles après *DIWSTOP*, ainsi 32 points seront toujours lus.

Les valeurs correctes se déduisent comme suit :

Estimation de DDFSTRT et de DDFSTOP en mode Lores

```
Hstart  = début horizontal de la fenêtre d'écran.
DDFSTRT = (Hstart/2 - 8.5) and $FFF8
DDFSTOP = DDFSTRT + (points par ligne/2 - 8)
```

Ceci correspond pour Hstart = $81 et 320 points par ligne :

```
DDFSTRT = ($81/2 - 8.5) and $FFF8 = $38
DDFSTOP = $38 + (320/2 - 8) = $D0
```

Estimation de DDFSTRT et de DDFSTOP en mode Hires

```
DDFSTRT = (Hstart/2 - 4.5) and $FFFC
DDFSTOP = DDFSTRT + (points par ligne/4 - 8)
```

Ceci correspond pour Hstart = $81 et 640 points par ligne :

```
DDFSTRT = ($81/2 - 4.5) and $FFFC = $3C
DDFSTOP = $3C + (640/4 - 8) = $D4
```

DDFSTRT ne doit pas être inférieur à $18. La limite maximale de *DDFSTOP* est $D8. Une valeur inférieure à $28 n'a aucun sens, étant donné que les points seront reproduits pendant le temps mort horizontal, ce qui n'est pas possible (exception faite pour le scrolling). Comme les positions *DDFSTRT* inférieures à $34 chevauchent les cycles DMA des bitplanes et des sprites, certains sprites ne sont pas représentables après *DDFSTRT*.

Décalage de la fenêtre d'écran

Si on désire décaler horizontalement la fenêtre d'écran au moyen de *DIWSTRT* et *STOP*, il peut arriver que la différence entre *DIWSTRT* et *DDFSTRT* ne soit pas exactement égale à 8.5 cycles de bus (17 points),

étant donné qu'on peut déterminer *DFFSTRT* suivar un pas de 8 cycles. A ce moment là, une partie du premier mot de données disparaitra dans la zone gauche, près de la limite de la fenêtre d'écran. Afin d'éviter cela, il y a la possibilité de décaler les données vers la droite, avant la sortie à l'écran, pour qu'elles correspondent avec le début de la fenêtre d'écran. La programmation de ce décalage sera expliquée dans le paragraphe concernant le scrolling.

Détermination des adresses Bitmap

Les valeurs dans *DDFSTRT* et *DDFSTOP* déterminent le nombre de mots de données par ligne. Pour chaque bitmap, l'adresse de départ doit être établie afin que le contrôleur DMA puisse savoir à partir de quel endroit les données de point doivent être lues. 12 registres renferment ces adresses. Deux registres (*BPLxPTH* et *BPLxPTL*) correspondent, à chaque fois, au même bitplane x. Ceux-ci peuvent être dénommés, plus simplement, *BPLxPT* (bitplane x pointer, pointeur du bitplane x).

Adresse	Nom	Fonction	
$0E0	BPL1PTH	adresse de départ	(bits 16-18)
$0E2	BPL1PTL	du bitplane 1	(bit 0-15)
$0E4	BPL2PTH	adresse de départ	(bits 16-18)
$0E6	BPL2PTL	du bitplane 2	(bit 0-15)
$0E8	BPL3PTH	adresse de départ	(bits 16-18)
$0EA	BPL3PTL	du bitplane 3	(bit 0-15)
$0EC	BPL4PTH	adresse de départ	(bits 16-18)
$0F0	BPL4PTL	du bitplane 4	(bit 0-15)
$0F2	BPL5PTH	adresse de départ	(bits 16-18)
$0F4	BPL5PTL	du bitplane 5	(bit 0-15)
$0F6	BPL6PTH	adresse de départ	(bits 16-18)
$0F8	BPL6PTL	du bitplane 6	(bit 0-15)

Le contrôleur DMA procède de la façon suivante lors de la reproduction des bitplanes : le DMA bitplane reste inactif, jusqu'à ce que la première ligne de la fenêtre d'écran soit atteinte (*DIWSTRT*).

Puis, il cherche dans *DDFSTRT* la colonne déterminée par les mots de données des différer bitplanes, tout en respectant le timing observé sur le schéma du fonctionnement du *Raster*. Il utilise *BPLxPT* comme pointeur sur les données se trouvant dans la *CHIP-RAM*. Après lecture de chaque mot, *BPLxPT* augmente d'un mot. Les mots lus passent dans le registre *BPLxDAT*. Ces derniers seront utilisés uniquement par le canal DMA. Si les 6 registres *BPLxDAT* sont occupés par les mots de données issus des bitplanes, les données sont transférées, bit à bit, vers la logique vidéo de DENISE, qui, suivant le choix du mode, sélectionne l'une des 4096 couleurs et l'affiche à l'écran.

En atteignant *DDFSTOP*, le DMA bitplane marque une pause jusqu'à *DDFSTRT* de la prochaine ligne, ce processus étant répété jusqu'à la fin de la dernière ligne de la fenêtre écran (*DIWSTOP*).

BPLxPT pointe alors sur le premier mot du bitplane. Etant donné que *BPLxPT* doit pointer à nouveau sur le premier mot du bitplane correspondant de la prochaine image, il doit être réinitialisé. C'est le rôle rapide et sans problème du *COPPER*. Une *liste du COPPER*, pour un playfield à 4 bitplanes, est du type :

```
AdrPlanexH = adresse du plan x, bits 16-18
AdrPlanexL = adresse du plan x, bits 0-15

MOVE #AdrPlane1H,BPL1PTH      ;pointeur sur le bitplane 1
MOVE #AdrPlane1L,BPL1PTL      ;initialisation
MOVE #AdrPlane2H,BPL2PTH      ;pointeur sur le bitplane 2
MOVE #AdrPlane2L,BPL2PTL      ;initialisation
MOVE #AdrPlane3H,BPL3PTH      ;pointeur sur le bitplane 3
MOVE #AdrPlane3L,BPL3PTL      ;initialisation
MOVE #AdrPlane4H,BPL4PTH      ;pointeur sur le bitplane 4
MOVE #AdrPlane4L,BPL4PTL      ;initialisation
WAIT ($FF,$FE)                ;fin de la liste COPPER (attente d'une
                              ;position impossible à l'écran)
```

La réinitialisation de *BPLxPT* est absolument nécessaire. Si on ne veut pas utiliser de listes, on doit laisser le processeur achever le travail, au moyen d'une interruption écran vide vertical.

Scrolling et grande taille des playfields

Tous les playfields étudiés jusqu'ici ont toujours eu la taille exacte de l'écran. Pourtant, il est souvent judicieux d'avoir un playfield de grande taille en mémoire, dont seule une partie est visible à l'écran, et que l'on peut décaler dans n'importe quelle direction. Ceci peut se faire sans problème sur l'Amiga. Le schéma suivant le montre, dans la direction des axes X et Y.

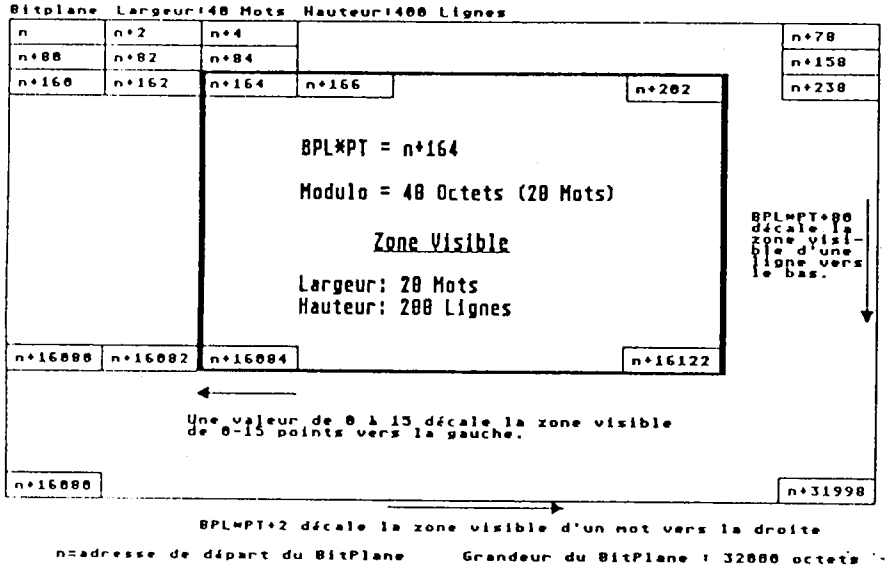


Figure 1.5.5.3

Hauteur du playfield et scrolling vertical

L'extension verticale est facilement réalisable. On initialise les bitplanes nécessaires en mémoire, mais cette fois-ci avec plus de lignes que l'écran ne peut reproduire. Par exemple, une fenêtre standard comporte 256 lignes, qu'on peut doubler pour former un playfield étendu en hauteur sur 512 lignes. Afin d'ammener ce playfield sur la fenêtre d'écran, il faut modifier la valeur de *BPLxPT*.

Si la fenêtre d'éc doit reproduire la zone s'étalant de la ligne 100 à la ligne 356, *BPLxPT* doit pointer le premier mot de la ligne 100. Une résolution de 320 points correspond à 20 mots (40 octets) par ligne. Si on multiplie par 100 lignes, le résultat sera 4000. En rajoutant l'adresse de départ du playfield, on obtient alors la valeur que doit renfermer *BPLxPT*. Pour déplacer le playfield dans la fenêtre d'écran, on modifie cette valeur avec chaque image, d'une ou plusieurs lignes, suivant la vitesse désirée. Etant donné qu'on ne peut modifier *BPLxPT* qu'en dehors de la zone visible, on doit se servir de la *liste du COPPER* énoncée plus haut.

On peut alors modifier l'adresse de n'importe quel point dans la *liste du COPPER*, ce dernier l'écrivant automatiquement dans le registre *BPLxPT*. On doit cependant faire attention à ne pas modifier les listes pendant que le *COPPER* accède à ces instructions. En effet, si à ce moment précis, on transforme un mot d'une adresse, le *COPPER* lira l'ensemble, c'est-à-dire une adresse erronée.

Largeur du playfield et scrolling horizontal

Pour le scrolling horizontal et les playfields d'extension large, des registres spéciaux sont à l'honneur (exclusivement accessibles en écriture).

$108 BPL1MOD valeur modulo des bitplanes impairs
$10A BPL2MOD valeur modulo des bitplanes pairs

BPLCON1 $102

Bit n° :	15-8	7	9	6	5	4	2	1	0
Fonction :	inutilisé	P2H3	P2H2	P2H1	P2H0	P1H3	P1H2	P1H1	P1H0

P1H0-P1H3 position des plans pairs (4 bits)
P2H0-P2H3 position des plans impairs (4 bits)

La valeur *modulo*, issue des registres *BPLxMOD* autorise la zone mémoire du coin droit. Ce principe sera encore souvent utilisé par le hardware de l'Amiga. Il permet la définition d'une petite zone mémoire à l'intérieur d'une grande zone où les lignes et colonnes sont définies.

Prenons par exemple, une grande zone mémoire de 640 points de large, et d'une hauteur de 256 points, constituant un playfield. Ceci correspond en fait à 256 lignes de 40 mots. La petite zone s'identifie donc à la taille normale de la fenêtre d'écran, c'est-à-dire 320 points par 200, ou 20 mots par lignes. Le problème est qu'après l'affichage d'une ligne, *BPLxPT* est augmenté de 20 mots. Pour pointer le début de la prochaine ligne de notre playfield, il faudrait qu'il soit en fait augmenté de 40 mots. On est donc obligé de rajouter une valeur de 20 mots à *BPLxPT* après reproduction de chaque ligne. Cette addition peut être réalisée automatiquement par l'Amiga. Il suffit d'écrire la différence entre les deux longueurs de lignes dans le registre *modulo*, cette valeur étant ajoutée automatiquement à la valeur de *BPLxPT* après affichage de chaque ligne.

Largeur du playfield : 80 octets (40 mots)

Largeur de la fenêtre d'écran : 40 octets (20 mots)

Valeur modulo : 40 octets (cette valeur doit toujours être un nombre pair)

Start = adresse de départ de la première ligne du playfield.

Affichage de la 1ère ligne :

Mot	:	0	1	2	3	...	19
BPLxPT	:	Start	Start+2	Start+4	Start+6		Start+38

Après l'affichage du dernier mot, *BPLxPT* sera augmenté d'un mot :

```
BPLxPT = Start + 40
```

A la fin de la ligne, la valeur *modulo* sera rajoutée au contenu du registre *BPLxPT* :

```
BPLxPT = BPLxPT + modulo BPLxPT = Start+40 + 40 = Start+80
```


Affichage de la 2ème ligne :

Mot	:	0	1	2	3	...	19
BPLxPT	:	Start+80	Start+82	Start+84	Start+86		Start+118

et ainsi de suite ...

En haut, la moitié gauche du grand champ sera reproduite dans la fenêtre écran. Si on veut débuter à une position horizontale différente, il suffit de rajouter à la valeur de départ de *BPLxPT* le nombre de mots souhaité, la valeur *modulo* restant la même.

Si la valeur de départ est la même que plus haut, *BPLxPT* aura une valeur de départ de Start+40, avec laquelle la première moitié du playfield sera pointée.

Affichage de la première ligne :

Mot	:	0	1	2	3	...	19
BPLxPT	:	Start+40	Start+42	Start+44	Start+46		Start+78

Après affichage du dernier mot :

```
BPLxPT = Start + 80
```

Addition de la valeur modulo

```
BPLxPT = BPLxPT + modulo BPLxPT = Start+80 + 40 = Start+120
```

Affichage de la deuxième ligne :

Mot	:	0	1	2	3	...	19
BPLxPT	:	Start+120	Start+122	Start+124	Start+126		Start+158

Les valeurs *modulo* des bitplanes pairs et impairs se laissent initialiser séparément. Ceci autorise la présence de deux playfields de grande extension, dans le mode *Dual-Playfield*. Si on ne travaille pas dans ce mode, il suffit de mettre la même valeur dans les fichiers *BPLxMOD*.

A l'aide des registres *BPLxPT* et *BPLxMOD*, l'écran peut être décalé d'un pas de 16 points. Un scrolling plus fin, d'un pas d'un point, est possible avec le registre *BPLCON1*. Les 4 bits inférieurs renferment la valeur du scrolling des plans pairs, les bits 4 à 7, celle des plans impairs. Ces valeurs de scrolling ralentissent la sortie des données de points des plans correspondant. Si ces valeurs sont nulles, les données seront émises tous les 8.5 cycles (4.5 en Hires) après la position *DDFSTRT* et si, au contraire, elles ne sont pas nulles, les données apparaitront, le nombre de points contenus dans *BPLCON1* plus tard, c'est-à-dire décalé de la valeur du nombre de points, vers la droite.

On peut obtenir un scrolling très fluide vers la droite du contenu de l'écran lorsqu'on augmente la valeur de *BPLCON1* progressivement, de 0 à 15, puis en la remettant à 0. On aura diminué la valeur de *BPLxPT* d'un mot.

Le scrolling vers la gauche est autorisé lorsqu'on décrémente la valeur du scrolling, de 15 à 0 par exemple. On aura augmenté la valeur de *BPLxPT* d'un mot. *BPLCON1* doit être modifié en dehors de la zone visible. On utilisera à cet effet, soit l'interruption *'écran vide vertical'* du processeur, soit le *COPPER*. On pourra alors modifier la valeur dans la *liste du COPPER* à tout moment, cette dernière étant toujours transférée dans le registre *BPLCON1*, lors du temps mort vertical.

Si on décale l'image vers la droite au moyen de la valeur *BPLCON1*, les points en trop seront effacés correctement sur le bord gauche, mais les nouveaux points sur la droite n'apparaitront pas, étant donné qu'aucune donnée de point ne sera lue. Afin d'éviter cela, la valeur de *DDFSTRT* doit être rallongée de 8 cycles de bus (4 cycles en mode Hires), par rapport au début normal.

On calcule, comme auparavant, la valeur *DDFSTRT* de la fenêtre d'écran, en la réduisant de 8 (4 Hires). A la valeur normale $38 se substituera la nouvelle valeur $30 (le sprite 7 sera désactivé). Ce mot à lire en plus n'est normalement pas visible. Lorsque la valeur du scrolling est à 0, ses points s'affichent sur une position libre, le bord gauche de la fenêtre d'écran. Si celle-ci a une largeur de 320 points, 21 mots de donnée seront lus, à la place des 20 normaux. Il faudra en tenir compte pour l'évaluation des bitplanes et de la valeur *modulo*.

A l'aide de la valeur du scrolling, on peut positionner la fenêtre d'écran n'importe où horizontalement. Si la différence entre *DIWSTRT* et *DFFSTRT* est supérieure à 17 points, les données seront décalées vers la droite du montant de la différence.

Le mode Interlace

Malgré le double affichage du nombre des lignes en mode *interlace*, ce dernier ne diffère de la représentation normale, en technique de programmation, que par une modification de la valeur *modulo* et par une nouvelle *liste du COPPER*. Comme cela a été expliqué au chapitre 1.5.2, le mode *interlace* se caractérise par l'alternance d'affichage des lignes paires et impaires.

Comme on peut reproduire, normalement, un playfield *interlace* en mémoire, on fixe la valeur *modulo* du nombre de mots d'une ligne. Après l'affichage d'une ligne, la valeur de *BPLxPT* sera augmentée de la longueur de cette dernière, ce qui correspond à un saut d'une ligne. A chaque image, seules toutes les deux lignes sont représentées. On devra donc initialiser *BPLxPT* en concordance avec le type de demi-image qui sera affiché, c'est-à-dire en alternant suivant la première ou la deuxième ligne du playfield, afin que, soit les lignes Impaires, soit les lignes paires puissent être pointées. Dans le cas *Long Frame* (lignes impaires), *BPLxPT* sera fixé sur la ligne 1 ; dans le cas *Short Frame* (lignes paires), *BPLxPT* sera fixé sur la ligne 2. La *liste du COPPER* est un peu plus compliquée pour un playfield *interlace*. En effet, 2 listes sont nécessaires pour les deux types de demi-image, afin de pouvoir être alternés avec chaque image.

Liste du COPPER pour un playfield en mode interlace :

ligne1 = adresse de la première ligne du bitplane
ligne2 = adresse de la deuxième ligne du bitplane

Copper1 :

```
MOVE #ligne1Hi,BPLxPTH    ;pointeur BPLxPTH sur
MOVE #ligne1Lo,BPLxPTL    ;sur la première ligne
... autres instructions du COPPER
```

```
MOVE #Copper2Hi,COP1LCH   ;adresse de la liste
MOVE #Copper2Lo,COP1LCL   ;initialisée à Coppe..
WAIT ($FF,$FE)            ;fin de la première liste
```

Copper2 :

```
MOVE #ligne2Hi,BPLxPTH    ;pointeur BPLxPTH sur
MOVE #ligne2Lo,BPLxPTL    ;sur la deuxième ligne
... autres instructions du COPPER
MOVE #Copper1Hi,COP1LCH   ;adresse de la liste
MOVE #Copper1Lo,COP1LCL   ;initialisée à Copper1
WAIT ($FF,$FE)            ;fin de la deuxième liste
```

Le *COPPER* change de liste après chaque image, grâce à la liste d'instructions se trouvant à la fin, qui charge l'adresse de l'autre liste dans *COP1LC*. Cette adresse sera chargée automatiquement dans le compteur programme du *COPPER*, au début de chaque image. L'initialisation du mode *interlace* doit être fait avec soin, afin que la *liste du COPPER* réservée aux lignes impaires traite bien une *Long Frames* :

- Initialiser *COP1LC* avec *Copper1*.

- Le bit *LOF* (bit 15) du registre *VPOS* ($2A) doit être mis à 0. Ceci assure la présence d'une *LongFrame* comme première demi-image, après démarrage du mode *interlace* et le passage de la liste sur *Copper1*. Le bit *LOF* sera inversé à chaque demi-image. Si on le fixe à 0, au début de la prochaine demi-image, il passera à 1.

- Mode *interlace* sélectionné.

- Attente de la première ligne de la prochaine image (ligne 0).

- *DMA COPPER* activé.

Toutes les autres fonctions des registres restent inchangées en mode *interlace*. Toutes les instructions concernent les lignes des demi-images actuelles (0-312 pour *LongFrame* et 0-311 pour *ShortFrame*). Si on sélectionne le mode *interlace* sans modifier les autres registres, on remarque un léger tremblement de l'image.

Ceci est dû au fait que chaque ligne d'une demi-image est remplacée par une autre. De plus, les deux images contiennent les mêmes données. Ce n'est qu'au moyen des *listes du COPPER* qu'on arrive à doubler le nombre de lignes, tout en modifiant la taille des bitplanes et en choisissant la valeur *modulo* correspondante, ceci de telle façon que chaque demi-image sera représentée par d'autres données.

Le mode *interlace* est accompagné d'un fort scintillement, étant donné que chaque ligne n'est rafraîchie que toutes les deux images, c'est-à-dire 25 fois par seconde. On peut atténuer ce scintillement en choisissant des couleurs différentes avec le moins de contrastes possibles, l'oeil de l'homme n'arrivant que difficilement à repérer un changement d'images aux couleurs faiblement contrastées.

Les registres de contrôle

Pour activer les différents modes, il existe trois registres de contrôle *BPLCON0* à *BPLCON2*. *BPLCON1* contient la valeur du scrolling. Les deux autres sont structurés de la manière suivante :

BPLCON0 $100

Bit n°	Nom	Fonction
15	HIRES	Mode haute résolution sélectionné (HIRES = 1).
14	BPU2	Les trois bits BPUx forment ensemble
13	BPU1	un compteur 3 bits, qui contient le
12	BPU0	nombre de bitplanes utilisés (0 à 6).
11	HOMOD	Mode Hold and Modify sélectionné (HOMOD = 1).
10	DBPLF	Mode Dual-Playfield sélectionné (DBPLF = 1).
9	COLOR	Sortie Vidéo couleur (COLOR = 1).
8	GAUD	Genlock audio sélectionné (GAUD = 1).
7-4	—	Inutilisés
3	LPEN	Entrée crayon lumineux activé (LPEN = 1).
2	LACE	Mode Interlace sélectionné (LACE = 1).
1	ERSY	Synchronisation externe sélectionnée (ERS
0	—	Inutilisé

HIRES

Ce bit sert à la sélection du mode *haute résolution* (Hires, 640 points par ligne).

BPL0-BPL2

Ces trois bits forment un compteur qui renferme le nombre de bitplanes activés. Seules les valeurs comprises entre 0 et 6 sont possibles.

HOMOD et DBPLF

Ces deux bits sélectionnent les modes correspondants. Ces derniers ne doivent pas être activés en même temps. Le mode *Extra-Half-Bright* sera sélectionné automatiquement, lorsque tous les 6 bitplanes seront activés.

LACE

Lorsque le bit *LACE* sera initialisé, le bit *LOF-Frame* du registre *VPOS*, sera inversé au début de chaque nouvelle image, afin que le changement entre *Long* et *Short Frame* puisse avoir lieu.

COLOR

Le bit *COLOR* active la sortie du signal couleur d'AGNUS. Lorsque ce dernier délivre ce signal, le mixeur vidéo peut générer un signal vidéo couleur, sinon le signal reste noir et blanc. La sortie RGB ne sera pas influencée par ce signal.

ERSY

Le bit *ERSY* commute les connexions des signaux de synchronisation horizontale et verticale, du mode sortie en mode entrée. Ainsi l'image de l'Amiga pourra être synchronisée par un signal extérieur.

L'interface *GENLOCK* utilise ce bit pour pouvoir mixer l'image de l'Amiga avec n'importe quelle image vidéo. Le bit *GAUD* est aussi utilisé par cette interface (cf. section 1.3.2).

BPLCON2 $104

Bit n° :	15-7	6	5	4	3	2	1	0
Fonction :	inuti.	PF2PRI	PF2PR2	PF2PR1	PF2PR0	PF1P2	PF1P1	PF1P0

PF2P0-PF2P2 et *PF1P0-PF1P2* déterminent la priorité des sprites dans le traitement d'un playfield (cf. prochain chapitre).

Si le bit *PF2PRI* est activé, les plans pairs sont prioritaires par rapport aux plans impairs, c'est-à-dire qu'ils s'affichent devant les plans impairs. Ce bit n'a de résultats visibles qu'en mode *Dual-Playfield*.

Affichage d'écran activé

Après avoir initialisé les registres vus plus haut avec les valeurs désirées, on doit encore activer le canal DMA bitplane et dans le cas où on utilise le *COPPER*, le canal *DMA COPPER*. C'est le rôle de l'instruction *MOVE* suivante, qui active les bits *DMAEN*, *BPLEN* et *COPEN* du registre de contrôle *DMACON* :

```
MOVE.W #$8310,$DFF096
```

Exemples de programmes

Programme 1 : Démonstration du mode Extra-Half-Bright

Ce programme génère un playfield en mode basse résolution, c'est-à-dire 320 points par 200. Il utilise 6 bitplanes, permettant ainsi la sélection du mode *Extra-Half-Bright*. Au départ, le programme gère la mémoire nécessaire. Etant donné que les adresses des bitplanes sont maintenant connues, la *liste du COPPER* ne sera pas issue du programme, mais directement placée dans la CHIP-RAM. Elle ne renferme que les instructions servant à initialiser le registre *BPLxPT*.

Afin que l'on puisse voir les 64 couleurs possibles, le programme dessine à des positions déterminées, des blocs de 16 ...s sur 16, dans toutes les couleurs. Le registre *VHPOS* sera utilisé comme générateur de nombres aléatoires.

```
;*** Dual-Playfield et Scrolling ***

;registres des circuits spécialisés

INTENA  = $9A      ; registre d'autorisation des interruptions (écriture)
DMACON  = $96      ; registre demande d'interruptions (lecture)
COLOR00 = $180     ; registre couleur 0
VHPOSR  = $6       ; position du faisceau d'électrons (lecture)

;registre COPPER

COP1LC  = $80      ; adresse de la 1ère liste
COP2LC  = $84      ; adresse de la 2ème liste
COPJMP1 = $88      ; saut à la 1ère liste
COPJMP2 = $8A      ; saut à la 2ème liste

;registre bitplane

BPLCON0 = $100     ; registre 0 de contrôle bitplane
BPLCON1 = $102     ; 1 (valeur pour le scrolling)
BPLCON2 = $104     ; 2 (sprite <> priorité playfield)
BPL1PTH = $0E0     ; pointeur sur le
BPL1PTL = $0E2     ; premier bitplane
BPL1MOD = $108     ; valeur modulo pour les plans impairs
BPL2MOD = $10A     ; valeur modulo pour les plans pairs
DIWSTRT = $08E     ; début de la fenêtre écran
DIWSTOP = $090     ; fin de la fenêtre écran
DDFSTRT = $092     ; début DMA bitplane
DDFSTOP = $094     ; fin DMA bitplane

;CIA-A registre port A (bouton souris)

CIAAPRA = $BFE001

;Exec Library Base Offsets
```



```
Forbid     = -30-102
Permit     = -30-109
AllocMem   = -30-168    ; ByteSize, Requirements / d0, d1
FreeMem    = -30-180    ; MemoryBlock, ByteSize / a1, d0

;graphics base

StartList = 38

;autres labels

Execbase  = 4
Planesize = 40*256      ; taille des bitplanes : 40 octets sur 256 lignes
CLsize    = 13*4        ; la liste COPPER contient 13 instructions
Chip      = 2           ; CHIP-RAM sollicitée
Clear     = Chip+$10000 ; CHIP-RAM réservée

Start :

;Allouer de la mémoire pour les bitplanes

  move.l Execbase,a6
  move.l #Planesize*6,d0    ; mémoire nécessaire à tous les plans
  move.l #Clear,d1          ; la mémoire doit être remplie de 0
  jsr AllocMem(a6)          ; mémoire sollicitée
  move.l d0,Planeadr        ; mise en mémoire de l'adresse du 1er plan
  beq Fin                   ; Erreur -> FIN

;Solliciter la mémoire pour la liste du COPPER

  moveq #CLsize,d0          ; taille de la liste COPPER
  moveq #Chip,d1
  jsr AllocMem(a6)
  move.l d0,CLadr
  beq FreePlane             ; erreur -> RAM bitplane libérée

;Installation de la liste COPPER

  moveq #5,d4               ; 6 plans = 6 boucles
```

```
  move.l d0,a0              ; adresse de la liste du COPPER dans a0
  move.l Planeadr,d1
  move.w #BPL1PTH,d3        ; premier registre dans d3

MakeCL :
  move.w d3,(a0)+           ; BPLxPTH dans la RAM
  addq.w #2,d3              ; prochain registre
  swap d1
  move.w d1,(a0)+           ; Haut Mot de l'adresse plan dans la RAM
  move.w d3,(a0)+           ; BPLxPTL dans la RAM
  addq.w #2,d3              ; prochain registre
  swap d1
  move.w d1,(a0)+           ; Bas Mot de l'adresse plan dans la RAM
  add.l #Planesize,d1       ; adresse du prochain plan calculée

  dbf d4,MakeCL

  move.l #$FFFFFFFE,(a0)    ; fin de la liste du COPPER

;DMA activé et Task Switching bloqué

  jsr Forbid(a6)
  lea $DFF000,a5
  move.w #$03E0,DMACON(A5)

;COPPER initialisé

  move.l CLadr,COP1LC(a5)
  clr.w COPJMP1(a5)

;création de la palette couleur

  moveq #31,d0              ; compteur des registres couleur
  lea COLOR00(a5),a1
  moveq #1,d1               ; première couleur
SetTab:
  move.w d1,(a1)+           ; couleur dans le registre correspondant
  mulu #3,d1                ; calcul de la prochaine couleur
  dbf d0,SetTab
```



```
;playfield initialisé

  move.w #$3081,DIWSTART(a5)            ; valeurs standard pour
  move.w #$30C1,DIWSTOP(a5)             ; la fenêtre écran
  move.w #$0038,DDFSTRT(a5)             ; et le DMA bitplane
  move.w #$00D0,DDFSTOP(a5)
  move.w #%0110001000000000,BPLCON0(a5) ; 6 bitplanes
  clr.w BPLCON1(a5)                     ; pas de scrolling
  clr.w BPLCON2(a5)                     ; pas de priorité
  clr.w BPL1MOD(a5)                     ; modulo de tous les plans = 0
  clr.w BPL2MOD(a5)

;DMA activé

  move.w #$8380,DMACON(a5)

;Bitplane modifié

  moveq #40,d5                  ; octets par ligne
  clr.l d2                      ; commencer avec la couleur 0

Loop :
  clr.l d0
  move.w VHPOSR(a5),d0          ; valeur aléatoire dans d0
  and.w #$3FFE,d0               ; éliminer les bits superflus

  cmp.w #$2580,d0               ; comparaison à la taille d'un plan
  bcs cont                      ; si plus petit, alors on continue
  and.w #$1FFE,d0               ; sinon, effacer le bit supérieur
Cont :
  move.l Planeadr,a4
  add.l d0,a4
  moveq #5,d4
  move.l d2,d3

Block :
  clr.l d1
  lsr #1,d3             ; un bit issu du numéro de couleur dans le flag x
  negx.w d1             ; d1 égalisé avec le flag x
```

```
  moveq #15,d0              ; 16 lignes par bloc
  move.l a4,a3              ; adresse bloc dans le re    re de travail

Fill :
  move.w d1,(a3)            ; mot dans le bitplane
  add.l d5,a3               ; calcul de la prochaine ligne
  dbf d0,Fill

  add.l #Planesize,a4       ; prochain bitplane
  dbf d4,Block

  addq.b #1,d2              ; prochaine couleur
  btst #6,CIAAPRA           ; test bouton souris
  bne Loop                  ; sinon -> on continue

;*** fin de programme ***

;liste ancienne du COPPER activée

  move.l #GRname,a1             ; paramètre pour initialiser OpenLibrary
  clr.l d0
  jsr OpenLibrary(a6)           ; ouverture de la librairie graphique
  move.l d0,a4
  move.l StartList(a4),COP1LC(a5) ; adresse du début de la liste
  clr.w  COPJMP1(a5)
  move.w #$8060,DMACON(a5)      ; réinitialisation du canal DMA
  jsr Permit(a6)                ; Task Switching autorisé

;mémoire libérée des listes du COPPER

  move.l CLadr,a1               ; paramètre initialisant FreeMem
  moveq #CLsize,d0
  jsr FreeMem(a6)               ; mémoire libérée

;mémoire des bitplanes libérée

FreePlane :
  move.l Planeadr,a1
  move.l #Planesize*6,d0
  jsr FreeMem(a6)
```



```
Fin :
  clr.l d0
  rts                           ; quitter le programme

;variables

CLadr:    dc.l 0
Planeadr: dc.l 0

;constante

GRname:   dc.b "graphics.library",0

    END

;fin du programme
```

Programme 2 : Dual-Playfield et Scrolling

Ce programme utilise plusieurs effets en même temps : premièrement, il définit un écran *Dual-Playfield* avec un bitplane basse résolution par playfield. Ensuite, il augmente la taille de la fenêtre d'écran, afin que les bordures ne soient plus visibles. Enfin, il scrolle le playfield 1 horizontalement et le playfield 2 verticalement.

Au début et à la fin, on utilisera les mêmes routines d'occupation de la mémoire que plus haut.

Les deux playfields seront occupés par un damier, avec des cases de 16 points sur 16.

La grande boucle du programme qui produit le scrolling attend tout d'abord une ligne du temps mort vertical, où le système d'exploitation a déjà traité toutes les éventuelles routines d'interruption et où le *COPPER* a déjà initialisé le pointeur *BPLxPT*. Puis il compte le nombre de lignes à scroller, calcule le nouveau pointeur pour le champ deux et l'écrit dans la *liste du COPPER*.

Les positions de scrolling sont obtenues par séparation [illegible] compteur *scroll* entre les 4 bits inférieurs et le reste. Les 4 bits de [illegible]aible poids seront transférés dans le registre *BPLCON1*, où ils représenteront la valeur du scrolling. A partir du 5ème bit, le nouveau pointeur *BPLxPT* sera calculé et copié dans la *liste du COPPER*.

Les deux compteurs de scrolling, vertical et horizontal, seront augmentés progressivement de 0 à 31 et puis remis à 0. Ceci aura pour résultat un effet de scrolling très fluide, étant donné que le contenu des playfields, suivant le modèle utilisé, sera répété tous les 32 points.

```
;*** Dual-Playfield et Scrolling ***

;registres circuits spécialisés

INTENA  = $9A      ; registre d'autorisation d'interruptions (écriture)
DMACON  = $96      ; registre de demande d'interruptions (lecture)
COLOR00 = $180     ; registre de contrôle DMA (écriture)
VPOSR   = $4       ; registre 0 palette couleur

;registre COPPER

COP1LC  = $80      ; adresse de la 1ère liste
COP2LC  = $84      ; adresse de la 2ème liste
COPJMP1 = $88      ; saut à la 1ère liste
COPJMP2 = $8A      ; saut à la 2ème liste

;registre bitplane

BPLCON0 = $100     ; registre 0 de contrôle bitplane
BPLCON1 = $102     ; 1 (valeur de scrolling)
BPLCON2 = $104     ; 2 (sprite <> priorité playfield)
BPL1PTH = $0E0     ; pointeur sur le 1er
BPL1PTL = $0E2     ; bitplane
BPL1MOD = $108     ; valeur modulo pour les plans impairs
BPL2MOD = $10A     ; valeur modulo pour les plans pairs
DIWSTRT = $08E     ; début de la fenêtre écran
DIWSTOP = $90      ; fin de la fenêtre écran
```



```
DDFSTRT = $92      ; débu   4A bitplane
DDFSTOP = $94      ; ar     DMA bitplane

;CIA-A registre port A (bouton souris)

CIAAPRA = $BFE001

;Exec Library Base Offset

OpenLibrary = -30-522      ; LibName, Version / a1, d0
Forbid      = -30-102
Permit      = -30-108
AllocMem    = -30-168      ; ByteSize,Requirements / d0, d1
FreeMem     = -30-180      ; MemoryBlock,ByteSize / a1, d0

;graphics base

StartList = 38

;autres labels

Execbase   = 4
Planesize  = 52*345        ; taille des bitplanes
Planewidth = 52
CLsize     = 5*4           ; la liste du COPPER contient 5 instructions
Chip       = 2             ; CHIP-RAM sollicitée
Clear      = Chip+$10000   ; CHIP-RAM réservée

;Initialisation

Start :

;réserver de la mémoire pour les bitplanes

   move.l Execbase,a6
   move.l #Planesize*2,d0    ; mémoire nécessaire aux plans
   move.l #Clear,d1
   jsr AllocMem(a6)          ; mémoire sollicitée
   move.l d0,Planeadr
   beq.l Fin                 ; erreur -> Fin
```

```
;réservation de mémoire pour la liste du COPPER

   moveq #CLsize,d0
   moveq #Chip,d1
   jsr AllocMem(a6)
   move.l d0,CLadr
   beq FreePlane           ; erreur -> mémoire des plans libérée

;mise en place de la liste du COPPER

   moveq #1,d4             ; deux bitplanes
   move.l d0,a0
   move.l Planeadr,d1
   move.w #BPL1PTH,d3

MakeCL:
   move.w d3,(a0)+
   addq.w #2,d3
   swap d1
   move.w d1,(a0)+
   move.w d3,(a0)+
   addq.w #2,d3
   swap d1
   move.w d1,(a0)+
   add.l #Planesize,d1     ; adresse du prochain plan

   dbf d4,MakeCL

   move.l #$FFFFFFFE,(a0)  ; fin de la liste du COPPER

;*** programme principal ***

;DMA activé et Task Switching bloqué

   jsr Forbid(a6)
   lea $DFF000,a5
   move.w #$01E0,DMACON(a5)
```



```
;COPPER initialisé

   move.l CLadr,COP1LC(a5)
   clr.w COPJMP1(a5)

;Playfield initialisé

   move.w #0,COLOR00(a5)
   move.w #0F00,COLOR00+2(a5)
   move.w #$000F,COLOR00+18(a5)

   move.w #$1A64,DIWSTRT(a5)       ; 26,100
   move.w #$39D1,DIWSTOP(a5)       ; 313,465
   move.w #$0020,DDFSTRT(a5)       ; lecture d'un mot de plus
   move.w #$00D8,DDFSTOP(a5)
   move.w #%0010011000000000,BPLCON0(a5) ; Dual-Playfield activé
   clr.w  BPLCON1(a5)              ; valeur de départ du scrolling à 0
   clr.w  BPLCON2(a5)              ; playfield 1 en avant du playfield 2

   move.w #4,BPL1MOD(a5)           ; valeur modulo = 2 mots
   move.w #4,BPL2MOD(a5)

;DMA activé

   move.w #$8180,DMACON(a5)

;remplissage des bitplanes avec un damier

   move.l Planeadr,a0
   move.w #Planesize/2-1,d0        ; compteur de boucle
   move.w #13*16,d1                ; maximum = 16 lignes
   move.l #$FFFF0000,d2            ; damier
   move.w d1,d3

Fill :
   move.l d2,(a0)+
   subq.w #1,d3
   bne.s cont
   swap d2                         ; changer de modèle
   move.w d1,d3
```

```
Cont:
   dbf d0,Fill

;Playfields scrollés

   clr.l d0                ; position de scroll vertical
   clr.l d1                ; position de scroll horizontal
   move.l CLadr,a1         ; adresse de la liste COPPER
   move.l Planadr,a0       ; adresse du premier bitplane

Wait :
   move.l VPOSR(a5),d2     ; lecture de la position
   and.l #$0001FF00,d2     ; bits horizontaux sélectionnés
   cmp.l #$00001000,d2     ; attendre la ligne 16
   bne.s wait

;scrolling vertical du playfield 1

   addq.b #2,d0            ; compteur scrolling vertical incrémenté
   cmp.w #$80,d0           ; test d'arrivée à la valeur 128 (4*32)
   bne.s PlusHaut
   clr.l d0                ; retour à 0
Plushaut :
   move.l d0,d2            ; compteur scrolling copié
   lsr.w #2,d2             ; copie divisée par 4
   mulu #52,d2             ; nombre d'octets/ligne * position de scroll
   add.l a0,d2             ; plus adresse du premier plan

   add.l #Planesize,d2     ; plus taille du plan
   move.w d2,14(a1)        ; donne l'adresse de fin de la liste COPPER
   swap d2
   move.w d2,10(a1)

;scrolling horizontal du playfield 2

   addq.b #1,d1            ; compteur scrolling horizontal incrémenté
   cmp.w #$80,d1
   bne.s PlusLoin
   clr.l d1
```



```
PlusLoin :
   move.l d1,d2
   lsr.w #2,d2
   move.l d2,d3            ; position scrolling copiée
   and.w #$FFF0,d2         ; 4 bits inférieurs forcés à 0
   sub.w d2,d3             ; 4 bits inférieurs dans d3 isolés
   move.w d4,BPLCON1(a5)   ; dernière valeur dans BPLCON1
   move.w d3,d4            ; nouvelle valeur de scrolling dans d4
   lsr.w #3,d2             ; nouvelle adresse pour la liste du COPPER
   add.l a0,d2             ; calcul
   move.w d2,6(a1)         ; et écriture dans la liste du COPPER
   swap d2
   move.w d2,2(a1)

   btst #6,CIAAPRA         ; souris activée (bouton pressé)
   bne.s Wait              ; non -> continuer

;*** fin de programme ***

;ancienne liste du COPPER réactivée

   move.l #GRname,a1       ; initialisation d'OpenLibrary
   clr.l d0
   jsr OpenLibrary(a6)     ; librairie graphique ouverte
   move.l d0,a4
   move.l StartList(a4),COP1LC(a5)
   clr.w COPJMP1(a5)
   move.w #$83E0,DMACON(a5)
   jsr Permit(a6)          ; Task Switching rétabli

;mémoire des listes du COPPER libérée

   move.l CLadr,a1
   moveq #CLsize,d0
   jsr FreeMem(a6)

Freeplane :
   move.l Planeadr,a1
   move.l #Planesize*2,d0
   jsr FreeMem(a6)
```

```
Fin:
   clr.l d0
   rts                          ; quitter le programme

;variables

CLadr :     dc.l 0
Planeadr :  dc.l 0
test :      dc.l 0

;constante

GRname :    dc.b "graphics.library",0

   END
;Fin du programme
```

1.5.6 SPRITES

Les sprites sont des éléments graphiques qui peuvent être utilisés indépendamment des playfields. Chaque sprite peut avoir une largeur de 16 points et une hauteur maximale analogue à celle de la fenêtre d'écran. Il peut occuper n'importe quelle position à l'écran. Normalement, les sprites se trouvent en avant du playfield et leurs points en cachent le graphisme. Le pointeur de la souris en est un exemple. Sur l'Amiga, seuls 8 sprites peuvent être reproduits, avec la possibilité de 3 couleurs. Il existe aussi un moyen de combiner deux sprites en un nouveau, qui acceptera lui, 15 couleurs.

La structure des sprites

Le choix des couleurs

Le choix des couleurs d'un sprite ressemble à celui du mode *Dual-Playfield*. Le sprite a une largeur de 16 points, ce qui représente deux mots de donnée, ce qui peut correspondre à deux 'mini-bitplanes'. Ainsi la couleur d'un point sera déterminée, comme pour les bitplanes, par la combinaison des bits le représentant.

La couleur du premier point (point le plus à gauche du sprite) sera déterminée par les deux bits de plus fort poids (bit 15) des deux mots de donnée. Les deux bits de plus faible poids correspondent à la couleur du dernier point. Chaque point est donc représenté par deux bits, ce qui amène la possibilité de 4 combinaisons différentes, donc de 4 couleurs différentes. Pour attribuer une couleur à ces points au travers de ces 4 valeurs, on utilisera à nouveau la palette couleur, étant donné qu'il n'existe pas de registre couleur propre aux sprites. Les couleurs des sprites seront représentées par la deuxième moitié des registres, c'est-à-dire les registres couleurs correspondant aux numéros variant de 16 à 31. Ainsi, les sprites et les playfields pourront entrer en conflit lorsque ce dernier comprendra plus de 16 couleurs.

Le tableau suivant nous montre les liens entre les registres couleurs et les sprites :

Sprite N°	Données sprite	Registre couleur
0 & 1	0 0	Transparent
	0 1	COLOR17
	1 0	COLOR18
	1 1	COLOR19
2 & 3	0 0	transparent
	0 1	COLOR21
	1 0	COLOR22
	1 1	COLOR23
4 & 5	0 0	transparent
	0 1	COLOR25
	1 0	COLOR26
	1 1	COLOR27
6 & 7	0 0	transparent
	0 1	COLOR29
	1 0	COLOR30
	1 1	COLOR31

Les deux sprites qui se suivent ont les mêmes couleurs.

Comme en mode *Dual-Playfield*, la combinaison de bits de deux 0 ne reproduit pas une couleur, mais s'affiche en transparence. A la place de cette couleur, on peut voir directement les couleurs des éléments se trouvant en position d'arrière plan, que ce soit d'autres sprites, un playfield ou plus simplement la couleur du fond.

Si les trois couleurs ne suffisent pas, on peut combiner deux sprites. Les deux combinaisons de bits des deux sprites donnent ensemble une valeur 4 bits. Les deux mots de donnée du sprite ayant le plus grand numéro correspondent aux deux bits de plus fort poids de la valeur 4 bits. Cette dernière pointe sur 15 registres de la palette couleur, la valeur nulle correspondant à l'état transparent.

Ces registres sont les mêmes pour les 4 combinaisons possibles de sprite : COLOR16 à COLOR31.

Données sprite	Registre couleur	Données sprite	Registre couleur
0 0 0 0	transparent	1 0 0 0	COLOR24
0 0 0 1	COLOR17	1 0 0 1	COLOR25
0 0 1 0	COLOR18	1 0 1 0	COLOR26
0 0 1 1	COLOR19	1 0 1 1	COLOR27
0 1 0 0	COLOR20	1 1 0 0	COLOR28
0 1 0 1	COLOR21	1 1 0 1	COLOR29
0 1 1 0	COLOR22	1 1 1 0	COLOR30
0 1 1 1	COLOR23	1 1 1 1	COLOR31

Le DMA Sprite

La programmation des sprites sur l'Amiga peut se réaliser assez facilement. La plus grande partie du travail est réalisée par le biais des canaux DMA sprite. Pour représenter un sprite à l'écran, une liste de données sprite spéciale est nécessaire en mémoire. Elle contient toutes les données importantes du sprite. Pour afficher ce dernier, seule l'adresse de cette liste est à communiquer au contrôleur DMA.

Le contrôleur DMA possède un canal DMA propre à chaque sprite.

Il ne peut malheureusement que lire deux mots de données, pendant l'affichage d'une ligne par le *Raster*. C'est la raison pour laquelle le sprite n'a qu'une largeur de 16 points et est limité à 4 couleurs. Etant donné que ces deux mots de données sont lus à chaque affichage d'une ligne, la hauteur d'un sprite n'est limité que par la fenêtre écran.

Structure d'une liste de données sprite

Une telle liste de données se compose de lignes renfermant chacune 2 mots de données. A chaque ligne affichée par le *Raster*, une de la liste sera lue via le DMA. Celle-ci peut contenir, soit deux mots de contrôle, soit deux mots avec les données points, lorsque le sprite doit être à reproduit à l'instant.

Les mots de contrôle déterminent la position horizontale et les première et dernière lignes du sprite.

Après lecture de ces mots par le contrôleur DMA et lorsque celui-ci les a placé dans les registres correspondants, il attend jusqu'à ce que le faisceau d'électrons ait atteint le début de la première ligne du sprite. Puis, à chaque ligne, deux mots de données seront lus, et par le hardware de DENISE, une ligne de sprite sera affichée à la position correspondante, jusqu'à la dernière définie. Les deux prochains mots correspondent à des mots de contrôle. S'ils sont à 0, le canal DMA termine son activité. Mais il est aussi possible de donner une nouvelle position sprite. Le contrôleur DMA attendra alors à nouveau la ligne de début d'affichage et le processus se répètera aussi longtemps que la liste ne se finira pas avec deux mots nuls.

Structure d'une liste de données sprite (Start = adresse de départ de la liste dans la CHIP-RAM) :

Adresse	**Contenu**
Start+4	1er et 2ème mot de donnée de la 1ère ligne du sprite
Start+8	1er et 2ème mot de donnée de la 2ème ligne du sprite
Start+12	1er et 2ème mot de donnée de la 3ème ligne du sprite
Start+4*n	1er et 2ème mot de donnée de la nième ligne du sprite
Start+4*(n+1)	0,0 fin de la liste de données sprite

Structure du premier mot de contrôle :

Bit n° :	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Fonction :	E7	E6	E5	E4	E3	E2	E1	E0	H8	H7	H6	H5	H4	H3	H2	H1

Structure du deuxième mot de contrôle :

Bit n° :	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Fonction :	L7	L6	L5	L4	L3	L2	L1	L0	AT	0	0	0	0	E8	L8	H0

H0-H8	position horizontale du sprite (HSTART)
E0-E8	première ligne du sprite (VSTART)
L0-L8	dernière ligne + 1 du sprite (VSTOP)
AT	bit de contrôle d'attache

Il existe donc 9 bits de position horizontale et verticale pour chaque sprite. Ces bits ne sont pourtant pas répartis de façon pratique entre les deux registres de contrôle.

La résolution correspond verticalement à une ligne du *Raster* et horizontalement, à un point en mode *Lores*. Ces valeurs ne peuvent être modifiées, étant donné qu'elles sont indépendantes du mode des playfields.

La hauteur des sprites est limitée d'après les valeurs *DIWSTRT* et *STOP*, définissant la fenêtre d'écran. Si les coordonnées se trouvant dans les mots de contrôle dépassent ces derniers, les sprites ne seront plus ou partiellement représentés, suivant les limites de la fenêtre.

La position de départ, horizontale et verticale, concerne le coin supérieur gauche du sprite. La position de fin verticale correspond à la première ligne après le sprite, c'est-à-dire à la dernière + 1. Le nombre de lignes d'un sprite est donc *VSTOP-VSTART*.

L'exemple suivant reproduit un sprite aux coordonnées 180, 160, à peu près au centre de l'écran. Il doit avoir une hauteur de 8 lignes, la valeur de *VSTOP* étant alors 168.

Lorsqu'on réunit les deux mots de donnée, on obtient un nombre compris entre 0 et 3 qui représente une des 3 couleurs et l'état transparent (0) du sprite. On peut donc écrire les sprites de la manière suivante :

```
0000002222000000
0000220000220000
0002200330022000
0022003113002200
0022003113002200
0002200330022000
0000220000220000
0000002222000000
```

Dans la liste de données, on doit fournir les deux mots séparés :

```
Start:
dc.w $A05A,$A800 ; HSTART = $B4, VSTART = $A0, VSTOP=$A8
dc.w %0000 0000 0000 0000,%0000 0011 1100 0000
dc.w %0000 0000 0000 0000,%0000 1100 0011 0000
dc.w %0000 0001 1000 0000,%0001 1001 1001 1000
dc.w %0000 0011 1100 0000,%0011 0010 0100 1100
dc.w %0000 0011 1100 0000,%0011 0010 0100 1100
dc.w %0000 0001 1000 0000,%0001 1001 1001 1000
dc.w %0000 0000 0000 0000,%0000 1100 0011 0000
dc.w %0000 0000 0000 0000,%0000 0011 1100 0000
dc.w 0,0   ;fin de la liste de données sprite
```

Le bit AT, du deuxième mot de contrôle, établit si deux sprites sont combinés l'un avec l'autre. Il n'a qu'une fonction dans les sprites à numéro impair (sprites 1, 3, 5, 7). S'il est activé pour le sprite1, par exemple, ses bits de données seront interprétés avec ceux du sprite0, en tant que pointeur 4 bits sur la palette couleur. L'ordre des bits sera le suivant :

Sprite 1 (numéro impair), deuxième mot de donnée :	bit 3 (MSB)
Sprite 1 premier mot de donnée :	bit 2
Sprite 0 (numéro pair), deuxième mot de donnée :	bit 1
sprite 0 premier mot de donnée :	bit 0 (LSB)

Pour que deux sprites puissent être combinés, le position doit concorder. Si ce n'est pas le cas, on se replacera au matiquement en mode trois couleurs. Le plus simple est de mettre les mêmes mots de contrôle dans les deux listes de données sprite.

Voici pour exemple, une liste de données sprite en 16 couleurs :

Pour plus de simplicité, le sprite représenté ne comprend que 4 lignes. Les chiffres reproduisent à nouveau les couleurs des points correspondant. Pour pouvoir représenter les 15 couleurs, on utilisera la notation hexadécimale.

```
0011111111111100
1123456789ABCD11
11EFEFEFEFEFEF11
0011111111111100
```

Voici le détail des mots de données nécessaires à la deuxième ligne du sprite :

Couleurs du sprite :	1123456789ABCD11
Sprite1, mot de donnée 2 :	0000000011111100
Sprite1, mot de donnée 1 :	0000111100001100
Sprite0, mot de donnée 2 :	0011001100110000
Sprite0, mot de donnée 1 :	1101010101010111

Les listes de données pour ce même sprite sont les suivantes :

La position horizontale (*HSTART*) est à nouveau 180. La première ligne du sprite (*VSTART*) correspond à 160, la dernière ligne à 164.

```
StartSprite0:
dc.w $A05A,$A400 ;HSTART=$B4, VSTART=$A0, VSTOP=$A4, AT=0
dc.w %0011 1111 1111 1100, %0000 0000 0000 0000
dc.w %1101 0101 0101 0111, %0011 0011 0011 0000
dc.w %1101 0101 0101 0111, %0011 1111 1111 1100
dc.w %0011 1111 1111 1100, %0000 0000 0000 0000
```



```
StartSprite1:
dc.w $A05A,$A480 ;HSTART=$B4, VSTART=$A0, VSTOP=$A4, AT=1
dc.w %0000 0000    0 0000, %0000 0000 0000 0000
dc.w %0000 1111 0000 1100, %0000 0000 1111 1100
dc.w %0011 1111 1111 1100, %0011 1111 1111 1100
dc.w %0000 0000 0000 0000, %0000 0000 0000 0000
```

Plusieurs sprites sur un canal DMA

Après l'affichage d'un sprite, son canal est à nouveau libre. Dans l'exemple plus haut, les dernières données sprite seront lues à la ligne 163. Puis le canal DMA sprite sera désactivé par la présence des deux zéros. Comme cela a été vu, il est possible d'utiliser à nouveau le canal, en écrivant à la place des deux mots nuls, deux nouveaux mots de contrôle. Il y a tout de même une restriction, car une ligne reste libre entre la première du nouveau sprite et la dernière du sprite affiché. Si celle-ci est reproduite à la ligne 163, la nouvelle ne commencera pas avant la ligne 165. La raison réside dans la présence intermédiaire de la ligne comprenant les mots de contrôle, qui seront eux aussi lus. Le fonctionnement du DMA sprites se déroule donc de la façon suivante :

Ligne	Données sur le canal DMA
162	Avant dernière ligne du 1er sprite
163	Dernière ligne du 1er sprite
164	Mot de contrôle du 2ème sprite
165	1ère ligne du 2ème sprite
166	2ème ligne du 2ème sprite

L'exemple suivant place le sprite 3 couleurs de notre première liste de données à deux positions différentes.

```
Start:
;Premier sprite sur le canal DMA à la ligne 160 ($A0)
;position horizontale: 180 ($B4)
dc.w $A05A,$A800 ; HSTART = $B4, VSTART = $A0, VSTOP=$A8
dc.w %0000 0000 0000 0000,%0000 0011 1100 0000
dc.w %0000 0000 0000 0000,%0000 1100 0011 0000
```

```
dc.w %0000 0001 1000 0000,%0001 1001 1001 1000
dc.w %0000 0011 1100 0000,%0011 0010 0100 1100
dc.w %0000 0011 1100 0000,%0011 0010 0100 1100
dc.w %0000 0001 1000 0000,%0001 1001 1001 1000
dc.w %0000 0000 0000 0000,%0000 1100 0011 0000
dc.w %0000 0000 0000 0000,%0000 0011 1100 0000
;deuxième sprite à la suite sur le canal DMA
;à la ligne 176 ($B0), et à la position horizontale 300 ($12C)
dc.w $B096,$B800 ; HSTART = $12C, VSTART = $B0, VSTOP=$B8
dc.w %0000 0000 0000 0000,%0000 0011 1100 0000
dc.w %0000 0000 0000 0000,%0000 1100 0011 0000
dc.w %0000 0001 1000 0000,%0001 1001 1001 1000
dc.w %0000 0011 1100 0000,%0011 0010 0100 1100
dc.w %0000 0011 1100 0000,%0011 0010 0100 1100
dc.w %0000 0001 1000 0000,%0001 1001 1001 1000
dc.w %0000 0000 0000 0000,%0000 1100 0011 0000
dc.w %0000 0000 0000 0000,%0000 0011 1100 0000
dc.w 0,0   ;fin de la liste de données sprite
```

Initialisation du sprite

Après avoir créé un sprite, en reproduisant une liste de données correctes dans la CHIP-RAM, et après avoir choisi les couleurs dans la palette, on doit encore communiquer au contrôleur DMA l'adresse où se trouve la liste, afin qu'il puisse initialiser le sprite. Ainsi chaque canal DMA sprite possède une paire de registres, dans laquelle l'adresse de départ de la liste de données doit être écrite :

Registre *SPRxPT (SPRitexPoinTeur*, pointeur sur la liste de données pour le canal DMA sprite).

Adresse	Nom	Fonction
$120	SPR0PTH	pointeur sur la liste de données sprite (bits 16-18)
$122	SPR0PTL	pour le canal DMA sprite 0 (bits 0-15)
$124	SPR1PTH	pointeur sur la liste de données sprite (bits 16-18)
$126	SPR1PTL	pour le canal DMA sprite 1 (bits 0-15)
$128	SPR2PTH	pointeur sur la liste de données sprite (bits 16-18)
$12A	SPR2PTL	pour le canal DMA sprite 2 (bits 0-15)

$12C	SPR3PTH	pointeur sur la liste de données sprite (bits 16-18)
$12E	SPR3PTL	pour le canal DMA sprite 3 (bits 0-15)
$130	SPR4PTH	pointeur sur la liste de données sprite (bits 16-18)
$132	SPR4PTL	pour le canal DMA sprite 4 (bits 0-15)
$134	SPR5PTH	pointeur sur la liste de données sprite (bits 16-18)
$136	SPR5PTL	pour le canal DMA sprite 5 (bits 0-15)
$138	SPR6PTH	pointeur sur la liste de données sprite (bits 16-18)
$13A	SPR6PTL	pour le canal DMA sprite 6 (bits 0-15)
$13C	SPR7PTH	pointeur sur la liste de données sprite (bits 16-18)
$13E	SPR7PTL	pour le canal DMA sprite 7 (bits 0-15)

SPRxPT n'est accessible qu'en mode écriture.

Le contrôleur DMA utilise ces registres comme pointeur sur l'adresse actuelle de la liste de données sprite. Au début de chaque image, les registres renferment l'adresse du premier mot de contrôle. A chaque mot de données lu, ils sont augmentés d'un mot (16 bits), pour qu'ils pointent le mot suivant à la fin de la liste. Si le même sprite doit être représenté, à la même position, le pointeur devra être remis à l'adresse de départ.

Comme pour les pointeurs bitplanes *BPLxPT*, c'est le *COPPER* qui assure cette tâche, pendant le temps mort vertical. La partie correspondante de la *liste du COPPER* sera identique à ce qui suit :

StartSpritexH = adresse de départ de la liste de données du sprite x bits 16-18
StartSpritexL = bits 0-15

```
;début de la liste du COPPER
MOVE #StartSprite0H,SPR0PTH  ;initialisation du canal 0 DMA sprite
MOVE #StartSprite0L,SPR0PTL
MOVE #StartSprite1H,SPR0PTH  ;initialisation du canal 1 DMA sprite
MOVE #StartSprite1L,SPR0PTL
MOVE #StartSprite2H,SPR0PTH  ;initialisation du canal 2 DMA sprite
MOVE #StartSprite2L,SPR0PTL
MOVE #StartSprite7H,SPR0PTH  ;initialisation du canal 7 DMA sprite
MOVE #StartSprite7L,SPR0PTL
WAIT $FFFE                   ;fin de la liste COPPER
```

il n'y a aucune possibilité d'activer et de désactiver chaque canal DMA sprite séparément. Le bit *SPREN* (bit n° 5) du registre *DMACON* active les 8 canaux DMA en même temps. Si on ne veut pas tous les employer, on doit laisser travailler les canaux inutilisés avec des listes vides. On fixera les pointeurs sur deux mots mémoire à contenu nul, les deux mots de fin de liste du sprite utilisé, par exemple.

Les pointeurs *SPRxPT* doivent toujours être initialisés lors du temps mort vertical. Même quand les listes de données ne renferment rien d'autre que deux valeurs nulles, *SPRxPT* pointera toujours à la fin d'une image, sur le premier mot se tenant après elles.

Evidemment, l'initialisation des *SPRxPT* peut aussi être réalisée par le processeur, après l'interruption *'écran vide vertical'*.

Puis en dernier, il faut initialiser les 8 canaux DMA sprites, au moyen du bit *SPREN* du registre *DMACON*, par l'instruction *move* suivante :

```
MOVE.W #$8220,$DFF096 ;SPREN et DMAEN du registre DMACON initialisés
```

Mouvement des sprites

La position d'un sprite est déterminée par la valeur se trouvant dans les deux mots de contrôle de la liste de données *sprite*. Afin que le sprite acquiert un mouvement, il faut modifier cette valeur d'un certain pas. Ceci peut être exécuté par le processeur directement, au moyen de l'instruction *MOVE*. On doit cependant être prudent, car il peut apparaître des problèmes si on modifie les mots de contrôle à n'importe quel moment :

Le processeur modifie le premier mot et avant qu'il n'ait le temps de modifier le deuxième, le contrôleur *DMA* lit les deux valeurs. Comme ces deux dernières ensemble ne conviennent plus, l'affichage sera altéré.

On peut éviter ceci en modifiant les mots de contrôle pendant le *temps mort vertical* (pendant l'interruption *'écran vide vertical'*, après que le *COPPER* ait initialisé *SPRxPT*).

La priorité sprites/playfields

La priorité entre les sprites et un playfield détermine qui sera affiché entre ou derrière les éléments de l'écran. Le sprite avec la plus grande priorité se trouve devant tous les autres et ne peut être dissimulé. La priorité est déterminée par le numéro du sprite. Le numéro 0 correspond à la plus grande priorité.

Pour les playfields, c'est un bit de contrôle qui détermine lequel sera devant l'autre.

Le dernier cas à traiter est celui de la priorité entre les sprites et le playfield. Sur l'Amiga, il est possible de positionner un playfield entre n'importe quel sprite. Lors de la détermination de la priorité du *playfield* sur les sprites, ces derniers seront rassemblés et traités par paires. Ce sera à nouveau les mêmes combinaisons utilisées pour les sprites à 15 couleurs, c'est-à-dire, une paire formée du sprite à numéro impair et de son prédécesseur à numéro pair :

```
Sprite 0&1, sprite 2&3, sprite 4&5, sprite 6&7
```

On peut considérer ces quatre paires sous forme d'un tableau. Les trous correspondent aux points transparents des *bitplanes* ou des sprites et aux parties de l'écran, que la taille du sprite ne permet pas de cacher. La rangée d'un tel tableau n'est pas modifiable. On peut, par contre, insérer deux autres éléments, les playfields, à n'importe quelle place entre les paires de sprites. Cinq positions sont alors possibles :

Position	Suite des priorités				
0	PLF	SPR0&1	SPR2&3	SPR4&5	SPR6&7
1	SPR0&1	PLF	SPR2&3	SPR4&5	SPR6&7
2	SPR0&1	SPR2&3	PLF	SPR4&5	SPR6&7
3	SPR0&1	SPR2&3	SPR4&5	PLF	SPR6&7
4	SPR0&1	SPR2&3	SPR4&5	SPR6&7	PLF

sprites.

BPLCON2 $104 (écriture seulement)

Bit n°	: 15-7	6	5	4	3	2	1	0
Fonction	: inutilisé	PF2PRI	PF2P2	PF2P1	PF2P0	PF1P2	PF1P1	PF1P0

PF2PRI

Si ce bit est activé, le playfield 2 s'affiche devant le playfield 1.

PF1P0 à PF1P2

Ces trois bits forment un nombre 3 bits, qui correspond à la position du playfield n°1 (bitplanes impairs) entre les quatre paires de sprites. Seules les valeurs 0 à 4 sont possibles.

PF2P0 à PF2P2

Ces trois bits correspondent à la même fonction que les trois bits *PF1Px*, mais cette fois-ci pour le playfield n°2 (bitplanes pairs).

Exemple :

```
BPLCON2 = $0003
```

Ceci signifie que le playfield n°1 se trouve devant le n°2, que *PF2Px* = 2, *PF1Px* = 3. La suite logique entre les différents éléments est donc :

```
PLF2 SPR0&1 SPR2&3 SPR4&5 PLF1 SPR6&7
```

Cette suite est donc paradoxale, étant donné que le bit *PLF2PRI* est à 0 et que le n°1 devrait précéder le n°2, mais elle est juste.

dernier se tient deva e n°1, tout en respectant sa priorité. Etant donné que le n° trouve devant le n°2, les sprites visibles se tiennent à leur place, même si ceux-ci sont derrière le playfield n°2. Si ce dernier se trouve à une place commune d'un sprite, il couvrira suivant le rang de priorité le(s) sprite(s) d'une façon normale. Il en résulte donc que la priorité *playfield/playfield* passe avant la priorité *playfield/sprite*.

Si on n'utilise pas le mode *Dual-Playfield*, il n'y aura qu'un seul playfield qui sera formé d'un ensemble de bitplanes pair(s) et impair(s). Les bits *PLF2PRI* et *PL2Px* n'auront alors plus aucune fonction.

Collisions entre éléments graphiques

Il est souvent très utile de savoir si deux sprites entrent en collision l'un avec l'autre ou avec un élément de l'arrière plan. Ceci facilite, par exemple, la reconnaissance d'un coup au but dans un programme de jeu.

Lorsque les points de deux sprites se chevauchent à une position donnée sur l'écran, c'est-à-dire qu'ils possèdent les mêmes coordonnées tout en n'étant pas transparents, il y a collision. Cette dernière est aussi possible entre deux playfields ou un champ et un sprite.

Toutes les collisions reconnues sont mises en mémoire dans le registre de données collision *CLXDAT* :

CLXDAT $00E (lecture seulement)

Bit n°	Collision entre
15	inutilisé
14	sprite 4 (ou 5) et sprite 6 (ou 7)
13	sprite 2 (ou 3) et sprite 6 (ou 7)
12	sprite 2 (ou 3) et sprite 4 (ou 5)

10	sprite 0 (ou 1) et sprite 4 (ou 5)
9	sprite 0 (ou 1) et sprite 2 (ou 3)
8	playfield 2 (bitplane pair) et sprite 6 (ou 7)
7	playfield 2 (bitplane pair) et sprite 4 (ou 5)
6	playfield 2 (bitplane pair) et sprite 2 (ou 3)
5	playfield 2 (bitplane pair) et sprite 0 (ou 1)
4	playfield 1 (bitplane impair) et sprite 6 (ou 7)
3	playfield 1 (bitplane impair) et sprite 4 (ou 5)
2	playfield 1 (bitplane impair) et sprite 2 (ou 3)
1	playfield 1 (bitplane impair) et sprite 0 (ou 1)
0	playfield 1 et playfield 2

Alors que chaque point non transparent d'un sprite peut libérer une collision, on peut choisir librement dans le(s) champ(s) de jeu, la couleur du point qui caractérisera la reconnaissance d'une collision. De plus, il est possible d'autoriser ou non le test de collision pour tous les sprites à numéro impair. Toutes ces possibilités sont issues des bits du registre de contrôle collision *CLXCON*.

CLXCON $098 (écriture seulement)

Bit n°	Nom	Fonction
15	ENSP7	test de collision autorisé pour le sprite 7
14	ENSP5	test de collision autorisé pour le sprite 5
13	ENSP3	test de collision autorisé pour le sprite 3
12	ENSP1	test de collision autorisé pour le sprite 1
11	ENBP6	comparaison entre le bitplane 6 et MVBP6
10	ENBP5	comparaison entre le bitplane 5 et MVBP5
9	ENBP4	comparaison entre le bitplane 4 et MVBP4
8	ENBP3	comparaison entre le bitplane 3 et MVBP3
7	ENBP2	comparaison entre le bitplane 2 et MVBP2
6	ENBP1	comparaison entre le bitplane 1 et MVBP1
5	MVBP6	valeur pour la collision avec le bitplane 6
4	MVBP5	valeur pour la collision avec le bitplane 5

2	MVBP3	va r pour la collision avec le bitplane 3
1	MVBP2	eur pour la collision avec le bitplane 2
0	MVBP1	valeur pour la collision avec le bitplane 1

Les bits *ENSPx* (enable sprite x) déterminent si les collisions des sprites correspondants, à numéro impair, seront testées ou non. Par exemple, si le bit *ENSP1* est activé, la collision entre le sprite 1 et un autre sera enregistrée. Le bit correspondant du registre de données *collision* sera alors activé, comme pour le sprite 0. Il n'est donc pas possible de savoir, par l'intermédiaire du contenu du registre, lequel des deux sprites, 0 ou 1, a libéré la collision. Retenez bien cela dans l'utilisation des sprites.

Si on a combiné deux sprites, en vue de former un sprite 15 couleurs, le bit correspondant *ENSPx* doit être activé, afin de permettre un test de collision correct.

Dans un playfield, on peut établir soi-même quelle combinaison de bitplane est en mesure de libérer une collision ou non. Les bits *ENBPx* (enable bitplane x) déterminent quel bitplane sera sélectionné pour le test de collision. Si tous les bits *ENBPx* sont activés, la collision est possible avec tous les points. Les combinaisons de bits correspondent aux bits *MVBPx* (Match Value Bitplane x).

Les bits *ENBPx* déterminent si les bits des plans x doivent être comparés avec les valeurs des bits *MVBPx*. Si les bits de tous les plans d'un point concordent avec les bits *MVBPx*, ce point pourra libérer une collision. Pour plus de clarté, voici un exemple :

Les bits *ENBPX* sont activés, tout comme les bits *MVBPx*. Chaque point du playfield pourra libérer une collision, si sa combinaison de bits est 111111. Si seuls les trois bits inférieurs sont activés, une collision pourra être libérée lorsque les points du playfield auront la combinaison 000111.

combinaisons de bits suivantes, 000111, 000110, 000100 ou 000101, les bits *MVBPx* doivent correspondre au nombre suivant : 000100.

En effet ces deux bits inférieurs devront toujours exécuter les modalités de la collision, du fait de leur état à 0, la combinaison des bits ENBPx devant quant à elle correspondre à 111100.

Exemple de combinaison de bits possibles :

ENBPx	MVBPx	Collision possible
111111	111111	111111
111111	111000	111000
111100	1111xx	111100, 111101, 111110, 111111
011111	x00000	000000, 100000
000000	xxxxxx	collision possible pour toutes combinaisons de bits

(La valeur du bit caractérisé par x n'a pas d'importance).

Si tous les bitplanes ne sont pas actifs, les bits *ENBPx* des plans inutilisés doivent être mis à 0.

Les différentes possibilités de combinaisons des bits *ENBPx* et *MVBPx* permettent une grande quantité de reconnaissances de collisions différentes. On peut, par exemple, initialiser le registre *CLXCON* d'une façon telle que seuls les sprites à points rouges et verts peuvent entrer en collision avec le playfield.

Il est aussi possible d'autoriser une collision, lorsque les points transparents du playfield 1 chevauchent les points de couleur noir du playfield 2 etc...

Il existe pour chaque sprite, en dehors des registres *SPRxPT*, 4 autres registres. Les données qu'ils contiennent leur sont communiquées automatiquement par le contrôleur DMA. Mais il est aussi possible d'y accéder via le processeur.

SPRxPOS	premier mot de contrôle
SPRxCTL	deuxième mot de contrôle
SPRxDATA	premier mot de données d'une ligne (mot low)
SPRxDATB	deuxième mot de données d'une ligne (mot high)

(x correspond à un des numéros 0-7 possibles de sprite. Les adresses de ces registres se trouvent dans la liste des registres du chapitre 1.5.1).

Le contrôleur DMA écrit directement les deux mots de contrôle d'un sprite dans deux registres : *SPRxPOS* et *SPRxCTL*. Lorsqu'une valeur sera écrite dans le registre *SPRxCLT*, que ce soit par l'intermédiaire du DMA ou du 68000, la sortie sprite de DENISE sera désactivée. Le sprite ne sera plus envoyé à l'écran.

Le contrôleur DMA attend alors la ligne établie dans *VSTART*. Puis il écrit les deux premiers mots de données dans les registres *SPRxDATA* et *SPRxDATB*. C'est alors que DENISE activera à nouveau la sortie sprite. Il attend alors la position horizontale souhaitée en comparant les registres *SPRxCTL* et *SPRxPOS* avec la colonne actuelle de l'écran, et affiche le sprite à sa bonne place.

Le contrôleur DMA écrit, à chaque nouvelle ligne, deux nouveaux mots de données dans *SPRxDATA/B*, jusqu'à ce que la dernière ligne du sprite (VPOS) soit passée. Puis il prend les prochains mots de contrôle et les place dans *SPRxPOS* et *SPRxCTL*. Ainsi, le sprite sera à nouveau désactivé, jusqu'à ce que la position *VSTART* soit atteinte. Si ces deux mots de contrôle étaient nuls, le contrôleur DMA termine ses accès DMA sprites, par le biais des canaux correspondants, jusqu'au début de l'affichage de la prochaine image. A la fin du *temps mort vertical*, il recommencera à l'adresse se trouvant actuellement dans *SPRxPT*.

On peut afficher un sprite sans canaux DMA, sans problème. Il faut alors écrire directement les mots de contrôle souhaités dans les registres *SPRxPOS* et *SPRxCTL*. La position *HSTART* et le bit *AT* devront obligatoirement renfermer des valeurs valides. *VSTART* et *VSTOP* seront utilisés exclusivement par les canaux DMA.

On peut activer la sortie sprite, à n'importe quelle ligne, dès qu'on écrit les deux mots de données dans *SPRxDATA/B*. Comme *SPRxDATA* active la sortie sprite, il est préférable d'écrire le deuxième mot de données dans *SPRxDATB* en premier. Si on ne modifie pas les données des deux registres, elles seront affichées à chaque ligne, pour devenir un montant vertical.

Si on désire désactiver à nouveau le sprite, il suffit d'écrire n'importe quelle valeur dans *SPRxPOS*.

1.5.7 LE BLITTER

Le nom *Blitter* est un raccourci pour l'expression anglaise : *'BLock Image Transferrer'*, ce qui signifie *'transfert de blocs de données image'*. Son nom traduit exactement le rôle qu'il joue au sein de l'Amiga, c'est-à-dire, décaler et copier des blocs de données en mémoire, ces données étant pour la plupart, de type graphique. Le *Blitter* peut aussi réunir plusieurs zones mémoires de façon logique et replacer le résultat en mémoire. Il exécute toutes ces tâches très rapidement. Il peut réaliser un décalage de données en traitant plus de 16 millions de données dans la même seconde.

De plus, le *Blitter* peut aussi remplir des surfaces et tirer des lignes. La combinaison de ces deux possibilités autorise le dessin de n'importe quelle figure à plusieurs côtés et ceci beaucoup plus rapidement que n'aurait pu le faire le 68000.

Le système d'exploitation utilise le *Blitter* pour pratiquement toutes les opérations graphiques. Ce dernier s'occupe de l'affichage du texte, de dessiner les gadgets, de décaler les fenêtres etc...

il a aussi la responsa[illegible]te du décodage des [illegible] prouve que le *Blit*[illegible]e se limite pas aux données graphiques.

L'utilisation du blitter dans la copie de données

Le *Blitter* copie toujours les données suivant le même modèle : une à trois zones mémoires différentes, qui représentent la source des données, seront réunies, suivant les conditions logiques choisies et le résultat sera à nouveau réécrit en mémoire. Ce dernier peut aller d'une simple copie à une union complexe de plusieurs zones de données. Les adresses des zones sources de données se nomment A, B et C, la zone cible étant D. Celles-ci peuvent être choisies librement dans toute la *CHIP-RAM* (adresses de 0 à $7FFFF).

Le nombre de mots qui peuvent être traités en une opération du *Blitter*, peut atteindre 65536. Il peut donc y avoir transfert en mémoire de 128 Koctets en une seule fois.

Le *Blitter* forme donc des zones mémoires *'à angle droit'*, c'est-à-dire que la mémoire est découpée en colonnes et en lignes, comme pour une *bitmap*. Il est aussi possible de traiter une petite zone incluse dans une grande *bitmap*, en utilisant des valeurs *modulo*.

Ces dernières ont aussi été utilisées dans la définition des bitplanes d'un playfield et avaient alors la largeur de fenêtre d'écran.

Pour initialiser une opération du *Blitter*, les démarches suivantes sont importantes :

- Choix du mode du *Blitter* : copie des données.
- Sélection de la zone de données *source* (l'utilisation de trois zones mémoires n'est pas obligatoire) et de la zone *cible*.
- Choix de la liaison logique.

- Définition d'autres paramètres (scrolling, sques, sens d'adressage).

- Détermination de la fenêtre dans laquelle l'opération *Blitter* sera exécutée, et démarrage du *Blitter*.

Etablir une fenêtre Blitter

Vous vous demandez peut-être pourquoi nous commençons par la description du dernier point de la suite de démarches énoncées plus haut. En fait, la définition de la fenêtre souhaitée est l'élément de base de toutes les autres opérations. Mais lorsqu'on programme le *Blitter*, cette valeur sera écrite à la fin dans le registre correspondant, étant donné que le *Blitter* débutera ses opérations avec cette valeur. C'est pour cette raison que ce point se trouve à la fin de la liste. Mais pour la compréhension des autres valeurs, il est cependant nécessaire de connaître la notion de fenêtre *Blitter*.

La fenêtre *Blitter* est la zone mémoire traitée par le *Blitter* lors de ses opérations. Elle est structurée comme un bitplane, c'est-à-dire découpée en lignes et colonnes, où une colonne correspond à un mot. Le nombre de mots dans une fenêtre est donc le produit des lignes par les colonnes : *L*C*.

A travers ce découpage de la zone mémoire, le traitement des bitplanes par le *Blitter* est grandement facilité et ceci d'une telle manière qu'on peut parler ici de *zones mémoires linéaires*.

Le découpage en lignes et colonnes n'est là que pour faciliter la programmation. En vérité, l'ordre de toutes les lignes en mémoire correspond à une suite régulière d'adresses. Pour un petit champ de données, qui n'est pas découpé en lignes et colonnes, il est possible d'initialiser une fenêtre d'une taille 1.

Le *Blitter* traite sa fenêtre ligne par ligne. Les opérations *Blitter* débutent avec le premier mot de la première ligne et finissent avec le dernier mot de la dernière ligne.

Le registre *BLTSIZE* renferme la taille de la fenêtre.

BLTSIZE $058 (écritu seulement)

Bit n° :	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Fonction :	H9	H8	H7	H6	H5	H4	H3	H2	H1	H0	W5	W4	W3	W2	W1	W0

H0-H9 Ces dix bits reproduisent la hauteur (*Height*) de la fenêtre *Blitter*. La fenêtre peut avoir une hauteur variant de 1 à 1024 lignes (2^{10} = 1024). La hauteur maximale est sélectionnée en fixant *Height* à 0. Toutes les autres valeurs correspondent directement au nombre de lignes. Une hauteur de 0 ligne est impossible.

W0-W5 Ces six bits représentent la largeur (*Width*) de la fenêtre. Cette dernière peut comporter de 1 à 64 mots (2^6 = 64). Si on convertit cette valeur en points graphiques (1 mot = 16 points), on obtient un maximum de 1024 points. Ce dernier est sélectionné si on met *Width* à 0.

Pour trouver la valeur du registre *BLTSIZE* à partir de la hauteur et de la largeur, on se sert de la formule suivante :

```
BLTSIZE = Hauteur*64 + Largeur
```

Si on veut atteindre les deux maxima *Height = 1024* et *Width = 64*, on est obligé de la modifier quelque peu.

```
BLTSIZE = (Hauteur AND $3FF)*64 + (Largeur AND $3F)
```

Le registre *BLTSIZE* devra toujours être initialisé en dernier. Le *Blitter* démarrera automatiquement lors de l'accès écriture à ce registre.

Zone de données Source et Cible

Lors d'une opération *Blitter*, les données issues de zones mémoire différentes seront reliées entre elles. Même lorsque la fenêtre *Blitter* établit le nombre et l'ordre de traitement des données, le positionnement des trois *zones sources* et de la *zone cible* de cette fenêtre doit être déterminé.

Prenons pour exemple un petit graphique à angles [illegible], que le *Blitter* doit copier dans la mémoire d'écran et qui se trouve à un endroit quelconque de la *CHIP-RAM*. Il n'y aura, pour cette simple tâche, qu'une *zone source*. Le choix de la fenêtre *Blitter* est simple. Le graphique devant être copié entièrement, la taille de la fenêtre correspondra à la hauteur et à la largeur du graphique en mémoire.

Pour que le *Blitter* sache aussi où ce graphique se trouve, il suffit d'écrire l'adresse du premier mot de la ligne supérieure dans le registre correspondant.

La définition de la *zone cible* est sensiblement différente. Le graphique devant être copié dans la mémoire d'écran, il sera obligatoirement transféré dans le *bitplane* actuel. Le problème est que le *bitplane* est plus étendu que notre simple graphique.

La copie directe dans le *bitplane* par le *Blitter*, pourrait paraître quelque peu curieuse. Ainsi, à côté de l'adresse de la *zone cible*, la largeur du graphique devra aussi être communiquée au *Blitter*, au moyen d'une valeur *modulo*. Cette dernière sera additionnée à l'adresse du pointeur, après chaque traitement de ligne. Par conséquent, les mots non influencés du *bitplane* seront sautés et le pointeur se tiendra à nouveau au début de la prochaine ligne. Comme la *zone mémoire source* et la *zone cible* peuvent avoir une largeur différente, il existe pour chaque zone, un registre *modulo* indépendant.

Recopier un graphique dans un BitPlane

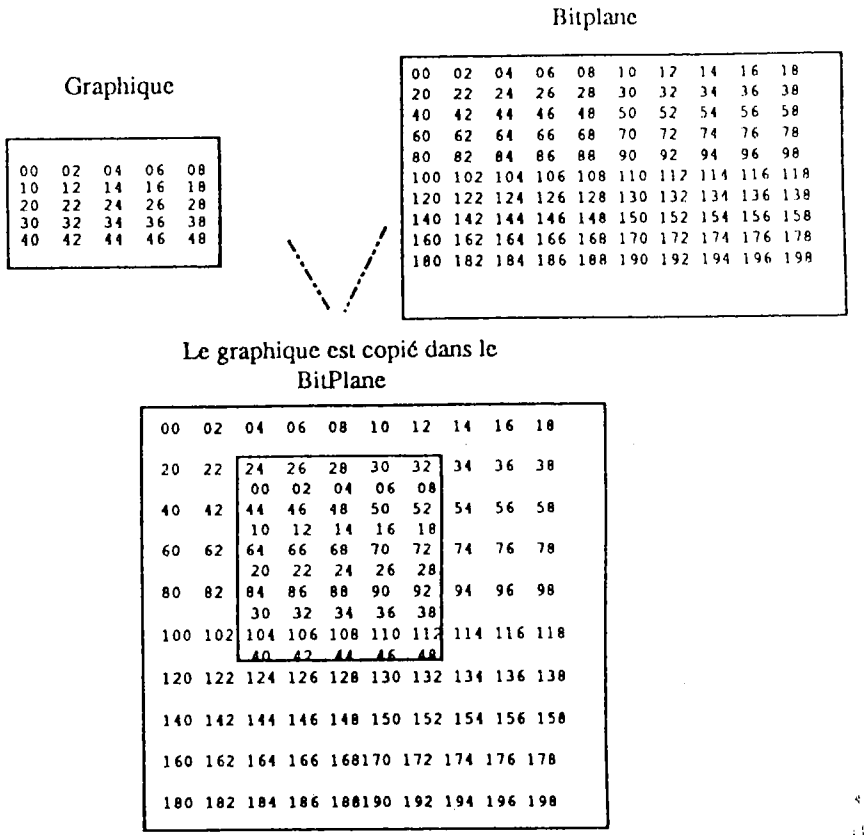


Figure 1.5.7.1

Ce schéma illustre bien notre exemple. Le graphique est composé de 5 lignes d'une largeur de 10 mots. Le *bitplane* a une hauteur de 10 lignes sur 20 mots. Le choix de la fenêtre *Blitter* de l'adresse de départ et de la valeur *modulo* se fera de la manière suivante :

La fenêtre *Blitter* doit correspondre au graphique pour que ce dernier puisse être copié entièrement, c'est-à-dire que la hauteur de la fenêtre sera de 5 lignes et que la largeur sera de 10 mots. La valeur qui sera mise en dernier dans le registre *BLTSIZE*, correspondra à 330 ou, en valeur hexadécimale, à $014A.

L'adresse de départ de la source de données est la même que celle du premier mot de données du graphique. Etant donné que la largeur d'une ligne du graphique est la même que celle de la fenêtre *Blitter*, la valeur *modulo* de la source reste à 0.

Il faut maintenant déterminer la valeur *modulo* de la *zone cible de données*. On établit pour cela la différence entre la largeur de la *zone cible* et de la fenêtre *Blitter*. Dans notre exemple, 20 mots - 10 mots : la valeur *modulo* de la *zone cible* correspondra à 10 mots. Dans le registre *modulo* du *Blitter*, la valeur *modulo* doit être donnée en octets : *Valeur modulo = Modulo en mots* *2.

En dernier, le *Blitter* nécessite encore l'adresse de départ de la *zone cible*. Elle détermine la position à laquelle le graphique sera copié dans le *bitplane*. Elle correspond donc à la somme de l'adresse de départ du *bitplane* et de l'adresse du mot, qui doit être placé dans le coin gauche supérieur du graphique. Sur le schéma, l'adresse de départ de la *zone cible* correspond à l'adresse du *bitplane* plus 24.

Déroulement d'une opération Blitter

Une fois que les adresses et la valeur *modulo* ont été établies, le *Blitter* commence la copie des données, après initialisation du registre *BLTSIZE*. Il prend le mot se trouvant à l'adresse de départ de la source de données et le met en mémoire à partir de l'adresse cible. Puis il additionne un mot aux deux adresses et copie le prochain mot. Ceci sera répété jusqu'à ce que le nombre de mots par ligne, contenus dans *BLTSIZE*, soit traité. Avant que le *Blitter* ne recommence à la prochaine ligne, il additionne la valeur *modulo* au pointeur d'adresses, afin que la prochaine ligne débute bien à la bonne adresse.

Lorsque toutes les lignes ont été copiées, le *Blitter* se désactive et attend le prochain transfert.

Le registre *adresse* renferme, après une opération du *Blitter*, l'adresse du dernier mot copié deux, plus la valeur *modulo*.

Les registres *adresses* se nomment *BLTxPT*, où x correspond à l'une des trois sources A, B ou C, ou à la *zone cible* D. Le registre d'adresses est formé d'un registre renfermant les bits 0-15 et d'un autre renfermant les bits 16-18 :

Adresse	Nom	Fonction	
048	BLTCPTH	Adresse de départ de la	(bits 16-18)
04A	BLTCPTL	zone source de données C.	(bits 0-15)
04C	BLTBPTH	Adresse de départ de la	(bits 16-18)
04E	BLTBPTL	zone source de données B.	(bits 0-15)
050	BLTAPTH	Adresse de départ de la	(bits 16-18)
052	BLTAPTL	zone source de données A.	(bits 0-15)
054	BLTDPTH	Adresse de départ de la	(bits 16-18)
056	BLTDPTL	zone cible de données D.	(bits 0-15)

Chacune des quatre zones possède un registre *modulo* propre :

060	BLTCMOD	Valeur modulo pour la source C
062	BLTBMOD	Valeur modulo pour la source B
064	BLTAMOD	Valeur modulo pour la source A
068	BLTDMOD	Valeur modulo pour la zone cible D

Copie par incrémentation ou décrémentation d'adresse

Dans notre exemple, le *Blitter* travaille par *incrémentation d'adresse*, c'est-à-dire qu'il commence à une adresse de départ, qui sera augmentée continuellement d'un certain pas et ce jusqu'à l'adresse de fin. Logiquement, cette dernière est plus haute que l'adresse de départ.

Il se peut qu'un tel adressage nous mène à l'erreur : en effet, la copie d'une zone mémoire à une haute adresse peut avoir pour conséquence le chevauchement partiel des zones *source* et *cible*.

Voici un exemple :

Adresse	Données sources	Données cibles	Résultat Souhaité	Résultat Accidentel
0	source 1			
2	source 2			
4	source 3			
6	source 4	cible 1	source 1	source 1
8	source 5	cible 2	source 2	source 2
10		cible 3	source 3	source 3
12		cible 4	source 4	source 1 !!
14		cible 5	source 5	source 2 !!

Les 5 mots de données *source* doivent être transférés à l'adresse de données *cible*. Si le *Blitter* commence à la source 1, il écrira sur la source 4, lorsqu'il aura mis la source 1 en mémoire à l'adresse *cible* souhaitée. Ceci résulte du fait que cible 1 et source 4 possèdent les mêmes adresses, les deux zones se chevauchant. Il se passe la même chose pour source 5 et cible 2.

Lorsque le *Blitter* arrive à l'adresse de la source 4, il y trouvera la source 1. Ainsi cette dernière sera chargée à la place de la source 4 dans la zone cible 4. Ce sera à nouveau identique pour la source 2 dans la zone cible 5, alors que les sources 4 et 5 seront perdues.

Comme solution à ce problème, le *Blitter* possède en plus du mode *ascending* (adressage par incrémentation), le mode *descending* (adressage par décrémentation). Dans ce dernier mode, il débute à l'adresse se trouvant dans le pointeur *BLTxPT*, puis à chaque mot copié, il le diminue de deux octets. La valeur *modulo* ne sera pas additionnée, mais soustraite. L'adresse de fin se trouve donc avant l'adresse de départ.

Il faudra, bien sûr, en tenir compte lors de l'initialisation de *BLTxPT*. En *mode normal* on prend comme référence le coin supérieur gauche.

En *mode descending* on prendra le coin inférieur droit, étant donné que l'adressage se fera par décrémentation.

Les valeurs *modulo* et *BLTSIZE* se déterminent de la même façon que celles du mode *ascending*.

Suivant les modes choisis, on peut faire les remarques suivantes :

1) Aucun chevauchement entre les zones *source* et *cible* :

   Que ce soit en mode *ascending* ou *descending*, les deux travaillent correctement dans ce cas.

2) Chevauchement partiel entre les zones *source* et *cible* : (la zone *cible* se trouve avant la zone *source*).

   Seul le mode *ascending* travaille correctement.

3) Chevauchement partiel entre les zones *source* et *cible* : (la zone *cible* se trouve derrière la zone *source*).

   Seul le mode *descending* travaille correctement.

La sélection des liaisons logiques

Comme cela a déjà été vu, il peut y avoir trois sources de données qui seront reliées pour donner les données *cible*. Ce lien logique ne se déroule que bit à bit, c'est-à-dire que les trois bits de données A, B et C doivent donner le bit cible D.

Le *Blitter* connaît 256 différentes liaisons. Elles peuvent se différencier en deux groupes :

1) On peut appliquer 8 équations booléennes différentes sur les trois bits de données. Chacune libère comme résultat un 1, après combinaison de A, B et C.

2) Les huit résultats des équations, citées plus haut, seront réunis par un *OU* logique. Le résultat sera le bit cible D.

La notion *d'équation booléenne* correspond à une f ıule mathématique qui reproduit une combinaison d'une liaison l ,,e.

Cette façon de calculer est appelée *algèbre booléenne*, du mathématicien *Georges BOOLE (1815 à 1864)*. Les explications à venir des fonctions logiques du *Blitter* ne peuvent être comprises sans quelques connaissances d'algèbre booléenne. Les équations booléennes seront toutefois exposées.

Trois bits donnent huit combinaisons possibles. Chacune des huit équations révèle un résultat vrai (résultat = 1) dans une des combinaisons. On peut choisir, au moyen de huit bits de contrôle LF0 à LF7, si le résultat de l'équation doit être sélectionné ou non, pour former D. Chaque résultat sous forme de bit, les bits correspondants étant à 1, seront liés entre eux par un *OU* logique. Une liaison *OU* signifie que le résultat est 1, si au moins une entrée est à 1. Autrement dit, un *OU* logique libère un 0 lorsque toutes les entrées sont à 0.

Au moyen des 8 bits LFx, on peut ainsi choisir par quelle combinaison des trois bits d'entrée A, B et C, on obtient le bit de sortie D, égal à 1.

Les 8 équations booléennes d'entrée sont caractérisées en anglais par le terme, *'Minterms'*.

Le tableau suivant donne un aperçu des combinaisons d'entrée, obtenues avec les bits LFx (dans la colonne *Minterm*, le caractère minuscule désigne une liaison *NOT* du bit d'entrée correspondant. En temps normal, cet état est caractérisé par une barre horizontale au dessus du caractère).

La colonne *'bits d'entrée'* renferme les combinaisons de bits telles qu'elles seront exécutées par les équations correspondantes. L'ordre des bits est le suivant : A, B puis C.

	LF7	LF6	LF5	LF4	LF3	LF2	LF1	LF0
Minterm :	ABC	ABc	AbC	Abc	aBC	aBc	abC	abc
Bits d'entrée :	111	110	101	100	011	010	001	000

Le choix d'un *Minte* est simple. Il suffit d'activer tous les bits LFx, de telle façon que le combinaison d'entrée forme le bit de sortie D égal à 1.

Dans notre premier exemple, les données *sources* A devaient être copiées directement dans la zone *cible* D. Les *sources* B et C ne sont pas utilisées. La sélection du *Minterm* se passera donc de la manière suivante :

D doit être à 1, lorsque A = 1. Seuls les 4 termes supérieurs obéissent à ces conditions, étant donné que A est à 1. Comme B ne joue aucun rôle, on peut choisir deux termes, où B peut être soit à 1 soit à 0, qui sont cependant identiques. B n'a plus aucune conséquence sur D, le reste de l'équation n'étant en rien influencé par les deux états possibles de B, et le résultat ne dépendant que de ce reste. Il se passe la même chose pour le bit C. Si on se réfère au tableau, on remarque que les 4 termes, LF4 à LF7, doivent être activés. Le résultat ne dépendra que de A, car après chaque combinaison de B et C, une de ces quatre équations sera toujours vrai pour A = 1 et D sera égal à 1. Si A = 0 les quatre équations seront fausses et D sera égal à 0.

Si vous avez quelques connaissances en algèbre booléenne, vous pouvez obtenir le *Minterm* suivant une approche différente. La formule à obtenir est A=D. Etant donné que B et C sont toujours présents dans le *Blitter*, vous devez les intégrer dans l'équation de la manière suivante :

```
A*(b+B)*(c+C)=D
```

Le terme x+X est toujours vrai (égal à 1) et sera utilisé lorsque la valeur de x sera sans importance pour le résultat D. Afin d'obtenir le *Minterm*, on est obligé de développer la multiplication :

1. A*(b+B)*(c+C)=D
2. (A*b+A*B)*(c+C)=D
3. A*b*c + A*B*c + A*b*C + A*B*C = D

En enlevant le signe multiplicateur, on obtient :

```
Abc + ABc + AbC + ABC = D
```

Il ne reste plus alors qu'à activer les bits LFx des *Minterms* correspondants. Comme on peut le voir, on accède aussi au résultat par le biais de l'algèbre booléenne.

Voici d'autres exemples de combinaisons des bits LFx, utiles pour les opérations du *Blitter* :

- Inversion d'une zone de données : a=D

  Combinaison LFx à obtenir : 00001111

  En algèbre booléenne :
  ```
  a=D
  a*(b+B)*(c+C)=D
  (a*b+a*B)*(c+C)=D
  abc + aBc + abC + aBC = D
  ```

- Copie d'un graphique dans un *bitplane*, sans modifier son contenu ; ceci correspond à un *OU* logique entre le graphique A et le *bitplane B* : A+B = D.

  Combinaison LFx à obtenir : 11111100

  En algèbre booléenne :
  ```
  A + B = D
  A(b+B)(c+C) + B(a+A)(c+C) = D
  (Ab+AB)(c+C) + (Ba+BA)(c+C) = D
  Abc+AbC+ABc+ABC+Bac+BaC+BAc+BAC = D
  Abc+ABc+AbC+ABC+aBc+aBC = D
  ```

Pour finir, voici encore les règles d'obtention des bits LFx nécessaires :

1) Trouver par laquelles des 8 combinaisons de ABC, on obtient D=1.

2) Activer les bits LFx de la combinaison correspondante.

3) Si les trois sources ne sont pas indispensables, toutes les combinaisons peuvent être choisies, dans lesquelles les bits inutilisés apparaissent et où les bits souhaités possèdent la bonne valeur.

Décalage des valeurs d'entrée

Pour certaines tâches, le fait que le *Blitter* soit relié au mot *limite* peut poser un problème. Il peut arriver, par exemple, qu'une zone déterminée se trouvant à l'intérieur d'une *bitmap* doit être décalée d'un certain nombre de points. Dans ce cas, le *Blitter* ne peut déplacer les bits de données que sur la partie d'un mot. Autrement le *Blitter* est obligé d'écrire un graphique à une position déterminée de la mémoire d'écran, les coordonnées ne concordant évidemment pas avec une limite de mot.

Pour résoudre ce problème, le *Blitter* a la possibilité de décaler les mots de données des sources A et B de 15 bits vers la droite. Il est alors en situation de mettre les données à chaque position de point souhaitée. Tous les bits qui seront décalés vers la droite par ce processus de déplacement, arriveront dans le mot suivant, où les bits ont été libérés.

Pratiquement toute la ligne sera décalée. Ce déplacement se nomme *décalage en cylindre.* Le *Blitter* ne nécessite pas de temps en plus pour ce genre de processus et ceci indépendamment du nombre de bits décalés. Le décalage de données ne limite en aucun cas la vitesse du *Blitter*.

Exemple de décalage de données sur trois bits :

AVANT

Mot de données 1	Mot de données 2	Mot de données 3
00011111 10011100	00010101 01111111	11100001 11100101

APRES

Mot de données 1	Mot de données 2	Mot de données 3
xxx00011 11110011	10000010 10101111	11111100 00111100

Les trois bits xxx dépendent du mot de données pré[illegible]ent, d'où ils proviennent.

Les masques

Il est possible que le *Blitter* soit obligé de copier un graphique de la mémoire d'écran, dont les bords ne correspondent pas au mot *limite*. Les données qui se trouvent à gauche de la bordure du graphique, tout en appartenant aux premiers mots de données, ne doivent pas être copiées. Afin de rendre ceci possible, le *Blitter* peut traiter le premier et dernier mot de données d'une ligne avec un masque. Ceci signifie qu'on peut choisir les bits de ces mots, qui seront pris en charge. On pourra ainsi éliminer les bits indésirables des bordures.

Cette possibilité de masquer n'existe que pour la source A. Deux registres contiennent les masques des deux bordures. Seuls les bits activés seront pris en compte, les autres étant éliminés.

$044 BLTAFWM Blitter Source A First Word Mask
(Masque pour le premier mot de données)

$046 BLTALWM Blitter Source A Last Word Mask
(Masque pour le dernier mot de données de la ligne)

Les bits 0-15 renferment les bits masque correspondants.

Exemple :

('1' correspond à un bit activé, '.' correspond à un bit masqué).

Données graphiques - Bitplane :

Colonne 1	Colonne 2	Colonne 3
........11111111	1111111111111111	1.............11
111111......1111	11..........1111	1111........1111
....11........11	1111.........111	11111....1111111
....11.........1	11111.........11	1111111111111111
....11.........1	11111.........11	1111111111111111
....11........11	1111.........111	11111....1111111
111111......1111	11..........1111	1111........1111
........11111111	1111111111111111	1.............11

Premier mot masque :
0000000011111111

Dernier mot masque :
1111110000000000

Résultat :

Colonne 1	Colonne 2	Colonne 3
........11111111	1111111111111111	1...............
............1111	11..........1111	1111............
..............11	1111.........111	11111...........
...............1	11111.........11	111111..........
...............1	11111.........11	111111..........
..............11	1111.........111	11111...........
............1111	11..........1111	1111............
........11111111	1111111111111111	1...............

Après avoir masqué les éléments indésirables de l'image sur les bordures, on obtient le graphique souhaité.

Si la largeur de la fenêtre d'écran correspond exclusivement à un mot (*BLTSIZE Width=1*), les deux masques sont appliqués ensemble. Ils influencent ensemble le mot d'entrée. Seuls les bits d'entrées, activés dans les deux masques, seront pris en compte.

Les registres de contrôle Blitter

Le *Blitter* possède deux registres de contrôle, *BLTCON0* et *BLTCON1*. On y trouve différents bits de contrôle nécessaires à la gestion du *Blitter*.

BLTCON0 $040

Bit n°	Nom	Fonction
15	ASH3	Ces quatre bits renferment la valeur de décalage
14	ASH2	des données d'entrée de la source A
13	ASH1	ASH0-3 = 0 ne produit aucun décalage
12	ASH0	
11	USEA	Active le canal DMA de la source A
10	USEB	Active le canal DMA de la source B
9	USEC	Active le canal DMA de la source C
8	USED	Active le canal DMA de la zone cible D
7	LF7	Choix Minterm ABC (combinaison de bit ABC: 111)
6	LF6	Choix Minterm ABc (combinaison de bit ABC: 110)
5	LF5	Choix Minterm AbC (combinaison de bit ABC: 101)
4	LF4	Choix Minterm Abc (combinaison de bit ABC: 100)
3	LF3	Choix Minterm aBC (combinaison de bit ABC: 011)
2	LF2	Choix Minterm aBc (combinaison de bit ABC: 010)
1	LF1	Choix Minterm abC (combinaison de bit ABC: 001)
0	LF0	Choix Minterm abc (combinaison de bit ABC: 000)

Bit n°	Nom	Fonction
15	BSH3	Ces quatre bits renferment la valeur
14	BSH2	de décalage des données d'entrée de la source B
13	BSH1	BSH0-3 = 0 ne produit aucun décalage
12	BSH0	
10-5		inutilisé
4	EFE	Exclusive Fill Enable
3	IFE	Inclusive Fill Enable
2	FCI	Fill Carry In
1	DESC	DESC = 1 commute en mode descending
0	LINE	LINE = 1 active le mode lignes

Le bit *LINE* commute le *Blitter* en mode *dessin de ligne*. Si on utilise le *Blitter* pour copier des données, ce bit doit être à 0.

Avec le bit *DESC*, on peut choisir entre le mode d'adressage par *incrémentation* et le mode par *décrémentation*. Si *DESC = 0*, c'est le mode *ascending* qui sera utilisé, le mode *descending* étant utilisé avec le bit *DESC = 1*.

Les bits *EFE* et *IFE* activent la routine de remplissage des surfaces. Les deux doivent être à 0 si on veut utiliser le *Blitter* normalement.

Le bit *FCI* a une fonction liée au mode de remplissage.

Le DMA Blitter

Les données des zones source A, B et C et cible D ont accès à la mémoire, que ce soit en mode lecture ou écriture, au moyen de 4 canaux DMA. On peut activer ces *DMA Blitter* par l'intermédiaire du bit *BLTEN* (bit 6) du registre *DMACON*. Les 4 canaux seront activés ensemble. Le *Blitter* possède pour ses transferts DMA, 4 registres de données :

\$000	BLTDDAT	Sortie de données D
\$070	BLTCDAT	Registre de données de la source C
\$072	BLTBDAT	Registre de données de la source B
\$076	BLTADAT	Registre de données de la source A

Le contrôleur DMA lit les données d'entrées nécessaires dans la mémoire et les écrit dans le registre de données. Si le *Blitter* a traité les données d'entrée, c'est *BLTDDAT* qui contiendra le résultat. Le contrôleur DMA transfèrera alors le contenu de ce registre dans la *CHIP-RAM*.

On commute le transfert DMA des 4 registres au moyen des bits *USEx*. Si *USEA=0*, on désactivera par exemple, le canal DMA sur le registre A. Le *Blitter* continuera à accéder à la valeur dans *BLTADAT*. Autrement dit, à chaque nouveau mot de la source active, le même mot sera issu de la source A. Pour cette raison, il est préférable de mettre les bits *USEx* à 0 pour les sources inutilisées, ainsi que d'exclure toute influence de celles-ci dans la sélection correspondante du *Minterm*.

Une autre possibilité est l'utilisation consciente du suivant, qu'après désactivation du canal DMA, ce sera toujours le même mot de données qui sera utilisé. On pourra ainsi, par exemple, remplir la mémoire suivant un modèle que l'on a écrit directement dans le registre *BLTxDAT*, à l'aide du processeur.

En dehors de *BLTEN*, 3 autres bits du registre *DMACON* appartiennent au *Blitter* :

Bit 10 BLTPRI

Le rôle de ce bit a déjà été expliqué dans le chapitre **'Eléments de base'**. S'il est à 1, le *Blitter* a la priorité absolue sur le processeur.

BBUSY signale l'état du *Blitter*. S'il est à 1, cela signifie qu'une opération se déroule à ce moment-là.

Après activation de la fenêtre *Blitter* dans *BLTSIZE*, le *Blitter* débute ses accès DMA tout en activant *BBUSY*, jusqu'à ce que le dernier mot de la fenêtre soit traité et à nouveau réécrit en mémoire. Il termine alors ses accès et désactive *BBUSY*.

En même temps, le bit *Blitter-Finished* sera initialisé dans le registre *Interrupt-Request*.

Bit 13 BZERO

Le bit *BZERO* indique si, lors d'une opération *Blitter*, tous les bits résultats sont nuls. En d'autres termes, *BZERO* est initialisé lorsque, dans tous les mots de données, aucune des liaisons choisies ne libère un 1 comme résultat. Avec l'aide de ce bit, on peut organiser un test de collision. Il suffit pour cela d'activer les *Minterms* d'une telle manière que D ne soit égal à 1 que lorsque les deux sources sont à 1. Si les graphiques se chevauchent dans les deux zones sources, même d'un seul point, le résultat sera 1. A la fin de l'opération *Blitter*, on pourra ainsi déterminer s'il y a eu collision ou non. *USED* sera initialisé à 0, afin que les données de sortie ne soient pas écrites en mémoire.

Utilisation du Blitter pour remplir des surfaces

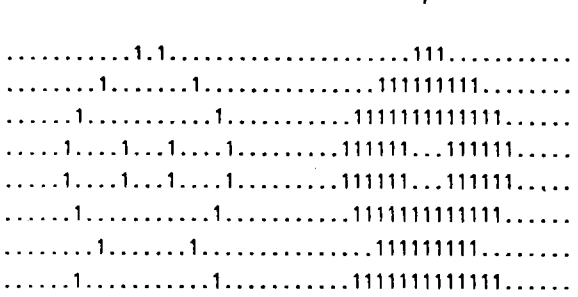
Le *Blitter* entend par surface une zone mémoire bidimensionnelle, qu'il doit remplir avec des points. Normalement, cette surface appartient à un graphique ou à un *bitplane*.

Afin de pouvoir remplir une surface, le *Blitter* doit connaître ses limites. Une définition compréhensible des lignes limites est absolument nécessaire pour le *Blitter*.

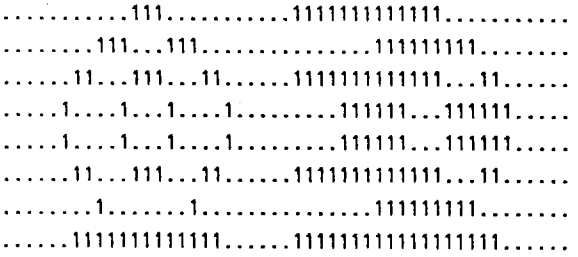
De nombreuses fonctions de remplissage existent d[illegible]s la plupart des programmes de dessin, ou par exemple e[illegible]Amiga-Basic avec l'instruction *PAINT*.

Avec elles, une zone de l'écran sera remplie à partir d'un point de départ, et ce jusqu'à ce que le programme rencontre une ligne frontière. Ainsi, n'importe quelle surface se laissera colorier, avec pour seule restriction qu'elle soit entièrement fermée par une ligne continue. Le *Blitter* n'a pas la possibilité d'exécuter une opération de remplissage aussi complexe. Il travaille seulement ligne par ligne, en remplissant les espaces vides entre deux bits activés, qui représentent les limites de la surface désirée. Les deux exemples suivant montre le déroulement du processus de remplissage du *Blitter* :

Opération de remplissage sans erreur :



Opération de remplissage avec erreurs, du fait du mauvais choix des bits de limite.



Dans le premier exe[illegible]ple, une surface correctement limitée sera correctement remp[illegible] n'est évidemment pas le cas sur la figure n°2. On a dessiné ici une limite entièrement close et continue. Si on cherche à remplir une telle figure par le *Blitter*, il n'en résultera que le chaos.

La raison réside dans l'algorithme du *Blitter*. Il est assez simple. Le *Blitter* commence sur le côté droit de la ligne.

Pour déterminer si un bit doit être activé, il utilise le bit *FillCarry (FC)*. Normalement, il est à zéro au départ. Le *Blitter* teste alors la valeur du bit le plus à droite. S'il est nul, la valeur du bit FC reste inchangée, et le bit de sortie prendra cette valeur, c'est-à-dire 0. Dans ce cas, le *Blitter* continuera avec le bit voisin, jusqu'à ce qu'il rencontre un bit d'entrée initialisé. Le bit FC sera mis à 1 et comme le bit de sortie correspond à la valeur actuelle de FC, il sera aussi mis à 1. Dès que le *Blitter* rencontrera à nouveau un bit initialisé, le bit FC sera à nouveau effacé et mis à 0. De cette manière, une zone se trouvant entre deux bits activés sera toujours remplie. Comme on peut le remarquer sur le deuxième exemple, la présence d'un nombre impair de bits perturbe la logique de remplissage.

La valeur de départ du bit FC détermine le bit *FCI (FillCarryIN)* dans le registre *BLTCON1*. Si FCI n'est pas activé, tout se déroule comme cela a été expliqué plus haut. Si FCI = 1, le *Blitter* commence à remplir la ligne, dès la bordure, jusqu'à ce qu'il rencontre un bit d'entrée activé. Le processus de remplissage se déroulera alors de la manière inverse.

Exemple du rôle du bit FCI :

Sur les exemples ci-dessus, les bits d'entrée, c'[illegible] à-dire les limites des bordures, restent présent après le rempl[illegible]ge de l'image. C'est toujours le cas lorsqu'on sélectionne le mode de remplissage par initialisation du bit *ICE (Inclusive Fill Enable)* dans le registre *BLTCON1*.

L'inverse s'obtient avec le mode *ECE (Exclusive Fill Enable)* et sera sélectionné en initialisant le bit de même nom dans le registre *BLTCON1*. Avec lui, le bit limite gauche d'une bordure d'une surface remplie (toujours quand le bit *Fill-Carry* change de 1 à 0) ne sera pas pris en compte dans l'image de sortie. La surface de cette dernière sera donc réduite d'un point. Il n'est possible qu'en mode *ECE* d'obtenir une surface d'une largeur d'un bit. Ceci n'est pas possible en mode *ICE* puisqu'au minimum deux points sont nécessaires pour former les limites d'une figure qui puisse s'afficher sans problème sur la sortie image.

Différences entre les modes *ICE* et *ECE* :

Sortie image	ICE	ECE
........11......11	........11......11	.........1.......1
......1...1....1.1	......11111....111	.......1111.....11
...1...11..1..1..1	...111111111..1111	....1111.111...111
1......11...11...1	1111111111111111	.1111111.1111.1111
..1.............1	..11111111111111	...11111111111111
....1...........1.	....111111111111.	.....111111111111.
......1...1...11..	......11111...11..	.......1111....1..

Déroulement bit à b[illegible] des différentes opérations de remplissage :

Modèle d'entrée : 11010010

Bit Nr.	bit d'entrée	FCI = 0 FC	FCI = 0 ICE	FCI = 0 ECE	FCI = 1 FC	FCI = 1 ICE	FCI = 1 ECE
-	11010010	FC=FCI			FC=FCI		
0	0	0	0	0	1	1	1
1	1	1	10	10	0	11	01
2	0	1	110	110	0	011	001
3	0	1	1110	1110	0	0011	0001
4	1	0	11110	01110	1	10011	10001
5	0	0	011110	001110	1	110011	110001
6	1	1	1011110	1001110	0	1110011	0110001
7	1	0	11011110	01001110	1	11110011	10110001

(*FC=FCI* signifie que le bit *FC* accepte la valeur du bit *FCI* issue de *BLTCON1*, avant le début du processus de remplissage).

Voici la façon de démarrer une opération de remplissage du *Blitter* :

Le *Blitter* peut exécuter une opération de remplissage en même temps qu'un processus de copie. Il sera initialisé après avoir sélectionné l'un des deux modes, *ICE* ou *ECE* dans le registre *BLTCON1*. Le *Blitter* formera comme toujours, à partir des trois sources A, B et C et du *Minterm* choisi, les données de sortie D. Si aucun des deux modes de remplissage n'est actif, le *Blitter* prend en charge ces données directement dans son registre de données de sortie (*BLTDDAT, $000*), les données accédant à la mémoire via les canaux DMA, lorsque *USED = 1*.

En mode de remplissage, les données de sortie D seront utilisées comme données d'entrée du circuit de remplissage. Le résultat de l'opération de remplissage sera écrit alors dans le registre de données de sortie *BLTDDAT*.

Pour exécuter une opération de remplissage, les démarches suivantes sont nécessaires :

- *BLTxPT, BLTMOD* et les *Minterms* doivent être choisis de telle manière que les données de sortie D doivent renfermer les bits corrects limites de la surface à remplir.

- Il faut choisir le mode *descending* (le *Blitter* rer t de la droite vers la gauche, ceci ne fonctionnant que l que les mots ont des adresses décrémentées).

- Il faut choisir le mode de remplissage souhaité, c'est-à-dire activer *ICE* ou *ECE*.

- *LINE = 0* (mode *lignes* désactivé).

- *BLTSIZE* initialisé à la taille du graphique à remplir.

Le *Blitter* débute alors le processus de remplissage. Lorsqu'il aura terminé, il mettra le bit *BLTBUSY* à 0.

La vitesse du *Blitter* ne sera pas influencée par l'activité du mode de remplissage. Le *Blitter* a la possibilité de remplir une surface avec une vitesse maximum de 16 millions de points par seconde. La principale utilisation du mode de remplissage est le dessin de figures pleines. Celles-ci seront dessinées dans une zone mémoire libre au moyen du mode *lignes*. Le *Blitter* remplira alors ces graphiques avec plus de rapidité.

Utilisation du Blitter pour tracer des lignes

Le *Blitter* offre de nombreuses possibilités. En dehors de la capacité à copier des données et à remplir des surfaces, il possède un mode de dessin de lignes. Comme les autres modes du *Blitter*, il est assez rapide : jusqu'à 1 million de points par seconde. Seule l'utilisation du processeur 68020 peut permettre d'atteindre cette vitesse, et encore, avec peine. Pour le 68000 cette vitesse est impossible.

Pour dessiner une ligne, on doit relier deux points entre eux par un certain nombre de ces unités.

Etant donné que la rés tion du graphisme d'ordinateur est limitée, on ne peut pas touj s choisir le point optimum. Les points réels se trouvent toujours un peu au-dessus ou au-dessous du point idéal. Une telle représentation de lignes ressemble à un escalier. Plus la résolution sera importante, plus les marches seront petites, mais on ne pourra jamais les éliminer totalement.

Exemple d'une ligne dans un graphisme d'ordinateur :

Les deux points	reliés entre eux forment une droite.
.....	.....
.....1...	.....111...
.....	.....111.....
.....	.....1111.....
.....	.....111.....
.....1.....	.....111.....
.....	.....

Le *Blitter* a la possibilité de dessiner de telles lignes, d'une longueur maximum de 1024 points. Mais on ne peut malheureusement pas donner tout simplement les coordonnées des deux extrémités. On doit définir la ligne *Blitter* correctement.

Celui-ci nécessite les octants à l'endroit où se trouve la ligne. Le partage des coordonnées en 8 parties, *les octants*, se retrouve dans plusieurs processeurs graphiques.

Schéma du choix des octants cor' :ts

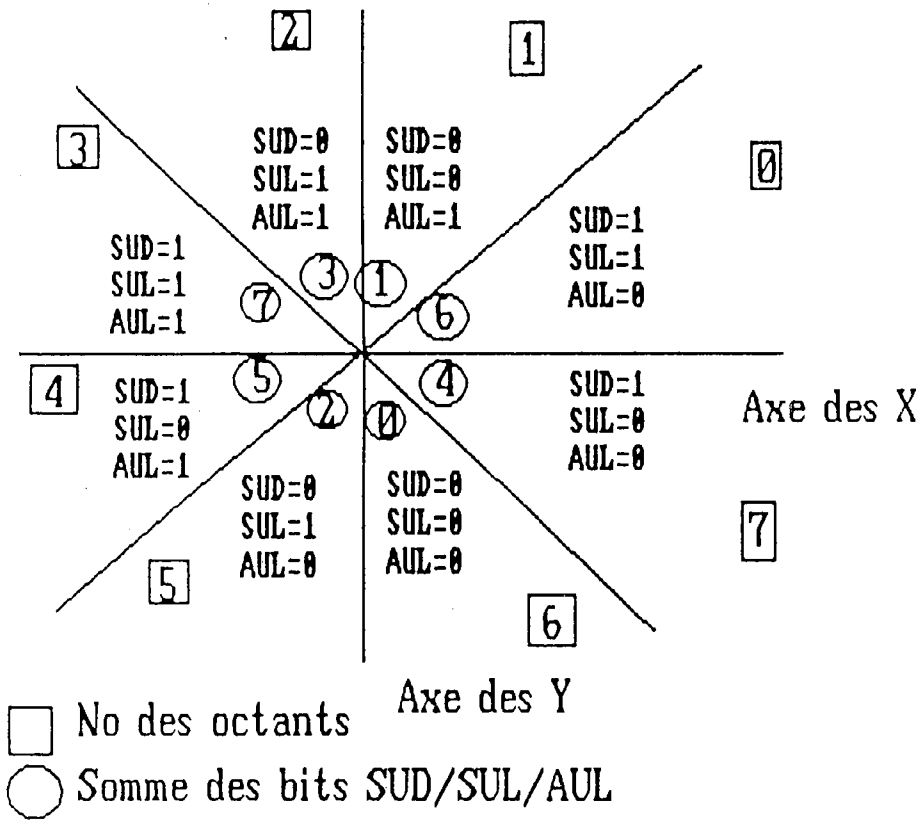


Figure 1.5.7.2

Schéma d'exemple d'une ligne

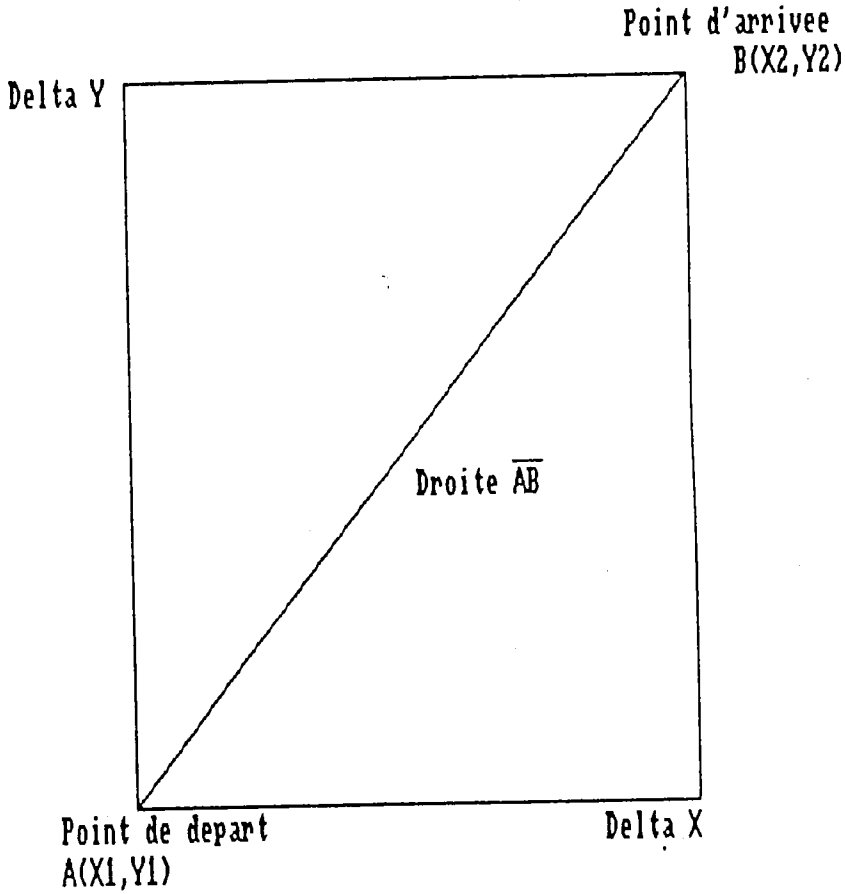


Figure 1.5.7.3

Le schéma du choix des octants nous montre cette répartition. Le point de départ d'une ligne se trouve à l'origine des axes de coordonnées (point de croisement des deux axes). Le point d'arrivée se trouve dans un des 8 octants. Au moyen de trois comparaisons logiques, on peut caractériser le numéro de ces octants, ainsi X1 et Y1 correspondent aux coordonnées du point de départ, X2 et Y2 étant les coordonnées du point d'arrivée :

Lorsque X1 est inférieur à X2, le point d'arrivée peut se trouver dans un des octants suivants : 0, 1, 6 ou 7. Si X1 est supérieur à X2 il se trouvera dans un des octants suivants : 2, 3, 4 ou 5. Si X1 et X2 sont égaux, il se trouve sur l'axe des Y, tous les octants étant alors possibles.

De la même façon, si Y1 est inférieur à Y2, les octants possibles du point d'arrivée sont 0, 1, 2 ou 3. Lorsque Y1 est supérieur à Y2, les octants possibles seront 4, 5, 6 ou 7. Y1 = Y2 indique que tous les octants sont possibles.

A partir des deux dernières égalités, on peut caractériser les différences des X et Y : *DeltaX* = $|X2-X1|$, *DeltaY* = $|Y2-Y1|$. Lorsque *DeltaX* est supérieur à *DeltaY*, le point d'arrivée se trouve dans un des octants suivants : 0, 3, 4, ou 7. Si *DeltaX* est inférieur à *DeltaY*, il se trouvera dans un des octants suivants : 1, 2, 5 ou 6. *DeltaX=DeltaY* indique que tous les octants sont possibles.

De ces trois comparaisons, on pourra déterminer dans quel octant précisément, se trouve le point d'arrivée. Si ce point se trouve sur une ligne frontière entre deux, l'octant choisi n'a pas d'importance.

Détermination de l'octant :

Coordonnées de point	Octant	Code	Coordonnées de point	Octant	Code
Y1 <= Y2 X1 <= X2 DeltaX>=DeltaY	0	6	Y1 >= Y2 X1 >= X2 DeltaX>=DeltaY	4	5
Y1 <= Y2 X1 <= X2 DeltaX<=DeltaY	1	1	Y1 >= Y2 X1 >= X2 DeltaX<=DeltaY	5	2
Y1 <= Y2 X1 >= X2 DeltaX<=DeltaY	2	3	Y1 >= Y2 X1 <= X2 DeltaX<=DeltaY	6	0
Y1 <= Y2 X1 >= X2 DeltaX>=DeltaY	3	7	Y1 >= Y2 X1 <= X2 DeltaX>=DeltaY	7	4

Le chiffre dans la colonne *'Code'* correspond au chiffre entouré d'un cercle sur le schéma du choix des octants. Le *Blitter* nécessite, pour chaque octant dans lequel se trouve le point d'arrivée de la ligne, une combinaison spéciale de trois bits.

Ces bits se nomment *SUD* (Sometimes Up or Down), *SUL* (Sometimes Up or Left) et *AUL* (Always Up or Left). *'Code'* est en fait le nombre résultant de la combinaison de ces trois bits (*SUD = MSB, AUL = LSB*).

Pour programmer une ligne, on doit tout d'abord découvrir l'octant du point d'arrivée, puis écrire dans le *Blitter* la valeur *Code* correspondante.

Lignes avec modèles

Le *Blitter* utilise un masque pour dessiner une ligne, afin de déterminer si les points d'une ligne doivent être activés, effacés ou affichés suivant un modèle. Le masque a une largeur de 16 bits, celui-ci se répétant sur la ligne tous les 16 points. L'emploi d'un modèle et son influence sur l'affichage ne peuvent être expliqués au mieux que par quelques exemples :

('.' = 0, '1' = 1, A = point de départ, B = point d'arrivée)

Sortie image : Masque = '1111111111111111'

```
.........11111111......B.
.......111......111......
.....11............11....
....11..............11...
....11..............11...
.....11............11....
.......111......111......
..A......11111111........
```

```
.........11111111....11B.
.......111......11111....
.....11.........11.11....
....11.......111....11...
....11....111.......11...
.....11111..........11....
.....11111......111......
..A11....11111111........
```

Les bits nuls du masque indiquent que les points correndants de la ligne seront effacés :

Sortie image : Masque = '0000000000000000'

```
....1111111111111111..B.        ....1111111111111111..B.
....1111111111111111....        ....1111111111111.......
....1111111111111111....        ....11111111111..111....
....1111111111111111....        ....111111111...11111....
....1111111111111111....        ....111111...11111111....
....1111111111111111....        ....1111..1111111111....
....1111111111111111....        ....1...111111111111....
..A.1111111111111111....        ..A..111111111111111....
```

La combinaison de 0 et de 1 dans un masque donne un modèle de ligne :

Masque = '1111111000111000'

```
.A111111........................................................
..........111...1...............................................
................111111..........................................
........................111...11.................................
..................................11111.........................
.....................................111...111..................
.....................................................1111...B
```

Dessin d'une ligne frontière

Dans le chapitre concernant le remplissage d'une surface par le *Blitter*, il est expliqué que les lignes frontières d'une surface doivent toujours avoir la largeur d'un pixel. Si on dessine un telle ligne avec le *Blitter*, il peut arriver que plusieurs points se trouvent sur la même ligne horizontale. Afin d'éviter cela, le *Blitter* se laisse manipuler d'une telle façon qu'il dessine la ligne avec seulement un point par position horizontale :

Ligne normale	Ligne avec un point/position hor.
.....1111	.....1...
.....1111....	.....1.....
.....1111.....	.....1.....
....1111.....	....1.....
1111.....	1.....

Définition de la pente

Afin que le Blitter sache où il doit tracer sa droite, il est nécessaire de définir la pente d'une droite. La façon la plus simple est de se baser sur trois termes issus des valeurs *delta X* et *delta Y*, valeurs qui ont été détaillées lors de l'explication des octants (*delta Y* et *delta X* reproduisent la largeur et la hauteur d'un rectangle dont la diagonale correspond à la droite. Cf schéma 1.5.7.3).

On devra tout d'abord comparer les deux valeurs afin de savoir quelle est la plus petite et quelle est la plus grande. La plus petite sera appelée *Pdelta*, la plus grande *Gdelta*. Puis le Blitter exécutera les opérations suivantes :

1) 2 * Pdelta
2) 2 * Pdelta - Gdelta
3) 2 * Pdelta - 2 * Gdelta

Le Blitter possède de plus un drapeau SIGN qui sera initialisé à 1 lorsque 2 * *Pdelta* sera inférieur à *Gdelta*.

Fonction des registres dans le mode Droite

Le Blitter utilise, lors du dessin d'une droite, des registres identiques à ceux utilisés lors de la copie des données, les fonctions étant toutefois différentes :

BLTAPTL
: dans ce registre sera écrite la valeur de l'opération 2 * Pdelta - Gdelta.

DLTCPT et BLTDPT

ces deux registres (BLTCPTH et BLTCPTL, BLTDPTH et BLTDPTL) doivent être initialisés avec l'adresse de départ de la droite. Ceci correspond à l'adresse du mot à laquelle se trouve le point de départ de la droite.

BLTAMOD

dans ce registre on trouve le résultat de l'opération 2 * Pdelta - 2 * Gdelta.

BLTBMOD

2 * Pdelta.

BLTCMOD et BLTDMOD

dans ces deux registres modulo, on trouve la largeur de l'image dans laquelle la droite devra être dessinée. Cette demière est toujours sous la forme d'un nombre pair d'octets. Pour un Bitplane normal, comportant 320 points (40 octets), sur l'axe des X, la valeur pour BLTCMOD sera égale à 40.

BLTSIZE

la largeur (Bit 0 à 5) devra être initialisée à 2. La hauteur (Bit 6 à 15) renferme la longueur de la droite en points. Une hauteur de 0 correspond à une droite d'une longueur de 1024 points.
La longueur correcte de la droite est toujours identique à la valeur de Gdelta.
Le dessin de la droite sera toujours démarré après l'écriture dans le registre BLTSIZE. Ce sera donc toujours le dernier registre à initialiser.

BLTADAT

ce registre doit être initialisé avec la valeur $8000.

BLTBDAT

ce registre correspond au masque avec lequel la droite devra être dessinée.

BLTAFWM

ce registre masque est initialisé avec la valeur $FFFF.

BLTCON 0

N° de bit	Nom	Fonction
15	START3	ces 4 bits STARTx correspondent à la position du point
14	START2	de départ de la droite.
13	START1	(BLTCPT/BLTDPT).
12	START0	En règle générale, ceux-ci correspondent aux 4 bits ci-dessous de coordonnée X du point de départ.
11	USEA = 1	cette combinaison des bits USEx est
10	USEB = 0	nécessaire au mode droite.
9	USEC = 1	
8	USED = 1	
7	LF7	ces bits LFx doivent être initialisés avec x
jusqu'à 0	LF0	$CA(D= aC+ AB).

BLTCON1

N° de bit	Nom	Fonction
15	Texture3	Ceci correspond à la valeur nécessaire
14	Texture2	au décalage du masque. En temps
13	Texture1	normal Texture0-3 doit être égal à START
12	Texture0	0-3. Le modèle présent dans le registre masque BLTBDAT commence alors avec le premier point de la droite.
11 à 7 = 0		inutilisé, toujours 0.
6	SIGN	lorsque 2 * Pdelta est inférieur à Gdelta SIGN est Initialisé à 1.
5		inutilisé, toujours 0.
4	SUL	Ces trois bits doivent être initialisés
3	SUD	avec les codes SUL/SUD/AUL des des octants correspondants.
2	AUL	
1	SIGN = 1	trace la droite avec un point par ligne
0	LINE = 1	commute le Blitter en mode tracé de droite

En conclusion voici un exemple de calcul :

Il s'agit de tracer une droite dans un Bitplane. Ce Bitplane possède une taille de 320 points par 200 et se trouve à l'adresse $40000. Le point de départ de la droite possède les coordonnées X=20 et Y=185. Le dernier point se trouve aux coordonnées X=210 et Y=35 (les coordonnées se trouvent dans le coin inférieur gauche du BitPlane).
DeltaX = 190, DeltaY = 150

1ère opération : recherche de l'octant du dernier point

Il s'agit ici d'exécuter trois comparaisons : X1 < X2, Y1 > Y2 et Delta X > Delta Y. Le résultat sera ici l'octant numéro 7 et une valeur pour les codes SUD/SUL/AUL = 4.

2ème opération : adresse du point de départ

Ceci se calcule de la façon suivante :

> l'adresse de départ du Bitplane + (le nombre de lignes - Y1 - 1) * nombre d'octets par ligne + 2*(X1/16).

La partie décimale de la division sera éliminée.

En remplaçant les variables par les chiffres :

> $40000 + (200-185-1)*40 + 2 = $40232

Cette valeur sera mise dans les registres *BLTCPT* et *BLTDPT*. Le nombre d'octets par ligne sera mis dans les registres *BLTCMOD* et *BLTDMOD*.

3ème opération : point de départ de la droite dans START0-3

Opération nécessaire : X1 AND $F

En numérique : START0-3 = 20 AND $F = 4

4ème opération : sur des registres BNTAPTL, BLTAMOD et BLTBMOD

DeltaY < DeltaX, Pdelta = DeltaY et Gdelta = DeltaX

BLTAPTL = 2*Pdelta - Gdelta, ce qui est égal à 2*150-190 = 110

BLTAMOD = 2*Pdelta - 2*Gdelta ce qui est égal 2*150- 2*190 = -80

BLTBMOD = 2*Pdelta = 300

2*Pdelta > Gdelta implique SIGN = 0

5ème opération : longueur de la droite dans BLTSIZE

Longueur = Gdelta = DeltaX = 190

La valeur du registre BLTSIZE se trouve sous la forme suivante :

longueur*64 + largeur. La largeur lors du tracé de la droite doit toujours être initialisée à 2. BLTSIZE est égal à DeltaX * 64 + 2 = 12162, soit $2F82.

6ème opération : Compilation des données pour les deux registres BLTCONx

Il s'agit ici d'initialiser la valeur START dans le registre BLTCON0 ainsi que $CA pour les bits LFx et la valeur 1 0 1 1 dans USEx. Dans notre exemple, l'ensemble donnera $ABCA.

BLTCON1 contient le code de l'octant et les bits de contrôle. Notre droite doit être dessinée en mode normal ce qui implique que SIGN = 0.

LINE doit être initialisé à 1. SIGN sera calculé et dans notre exemple, correspondra à 0. L'ensemble donnera $0011. En langage assembleur, l'initialisation des registres sera fait de la façon suivante :

```
LEA $DFF000,A5 ;Adresse de base Blitter dans A5
MOVE.L #$40232, BLTCPTH(A5) ;Adresse de départ dans BLTCPT
MOVE.L #$40232, BLTDPTH(A5) ;et dans BLTDPT
MOVE.W #40, BLTCMOD(A5) ;Largeur du Bitplane dans BLTCMOD
MOVE.W #40, BLTDMOD(A5) ;et dans BLTDMOD
MOVE.W #110, BLTAPTL(A5)
MOVE.W #-80, BLTAMOD(A5)
MOVE.W #300, BLTBMOD(A5)
MOVE.W #$ABCA, BLTCON0(A5)
MOVE.W #$11, BLTCON1(A5)
MOVE.W #12162, BLTSIZE(A5) ;Début de tracé de la droite par le Blitter
```

Autre mode de tracé

Jusqu'à présent la valeur des bits LFx a toujours été $CA. Ceci permet au point de la droite de suivre exactement le modèle du masque.

Mais il existe aussi d'autres combinaisons LFx.

Afin de bien le comprendre, on doit, en premier lieu, savoir comment les Bits LFx sont interprétés en mode tracé de droite :

Le Blitter ne peut adresser la mémoire que par des mots. En mode tracé de droite, les données d'entrée arrivent sur le canal C du Blitter. A cela se rajoute le registre B correspondant au masque. Le registre A détermine alors les points à tracer parmi les mots lus correspondants aux points de la droite. Celui-ci contient toujours un Bit initialisé, qui était décalé par le Blitter à la position souhaitée.

Initialisation de LFx avec la valeur $CA que tous les bits, dont les bits A correspondants sont égaux à 0, seront directement pris en charge par la source C. Si par contre A = 1, le bit masque correspondant sera utilisé comme bit *cible*.

Lorsqu'on sait comment utiliser les bits LFx, on peut déterminer d'autres modes de dessin. Par exemple $4A provoque l'inversion de tous les points de la droite.

Les cycles Blitter DM

Comme cela a déjà été expliqué dans le chapitre précédent, le Blitter n'occupe que les cycles de bus pairs. Comme il y a par ailleurs aussi la priorité sur le 68000, il est intéressant de savoir combien de cycles peuvent être occupés par le processeur.

Ceci dépend surtout du nombre de canaux DMA du Blitter (A, B, C et D) qui sont actifs.

Le tableau suivant montre le déroulement d'une opération Blitter dans les 15 combinaisons possibles des canaux DMA du Blitter.

Les caractères A, B, C et D correspondent aux canaux DMA. Le chiffre 1 correspond au 1er mot de l'opération Blitter, le chiffre 2 correspond au 2ème et le chiffre 3 au dernier mot de données. Les deux traits d'union signifient que le cycle de bus n'est pas occupé par le Blitter.

Occupation des cycles de bus pairs par le Blitter :

Canaux DMA actifs	Occupation des cycles de bus pairs
aucun	-- -- -- -- -- -- -- -- -- -- -- -- -- --
D	D0 -- D1 -- D2 -- -- -- -- -- -- -- -- --
C	C0 -- C1 -- C2 -- -- -- -- -- -- -- -- --
C D	C0 -- -- C1 D0 -- C2 D1 -- D2 -- -- --
B	B0 -- -- B1 -- -- B2 -- -- -- -- -- --
B D	B0 -- -- B1 D0 -- B2 D1 -- D2 -- -- --
B C	B0 C0 -- B1 C1 -- B2 C2 -- -- -- -- --
B C D	B0 C0 -- -- B1 C1 D0 -- B2 C2 D1 -- D2
A	A0 -- A1 -- A2 -- -- -- -- -- -- -- --
A D	A0 -- A1 D0 A2 D1 -- D2 -- -- -- -- --
A C	A0 C0 A1 C1 A2 C2 -- -- -- -- -- -- --
A C D	A0 C0 -- A1 C1 D0 A2 C2 D1 -- D2 -- --
A B	A0 B0 -- A1 B1 -- A2 B2 -- -- -- -- --
A B D	A0 B0 -- A1 B1 D0 A2 B2 D1 -- D2 -- --
A B C	A0 B0 C0 A1 B1 C1 A2 B2 C3 -- -- -- --
A B C D	A0 B0 C0 -- A1 B1 C1 D0 A2 B2 B3 D1 D2

Remarque :

Le tableau précédent est valide lorsque les considérations suivantes sont prises en compte :

1) Le Blitter ne doit pas être dérangé par les accès DMA Copper ou Bitplane.

2) Le Blitter fonctionne en mode normal (soit tracé de droite, soit remplissage de surface).

3) Le bit BLITPRI du registre DMACON est initialisé et le Blitter a la priorité absolue sur le 68000.

Explications :

Comme on peut le voir, il arrive que les données de sortie D0 arrivent dans la RAM après que les données A1, B1 et C1 aient été lues. Ceci provient du mode de traitement du Blitter en *"Pipelining"*. Cela signifie que le traitement des données à l'intérieur du Blitter fonctionne en plusieurs étapes indépendantes les unes des autres. Chaque étape est liée à la sortie de la précédente et à l'entrée de la suivante. La première étape correspond à l'entrée des données (A0, B0, C0) où ces dernières seront traitées et transférées à la deuxième étape. Pendant que ces données sont traitées, dans la deuxième étape, les données suivantes arrivent à la première étape (A1, B1, C1). Quand les premières données arrivent à la dernière étape, l'étape de sortie (D0), le Blitter est déjà en train de traiter les données suivantes. On trouve toujours lors d'une opération Blitter, et à tout moment, deux paires de données à une étape de traitement différente.

Cette étape permet aussi le calcul de la durée de déroulement d'une opération Blitter.

A chaque microseconde, le Blitter a, à sa disposition, deux cycles de bus. S'il doit par exemple copier une zone de 64 kilo Octets (32768 mots) de A vers D, le Blitter nécessite deux fois 32768 cycles.

Lorsque cette même zone doit être de plus combinée avec la source C, le Blitter nécessiter[illegible] * 32768 cycles, étant donné que pour chaque mot, il doit être lu un mot de donnée de la source C. Le tableau montre aussi que le Blitter n'a pas la possibilité d'utiliser tous les cycles de bus lorsque seul un canal DMA est actif. Cette étape permet aussi le calcul de la durée de déroulement d'une opération Blitter.

Exemples de programme

Programme 1 : tracé de droite avec le Blitter

Ce programme met en place une routine de tracé de droite avec le Blitter. Il montre comment calculer les valeurs nécessaires. Le programme est simple : en avant programme, on réservera la mémoire nécessaire et on structurera la liste du Copper. Seule la routine *OwnBlitter* est inconnue. Comme son nom l'indique, on peut, grâce à elle, se réserver le Blitter.

De la même façon, on retrouvera à la fin du programme la routine de désactivation, c'est-à-dire la routine *DisownBlitter* avec laquelle le Blitter retombe à nouveau sous le contrôle du système d'exploitation.

Le programme utilise une seul BitPlane *Hires* avec la résolution standard de 640 * 256 points. Dans la boucle principale, le programme trace des droites qui partent du bord de l'écran, passent par le point central, pour finir sur le côté opposé.

Lorsque l'écran est rempli, le programme décale le masque avec lequel la droite est tracée et recommence.

Remarque :

les coordonnées données dans le programme partent du point 0,0 qui se trouve dans le coin supérieur gauche de l'écran et ne correspondent pas à des coordonnées mathématiques.

```
;*** Tracé de droite par le Blitter

;Registres Custom chip

INTENA  = $9A   ;Registre InterruptEnable (écriture)
DMACON  = $96   ;Registre de contrôle DMA (écriture)
DMACONR = $2    ;Registre de contrôle DMA (lecture)
COLOR00 = $180  ;Registre palette de couleur 0
VHPOSR  = $6    ;Position faisceau (lecture)

;Registre Copper

COP1LC  = $80  ;Adresse de la 1ère liste Copper
COP2LC  = $84  ;Adresse de la 2ème liste Copper
COPJMP1 = $88  ;Saut à la 1ère liste Copper
COPJMP2 = $8a  ;Saut à la 2ème liste Copper

;Registre Bitplane

BPLCON0 = $100 ;Registre de contrôle Bitplane 0
BPLCON1 = $102 ;1 (Valeur de scrolling)
BPLCON2 = $104 ;2 (Sprite<>Priorité playfield)
BPL1PTH = $0E0 ;Pointeur sur le Bitplane 1
BPL1PTL = $0E2 ;
BPL1MOD = $108 ;Valeur Modulo pour BitPlane impair
BPL2MOD = $10A ;Valeur Modulo pour BitPlane pair
DIWSTRT = $08E ;Départ fenêtre écran
DIWSTOP = $090 ;Fin fenêtre écran
DDFSTRT = $092 ;BitPlane DMA Start
DDFSTOP = $094 ;BitPlane DMA Stop

;Registres Blitter

BLTCON0 = $40 ;Registre contrôle Blitter 0
BLTCON1 = $42 ;Registre contrôle Blitter 1
BLTCPTH = $48 ;Pointeur sur source C
BLTCPTL = $4a
BLTBPTH = $4c ;Pointeur sur source B
BLTBPTL = $4e
BLTAPTH = $50 ;Pointeur sur source A
```



```
BLTAPTL = $52
BLTDPTH = $54 ;Po  .eur sur données cible D
BLTDPTL = $56
BLTCMOD = $60 ;Valeur Modulo pour source C
BLTBMOD = $62 ;Valeur Modulo pour source B
BLTAMOD = $64 ;Valeur Modulo pour source A
BLTDMOD = $66 ;Valeur Modulo pour cible D
BLTSIZE = $58 ;Hauteur et largeur de la fenêtre blitter
BLTCDAT = $70 ;Registre de données source C
BLTBDAT = $72 ;Registre de données source B
BLTADAT = $74 ;Registre de données source A
BLTAFWM = $44 ;Masque 1er mot de données source A
BLTALWM = $46 ;Masque 1er mot de données source B

;CIA-A Registre Port A (bouton souris)

CIAAPRA = $bfe001

;Exec Library Base Offsets

OpenLibrary = -30-522 ;LibName,Version/a1,d0
Forbid      = -30-102
Permit      = -30-108
AllocMem    = -30-168 ;ByteSize,Requirements/d0,d1
FreeMem     = -30-180  ;MemoryBlock,ByteSize/a1,d0

;Graphics Library Base Offsets

OwnBlitter    = -30-426
DisownBlitter = -30-432

;graphics base

StartList = 38

;Autres labels

Execbase = 4
Planesize = 80*256 ;Taille Bitplane: 80 octets par 256 lignes
Planewidth = 80
```

```
CLsize = 3*4 ;La liste Copper comporte 3 instructions
Chip   = 2   ;Solliciter la Chip-RAM
Clear  = Chip+$10000 ;Effacer Chip-RAM précédente

;*** Avant programme ***

Start:

;Solliciter la mémoire pour les bitplanes

move.l Execbase,a6
move.l #Planesize,d0 ;mémoire pour le plan
move.l #clear,d1
jsr AllocMem(a6)     ;Sollicitation de la mémoire
move.l d0,Planeadr
beq Ende  ;Erreur! -> Fin

;Solliciter la mémoire pour les bitplanes

moveq #Clsize,d0
moveq #chip,d1
jsr AllocMem(a6)
move.l d0,CLadr
beq FreePlane ;Erreur! -> FreePlane

;Mise en place de la liste Copper

move.l d0,a0            ;Adresse de la liste Copper dans a0
move.l Planeadr,d0      ;Adresse du Bitplane
move.w #bpl1pth,(a0)+   ;Première instruction Copper dans la RAM
swap d0
move.w d0,(a0)+         ;Mot supérieur de l'adresse Bitplane dans la RAM
move.w #bpl1ptl,(a0)+   ;Deuxième instruction dans la RAM
swap d0
move.w d0,(a0)+         ;Mot inférieur de l'adresse Bitplane dans la RAM
move.l #$fffffffe,(a0) ;Fin de la liste Copper

;Solliciter le Blitter

move.l #GRname,a1
```



```
clr.l d0
jsr OpenLibrary(a
move.l a6,-(sp)    ;ExecBase sur la pile
move.l d0,a6       ;GraphicsBase dans a6
move.l a6,-(sp)    ;GraphicsBase sur la pile
jsr OwnBlitter(a6) ;Prise en charge Blitter

;*** Programme général ***

;Bloquer les commutations de tâche et accès DMA

move.l 4(sp),a6 ;ExecBase dans a6
jsr forbid(a6)  ;Désinitialisation du Task-Switching
lea $dff000,a5
move.w #$03e0,dmacon(a5)

;Initialisation du Copper

move.l CLadr,cop1lc(a5)
clr.w copjmp1(a5)

;Choix des couleurs

move.w #$0000,color00(a5)    ;Arrière plan noir
move.w #$0fa0,color00+2(a5)  ;Droites jaunes

;Initialisation du Playfield

move.w #$3081,diwstrt(a5) ;30,129
move.w #$30c1,diwstop(a5) ;30,449
move.w #$003c,ddfstrt(a5) ;Ecran Hires normal
move.w #$00d4,ddfstop(a5)
move.w #%1001001000000000,bplcon0(a5)
clr.w bplcon1(a5)
clr.w bplcon2(a5)
clr.w bpl1mod(a5)
clr.w bpl2mod(a5)

;DMA activé
move.w #$83C0,dmacon(a5)
```

```
;Tracé de lignes

;Détermination des valeurs de départ

move.l Planeadr,a0      ;Paramètre Constant pour DrawLine
move.w #Planewidth,a1   ;dans registre correspondant
move.w #255,a3          ;Taille du Bitplane dans registre
move.w #639,a4
move.w #$0303,d7        ;Modèle de départ

Loop: rol.w #2,d7       ;Décalage du modèle
move.w d7,a2            ;Modèle dans registre pour DrawLine

clr.w d6                ;Effacer la variable de la boucle
BoucleX:
clr.w d1                ;Y1 = 0
move.w a3,d3            ;Y2 = 255
move.w d6,d0            ;X1 = variable boucle
move.w a4,d2
sub.w d6,d2             ;X2 = 639-Variable boucle

bsr DrawLine            ;Tracé de droite

addq.w #4,d6            ;Incrémenter variable de boucle
cmp.w a4,d6             ;Test supérieur ou égal à 639
ble.s boucleX           ;Sinon, boucle continue

clr.w d6                ;Effacer la variable boucle
BoucleY:
move.w a4,d0            ;X1 = 639
clr.w d2                ;X2 = 0
move.w d6,d1            ;Y1 = Variable boucle
move.w a3,d3
sub.w d6,d3             ;Y2 = 255-Variable boucle

bsr DrawLine            ;Tracé de droite

addq.w #2,d6            ;Incrémenter la variable boucle
cmp.w a3,d6             ;Test > 255
ble.s BoucleY           ;Sinon, la boucle continue
```



```
btst #6,ciaapra         ;uton souris ?
bne Loop                ;Non -> continue

;*** Fin du programme ***

;Attente jusqu'à ce que le blitter ait terminé

Wait: btst #14,dmaconr(a5)
bne Wait

;Liste Copper précédente à nouveau activée

move.l (sp)+,a6         ;GraphicsBase pris sur la pile
move.l StartList(a6),cop1lc(a5)
clr.w copJmp1(a5)       ;Liste Copper Startup activé
move.w #$8020,dmacon(a5)
jsr DisownBlitter(a6) ;Blitter libéré
move.l (sp)+,a6         ;ExecBase pris sur la pile
jsr Permit(a6)          ;Task Switching autorisé

;Mémoire libérée pour la liste Copper

move.l CLadr,a1         ;Paramètre pour FreeMem
moveq #CLsize,d0
jsr FreeMem(a6)         ;Mémoire libérée


;Mémoire libérée pour bitplane

FreePlane:
move.l Planeadr,a1
move.l #Planesize,d0
jsr FreeMem(a6)

Fin:
clr.l d0
rts   ;Fin du programme

;Variables
```

```
CLadr:    dc.l 0   ;Adresse de la liste Copper
Planeadr: dc.l 0   ;Adresse du Bitplane

;Constantes

GRname: dc.b "graphics.library",0

even

;*** DrawLine Routine ***
;DrawLine dessine une droite avec le Blitter.
;Les paramètres suivants seront nécessaires :
;d0  =  X1   Coordonnées X du point de départ
;d1  =  Y1   Coordonnées Y du point de départ
;d2  =  X2   Coordonnées X du dernier point
;d3  =  Y2   Coordonnées Y du dernier point
;a0 pointe sur le 1er mot du Bitplane
;a1 contient la largeur du Bitplane (octets)
;a2 est écrit dans le registre masque
;d4 à d6 seront utilisés en tant que registres de traitement

DrawLine:

;Calcul de l'adresse de départ de la droite

move.l a1,d4     ;Largeur du registre de travail
mulu d1,d4       ;Y1 * Nombre d'octets par ligne
moveq #-$10,d5   ;Tracer : $F0
and.w d0,d5      ;4 bits inférieurs de X1 masqués
lsr.w #3,d5      ;Division par 8 du reste
add.w d5,d4      ;Y1 * Nombre d'octets par ligne + X1/8
add.l a0,d4      ;+ adresse de départ du Bitplane
;d4 contient l'adresse de départ
;de la droite
;Calcul de l'octant et des Deltas

clr.l d5         ;Effacer le registre de travail
sub.w d1,d3      ;Y2-Y1  DeltaY dans D3
roxl.b #1,d5     ;Signe de DeltaY dans d5
tst.w d3         ;Flag N restauré
```



```
bge.s y2gy1      ;Si DeltaY positif, branchement à Y2gY1
neg.w d3         ;Inversion DeltaY
y2gy1:
sub.w d0,d2      ;X2-X1  DeltaX dans D2
roxl.b #1,d5     ;Signe de DeltaX dans d5
tst.w d2         ;Flag N restauré
bge.s x2gx1      ;Si DeltaX positif, branchement à X2gX1
neg.w d2         ;Inversion DeltaX
x2gx1:
move.w d3,d1     ;DeltaY dans d1
sub.w d2,d1      ;DeltaY-DeltaX
bge.s dygdx      ;Si DeltaY > DeltaX, branchement à dygdx
exg d2,d3        ;Delta < dans d2
dygdx: roxl.b #1,d5 ;d5 contient le résultat des 3 opérations
move.b Table-Octants(pc,d5),d5 ;prise en compte de l'octant correspondant
add.w d2,d2      ;petit Delta * 2

;Test si le Blitter a déjà terminé la dernière opération

WBlit:  btst #14,dmaconr(a5) ;Test du bit BBUSY
bne.s WBlit                  ;Attente jusqu'à ce qu'il soit à 0

move.w d2,bltbmod(a5)  ;2* PDelta dans BLTBMOD
sub.w d3,d2            ;2* PDelta - GDelta
bge.s signnl           ;Si 2*PDelta > GDelta : saut à signal
or.b #$40,d5           ;Initialisation de SignFlag
signnl: move.w d2,bltaptl(a5) ;2*PDelta-GDelta dans BLTAPTL
sub.w d3,d2            ;2* PDelta - 2* GDelta
move.w d2,bltamod(a5)  ;dans BLTAMOD

;Initialisation des registres

move.w #$8000,bltadat(a5)
move.w a2,bltbdat(a5)  ;Masque issu de a2 dans BLTBDAT
move.w #$ffff,bltafwm(a5)
and.w #$000f,d0        ;4 Bits inférieurs de X1 décalés
ror.w #4,d0            ;vers START0-3
or.w #$0bca,d0         ;Initialisation USEx et LFx
move.w d0,bltcon0(a5)
move.w d5,bltcon1(a5)  ;Octant dans le Blitter
```

```
move.l d4,bltcpth(a5)  ;Adresse de départ de la droite
move.l d4,bltdpth(a5)  ;dans BLTCPT et BLTDPT
move.w a1,bltcmod(a5)  ;Largeur du Bitplane dans
move.w a1,bltdmod(a5)  ;les 2 registres Modulo

;Initialisation de BLTSIZE et démarrage du Blitter

lsl.w #6,d3    ;Longueur * 64
addq.w #2,d3   ;plus (Width = 2)
move.w d3,bltsize(a5)

rts

;Table d'octants avec LINE = 1:
;La table d'octants contient pour chaque octant
;la valeur code correspondante qui est déjà décalée
;à la bonne position. De plus, le bit LINE est initialisé.

Table-Octants :

 dc.b 0 *4+1 ;y1<y2, x1<x2, dx<dy = Oct6
 dc.b 4 *4+1 ;y1<y2, x1<x2, dx>dy = Oct7
 dc.b 2 *4+1 ;y1<y2, x1>x2, dx<dy = Oct5
 dc.b 5 *4+1 ;y1<y2, x1>x2, dx>dy = Oct4
 dc.b 1 *4+1 ;y1>y2, x1<x2, dx<dy = Oct1
 dc.b 6 *4+1 ;y1>y2, x1<x2, dx>dy = Oct0
 dc.b 3 *4+1 ;y1>y2, x1>x2, dx<dy = Oct2
 dc.b 7 *4+1 ;y1>y2, x1>x2, dx>dy = Oct3
```

Programme 2 : Remplissage d'une surface avec le Blitter

Ce programme ressemble beaucoup au premier programme ; il montre comment le Blitter génère un polygone et comment il remplit cette surface.

La partie la plus importante est identique à celle du premier programme, seule la boucle de tracé du polygone ainsi que le remplissage ont été préservés.

La partie à rajouter de .te au commentaire *tracé de droite* jusqu'au commentaire *** *avant-programme* ***. Cette zone devra se mettre à la place et à la ligne où apparaît *partie 1.*

De plus, la table d'octants se trouvant à la fin du programme devra être mise à partir de la ligne où se trouve l'inscription *partie 2.*

La nouvelle table d'octants est nécessaire étant donné que le Blitter nécessite pour le remplissage d'une surface, une limite possédant un point par ligne. Dans la nouvelle table d'octants, les bits LIMMSIGN sont initialisés en plus. Le programme indiqué par *partie 1* trace deux droites et remplit la zone intermédiaire par le Blitter.

```
;*** Remplissage d'une surface avec le Blitter ***

Partie 1:

;Dessin d'un triangle vide

;Valeur de départ

move.l Planeadr,a0     ;Paramètre Constant pour initialiser
move.w #Planewidth,a1  ;la routine LineDraw
move.w #$ffff,a2       ;Masque = $FFFF -> pas de modèle

;* Droites limites dessinées *

;Droite de 320,10 à 600,230
move.w #320,d0
move.w #10,d1
move.w #600,d2
move.w #230,d3
bsr.L drawline ;Dessin de la droite

;Droite de 319,10 à 40,230
move.w #319,d0
move.w #10,d1
move.w #40,d2
move.w #230,d3
bsr.L drawline ;Dessin de la droite
```

```
;* Remplissage d'une surface *

;Attente jusqu'à ce que le blitter ait terminé le dessin de la dernière
droite

Wline: btst #14,dmaconr(a5) ;Test BBUSY
bne.S Wline

add.l #Planesize-2,a0       ;Adresse des derniers mots
move.w #$09f0,bltcon0(a5)   ;USEA et D, LFx: D = A
move.w #$000a,bltcon1(a5)   ;Initialisation de Fill
move.w #$ffff,bltafwm(a5)   ;inclusive et lastwordmask
move.w #$ffff,bltalwm(a5)
move.l a0,bltapth(a5)       ;Adresse du dernier mot du bitplane
move.l a0,bltdpth(a5)       ;dans le registre pointeur d'adresses
move.w #0,bltamod(a5)       ;Pas de Modulo
move.w #0,bltdmod(a5)
move.w #$ff*64+40,bltsize(a5) ;Démarrage du Blitter

;Attente événement souris

fin: btst #6,ciaapra ;Test bouton souris ?
bne.S fin ;Non -> continue

Fin de la partie 1.

Partie 2:

;Table-Octants avec SING = 1 et LINE = 1:

Table-Octants :

 dc.b 0 *4+3 ;y1<y2, x1<x2, dx<dy = Oct6
 dc.b 4 *4+3 ;y1<y2, x1<x2, dx>dy = Oct7
 dc.b 2 *4+3 ;y1<y2, x1>x2, dx<dy = Oct5
 dc.b 5 *4+3 ;y1<y2, x1>x2, dx>dy = Oct4
 dc.b 1 *4+3 ;y1>y2, x1<x2, dx<dy = Oct1
 dc.b 6 *4+3 ;y1>y2, x1<x2, dx>dy = Oct0
 dc.b 3 *4+3 ;y1>y2, x1>x2, dx<dy = Oct2
 dc.b 7 *4+3 ;y1>y2, x1>x2, dx>dy = Oct3
```


1.5.8 LA SORTIE SO!

Eléments de base de la musique électronique

Lorsqu'on entend quelque chose, que ce soit de la musique, un bruit ou une parole, il s'agit en fait d'une vibration de l'air, l'onde sonore, qui atteint notre oreille. Un instrument de musique normal engendre cette vibration, soit directement, comme par exemple une flûte où on y insufle de l'air, soit indirectement, une partie de l'instrument générant le son, ce dernier étant alors émis dans l'air. C'est ce qui se passe, par exemple, pour tous les instruments à corde.

Un instrument électronique engendre des vibrations électriques dans ses circuits de commutation, qui dans leurs allures, correspondent aux sons souhaités. Ces vibrations deviennent audibles, lorsqu'on les transforme en vibrations sonores, au moyen d'un haut parleur. L'Amiga utilise celui qui est présent dans le moniteur. Malheureusement, du fait de sa taille et de sa qualité, il n'est pas en mesure de convertir les vibrations électriques en ondes sonores identiques. C'est pour cette raison qu'il est préférable de relier l'Amiga à un amplificateur, qui offrira, en même temps qu'un grand confort d'écoute, la stéréophonie.

Voici les paramètres qui établissent le son de l'ordinateur :

Fréquence

La fréquence d'un son détermine si celui-ci est haut ou bas. Physiquement, la fréquence correspond au nombre de vibrations par seconde, l'unité étant le *HERTZ*. Une vibration par seconde = un HERTZ, un *KiloHertz* correspondant à 1000 Hertz. L'oreille humaine perçoit des sons entre 16 et 16000 Hertz. Celui qui possède quelques notions de musique sait que le LA possède une fréquence de 440 Hertz. Le lien entre la fréquence et la hauteur d'un son est le suivant : à chaque octave, la fréquence est doublée. Le LA seconde a donc une fréquence de 880 Hertz, alors que le LA se trouvant une octave en dessous, a une fréquence de 220 Hertz.

La fréquence d'un son ne doit pas être forcément constante. Elle peut, par exemple, osciller périodiquement entre plusieurs Hertzs, pour un même son. Cet effet se nomme *Vibrato*.

Intensité

Le deuxième paramètre d'un son est son intensité. Par ce terme, on comprend l'amplitude d'une vibration. L'unité est ici le *décibel (db)*. La zone audible s'étant de 1 à 120 db. Tous les 10 db environ, l'intensité est doublée. L'intensité sonore est aussi désignée par le terme *pression acoustique*.

L'intensité peut être influencée par de nombreux facteurs. Le plus simple correspond naturellement au bouton d'intensité du moniteur. Sa seule fonction est de modifier l'amplitude des vibrations électriques. Mais la distance entre l'auditeur et le haut-parleur a aussi un effet sur l'intensité. Plus on s'éloigne de ce dernier, plus le son devient faible. De plus, l'installation d'une salle, la présence de portes ouvertes ou fermées ou autres, peut aussi modifier l'amplitude de l'onde sonore. Pour cette raison, l'intensité absolue n'est pas importante, les intensités relatives entre plusieurs sons l'étant beaucoup plus.

Il existe un rapport entre l'intensité d'un son et sa fréquence. La raison est la sensibilité de l'oreille humaine. Les sons hauts et bas sont plus difficilement audibles que les sons moyens, même si physiquement, ils possèdent la même pression acoustique en décibels. Cette zone moyenne de sons s'étend environ de 1000 à 3000 hertz. A l'intérieur de cette zone fréquentielle se trouvent les vibrations de la voix humaine, raison pour laquelle la sensibilité dans cet intervalle est importante.

L'intensité d'un son peut se modifier périodiquement, à l'intérieur d'un milieu connu, cet effet se nommant *Trémolo*. L'allure de l'intensité peut être modifiée entre le début et la fin d'un son. Ainsi, celui-ci peut débuter très fort et s'assourdir progressivement. Mais il peut aussi débuter très fort, puis baisser avec une mesure déterminée et s'arrêter brusquement. Il peut aussi commencer très faiblement et augmenter progressivement son intensité. Il y a, comme on peut le voir, énormément de combinaisons possibles.

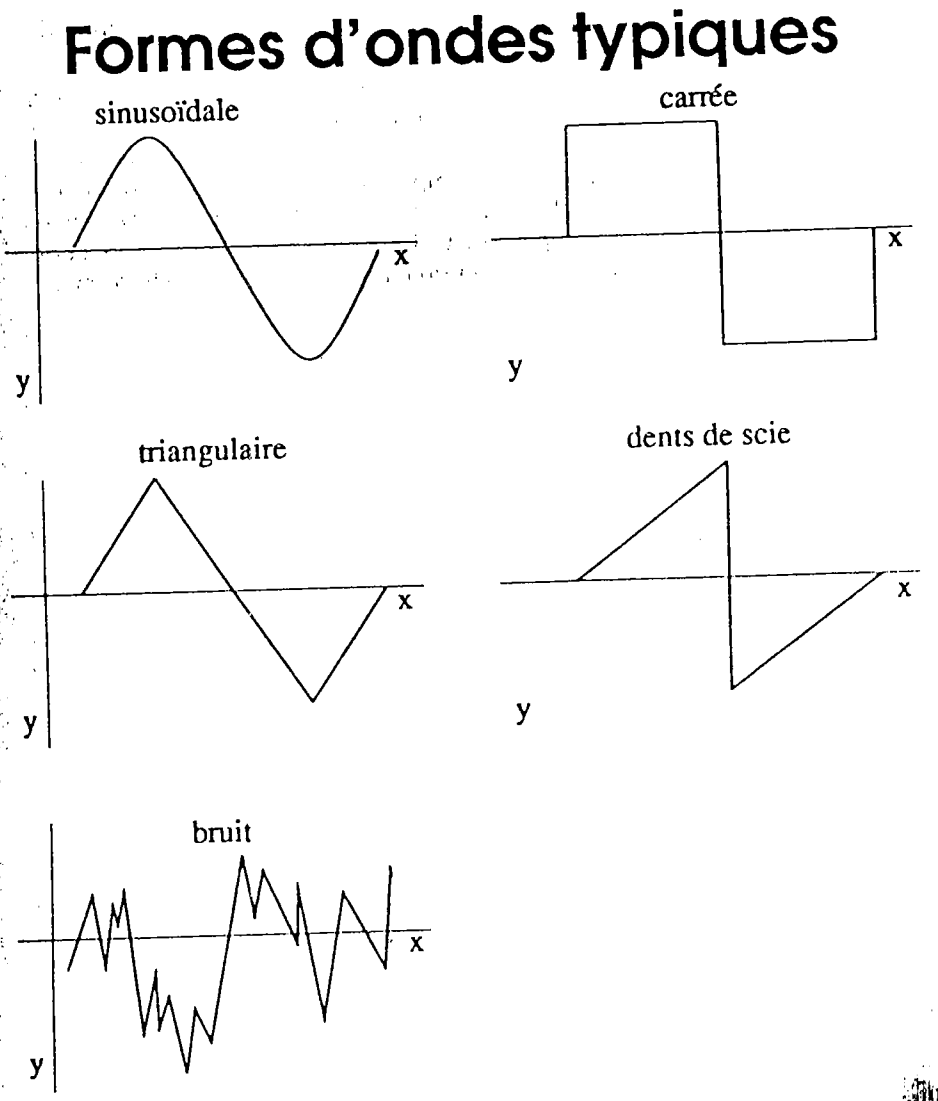


Figure 1.5.8.1

Le troisième et dernier paramètre d'un son est quel peu compliqué. Il s'agit du *timbre*, qui joue un rôle important. Il y a une centaine d'instruments différents qui peuvent jouer un son avec la même fréquence et à la même intensité. Le son résonnera pourtant de manière différente. La raison résidera dans la forme de la vibration. Le schéma précédent montre 4 formes d'ondes importantes. Chacune, quelle qu'elle soit, apparaît comme un mélange d'oscillations sinusoïdales, qui, entre elles, se tiennent à des proportions fréquentielles fixes. Pour une onde carrée, par exemple, la première partie du son est à la fréquence fondamentale, la deuxième a le triple de la fréquence, mais plus qu'un tiers de l'amplitude. La troisième partie du son possède une fréquence multipliée par cinq et un cinquième de l'amplitude etc...

Composition de différentes formes d'ondes, issues de vibrations sinusoïdales

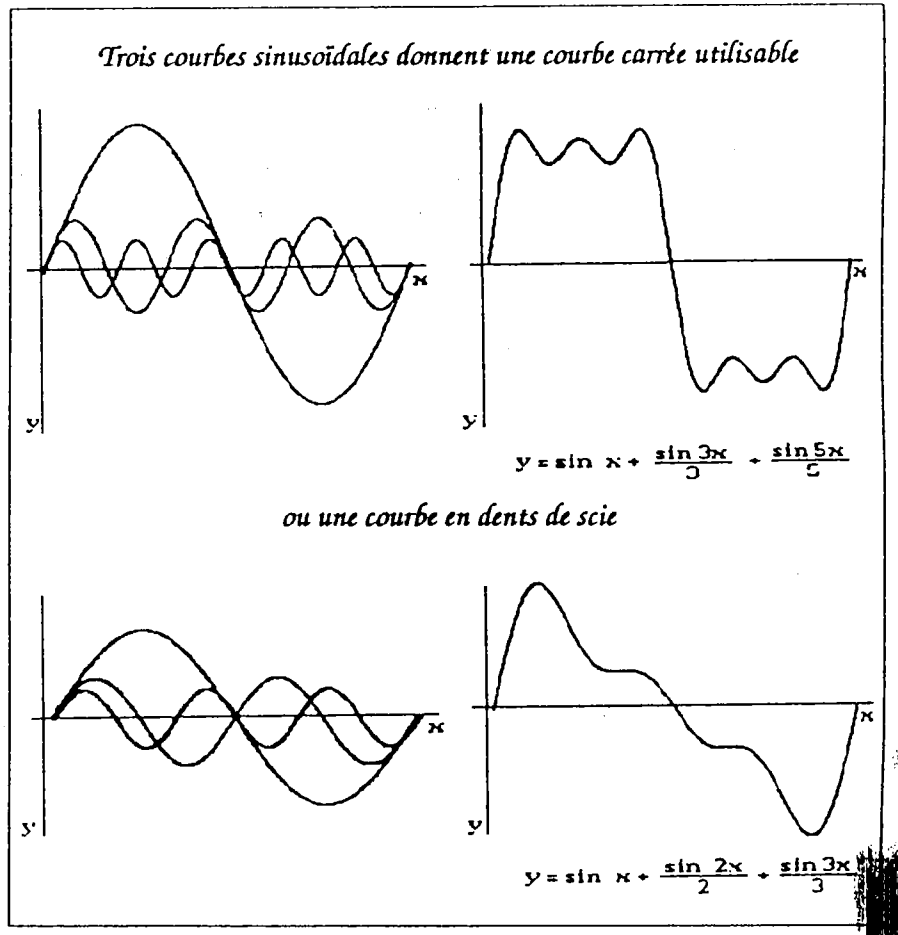


Figure 1.5.8.2

Ce schéma nous montre ainsi les différentes ›mpositions pour l'obtention d'une vibration carrée ou en dents de scie. Pour plus de simplicité, ces ondes ne sont formées que des trois premières parties du son.

Comme cela a été dit, toutes les formes d'ondes périodiques se composent d'oscillations sinusoïdales (ou série de Fourier). Celles-ci forment pour un son ses harmoniques. L'oscillation sinusoïdale pure apparaît seulement, en toute logique, avec la première partie du son. Une vibration carrée comporte une infinité d'harmoniques. Le nombre d'harmoniques, leur rapport fréquentiel et d'amplitudes déterminent le timbre d'un son. La série d'harmoniques est très importante, car l'oreille humaine ne réagit qu'avec des vibrations sinusoïdales. Un son dont la forme d'onde s'éloigne d'une sinusoïde pure, sera dans l'oreille avant d'être entendu, décomposé en parties de son ou partiel. On devra prendre en compte ce fait dans les explications à venir.

Les bruits

En dehors des sons, on trouve le *bruit.* Un son peut se définir très précisément et se générer électroniquement, ce qui pour un bruit est plus difficile à obtenir. Il possède soit une fréquence déterminée, soit une intensité à enveloppe définie et de plus, la forme de l'onde n'est pas unitaire. Il reproduit en fait une combinaison arbitraire d'événements sonores. Le souffle est l'élément de base du bruit car c'est un mélange d'une infinité de vibrations, dans lesquelles les fréquences et positions de phase n'ont aucun rapport entre elles. Par exemple, le vent souffle car les millions de molécules d'air s'entrechoquent les unes avec les autres ou avec la surface de la terre, la conséquence étant la formation et le déplacement de vibrations. Cet ensemble forme un son de combinaison qu'on nomme *souffle du vent.*

Digitilisation d'une forme d'onde

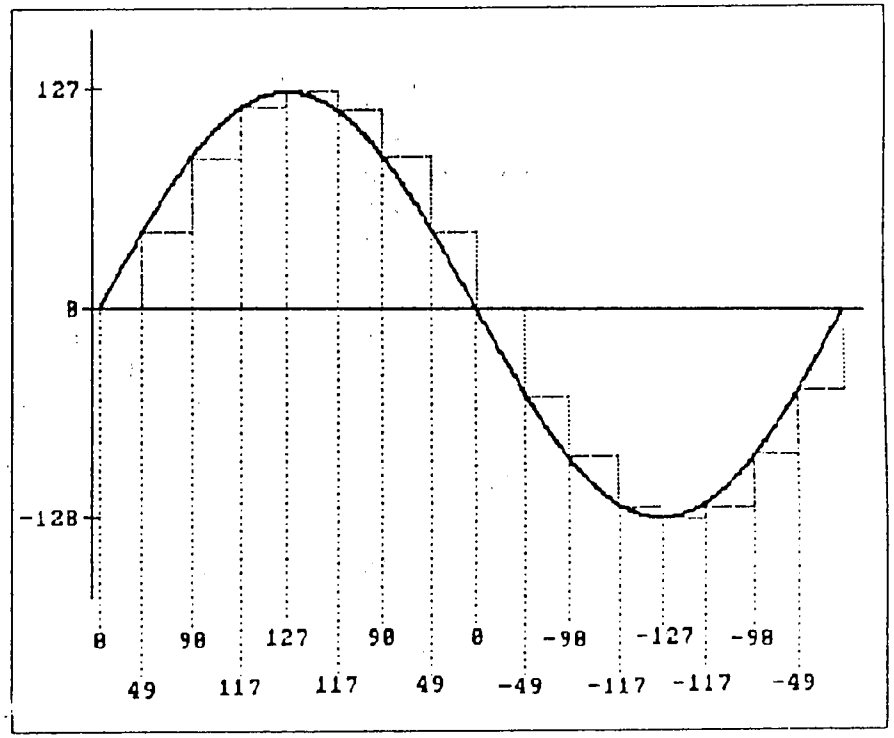


Figure 1.5.8.3

Le critère important pour l'analyse des capacités acoustiques d'un ordinateur est sa complexité. L'optimum serait que les trois paramètres d'un son (fréquence, intensité, timbre) se laissent manipuler librement.

Pour l'Amiga, on a cherché à s'approcher le plus près possible de cette condition. Pour ne pas être assujetti à une forme d'onde, l'équivalent digital sera mis en mémoire et sera modifié dans les oscillations électriques correspondantes, au moyen d'un transformateur digital/analogique.

Autrement dit, la vibration sera digitalisée et stockée mémoire. Pour la sortie, les données digitalisées seront à nouveau transformées en données analogiques et enfin transmises à l'amplificateur.

Le schéma 1.5.8.1 présente différentes formes d'ondes. Lorsque l'une d'elles doit être modifiée par l'ordinateur, son allure doit être reproduite à partir de calculs.

C'est pour cette raison qu'on partage l'oscillation d'une onde en un nombre pair de sections de taille égale. On débutera toujours ce fractionnement au passage à zéro de la courbe. Pour chacune des sections, on transfère la valeur Y correspondante en mémoire. On aura ainsi une suite de nombres, qui sur la courbe, représenteront un moment déterminé. Le terme anglais pour ces valeurs digitalisées est *'sample'*, ce qui signifie *échantillon*.

Lors de l'émission, l'Amiga modifie à nouveau la valeur des nombres issus de la mémoire, en tension de sortie correspondante. Etant donné que la vibration, issue de la digitalisation, ne peut se décomposer qu'en un nombre limité d'échantillons, la courbe de sortie ne pourra être reconstruite qu'avec ce nombre. C'est pourquoi cette dernière prend une allure en forme de marche, qu'on peut voir sur le schéma 1.5.8.3.

La qualité de reproduction d'une tonalité, proche de l'original, dépend essentiellement de deux facteurs :

Le premier est la *résolution du signal digitalisé*. On entend par là l'échelle de valeurs utilisée pour un échantillon. Sur l'Amiga, cette échelle est constituée d'un nombre 8 bits, pouvant varier de -128 à +127. Chaque valeur d'entrée peut donc prendre une des 256 valeurs en mémoire. Comme la résolution du signal d'entrée analogique est théoriquement infinie et que chaque échantillon est limité, il apparaît ici une erreur. On la nomme *erreur d'arrondi* (ou de *quantification)*. Ainsi, lorsqu'une valeur d'entrée se trouvera n'importe où, entre deux nombres ne correspondant en aucun cas à l'un des 256 pas de digitalisation, elle sera arrondie. L'erreur de quantification maximale se monte à 1/256 de la valeur digitalisée (on dit aussi : l'erreur atteint 1 LSB).

L'erreur de quantification
bruit se manifeste en augmentant avec la taille de l'erreur de quantification.

Une zone de valeur de 8 bits permet déjà une très bonne reproduction de la vibration originale. Une qualité Hifi nécessite une plus grande résolution. Une platine CD travaille, par exemple, avec 16 bits.

Le deuxième paramètre, pour une tonalité digitale de qualité, est le *'sampling-rate'*. On entend par ce terme le nombre d'échantillons par seconde, un nombre élevé donnant naturellement une meilleure restitution. Le taux d'échantillonnage se laisse librement déterminer sur l'Amiga, à l'intérieur de limites préétablies. On doit tout d'abord, considérer le nombre d'échantillons que l'on veut employer pour la digitalisation d'un son. Sur notre exemple (schéma 1.5.8.3), il y a 16 valeurs. Il en résulte une courbe sinusoïdale formée de marches peu différentes du signal normal.

La sortie d'un son digitalisé

L'Amiga possède 4 canaux audio, qui travaillent suivant le même principe :

Un son digitalisé en provenance de la mémoire, via les canaux DMA, sera émi au moyen d'un transformateur digital/analogique. Cette émission se répète continuellement, de telle sorte que la vibration se stabilise. Les canaux 0 et 3 ainsi que les canaux 1 et 2 forment respectivement le canal stéréo gauche et le canal stéréo droit.

A chaque canal audio correspond un canal DMA. Etant donné que les accès DMA de l'Amiga se déroulent mot par mot, on réunit deux échantillons sur un même mot de données. C'est pour cette raison qu'un nombre pair d'échantillons est nécessaire. La partie de poids fort du mot (bits 8-15) sera toujours prise en compte avant les bits inférieurs (bits 7-0). La liste de données pour notre son digitalisé apparaîtra en mémoire de la façon suivante :

(Start correspond à l'adresse de départ de la liste dans la CHIP-RAM) :

```
Start:
dc.b 0,49        ;1er mot de données, échantillon 1 et 2
dc.b 90,117      ;2ème mot de données, échantillon 3 et 4
dc.b 127,117     ;3ème mot de données, échantillon 5 et 6
dc.b 90,49       ;4ème mot de données, échantillon 7 et 8
dc.b 0,-49       ;5ème mot de données, échantillon 9 et 10
dc.b -90,117     ;6ème mot de données, échantillon 11 et 12
dc.b -127,-117   ;7ème mot de données, échantillon 13 et 14
dc.b -90,-49     ;7ème mot de données, échantillon 15 et 16
```

Le transformateur digital/analogique traite les échantillons par nombres 8 bits. Comme la technique digitale le précise plus haut, les échantillons doivent être émis sous la forme de deux compléments. Cette conversion est exécutée par l'assembleur, de telle manière que les valeurs négatives puissent être écrites directement dans la liste des données.

On doit alors choisir un des 4 canaux audio sur lequel le son sera émi. Ensuite, le canal DMA audio sera initialisé. Ce sont 5 registres qui établissent les paramètres d'exploitation. Les deux premiers forment une paire de registres d'adresse, identiques aux registres déjà rencontrés dans les autres accès DMA. Ils se nomment *AUDxLCH* et *AUDxLCL*, où x correspond au numéro de canal DMA.

Registre	Nom	Fonction
$0A0	AUD0LCH	Pointeur sur les données audio (bits 16-18)
$0A2	AUD0LCL	du canal 0 (bits 0-15)
$0B0	AUD1LCH	Pointeur sur les données audio (bits 16-18)
$0B2	AUD1LCL	du canal 1 (bits 0-15)
$0C0	AUD2LCH	Pointeur sur les données audio (bits 16-18)
$0C2	AUD2LCL	du canal 2 (bits 0-15)
$0D0	AUD3LCH	Pointeur sur les données audio (bits 16-18)
$0D2	AUD3LCL	du canal 3 (bits 0-15)

L'initialisation d'un pointeur [illegible] *MOVE.L* :

```
LEA $DDF000,a5 ;adresse de base des circuits spécialisés dans a5
MOVE.L #Start,AUD0LCH(a5) ;'Start' mis dans AUD0LC
```

Enfin, il faut communiquer au contrôleur DMA, la longueur d'une oscillation digitalisée, c'est-à-dire le nombre d'échantillons la représentant. Les registres correspondants sont *AUDxLEN* :

Adresse	Nom	Fonction
$0A4	AUD0LEN	nombre de mots de données audio du canal 0
$0B4	AUD1LEN	nombre de mots de données audio du canal 1
$0C4	AUD2LEN	nombre de mots de données audio du canal 2
$0D4	AUD3LEN	nombre de mots de données audio du canal 3

La longueur n'est pas donnée en nombre d'octets, mais en nombre de mots. C'est la raison pour laquelle le nombre d'octets doit être divisé par 2, avant d'être écrit dans le registre *AUDxLEN*.

L'initialisation du registre *AUDxLEN* peut se faire avec l'instruction *MOVE*. Pour éviter le décompte des mots, on définit deux labels : *'Start'* correspondant à l'adresse de départ de la liste de données, *'End'* correspondant à l'adresse de fin + 1 (cf. exemple de liste de données). Les adresses de base des circuits spécialisés se trouvent dans le registre *a5 ($DFF000)* :

```
MOVE.W #(End-Start)/2,AUD0LEN(a5)
```

On s'occupe ensuite de l'intensité des sons. Sur l'Amiga, on peut choisir l'intensité de chaque canal séparément. Ainsi, 65 pas sont à disposition, de 0 (inaudible) à 64 (intensité maximum). Les registres correspondants se nomment *AUDxVOL* :

Adresse	Nom	Fonction
$0A8	AUD0VOL	intensité du canal audio 0
$0B8	AUD1VOL	intensité du canal audio 1
$0C8	AUD2VOL	intensité du canal audio 2
$0D8	AUD3VOL	intensité du canal audio 3

Si on initialise le canal audio avec une intensité moyenne, on obtient :

```
MOVE.W #32,AUD0VOL(a5)
```

Le dernier paramètre manquant est le taux d'échantillonnage (*sampling-rate*). Il détermine l'intervalle de temps séparant l'émission de deux octets de données vers le transformateur digital/analogique. Ce taux détermine ainsi la fréquence des sons. Comme cela a déjà été défini au départ, la fréquence correspond aux nombre de vibrations par seconde. Chaque oscillation est constituée d'un nombre variable d'échantillons ; dans notre exemple, le nombre se monte à 16. Lorsque le taux d'échantillonnage reproduit le nombre d'échantillons par seconde, la fréquence du son correspond à ce taux divisé par le nombre d'échantillons par oscillation :

```
Fréquence du son = taux d'échantillonnage/échantillons par oscillation
```

Malheureusement, le taux d'échantillonnage n'est pas donné exactement en Hertz. Le contrôleur DMA saura, par contre, le nombre de cycles de bus présents entre deux sorties. Un cycle a une durée exacte de 279,365 nanosecondes (milliardièmes de seconde), ou 2.79365 * 10^{-7}, ou 0.000000279365 secondes.

Pour arriver au taux d'échantillonnage par l'intermédiaire du nombre des cycles de bus, il suffit d'obtenir la valeur inverse du taux et on aura ainsi la durée d'un échantillon. Si on divise cette valeur par la durée en secondes d'un cycle de bus entre deux échantillons, on obtient la période d'échantillon :

```
Période d'échantillon = 1 / taux d'échantillonnage * 2.79365*10^-7
```


c'est-à-dire un LA, le ux d'échantillonnage se calculera comme suit :

```
taux d'échantillonnage = fréquence * échantillons par oscillation

taux d'échantillonnage = 440 Hz * 16 = 7040 Hz
```

La période d'échantillon sera vite déduite en mettant les valeurs à la bonne place :

```
période d'échantillon = 1 / (7040 * 2.79365*10^-7) = 508.4583
```

Etant donné que seules les valeurs entières sont acceptées, le résultat arrondi sera 508. Ainsi la fréquence de sortie ne sera plus exactement 440 Hz, mais l'écart se montera à 0.4 Hz.

La période d'échantillon peut théoriquement accepter les valeurs entre 0 et 65535. La zone réelle sera cependant limitée vers le haut. Comme cela a été conclu sur le schéma 1.5.3.2 du chapitre **'Elements de base'**, chaque canal audio possède un *slot DMA* par ligne du *RASTER*, c'est-à- dire qu'à chaque ligne du *RASTER*, un mot, respectivement deux échantillons, peut être lu dans la mémoire. Ainsi la valeur minimale pour la période est de 124. La fréquence d'échantillon correspond alors à 28867 Hertz. Si on raccourcit la période, il peut arriver qu'un mot de données soit émis deux fois, étant donné que le prochain ne pourra être lu à temps.

Les registres de période d'échantillon se nomment *AUDxPER* :

Adresse	Nom	Fonction
$0A6	AUD0PER	période d'échantillon du canal audio 0
$0B6	AUD1PER	période d'échantillon du canal audio 1
$0C6	AUD2PER	période d'échantillon du canal audio 2
$0D6	AUD3PER	période d'échantillon du canal audio 3

L'instruction *MOVE.W #300,AUD0PER(a5)* d'échantillonnage dans le registre *AUD0PER*. Ainsi, tous les registres du canal audio 0 seront munis de la bonne valeur, pour l'émission de notre son. Pour l'entendre, il faut activer les accès DMA sur le canal *DMA audio 0*. C'est ce que permettent 4 bits du registre *DMACON* :

N° bit DMACON	Nom	N° canal DMA audio
3	AUD3EN	3
2	AUD2EN	2
1	AUD1EN	1
0	AUD0EN	0

Si le canal *DMA audio* 0 doit être activé, le bit *AUD0EN* doit être mis à 1. Au même moment, il sera nécessaire d'initialiser le bit *DMAEN* (cf. **Eléments de base**).

```
MOVE.W #$8201,DMACON(a5)      ;Initialisation de AUD0EN et de DMAEN
```

A ce moment, le contrôleur DMA commencera à prendre les données audio de la mémoire et à les transférer vers le transformateur digital/ analogique. Le son pourra être entendu via le haut-parleur. Si on désire le désactiver, il suffit de mettre le bit *AUD0EN* à 0.

Lorsqu'on met *AUDxEN* à 1, les accès DMA débutent à l'adresse contenue dans *AUDxLC*. Il y a évidemment une exception : si le canal DMA est activé (*AUDxEN=1*) et si on met le bit rapidement à 0, puis nouveau à 1, sans que le canal DMA n'ait lu, entre-temps, un nouveau mot, le contrôleur DMA poursuit à l'ancienne adresse.

Interruption Audio

Les accès *DMA audio* débutent toujours avec l'octet de données se trouvant à l'adresse *AUDxLC*. Si autant de mots de données sont issus de la mémoire qu'émis, comme l'établit *AUDxLEN*, les accès DMA recommenceront à l'adresse *AUDxLC*.

contenu des registres *AUDxLC* ne peut pas être modifié lors des accès. Il existe en fait pour chaque canal *DMA audio*, un registre adresse de plus. Avant que le contrôleur DMA ne prélève le premier octet de données de la mémoire, il copie la valeur du registre *AUDxLEN* dans ses registres adresse internes. Le registre *AUDxLEN* est aussi transféré dans un compteur interne. Lorsque ces deux transferts sont réalisés, une interruption est libérée. Comme cela a été vu au chapitre **'Interruptions'**, il existe pour chaque canal audio, un bit d'interruption propre. L'interruption processeur de niveau 4 est essentiellement réservée pour ces bits.

Pendant que le contrôleur DMA prélève mot de données sur mot de données dans la mémoire, le processeur peut s'occuper des nouvelles données de *AUDxLC* et de *AUDxLEN*, étant donné que les deux registres sont stockés en mémoire de façon interne. Lorsque le compteur, qui a été initialisé au départ avec la valeur de *AUDxLEN*, arrive à 0, les données de *AUDxLC* et *AUDxLEN* seront à nouveau lues. Le processeur a donc ainsi assez de temps pour modifier la valeur des deux registres si nécessaire. La sortie son est donc possible sans interruption.

Après chaque oscillation complète, une interruption sera donc libérée. Avec un son à haute fréquence, les interruptions apparaissent souvent. On devra activer le bit *Interrupt-Enable* de l'interruption audio, seulement si cette dernière est absolument nécessaire. Le processeur risque alors de saturer, à force d'appels d'interruptions.

Modulation de l'intensité et de la fréquence

La possibilité de moduler la fréquence ou l'intensité permet de générer des effets de tonalité déterminée. Un canal DMA travaille donc toujours en tant que modulateur, qui modifie les paramètres des autres canaux. L'oscillateur de modulation prend les données de la mémoire comme cela a été expliqué plus haut. Mais au lieu de les transférer vers le transformateur digital/analogique, il les écrit, soit dans le registre fréquence, soit dans le registre intensité de l'oscillateur, qu'il doit modifier *(AUDxVOL ou AUDxLEN)*. Il peut aussi influencer ces deux registres en même temps.

Dans ce cas, les données issues de la liste seront alter ɔs dans les registres *AUDxVOL* et *AUDxLEN*. Les mots de données t le même format que ceux du registre cible :

Volume :	bits 7-15	inutilisés
	bits 0-6	valeur d'intensité comprise entre 0 et 64
Fréquence :	bits 0-15	période d'échantillon

Le tableau suivant montre l'utilisation des mots de données de l'oscillateur de modulation dans les trois cas possibles :

		Oscillateur module :	
N° mot de données	Fréquence	Intensité	Fréquence & intensité
1	période 1	Volume 1	Volume 1
2	période 2	Volume 2	Période 1
3	période 3	Volume 3	Volume 2
4	période 4	Volume 4	Période 2

Pour activer un canal audio en tant que modulateur, on doit initialiser le(s) bit(s) correspondant(s) du registre contrôle audio-disque *(ADKCON)*. Chaque canal ne module que le suivant, c'est-à-dire : le canal 0 module le canal 1, 1 module 2 et 2 module 3. Le canal 3 peut aussi être commuté en modulateur, mais ses données n'étant pas utilisées pour une modulation, elles seront perdues. Enfin, si on utilise un canal audio en temps que modulateur, sa sortie audio sera désactivée.

Le registre *ADKCON* renferme aussi, comme son nom l'indique, des bits de gestion, pour le contrôleur disque. Ils ne sont pas utilisés ici et seront expliqués dans le chapitre suivant.

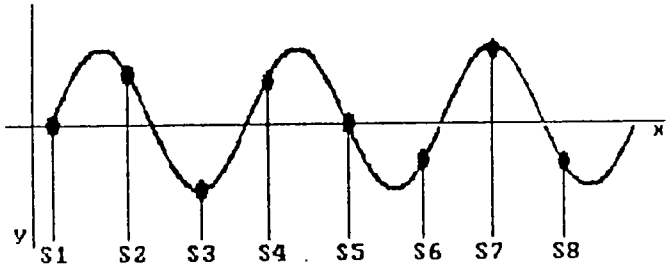
ADKCON $09E (écriture 010 (lecture)

Bit n°	Nom	Fonction
15	SET/CLR	Bits activés (SET/CLR=1) ou désactivés
14 à 8		Bits de gestion du contrôleur disque
7	USE3PN	le canal audio 3 ne module rien
6	USE2P3	le canal audio 2 module la période du canal 3
5	USE1P2	le canal audio 1 module la période du canal 2
4	USE0P1	le canal audio 0 module la période du canal 1
3	USE3VN	le canal audio 3 ne module rien
2	USE2V3	le canal audio 2 module le volume du canal 3
1	USE1V2	le canal audio 1 module le volume du canal 2
0	USE0V1	le canal audio 0 module le volume du canal 1

Si on utilise un canal pour en moduler un autre, ses mots de données seront écrits dans le registre correspondant et serviront à la modulation. Sinon les deux canaux travailleront indépendamment l'un de l'autre.

Problème de la création des sons digitalisés

Digitalisation de plusieurs oscillations, permettant d'améliorer la qualité des sons



S=Sample

Figure 1.5.8.4

Cet exemple d'oscillation est défini au moyen de 16 échantillons. Le taux d'échantillonnage maximum correspond à 28867 Hz. Ceci nous donne une fréquence maximum de 28867/16 = 1460,4 Hz. Ceci correspond en musique, à un FA troisième (1480 Hz).

Si on veut aller plus haut, il faut diminuer le nombre d'échantillons. Si notre sinusoïde était définie avec la moitié des échantillons, la limite fréquentielle doublerait pour atteindre 3020.8 Hz. Seulement, 8 bits de données est un petit nombre pour reproduire une sinusoïde.

Pour des sons plus hauts, le nombre d'échantillons se réduit encore. Pour 6041.6 Hz, seuls 4 sont présents. A ce moment, les formes d'ondes ne peuvent plus être différenciées dans leur représentation.

Toutefois, ces différenciations ne sont plus perceptibles par notre oreille, en hautes fréquences. En effet, plus cette dernière est haute, plus les différentes tonalités seront difficilement audibles.

Pourtant, la qualité des sons pourra être améliorée lorsqu'on utilisera, en haute fréquence, plusieurs oscillations pour former l'onde souhaitée (cf. schéma 1.5.8.4).

Le filtre passe-bas

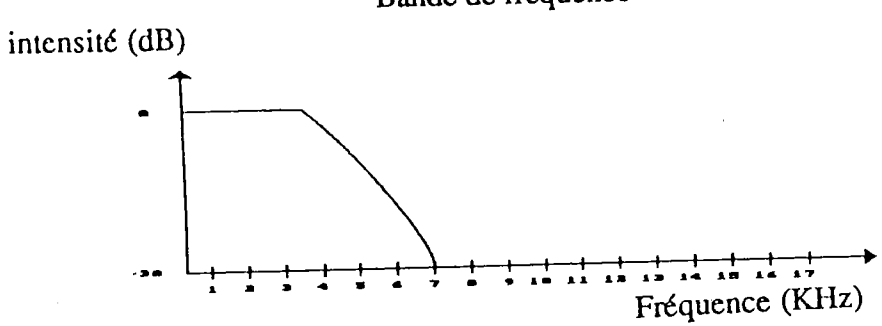


Figure 1.5.8.5 (a)

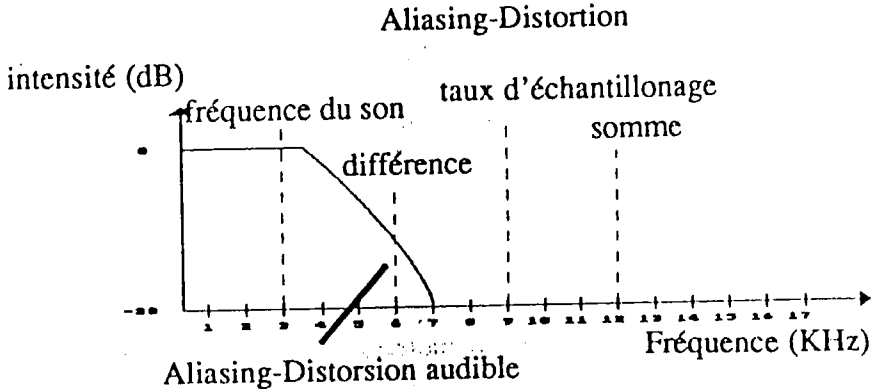


Figure 1.5.8.5 (b)

La frontière fréquentielle de la sortie son de l'Amiga sera encore limitée en d'autres circonstances. Lors de l'analogie des différentes données digitales, il apparaît deux fréquences parasites, par interaction entre le taux d'échantillonnage et la fréquence désirée. L'une correspond à la somme entre le taux et la fréquence, l'autre à la différence.

Ce phénomène porte le nom de *'Aliasing distorsion'*.

Par exemple, pour un son d'une fréquence de 3 KHz et un taux de 12 KHz, la différence se montera à 9 et la somme à 15 KHz.

Pour éliminer cette fréquence parasite, on a installé un filtre passe bas entre la sortie du transformateur analogique/digital et le boitier audio. Sa fonction est représentée par le schéma 1.5.8.5 (a et b). Toutes les fréquences inférieures à 4 KHz peuvent passer sans être parasitées. Entre 4 et 7 KHz, le signal sera de plus en plus atténué. Au dessus de 7 KHz, plus aucune fréquence ne passera. Si on prend notre exemple plus haut : le son 3 KHz ne sera pas atténué, mais la somme et la différence fréquentielle qui se montent à, respectivement, 15 et 9 KHz, ne passeront pas la limite de fréquence du filtre à 7 KHz.

De ce fait, ces deux fréquences parasites ne seront pas perceptibles à partir du haut-parleur. Si on cherche à produire un son à 3 KHz, avec un taux d'échantillonnage de 9 KHz, la différence fréquentielle correspondra à 9-3 = 6 KHz, valeur qui ne sera pas éliminée par le filtre. Si on veut être certain que cette différence reste toujours supérieure à la limite du filtre, on doit respecter la règle suivante :

```
taux d'échantillonnage > plus haute composante fréquentielle + 7 KHz
```

Il ne suffit pas que la différence entre la fréquence d'échantillonnage et la fréquence de sortie souhaitée soit supérieure à 7 KHz. Lorsqu'on utilise une forme d'onde, qui comprend beaucoup d'harmoniques, chacune de ces dernières produit une telle différence. Pour cette raison, on sera obligé de mettre dans la formule ci-dessus, la plus grande composante fréquentielle de la forme d'onde utilisée.

Le filtre passe-bas ne refoule pas seulement les fréquences parasites, il limite aussi la bande de fréquences de l'Amiga. En effet, les sons rares d'un morceau de musique apparaissent avec une fréquence de base comprise entre 4 et 7 KHz, mais les harmoniques des formes d'ondes déterminées se trouvent déjà à des fréquences bien plus basses. Cela se répercute particulièrement sur les ondes carrées. Le schéma 1.5.8.2. montre la formation d'une onde carrée, à partir de la combinaison de plusieurs ondes sinusoïdes, qui se trouvent dans le même rapport de fréquences. Sur le schéma, la carrée se compose exclusivement des trois partiels (parties de sons), une onde carrée réelle étant composée d'une infinité d'harmoniques. Si l'harmonique de haute fréquence est découpée ou limitée par le filtre, il en résultera un signal déformé. Dans le cas extrême où la fréquence de base de l'onde carrée approche la limite fréquentielle du filtre, il peut arriver que seule la première harmonique subsiste. L'onde carrée originelle deviendra donc une onde sinusoïde.

Développement d'intensité d'un son

La tonalité d'un instrument sera déterminée non seulement par la forme de l'onde, mais aussi par le développement d'intensité du son. Sur le premier point, l'Amiga est capable de tout prendre en charge. Le point important est la programmation du développement d'intensité.

Le développement d'intensité d'un son se divise en quatre parties : la *phase d'attaque, la phase d'affaiblissement, la phase de maintien* et *la phase de relâchement*.

Dès que le son sera joué, la phase *d'attaque* débute. Elle détermine la rapidité de l'intensité à passer de 0 jusqu'à la valeur maximale. Puis l'intensité baissera pendant la phase d'affaiblissement jusqu'à la valeur de maintien. Pendant la phase de maintien, le son retentira avec cette intensité. La phase de relâchement marque la fin du son, où l'intensité passera de la valeur de maintien à 0.

Ce déroulement peut être représenté sous la forme d'une courbe, que l'on nomme *courbe enveloppe*.

Il existe trois possibilités différentes de réalisation d'une telle courbe sur l'Amiga :

1. Modulation de l'intensité

On utilise un deuxième canal audio, afin de moduler le son (par exemple, le canal 0 sera le modulateur du canal 1). Le canal 1 servira donc à émettre le son. L'intensité de départ sera mise à 0.

La courbe enveloppe souhaitée sera divisée en deux parties : la *phase d'attaque* et la *phase de relâchement*. Son développement sera digitalisé (ceci fonctionnant de la même manière qu'avec une forme d'onde) et stocké en mémoire dans deux listes de données. Si le son doit être joué, il suffit de mettre le canal 0 sur l'adresse des données de la première phase et de le démarrer. Comme l'intensité du canal 1 est modulée, l'intensité du son suivra exactement la phase d'attaque. Lorsque cette dernière atteint la valeur de maintien, la liste de données du canal 0 a été traitée. Ce dernier génère alors une interruption et voudra recommencer la liste de données par le début. A ce moment, le processeur doit réagir à l'interruption et au moyen du bit *AUD0EN* du registre *DMA0CON*, désactiver le canal 0.

Pour désactiver le son, le canal 0 est initialisé à l'adresse de la phase de relâchement et redémarré à nouveau. On attendra alors l'interruption qui montre que cette phase est terminée et on désactivera le canal 0.

Les registres du canal 0 devront être initialisés de la mar e suivante, pour ce processus :

USE0V1 Ce bit du registre *ADKCON* doit être mis à 1, afin que le canal 0 puisse moduler l'intensité du canal 1.

AUD0LC Ce registre sera initialisé une première fois avec l'adresse de la liste de données de la phase d'attaque, puis une deuxième fois, avec l'adresse de la liste de la deuxième phase.

AUD0LEN Ce registre contiendra, comme les adresses dans *AUD0LC*, la longueur des données de la première phase, puis de la deuxième phase.

AUD0VOL Ce registre n'a aucune fonction, étant donné que la sortie audio du canal 0 est désactivée.

AUD0PER Le contenu de ce registre détermine la vitesse avec laquelle les données d'intensité seront issues de la mémoire. On pourra ainsi contrôler la durée des 2 phases.

Au moyen de cette méthode, il est possible de former une enveloppe de courbe parfaite. Il y a pourtant un inconvénient de taille : la sortie d'un son nécessite deux canaux audio. Si on est obligé d'utiliser 4 canaux audio différents, il faudra opter pour la deuxième méthode.

2. Gestion de l'intensité au moyen du processeur

La courbe enveloppe sera transférée dans la mémoire de la même manière. Cependant, on modifie cette fois-ci l'intensité à partir du processeur. Ce dernier prélève la valeur actuelle de l'intensité dans la mémoire et l'écrit dans le registre d'intensité du canal audio correspondant, ceci de manière régulière et périodique. On doit alors laisser se dérouler le programme comme une routine d'interruption. Ceci peut se passer lors de l'interruption *écran vide vertical*, ou alors on utilisera une interruption minuterie du *CIA-B*.

L'inconvénient de cett méthode est la nécessité d'un temps de calcul, étant donné que la g on de l'intensité ne se fait plus via les canaux DMA. Mais c'est une méthode qui conviendra dans la plupart des cas d'utilisation.

3. Structure de la courbe enveloppe à partir des données Vibration

Cette méthode est avantageuse pour tout ce qui est courte tonalité ou bruitage. Au lieu de digitaliser une oscillation de la forme d'onde souhaitée, on écrit le déroulement entier en mémoire. Ceci peut être soit évalué par un programme, soit on utilise un digitaliseur audio. Avec ce dernier, un bruit pourra être digitalisé au moyen d'un micro et d'un transformateur analogique/digital.

Plusieurs appareils sont proposés pour l'Amiga par différentes firmes. Lorsque les données sont dans l'Amiga, on peut à chaque hauteur de son, jouer sur la vitesse. On pourra ainsi imiter des effets complexes tels que les rires où les cris humains.

Cette méthode a aussi son inconvénient : elle requiert soit des calculs fastidieux, soit un périphérique de plus pour stocker en mémoire la forme digitalisée d'une tonalité. De plus, le besoin en place mémoire est assez important. Une sonorité d'une durée d'une seconde, avec un taux d'échantillonnage de 20 KHz, est issue d'une liste de données prenant 20 Koctets de mémoire.

Trucs, Astuces et autres

La qualité d'un son

La zone de valeur des données digitales s'étend de -128 à 127. Cette zone doit être entièrement utilisée, le plus souvent possible. Il est préférable d'avoir une oscillation digitalisée avec une amplitude égale à 256. Sinon la qualité du son diminuera perceptiblement, étant donné que la taille de l'erreur de quantification est inversement proportionnelle à cette zone, et que le souffle dû à cette quantification atteint très vite une dimension de brouillage.

C'est pour cette raison qu'il faut éviter d'utiliser l'amplitude d'une oscillation digitalisée pour la gestion de l'intensité et que chaque canal possède son registre *AUDxVOL*. Si on diminue l'intensité avec ce dernier, le rapport entre le son souhaité et le bruit parasite sera entièrement conservé, ainsi que la qualité sonore de l'Amiga.

Changement de forme d'onde sans brouillage

Pour éviter le brouillage des sons, comme les craquements ou les sauts d'intensité, on doit appliquer les règles suivantes :

Chaque oscillation ne doit être digitalisée qu'entre deux passages à 0, c'est-à-dire qu'on ne commence à transférer les données en mémoire qu'avec un point d'intersection avec l'axe des X. Si on se tient à cette règle, il est évident que les formes d'ondes débuteront ou se termineront avec la même valeur en mémoire, c'est-à-dire 0. Ainsi, plusieurs oscillations différentes pourront apparaître à la suite et ceci sans entendre des bruits parasites.

Deuxièmement, il faut faire attention à ce que l'intensité des deux oscillations soit à peu près identique. Cette intensité correspondra à la valeur effective de la vibration. La valeur effective d'une oscillation ou vibration est égale à l'amplitude d'un signal carré, la superficie en-dessous de la courbe étant à peu près la même que celle de l'oscillation.

Cette valeur effective détermine l'intensité d'une vibration. Seulement dans le cas d'une carrée, cette valeur est égale à l'amplitude. Si on passe d'une forme d'onde à une autre forme possédant une valeur effective plus haute, cette dernière résonnera beaucoup plus fort que la précédente.

La valeur effective d'une oscillation se laisse calculer assez facilement, à partir des données digitalisées :

On additionne les montants complets des octets et on les divise par le nombre d'octets de données.

Si on veut utiliser la zone complète de valeur 8 bits du transformateur digital/analogique pour toutes les formes d'ondes, leur valeur effective ne pourra pas toujours correspondre. On devra alors s'ajuster, lors d'un changement de forme d'onde, à la valeur d'intensité du registre *AUDxVOL* correspondant.

Jouer une note

Normalement, un morceau de musique est composé d'une suite de notes. Si on veut exécuter un telle partition sur l'Amiga, on doit transformer les valeurs des notes en périodes d'échantillons correspondantes. Pour éviter des calculs fastidieux, on préfère utiliser un tableau qui comprend les valeurs des périodes d'échantillons pour tous les demi-ton d'une octave :

Tableau des valeurs de période d'échantillon des notes de musique :

Note	Fréquence (Hz)	Période d'échantillon pour AUDxLEN = 16	
C : do	261.7	427	(262.0)
C# : do dièse	277.2	404	(276.9)
D : ré	293.7	381	(293.6)
D# : ré dièse	311.2	359	(311.6)
E : mi	329.7	339	(330.0)
F : fa	349.3	320	(349.6)
F# : fa dièse	370.0	302	(370.4)
G : sol	392.0	285	(392.5)
G# : sol dièse	415.3	269	(415.8)
A : la	440.0	254	(440.4)
A# : la dièse	466.2	240	(466.0)
B : si	493.9	226	(495.0)
C : do	523.3	214	(522.7)

(les valeurs entre parenthèses indiquent les fréquences réelles des périodes d'échantillonnage correspondantes).

Voici encore une petite remarque pour le calcul des valeurs :

La fréquence d'un demi-ton est toujours supérieure d'un facteur racine douzième de 2 que le précédent.

440 (La) * $2^{(1/12)}$ = 466.2 (la dièse)
466.2 (La#) * $2^{(1/12)}$ = 493.9 (Si)
etc ...

Une octave correspond toujours à une fréquence doublée.

Si on veut jouer maintenant une note d'une octave qui ne se trouve pas dans le tableau, on a deux possibilités :

1. On modifie la période d'échantillonnage. Pour chaque octave vers le haut, on doit diviser la valeur par deux. Une octave plus bas correspond à doubler la valeur. Ceci est assez simple, mais on atteint vite les limites connues. Pour un champ de données de 32 octets *(AUDxLEN = 16)*, comme dans notre tableau, la plus petite période d'échantillonnage possible est déjà atteinte avec un Fa seconde. On devra alors diminuer la liste de données.

   Dans ce cas, un problème risque d'apparaître dans la zone des sons bas, étant donné que la fréquence parasite de *l'Aliasing distortion* est perceptible.

La deuxième solution semble être la meilleure :

2. On compose pour chaque octave, une liste de données propres. La valeur de la période d'échantillonnage restera ainsi constante pour chaque octave. Elle ne servira qu'au choix des demi-tons. S'il existe un son d'une octave supérieure dans le tableau, on utilise une liste de données moitié moins grande. Respectivement, elle aura une longueur doublée pour une octave inférieure.

   L'étendue normale des sons comprend environ 8 octaves, c'est-à-dire que 8 listes de données sont nécessaires par forme d'onde.

avec ce procédé, une [illegible]orité optimale indépendamment de la hauteur du son.

Générer des hautes fréquences

La période d'échantillonnage minimale correspond à 124. La raison est que le *DMA audio* n'a pas la capacité de lire les données audio à temps, avec une période d'échantillonnage plus courte. Le mot de données précédent sera émis plusieurs fois. Cet effet peut être très bien utilisé. Comme le mot de donnée lu renferme 2 échantillons, on peut engendrer un signal carré de haute fréquence. Avec une période d'échantillonnage de 1, on obtient une fréquence d'échantillonnage de 3.58 MHz et une fréquence de sortie de 1.74 MHz. Pour pouvoir utiliser ces hautes fréquences comme signal de sortie, on est obligé de les faire passer sur le filtre passe bas. Pour ce faire, on pourra employer l'entrée *AUDIN* (broche 16) du connecteur série (RS232).

Pour générer de telles hautes fréquences, on devra mettre dans le registre *AUDxVOL*, l'intensité maximum *(AUDxVOL = 64)*.

Jouer de la musique sur plusieurs voix

Comme l'Amiga possède 4 canaux audio entièrement indépendants, il est possible de générer 4 sons différents au même moment. On pourra ainsi exécuter directement un morceau de musique à quatre voix.

On peut faire mieux. En effet, 4 canaux audio ne signifient pas que le maximum de voix corresponde à 4. On a vu précédemment qu'une forme d'onde est reproduite par une combinaison de plusieurs signaux sinusoïdaux. De la même façon que ces harmoniques forment une onde, on peut, par combinaison de plusieurs formes d'ondes, générer une sonorité à plusieurs voix. Les signaux de sortie des canaux audio 0 et 3 seront mixés ensemble, dans un canal stéréo, inclus dans PAULA. Ainsi les formes d'ondes des deux canaux seront combinées, pour former un seul signal à deux voix.

Ce qu'il est possible de faire avec des signaux analo[illegible]s électroniques, l'est aussi par calcul sur des données dig[illegible]s. On peut additionner des données digitales de deux formes d'ondes différentes, et émettre le résultat sur le canal audio. On aura alors deux voix par canal audio, ce qui n'est pas une limite, car théoriquement, n'importe quel nombre de voix est possible sur le même canal audio.

Ce nombre est vite limité par la vitesse de calcul, mais 16 voix sont tout de même possibles.

La détermination de la somme des signaux est assez simple. A chaque moment, la valeur actuelle de tous les sons sera additionnée, puis le résultat sera divisé par leur nombre. On obtiendra un signal carré de manière analogue lorsqu'on additionnera les signaux sinusoïdaux du même rapport de fréquence.

Sortie Audio sans DMA

Comme pour tous les canaux DMA, il existe pour le *DMA audio*, des registres de données dans lesquels les canaux DMA transfèrent des données et où le processeur peut avoir un accès écriture.

Les registres de données audio :

Adresse	Nom	Fonction
$0AA $0BA $0CA $0DA	AUD0DAT AUD1DAT AUD2DAT AUD3DAT	Ces quatre registres renferment toujours les mots de données, sous la forme d'une paire d'échantillons. Celui qui correspond aux bits supérieurs (8-15) sera toujours transféré en premier.

Pour pouvoir constituer les registres de données audio à partir du processeur, on doit désactiver les accès DMA en mettant *AUDxEN* à 0. La génération des interruptions audio sera alors modifiée.

Elles apparaîtront toujo[illegible] après la sortie des deux octets [illegible] les registres *AUDxDA*[illegible], non comme avant, au début de chaque liste de données audio.

Si on ne charge pas à temps les nouvelles données dans *AUDxDAT*, les deux derniers échantillons ne seront pas répétés par la routine DMA, mais la sortie restera à la valeur du dernier octet de données (l'octet de poids faible de *AUDxDAT*).

La programmation directe des registres de données audio coûte très cher en temps de calcul. A part les cas spéciaux, il est préférable d'utiliser les accès *DMA audio*.

Quelques résultats

Valeurs des registres *AUDxVOL* en décibels (0 dB = pleine intensité) :

AUDxVOL	dB	AUDxVOL	dB	AUDxVOL	dB	AUDxVOL	dB
64	0.0	48	-2.5	32	-6.0	16	-12.0
63	-0.1	47	-2.7	31	-6.3	15	-12.6
62	-0.3	46	-2.9	30	-6.6	14	-13.2
61	-0.4	45	-3.1	29	-6.9	13	-13.8
60	-0.6	44	-3.3	28	-7.2	12	-14.5
59	-0.7	43	-3.5	27	-7.5	11	-15.3
58	-0.9	42	-3.7	26	-7.8	10	-16.1
57	-1.0	41	-3.9	25	-8.2	9	-17.0
56	-1.2	40	-4.1	24	-8.5	8	-18.1
55	-1.3	39	-4.3	23	-8.9	7	-19.2
54	-1.5	38	-4.5	22	-9.3	6	-20.6
53	-1.6	37	-4.8	21	-9.7	5	-22.1
52	-1.8	36	-5.0	20	-10.1	4	-24.1
51	-2.0	35	-5.2	19	-10.5	3	-26.6
50	-2.1	34	-5.5	18	-11.0	2	-30.1
49	-2.3	33	-5.8	17	-11.5	1	-36.1

(*AUDxVOL = 0* correspond à une valeur dB minimum infinie).

Lorsque *AUDxVOL = 64*, la valeur digitale de 127 correspond à une tension de sortie d'environ 400 millivolts, -128 correspondant à -400 millivolts. Une modification de 1 LSB produit une fluctuation d'environ 3 millivolts.

PROGRAMME 1 : Génération d'un son sinusoïdal simple

Ce programme génère un son sinusoïdal avec une fréquence de 440 Hz. On utilisera le même tableau d'échantillons employé dans notre premier exemple. La plus grande partie du programme concerne l'allocation de la CHIP-RAM pour la liste des données audio.

Le son sera émis par le canal audio 0, jusqu'à ce que le bouton de la souris soit enfoncé. Le programme restituera alors la zone mémoire employée.

```
;*** génération d'un son simple ***

;registres des circuits spécialisés

INTENA = $9A        ;registre d'autorisation d'interruption (écriture)
DMACON = $96        ;registre de contrôle DMA (écriture)

;registres audio

AUD0LC  = $A0       ;adresse de la liste de données audio
AUD0LEN = $A4       ;longueur de la liste de données audio
AUD0PER = $A6       ;période d'échantillonnage
AUD0VOL = $A8       ;intensité

ADKCON = $9E        ;gestion de la modulation

;registre port A du CIA-A (bouton de la souris)

CIAAPRA = $BFE001

;Exec Library Base Offsets

AllocMem = -30-168  ;ByteSize,Requirements/d0,d1
FreeMem = -30-180   ;MemoryBlock,ByteSize/a1,d1

;autres labels
```



```
Execbase = 4
Chip = 2            ;CHIP-RAM sollicitée

;*** avant programme ***

Start:

;solliciter la mémoire pour la liste de données audio

move.l Execbase,a6
moveq #Alsize,d0    ;taille de la liste des données audio
moveq #Chip,d1
jsr AllocMem(a6)    ;mémoire sollicitée
beq Fin             ;erreur -> fin du programme

;copie de la liste de données audio dans la CHIP-RAM

move.l d0,a0        ;adresse dans la CHIP-RAM
move.l #ALstart,a1  ;adresse dans le programme
moveq #ALsize-1,d1  ;compteur de boucle

Loop: move.b (a1)+,(a0)  ;liste de données dans la CHIP-RAM
dbf d1,Loop

;*** programme général ***

;initialisation des registres audio

lea $DFF00,a5
move.w #$000F,DMACON(a)       ;DMA audio désactivé
move.l d0,AUD0LC(a5)          ;adresse de la liste des données activée
move.w #Alsize/2,AUD0LEN(a5) ;longueur en mot
move.w #32,AUD0VOL(a5)        ;intensité moyenne
move.w #508,AUD0PER(a5)       ;fréquence: 440 Hz

move.w #$00FF,ADKCON(a5)      ;modulation désactivée

;activer les accès DMA audio

move.w #$8201,DMACON(a5)      ;canal 0 activé
```

```
;attendre l'action du bouton de la souris

Wait: btst #6,CIAAPRA
bne Wait

;désactiver les accès DMA audio

move.w #$0001,DMACON(a5)   ;canal 0 désactivé

;*** Fin de programme ***

move.l d0,a1               ;adresse de la liste de données
moveq #Alsize,d0           ;longueur
jsr FreeMem(a6)            ;mémoire occupée, libérée

Fin: clr.l d0
rts

;liste de données audio

Alstart:
dc.b 0,49
dc.b 90,117
dc.b 127,117
dc.b 90,49
dc.b 0,-49
dc.b -90,-117
dc.b -127,-117
dc.b -90,-49
Alfin:
Alsize = Alfin - Alstart    ;longueur de la liste de données audio
```

PROGRAMME 2 : Son sinusoïdal avec Vibrato

Ce programme correspond à une suite du précédent. Il émet le même son sinusoïdal sur le canal 1. Le canal 0 module la fréquence du canal 1 et génère ainsi un vibrato. Les données du vibrato reproduisent une oscillation sinusoïdale digitalisée, la valeur de la période d'échantillonage du point 0 correspondant à un FA, c'est-à-dire 508.


```
;*** générer un vibrato ***

;registres des circuits sp   ılisés

INTENA = $9A        ;registre d'autorisation d'interruption (écriture)
DMACON = $96        ;registre de contrôle DMA (écriture)

;registres audio

AUD0LC  = $A0       ;adresse de la liste de données audio
AUD0LEN = $A4       ;longueur de la liste de données audio
AUD0PER = $A6       ;période d'échantillonnage
AUD0VOL = $A8       ;intensité

AUD1LC  = $B0
AUD1LEN = $B4
AUD1PER = $B6
AUD1VOL = $B8

ADKCON = $9E            ;gestion de la modulation

;CIA-A registre port A (bouton de la souris)

CIAAPRA = $BFE001

;Exec Library Base Offsets

AllocMem = -30-168      ;ByteSize,Requirements/d0,d1
FreeMem = -30-180       ;MemoryBlocK,ByteSize/a1,d0

;autres labels

Execbase = 4
Chip = 2

;*** avant programme ***
Start:

;solliciter la mémoire pour les listes de données
```

```
move.l Execbase,a6
move.l #Size,d0          ;longueur des deux listes
moveq #Chip,d1
jsr AllocMem(a6)         ;mémoire sollicitée
beq fin

;copie de la liste de données audio dans la CHIP-RAM

move.l d0,a0             ;adresse dans la CHIP-RAM
move.l #ALstart,a1       ;adresse dans le programme
move.w #Size-1,d1        ;compteur de boucle

Loop: move.b (a1)+,(a0)+ ;listes dans la CHIP-RAM
dbf d1,Loop

;*** programme général ***

;initialisation des registres audio

move.l d0,d1                    ;adresse de la liste de données audio
add.l #ALsize,d1                ;adresse du tableau Vibrato
lea $DFF000,a5
move.w #$000F,DMACON(a5)        ;accès DMA désactivés
move.l d1,AUD0LC(a5)            ;pointeur sur le tableau vibrato
move.w #Vibsize/2,AUD0LEN(a5) ;longueur du tableau vibrato

move.w #8961,AUD0PER(a5)        ;fréquence vibrato

move.l d0,AUD1LC(a5)            ;liste de données audio dans canal 1
move.w #Alsize/2,AUD1LEN(a5)  ;longueur de la liste
move.w #32,AUD1VOL(a5)          ;intensité moyenne

move.w #$00FF,ADKCON(a5)        ;autres modulations activées
move.w #$8010,ADKCON(a5)        ;canal 0 module la période du canal 1

;accès DMA audio activés

move.w #$8203,DMACON(a5)        ;canaux 0 et 1 activés

;attente de l'action du bouton de la souris
```



```
Wait: btst #6,CIAAPRA
bne Wait

;accès DMA audio désactivés

move.w #$0003,DMACON(a5)        ;canaux 0 et 1 désactivés

;*** fin du programme ***

move.l d0,a1                    ;adresse des listes
move.l #Size,d0                 ;longueur
jsr FreeMem(a6)                 ;mémoire libérée

Fin: clr.l d0
rts

;liste de données audio

Alstart:
dc.b 0,49
dc.b 90,117
dc.b 127,117
dc.b 90,49
dc.b 0,-49
dc.b -90,-117
dc.b -127,-117
dc.b -90,-49
Alfin:
Alsize = Alfin - Alstart    ;longueur de la liste de données audio

;tableau vibrato

Vibstart:
dc.w 508,513,518,522,524,525,524,522,518,513
dc.w 508,503,498,494,492,491,492,494,498,503
Vibfin:
Vibsize = Vibfin - Vibstart    ;longueur du tableau vibrato
Size = Alsize + Vibsize        ;longueur totale des deux listes

;fin du programme
```

1.5.9 SOURIS, JOYSTICK ET PADDLES

La souris, le joystick ou le paddle peuvent être, tous les trois, raccordés à l'Amiga. Ces derniers seront traités ensembles, avec leurs registres respectifs. Les références du gameport, auquel on raccordera ces périphériques, ont été décrites dans le chapitre des connexions. On commencera par la souris :

La souris

La souris est le périphérique le plus employé. Voici l'explication de son fonctionnement et du lien existant entre elle et la flèche à l'écran :

Si on retourne la souris, on remarque une boule dense, entourée de caoutchouc, qui roule lors d'un déplacement de la souris. Ce mouvement de la boule sera enregistré sur deux axes perpendiculaires. Ils sont disposés de telle manière qu'ils pivotent, l'un pour un mouvement le long de l'axe des X, l'autre pour n'importe quelle direction de l'axe Y. Si on déplace la souris en diagonale, les deux axes tournent suivant les composantes X et Y du mouvement de la souris.

A l'extrémité de chaque axe, un disque perforé est disposé. Lors d'une rotation, il interrompt sans cesse un faisceau lumineux. Ce dernier sera transformé en signal électrique et intensifié pour finalement accéder à l'ordinateur, au moyen du câble de la souris.

L'Amiga a maintenant la possibilité d'établir si et avec quelle vitesse la souris se déplace. Mais il ne sait toujours pas dans quelle direction, c'est-à-dire vers la droite ou la gauche, vers le haut ou le bas, est le mouvement.

Une petite astuce résoud ce problème. Sur chaque disque perforé sont disposées deux cellules photoélectriques, qui sont déplacées l'une vers l'autre, d'une demi perforation. Si le disque tourne dans un sens, un des faisceaux sera interrompu avant l'autre.

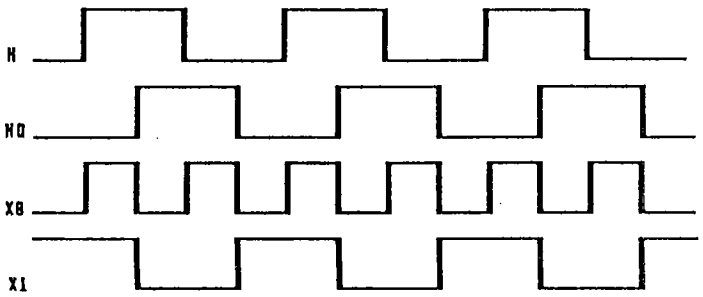
Si on change le sens du mouvement, l'état des signaux photo-électriques se modifier aussi. Ainsi, l'Amiga pourra déterminer le sens du mouvement de la souris.

Cette dernière délivre donc 4 signaux, 2 par axe. Ils portent les noms suivants :

- *vertical pulse*
- *vertical quadrature pulse*
- *horizontal pulse*
- *horizontal quadrature pulse*

Le signal de la souris

1. Mouvement vers la droite (incrémentation)



Le signal de la souris

2. Mouvement vers la gauche (décrémentation)

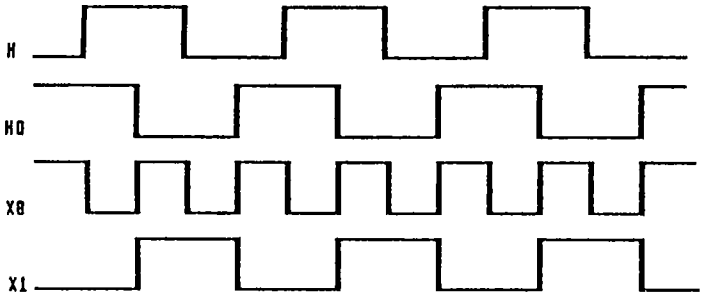


Table de vérité pour X0 et X1

H	HQ	X0
0	0	0
0	1	1
1	0	1
1	1	0

HQ	X1
0	1
1	0

Figure 1.5.9

Le schéma montre les [illegible] *'pulse'* (H) et *'quadrature pulse'* (Q) ; ceci est aussi valide pour les signaux verticaux. On remarque clairement le décalage de phase de H et HQ, pour chaque direction de mouvement. L'Amiga acquiert deux nouveaux signaux, X0 et X1, issus des impulsions H et HQ, par opérations logiques. X1 correspond à l'inverse de HQ, alors que X0 est le résultat d'un OU exclusif entre H et HQ, c'est-à-dire que X0 sera toujours à 1 lorsque les niveaux de H et HQ seront différents. Avec ces deux signaux, l'Amiga gère un compteur 6 bits, qui s'incrémentera ou se décrémentera suivant la direction de X1. L'ensemble X0, X1 correspond à une valeur 8 bits, qui représente la position actuelle de la souris.

Si la souris est déplacée vers la droite ou vers le bas, cette valeur sera incrémentée. Si la souris est déplacée vers la gauche ou vers le haut, cette valeur sera décrémentée.

Il existe 4 compteurs semblables, inclus dans DENISE, deux par *gameport*, où peut être branchée la souris. Ils se nomment *JOYDAT0* et *JOYDAT1*.

JOY0DAT $00A
(souris sur le gameport 0)

JOY1DAT $00C
(souris sur le gameport 1)

Bit	:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Fonction	:	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	X7	X6	X5	X4	X3	X2	X1	X0

(Les deux registres ne sont accessibles qu'en lecture).

Y0-7 compteur pour les mouvements verticaux (direction Y).
X0-7 compteur pour les mouvements horizontaux (direction X).

La souris génère deux cent impulsions compteur par pouce (79 par centimètre) c'est-à-dire que la limite du compteur est vite atteinte. Un nombre 8 bits permet d'obtenir 256 valeurs. On ne peut déplacer la souris que de 4 centimètres, avant de dépasser ce chiffre de 256. Ceci est possible, aussi bien par *incrémentation* (saut de 0 à 255) que par *décrémentation* (saut de 255 à 0). C'est pour cette raison qu'il faut interroger le registre compteur à des intervalles déterminés et vérifier s'il y a eu dépassement ou non.

Le système d'exploitation prend ce test en charge pendant l'interruption *écran vide vertical*.

On pourra ainsi considérer que le mouvement de la souris, entre deux tests, ne dépasse pas 127 pas du compteur. L'ordinateur compare alors le nouvel état avec le précédent. Si la différence est supérieure à 127, le compteur a un dépassement indiquant que la souris a été déplacée vers la droite (ou vers le haut). Si elle est inférieure à -127, le compteur a un dépassement, indiquant un mouvement vers la gauche (ou vers le haut).

Etat compteur précédent	Etat compteur actuel	Différence réelle	Mouvement de souris	Dépassement haut/bas
100	200	-100	+100	non
200	100	+100	-100	non
50	200	-150	-105	bas
200	50	+150	+105	haut

La différence = état précédent - état actuel

S'il y a dépassement vers le bas, le mouvement réel de la souris sera calculé comme suit :

-255 - différence, ce qui donne par calcul : -255 - (50-200) = -105

S'il y a dépassement vers le haut, le mouvement réel de la souris sera calculé comme suit :

255 - différence, ce qui donne par calcul : 255 - (200-50) = +105

Un mouvement positif de la souris correspond à un déplacement vers la droite (ou vers le bas), alors qu'une valeur négative correspond à un déplacement vers la gauche (ou vers le haut).

Le compteur souris peut être déterminé par le software. On pourra ainsi écrire n'importe quelle valeur dans ce dernier au moyen du registre *JOYTEST*.

Ce dernier influence [illegible] deux gameports. Cela veut dire que les compteurs horizontaux et verticaux seront initialisés avec la même valeur (*JOY0DAT = JOY1DAT*).

JOYTEST $036 (écriture)

Bit	:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Fonction	:	Y7	Y6	Y5	Y4	Y3	Y2	xx	xx	X7	X6	X5	X4	X3	X2	xx	xx

Comme on peut le remarquer, seuls 6 bits du compteur seront influencés. Ceci est logique si on se rappelle que les deux bits inférieurs proviennent directement du signal souris et non d'un registre interne se trouvant en mémoire, que l'on pourrait modifier.

Les joysticks

Lorsqu'on examine l'organisation des *gameports*, on s'aperçoit que les quatre directions du joystick correspondent aux quatre signaux de la souris. Les signaux du joystick seront donc traités de la même façon que ceux de la souris et au moyen des mêmes registres. Ainsi chaque paire de signaux sera combinée avec les bits X0 et X1 (ou Y0 et Y1). Les positions du joystick seront communiquées de la manière suivante :

Joystick à droite	X1=1	(bit 1 JOYxDAT)
Joystick à gauche	Y1=1	(bit 9 JOYxDAT)
Joystick vers l'arrière	X0 EOR X1 = 1	(bits 0 et 1 JOYxDAT)
Joystick vers l'avant	Y0 EOR Y1 = 1	(bits 8 et 9 JOYxDAT)

Pour établir la position du joystick soit vers l'avant, soit vers l'arrière, on doit exécuter une opération logique OU exclusif entre respectivement, Y0 et Y1, et X0 et X1.

Si le résultat correspond à 1, c'est que le joystick se trouve dans la position testée. Le programme en assembleur, ci-dessous, prend en charge le test de position sur le *gameport 1* :

```
Testjoystick:
move.w $DFF00C,d0       ;JOY1DAT transféré dans d
btst #1,d0              ;test du bit n°1
bne droite              ;s'il est activé, le joystick est à droite
btst #9,d0              ;test du bit n°1
bne gauche              ;s'il est activé, le joystick est à gauche
move.w d0,d1            ;copie de d0 dans d1
lsr.w #1,d1             ;Y1 et X1 sur la position de Y0 et X0
eor.w d0,d1             ;ou exclusif : Y1 eor Y0 et X1 eor X0
btst #0,d1              ;résultat du test X1 eor X0
bne arrière             ;résultat = 1 -> joystick vers l'arrière
btst #8,d1              ;résultat du test Y1 eor Y0
bne avant               ;résultat = 1 -> joystick vers l'avant
bra milieu              ;le joystick est en position centrale
```

L'opération OU exclusif de ce programme est employée de la manière suivante :

Une copie du registre *JOY1DAT* est réalisée dans d1, afin de décaler le contenu d'un bit vers la droite. Ainsi X1 dans d1 et X0 dans d0 possèdent la même position de bit. Le processus est le même pour les signaux Y1 et Y0. Une opération EOR entre d0 et d1 teste immédiatemment X1 avec X0 et Y1 avec Y0. Puis le résultat de ce test sera mis dans d1, pour être traité par la suite avec l'instruction *BTST*.

Ce programme ne prend pas en compte la position diagonale.

Les paddles

L'Amiga possède deux entrées analogiques par *gameport*, auxquelles on peut raccorder deux résistances variables ou potentiomètres. Ces derniers possèdent pour chaque position, une résistance déterminée que le hardware communique à PAULA. Un paddle renferme un tel potentiomètre, qui est activé grâce à un bouton de commande. Les joysticks analogiques fonctionnent de la même façon où chaque potentiomètre des directions X et Y donne la position réelle.

Deux registres renferme[illegible] es 4 valeurs 6 bits des entrées analogiques *POT0DAT* (gamepor[illegible] et *POT1DAT* (gameport 1) :

POT0DAT $012 et POT1DAT $014

Bit n°	:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Fonction	:	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	X7	X6	X5	X4	X3	X2	X1	X0

(Ces deux registres ne sont accessibles qu'en lecture).

Etant donné qu'un ordinateur ne peut traiter que des signaux digitaux, celui-ci nécessite un circuit spécial lorsqu'il doit traiter un signal analogique. Sur l'Amiga, la détermination de la valeur de la résistance externe fonctionne de la manière suivante :

Les potentiomètres doivent avoir une valeur de résistance maximum de 470 kilo-Ohm (+- 10%). Il est relié d'un côté avec la sortie +5 volts et de l'autre avec l'une des 4 entrées paddle du gameport. Ces dernières sont reliées aux entrées correspondantes de PAULA, et à 4 condensateurs qui se trouvent entre l'entrée et la masse.

La mesure commence au moyen d'un bit spécial de départ. Paula met alors rapidement les entrées paddle sur la masse, ce qui permet de décharger les condensateurs, tout en initialisant les registres *POTxDAT*. Puis, ce compteur est incrémenté d'une unité, avec chaque ligne de l'écran, pendant que les condensateurs se chargent à nouveau lentement sur les résistances. Si la tension des condensateurs dépasse une valeur déterminée, le compteur correspondant sera stoppé. C'est pour cette raison que l'état du compteur correspond exactement à la taille de la résistance (une petite résistance correspond à un état bas du compteur, une grande résistance correspond à un état haut du compteur).

Le bit de départ se trouve dans le registre POTGO :

POTGO $034 (écriture) et POTGOR $016 (lecture)

Bit	Nom	Fonction
15	OUTRY	gameport 1 POTY commuté en sortie
14	DATRY	bit de données gameport 1 POTY
13	OUTRX	gameport 1 POTX commuté en sortie
12	DATRX	bit de données gameport 1 POTX
11	OUTLY	gameport 0 POTY commuté en sortie
10	DATLY	bit de données gameport 0 POTY
9	OUTLX	gameport 0 POTX commuté en sortie
8	DATLX	bit de données gameport 0 POTX
7 à 1		inutilisés
0	START	déchargement des condensateurs début de la mesure.

(Un accès écriture sur *POTGO* initialise les deux registres *POTxDAT*).

Normalement, le bit *START* est initialisé à 1, pendant le temps mort vertical. Pendant l'affichage de l'image, les condensateurs se chargent, atteignant leur valeur seuil et stoppant le compteur. Pendant le temps mort suivant, on peut lire dans les registres *POTxDAT*, l'état du potentiomètre.

Les 4 entrées analogiques se laissent programmer comme des entrées/sorties normales digitales. Les bits de gestion et de données correspondants, se trouvent dans le registre *POTGO*, comme le bit *START*. Au moyen des bits *OUTxx (OUTxx = 1)*, on peut commuter chaque signal sur le mode sortie. Ainsi chaque ligne sera séparée du circuit de gestion des condensateurs et la valeur du bit *DATxx (POTGO)* pourra y être émise.

La lecture des bits *DATxx* de *POTGOR* donnera toujours l'état actuel du signal concerné.

Si on utilise les ports analogiques comme sortie, on doit faire attention à ce qui suit :

Comme les 4 ports analogiques internes sont reliés aux condensateurs pour la mesure de la résistance (47 nF), il peut s'écouler 300 microsecondes jusqu'à ce que le signal prenne le niveau souhaité, et ceci à chaque fois que le condensateur a une inversion de charge.

Le bouton de ces outils périphériques

Chacun des trois outils étudiés plus haut possèdent un ou plusieurs boutons. Le tableau suivant montre les registres concernant le statut du bouton de la *souris*, du *paddle* ou du *joystick*.

Gameport 0 :

Bouton gauche souris	CIA, port parallèle A, bit de port 6
Bouton droit souris	POTGOR, DATLY
Troisième bouton souris	POTGOR, DATLX
Bouton feu joystick	CIA-A, port parallèle A, bit de port 6
Bouton gauche paddle	JOY0DAT, bit 9 (1 = bouton activé)
Bouton droit paddle	JOY0DAT, bit 1 (1 = bouton activé)

Gameport 1 :

Bouton gauche souris	CIA, port parallèle A, bit de port 7
Bouton droit souris	POTGOR, DATRY
Troisième bouton souris	POTGOR, DATRX
Bouton feu joystick	CIA-A, port parallèle A, bit de port 7
Bouton gauche paddle	JOY1DAT, bit 9 (1 = bouton activé)
Bouton droit paddle	JOY1DAT, bit 1 (1 = bouton activé)

(Lorsque ce n'est pas précisé, les bits sont 0-actifs, c'est-à-dire que 0 correspond à un bouton activé).

1.5.10 LE CONNECTEUR SERIE

Comme cela a été vu au chapitre 1.3.4, l'Amiga possède un connecteur RS232 standard. Les signaux de ce boitier peuvent être divisés en deux groupes de signaux :

1. *Les signaux de données séries*
2. *Les signaux Handshake*

Examinons d'abord le groupe 2. Le connecteur RS232 possède une grande quantité de signaux *Handshake*. La plupart ne seront d'ailleurs pas du tout utilisés. De plus, les contenus de ces signaux ne sont pas les mêmes sur chaque boitier RS232. Leur fonction et programmation ont déjà été décrites dans le chapitre 1.3.4.

Sur les deux signaux du groupe 1 se déroule le transfert de toutes les informations. Le signal *RXD* reçoit les données et le signal *TXD* les émet. La communication RS232 peut donc se faire dans les deux sens au même moment, lorsque deux appareils sont reliés ensemble aux signaux *RXD* et *TXD*. On branchera ainsi le signal *RXD* d'un des appareils avec le signal *TXD* de l'autre et inversement.

Principe du transfert de données séries RS232

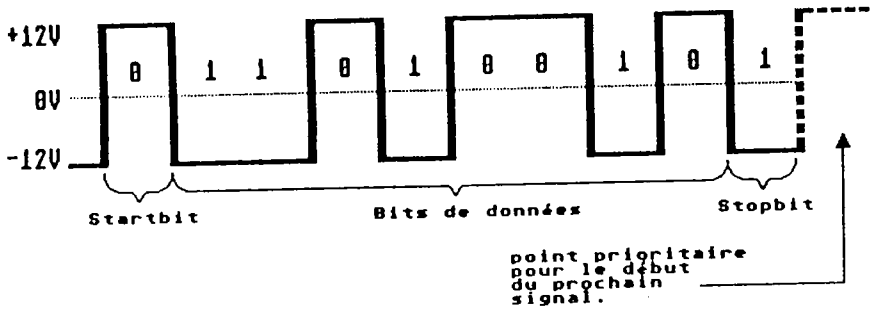


Schéma du déroulement de transfert de données séries.

Figure 1.5.10

Etant donné que le transfert de données pour une direction n'occupe qu'un seul signal, les mots de données doivent être convertis en courant de données, le transfert s'effectuant alors bit après bit. La norme RS232 ne prévoit aucun signal d'horloge. Pour que le destinataire sache quand le prochain bit doit être lu, le temps par bit doit être constant, c'est-à-dire que la vitesse de lecture ou d'émission de données doit être déterminée. C'est ce qu'on fait en établissant le *baudrate (taux)*. Ce dernier détermine le nombre de bits transférés par seconde. Les taux de transfert usuels sont 300, 1200, 2400, 4800, 9600 bauds.

On n'est cependant pas obligé de s'en tenir à ces valeurs, mais il faut faire attention, lors de l'utilisation de taux peu communs, à ce que l'émetteur et le destinataire autorisent la même vitesse de transmission.

Pour que le transfert fonctionne, le destinataire doit encore connaître le moment où un octet débute et celui où il se finit. Le schéma 1.5.10 montre le *timing* du transfert d'un octet sur les signaux de données. Chaque octet commence avec un bit de départ, qui ne se différencie en rien des bits de données, si ce n'est qu'il est toujours à 0. Puis suivent les bits dans l'ordre croissant, c'est-à-dire, du *LSB* au *MSB*. La fin est marquée par un ou deux bits d'arrêt. Le destinataire reconnaît l'alternance des octets par le changement de niveau de 1 à 0, qui résulte de la suite bit d'arrêt, bit de départ.

Le circuit qui exécute le transfert série, se nomme *Universal Asychronous Receive Transmit (UART)*. Il est inclus dans PAULA et ses registres se trouvent dans la zone registre du circuit spécialisé :

Les registres UART

SERPER $032 (écriture)

Bit	Nom	Fonction
15	LONG	La longueur des données reçues est de 9 bits
0 à 14	RATE	Ce nombre 15 bits correspond au Baudrate (taux)

SERDAT $030 (écriture)

Ce registre contient les données émises.

SERDATR $018 (écriture)

Bit	Nom	Fonction
15	OVRUN	Dépassement du registre à décalage de destination
14	RBF	Le tampon des données reçues est plein
13	TBE	Le tampon des données émises est vide
12	TSRE	Le registre à décalage d'émission est vide
11	RXD	Correspond au niveau du signal RXD
10	---	inutilisé
9	STP	Bit d'arrêt
8	STP ou DB8	Dépend de la longueur des données
7	DB7	Le tampon des données reçues/bit de données n°7
6	DB6	Le tampon des données reçues/ bit de données n°6
5	DB5	Le tampon des données reçues/ bit de données n°5
4	DB4	Le tampon des données reçues/ bit de données n°4
3	DB3	Le tampon des données reçues/ bit de données n°3
2	DB2	Le tampon des données reçues/ bit de données n°2
1	DB1	Le tampon des données reçues/ bit de données n°1
0	DB0	Le tampon des données reçues/ bit de données n°0 .

Un bit du registre *ADKCON* appartient aussi à la gestion *UART* :

ADKCON $09E (écriture) ADKCONR $010 (lecture)

Bit n°11 : UARTBRK

Ce bit stoppe la sortie sérielle et met TXD à 0.

Le déroulement du tr sfert de données UART de l'Amiga

La réception

La réception de données séries se déroule en deux stades. Les bits, arrivant sur la broche *RXD*, seront pris en charge par le registre à décalage à la vitesse déterminée par le *baudrate* et reconstitués pour données un mot de données parallèle. Si le registre à décalage est plein, son contenu sera écrit dans le tampon de données reçues. Il sera ainsi libre de suite pour les prochaines données.

Le processeur peut lire ce tampon, mais non le registre à décalage. Les bits de données correspondants, *DB0-7*, se trouvent dans le registre *SERDATR*.

L'Amiga peut réceptionner 8 ou 9 bits de données. *L'UART* est commuté en mode 9 bits, au moyen du bit *LONG* du registre *SERPER* (LONG = 1).

La longueur fixée des données détermine le format utilisé dans le registre *SERDATR*. Pour 9 bits, le bit 8 de *SERDAT* contient le 9ème bit de données, alors que le bit d'arrêt correspond au bit 9. Pour 8 bits de données, le bit 8 contient déjà le bit d'arrêt, le bit 9 étant réservé au deuxième bit d'arrêt.

L'état du registre à décalage et du tampon de données est donné par deux bits présents dans *SERDATR* :

RBF correspond au signal *'receive-buffer-full'*, c'est-à-dire que dès qu'un mot de données, issu du registre à décalage, est transféré dans le buffer, ce bit se met à 1, signalisant ainsi au microprocesseur qu'il doit lire les données dans *SERDATR*.

Ce bit existe aussi dans les registres d'interruption (*RBF, INTREQ/ INTEN* bit n° 11). Après que le processeur ait lu les données, il doit remettre *RBF* à 0 dans *INTREQ*. Ceux des registres *INTREQR* et *SERDATR* seront aussi remis alors à 0.

```
move.w #$0800,$DFF000+INTREQ   ;désactiver RBF dans INTREQ et SERDATR
```

Si on s'abstient, le registre à décalage réceptionner[illegible]ouveau mot de données complet, *l'UART* initialisant le bit *OVRUN*. Ceci signale qu'aucune donnée suivante ne peut être prise en compte, étant donné que le buffer *(RBF=1)* et le registre à décalage *(OVRUN=1)* sont pleins. *OVRUN* sera mis à 0 lorsque *RBF* sera désactivé. Puis *RBF* se remettra à 1, étant donné que le contenu du registre à décalage sera aussitôt transféré dans le buffer.

L'émission

Le processus d'émission se déroule aussi suivant deux phases. Le tampon d'émission de données se trouve dans le registre *SERDAT*. Dès qu'un mot de données y est inscrit, il sera transféré dans le registre à décalage de sortie. C'est ce que signale le bit *TBE*. *TBE* correspond à *Transmit Buffer Empty* et indique que *SERDAT* est prêt à prendre en compte les prochaines données. *TBE* est aussi présent dans les registres d'interruption (*TBE*, *INTREQ/INTEN* bit n°0). Comme *RBF*, *TBE* sera désactivé par le biais du registre *INTREQ*.

Dès que le registre à décalage a émis le mot de données, un nouveau mot sera prélevé automatiquement dans le buffer d'émission de données. Si ceci n'est pas possible, *l'UART* met le bit *TSRE* à 1 *(Transmit-Shift-Register-Empty)*. Ce bit sera remis à 0 par la désactivation du bit *TBE*.

La longueur des mots de données et le nombre de bits d'arrêt déterminent le format des données dans *SERDAT*. On écrit simplement le mot de données sur les 8 ou 9 bits inférieurs de *SERDAT*, ces derniers étant sulvis d'un ou deux bits d'arrêt (bit(s) mis à 1). Un mot de données 8 bits avec deux bits d'arrêt sera de la forme suivante :

Bit	:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Fonction	:	0	0	0	0	0	0	1	1	D7	D6	D5	D4	D3	D2	D1	D0

D0-7 sont les 8 bits de données.

Les deux 1 correspondent aux deux bits d'arrêt.

Un mot de données 9 bits avec un bit d'arrêt sera de la forme suivante :

Bit	:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Fonction	:	0	0	0	0	0	0	1	D8	D7	D6	D5	D4	D3	D2	D1	D0

Un mot de données 8 bits avec un bit d'arrêt sera de la forme suivante :

Bit	:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Fonction	:	0	0	0	0	0	0	0	1	D7	D6	D5	D4	D3	D2	D1	D0

Le bit *LONG* du registre *SERPER* a juste une influence sur la longueur des données réceptionnées. Le format des données émises sera déterminé seul avec la valeur présente dans le registre *SERDAT*.

Détermination du baudrate

Le taux d'émission ou de réception des données doit être écrit dans les 15 bits inférieurs du registre *SERPER*. On ne peut malheureusement pas établir directement ce taux, mais on doit choisir le nombre de cycles de bus présents entre deux bits (1 cycle de bus a une durée de $2.79365 * 10^{-7}$ secondes). Si tous les n cycles de bus, une donnée doit être émise, on devra écrire dans le registre *SERPER* la valeur n-1. Avec la formule suivante, on pourra transformer le taux en valeur utilisable :

$$SERPER = (1 / (baudrate * 2.79365 * 10^{-7})) - 1$$

Par exemple pour un taux de 4800 baud :

$$SERPER = (1 / (4800*2.7936510*10^{-7})) - 1 = 744,74$$

La valeur calculée sera ensuite arrondie et écrite dans le registre *SERPER* :

```
        move.w #745,$DFF000+SERPER          ;SERPER activé, LONG=0
ou
        move.w #$8000+745,$DFF000+SERPER    ;LONG=1
```

1.5.11 LE CONTROLEUR DISQUETTE

La gestion Hardware du lecteur de disquette se divise en deux parties.

La première correspond aux signaux de gestion, qui activent l'unité, le moteur, qui bougent la tête de lecture/écriture etc...

Ils sont tous reliés aux signaux correspondants de port du CIA. Leur fonction et initialisation ont déjà été décrites au chapitre 1.3.

Ce qui sera expliqué dans ce chapitre concerne essentiellement les signaux de données. C'est grâce à eux que les données seront enregistrées sur disquette ou chargées à partir de cette dernière. C'est un circuit spécial inclus dans PAULA, le contrôleur disquette, qui se charge du traitement de ces données.

Il possède un canal DMA qui se charge de lire ou d'écrire les données disquette.

La programmation du DMA disquette

Avant d'initialiser un accès DMA disquette, on doit vérifier que le précédent est terminé. Si on arrête le déroulement d'un accès écriture, on peut détruire les données de la piste correspondante. Soyez donc attentif à ce que le dernier accès DMA soit arrêté.

La première chose à faire est de déterminer l'adresse mémoire du buffer de données. Le DMA disquette utilise une des paires de registres adresses suivantes, *DSKPTH* et *DSKPTL*, comme pointeur sur la *CHIP-RAM*.

$20 DSKPTH pointeur de données disquette (bits 16-18)
$22 DSKPTL pointeur de données disquette (bits 0-15)

Puis il faut initialiser le registre *DSKLEN*. Ce dernier est organisé de la manière suivante :

DSKLEN $024 (écriture)

Bit	Nom	Fonction
15	DMAEN	Initialisation des accès DMA disquette
14	WRITE	Ecriture des données sur la disquette
0-13	LENGTH	Nombre de mots de données à transférer

LENGTH

Les 14 bits inférieurs du registre *DSKLEN* renferment le nombre de mots de données à transférer.

WRITE

Au moyen de *WRITE* (WRITE = 1), on commute le contrôleur disquette du mode lecture en mode écriture.

DMAEN

Lorsqu'on met le bit *DMAEN* à 1, le transfert de données commence. On doit cependant faire attention aux faits suivants :

1. Le bit *Disk-DMA-Enable* du registre *DMACON (DSKEN*, bit n°4) doit toujours être activé.

2. Pour éviter un accès écriture, initialisé par mégarde, on doit activer deux fois, et à la suite, le bit *DMAEN*. Puis l'accès DMA pourra débuter. Le bit *WRITE*, pour plus de sécurité, ne doit être activé que lors d'un accès écriture. Une séquence d'initialisation ordonnée pour un accès DMA disquette sera de la forme suivante :

   1. Désactiver *DMAEN* en mettant *DSKLEN* à 0.

   2. Tant que *DSKEN* du registre *DMACON* n'est pas activé, on doit exécuter ce qui suit à sa place.

3. Mettre l'adresse souhaitée dans *DSKPTH* et *DSK*

4. Mettre la bonne valeur (incluant *LENGTH*, *WRITE* et *DMAEN*) dans *DSKLEN*.

5. Réécrire la même valeur dans *DSKLEN*.

6. Attendre que les accès DMA disquette soit terminés.

7. Par mesure de sécurité, remettre *DSKLEN* à 0.

Afin que le processeur sache à quel moment le contrôleur disquette aura terminé le transfert des mots de données (dont le nombre est déterminé par *LENGTH*), il y aura interruption *DSKBLK (Disk BlocK Finished*, bit n°1 dans *INTREQ/INTEN)*. Elle sera libérée après lecture ou écriture du dernier mot de données.

Le statut actuel du contrôleur disquette se trouve dans le registre *DSKBYTR* :

DSKBYTR $01A (lecture)

Bit	Nom	Fonction
15	BYTEREADY	Ce bit signale que l'octet de données est valide pour les 8 bits inférieurs.
14	DMAON	DMAON indique si l'accès DMA disquette est initialisé. Dès que DMAON = 1, DMAEN dans DSKLEN et DSKEN dans DMACON doivent être initialisés.
13	DSKWRITE	Donne l'état de WRITE du registre DSKLEN.
12	WORDEQUAL	Données disquettes = DSKSYNC.
11 à 8		Inutilisés.
7 à 0	DATA	Octets de données actuels de la disquette.

Au moyen des 8 bits *DA* et du drapeau *BYTEREADY*, on peut données sur la disc par le processeur. A chaque fois qu'un octet données complet est arrivé à destination, le contrôleur disquette active le bit *BYTEREADY*. Ainsi, le processeur sait que l'octet présent dans les 8 bits *DATA* est valide. Après lecture du registre *DSKBYTR*, le bit *BYTEREADY* sera désactivé.

Il peut arriver qu'on ne veuille pas lire une piste entière. On aura alors la possibilité de démarrer les accès DMA à une position déterminée. On écrira, à ce moment là, le mot de données où le contrôleur disquette doit commencer, dans le registre *DSKSYNC*.

DSKSYNC $07E (écriture)

Le registre *DSKSYNC* contient le mot de données, où le transfert doit débuter.

Après initialisation des accès DMA disquette, le contrôleur disquette commence à lire les données sur la disquette, mais sans les écrire en mémoire. A la place, il les compare avec le mot de données présent dans *DSKSYNC*. Lorsque les deux concordent, il commencera un transfert normal. On pourra ainsi programmer le contrôleur disquette de telle façon, qu'il soit obligé d'attendre la marque de synchronisation au début d'un bloc de données.

Le bit *WORDEQUAL* du registre *DSKBYTR* sera mis à 1 dès que la donnée lue et celle présente dans *DSKSYNC* concordent. Etant donné que cette concordance ne dure que 2 microsecondes, le bit *WORDEQUAL* ne sera activé que pendant un temps sensiblement égal. De plus, une interruption sera générée chaque fois que *WORDEQUAL* sera à 1.

Le bit n° 12 du registre *INTREQ* (respectivement *INTEN*) correspond au bit d'interruption *DSKSYNC*. Il sera activé lorsque les données disquettes concorderont avec *DSKSYNC*.

Paramètres d'exploitation

Les données ne peuvent pas être écrites sur disquette, avec le format utilisé en mémoire. Elles doivent être décodées au préalable. Normalement, l'Amiga travaille avec le code *MFM*, mais il peut aussi utiliser le code *GCR*. Deux opérations sont nécessaires pour sélectionner le code souhaité :

1. Une routine doit coder les données avant l'écriture sur disquette, et décoder les données lues.

2. Le contrôleur disquette doit mettre au point le codage correspondant. C'est ce que permet le registre *ADKCON*.

ADKCON $09E (écriture) ADKCONR $010 (lecture)

Bit	Nom	Fonction
15	SET/CLR	Bits activés (SET/CLR=1) ou désactivés
14-13	PRECOMP	Ces bits contiennent la valeur PRECOMP :
		Bit 14 Bit 13 Temps PRECOMP
		0 0 nul
		0 1 140 nanosecondes
		1 0 280 nanosecondes
		1 1 560 nanosecondes
12	MFMPREC	GCR = 0, MFM = 1
11	UARTBRK	Cf. UART
10	WORDSYNC	WORDSYNC = 1 active la synchronisation GCR du contrôleur disquette sur le mot présent dans le registre DSKSYNC.
9	MSBSYNC	MSBSYNC = 1 active la synchronisation MSB
8	FAST	Taux d'horloge du contrôleur disquette FAST = 1: 2 microsecondes/bit (MFM) FAST = 0: 4 microsecondes/bit (GCR)
7-0		Cf. AUDIO chapitre 1.5.8

Le contrôleur DMA transfère les données de la mémoire vers un registre données correspondant. Le contrôleur disquette possède un registre données pour celles issues de la disquette et un pour celles écrites sur disquette.

DSKDAT $026 (écriture)

Ce registre renferme les données qui doivent être écrites sur disquette.

DSKDAT $008 (lecture)

Ce registre renferme les données issues de la disquette. C'est un registre du type *Early Read Register*, qui ne peut pas être accessible au processeur.

2. EXEC

Ce chapitre portera sur le système d'exploitation de l'Amiga, que tout utilisateur doit connaître, au moins dans ses grandes lignes.

2.1 Eléments de base du système d'exploitation

Le système d'exploitation, qui est chargé sur l'Amiga 1000 à partir de la disquette *KICKSTART*, se trouve dans la zone d'adressage supérieure sur environ 256 Koctets ($FC0000 - $FFFFFF). Dans cette grande zone mémoire, un nombre important de routines trouvent leur place, et permettent d'alléger considérablement le travail du programmeur. Ces routines sont classées sur l'Amiga, suivant les différentes tâches. On peut se représenter le système comme un grand nombre de modules indépendants, qui se partagent les différentes tâches. Les parties importantes sont : *DOS* (gestion des entrées/sorties), *graphisme*, *Intuition* (ensemble des routines qui s'occupent de la gestion des fenêtres et des écrans) et *Exec*.

Le rôle *d'Exec* est de gérer le multi-tâches et de permettre ainsi le travail simultané de plusieurs programmes. De plus, *Exec* s'occupe des niveaux inférieurs entre le programme et le hardware. La responsabilité de ces tâches font donc de *Exec* la partie la plus importante du système d'exploitation de l'Amiga.

Chaque partie du système met à disposition un nombre important de routines, qui peuvent être utilisées assez facilement par le programmeur. La façon dont ces routines sont appelées est expliquée au chapitre 2.3.

Les communications entre *Exec* et le hardware ne se déroulent pas directement, mais seront prises en charge par un *Device-Handler* (pilote de périphérique).

Des routines complètes, qui permettent d'accéder au *Device-Handler*, sont intégrées dans *Exec*.

machine, mais en renonçant aux capacités multi-tâches du système.

2.2 Introduction à la programmation

Avant de décrire la structure du système d'exploitation, il faut éclaircir certains points de programmation.

Remarque :

Les listages assembleur issus de la zone kickstart concernent la version 1.2, qui est implantée dans la ROM de l'Amiga 500 et 2000. Pour les possesseurs de l'Amiga 1000, les routines ne correspondront que pour la même version du kickstart.

2.2.1 DISTINCTION ENTRE C ET ASSEMBLEUR

Il est utile de connaître un langage tel que le C, ainsi que de pouvoir programmer en assembleur, ce dernier permettant surtout d'accéder à une grande vitesse de traitement. L'utilisation des routines du système d'exploitation ou de certaines structures ne demande cependant qu'un minimum de connaissances.

Nous allons débuter par la syntaxe des appels de routine avec choix de paramètres. Cet appel, en C, est de la forme suivante :

```
Enregistrement = FindName(Liste,"nom")
      d0                  a0      a1
```

FindName est une routine dont le but est de trouver un enregistrement dans une liste d'enregistrements, avec l'aide de son nom. Les paramètres, qui doivent être fournis, sont des pointeurs sur le début de la liste composée, et sur le nom de l'enregistrement qui est cherché.

La programmation en assembleur se fait à l'aide des routines (fonctions) définies des registres, dans lesquelles on transmet les paramètres utiles à l'appel de la routine.

On établit alors le pointeur sur l'enregistrement se trouvant en mémoire. Le pointeur ainsi sur la fonction sera, pour l'appel de cette dernière ou autres, transmis dans le registre d0. Lorsqu'il y aura emploi de plus d'un paramètre, on utilisera un autre registre.

L'appel d'une routine en assembleur est similaire. Il suffit de mettre les paramètres de cette dernière dans les registres correspondants. En voici un exemple :

```
lea.l   Liste,a0
lea.l   Nom,a1
jsr     FindName
move.l  d0,mémoire
        .
        .
        .
Nom:
dc.b    "Nom",0
```

Le pointeur de la liste sera mis dans a0, alors que a1 contient le pointeur sur le nom cherché. La chaîne de caractères doit être suivie d'un 0. Après quoi, la routine sera appelée et le pointeur de l'enregistrement sera transmis à d0, d'où il sera stocké en mémoire. Il est d'usage que l'appel se fasse par d0. L'appel écrit en C peut être interprété de la même manière.

Pour finir, nous allons voir comment les structures en C apparaissent en assembleur.

```
struct Node {
   struct Node *ln_Succ;
   struct Node *ln_Pred;
   UBYTE        ln_Type;
   BYTE         ln_Pri;
   char        *ln_Name;
}MonNode;
```

le rôle de cette structure ne sera pas abordé ici. Elle sert exclusivement à illustrer les différences qui existe entre le C et l'assembleur. L'initialisation d'une structure en C ne doit présenter aucun problème.

Exemple :

```
MonNode.ln_Pri = 20;
```

La valeur 20 a été affectée au champ ln_Pri.

En assembleur, une telle structure C se présente comme un tableau. Les valeurs d'une structure donnée seront présentées à la suite, avec leur longueur propre. On doit évidemment connaître l'adresse de base d'un tel tableau pour y accéder.

Exemple :

```
base + 0  $00000000    pointeur sur la suite ln_Succ
base + 4  $00000000    pointeur sur le prédecesseur ln_Pred
base + 8  $00          ln_Type
base + 9  $00          ln_Pri
base + 10 $00000000    pointeur sur le nom ln_Name
```

Les 0 remplacent n'importe quelle valeur. Pour mettre ln_Pri à 20, comme on vient de le faire en C plus haut, l'adresse de base de la structure doit être connue.

```
lea.l   base+9,a0
move.b  #20,(a0)
```

Vous pouvez voir que l'accès à une structure en assembleur ne pose aucun problème. Evidemment, ce n'est pas aussi simple qu'en C, puisque l'adresse de base de la stucture doit être connue. L'assembleur a tout de même l'avantage d'être plus rapide que le C, et on pourra, de plus, s'orienter sur le hardware en programmant de cette manière.

Avec l'assembleur, on pourra accéder à de nombreuses routines, inaccessibles en C. Ces routines gèrent une partie du système d'exploitation, peu utilisée par le programmeur, mais qui par de nombreuses astuces, notamment dans la gestion du multi-tâches, sont indispensables.

2.2.2 STRUCTURE DES NOEUDS

Les éléments de base des structures ont été décrits dans le chapitre précédent, car leur connaissance est essentielle pour la compréhension de ce qui suit.

La structure *Node* (noeud) est utilisée pour établir les enchaînements de listes, qui seront souvent employées. Cette structure est la plus utilisée du système d'exploitation de l'Amiga. Sa forme est la suivante :

En C :

```
struct Node {
      struct Node *ln_Succ;
      struct Node *ln_Pred;
      UBYTE        ln_Type;
      BYTE         ln_Pri;
      char        *ln_Name;
     };
```

En assembleur :

```
base + 0  $00000000    pointeur sur la suite ln_Succ
base + 4  $00000000    pointeur sur le prédecesseur ln_Pred
base + 8  $00          ln_Type
base + 9  $00          ln_Pri
base + 10 $00000000    pointeur sur le nom ln_Name
```

Cette structure peut se diviser en deux parties. La première concerne l'enchaînement *(ln_Succ et ln_Pred)*, la seconde les données *(Type, Pri* et *Name)*.

*ln_Succ

Pointeur sur le prochain noeud (Successor = prochain).

*ln_Pred

Pointeur sur le noeud précédent (Predecessor = précedent).

ln_Type

A chaque type de noeud correspond un numéro particulier qui sera rangé dans cet octet.

ln_Pri

Ceci marque la priorité du noeud. Ce champ, initialisé par une valeur différente de 0, n'est pris en compte que dans certains cas, comme par exemple un noeud de type *Task*.

*ln_Name

Dans ce mot long, il sera mis en mémoire un pointeur sur une chaîne de caractères terminée par un 0. C'est le nom du noeud qui est choisi d'une telle manière, que l'on sache immédiatement de quel noeud il s'agit, évitant ainsi bien des erreurs.

Initialisation d'un noeud

Avant qu'un noeud ne soit relié à une liste, il doit tout d'abord être initialisé. C'est pourquoi il est nécessaire d'établir son type. Voici plusieurs types standard :

Node-type	Type de noeud
NT_TASK	01
NT_INTERRUPT	02
NT_DEVICE	03
NT_MSGPORT	04
NT_MESSAGE	05
NT_FREEMSG	06
NT_REPLYMSG	07
NT_RESOURCE	08
NT_LIBRARY	09
NT_MEMORY	10
NT_SOFTINT	11
NT_FONT	12
NT_PROCESS	13
NT_SEMAPHORE	14

Un noeud se trouvant dans une structure *Task*, doit être initialisé. Pour comprendre comment cette initialisation se passe, nous allons montrer la construction de cette structure avec les parties importantes.

```
struct Task {
 struct Node tc_Node;
        .
        .
        .
};
```

Initialisation en C d'un noeud dont le type est *Task* :

```
 struct Task matask;  /* matask est le nom de la structure task */

 matask.tc_Node.ln_Type = NT_TASK
```

Enfin, l'initialisation équivalente en assembleur.

```
lea.l matask,a0     ;adresse de base de task dans a0
move.l #01,8(a0)    ;Type = task (valeur 01) initiali
```

Quand le type est déterminé, il s'agit alors d'établir la priorité du noeud par comparaison avec les autres. Il peut prendre une valeur comprise entre -128 et +127. La priorité la plus importante correspond à la valeur +127, -128 étant le niveau de priorité le plus faible.

Quelques listes *Exec* sont classées suivant l'ordre de priorité, où l'enregistrement prioritaire correspond au début. La plupart n'utilisent pas l'enregistrement *ln_Pri.* Il est souvent important de mettre les priorités d'une telle liste à 0.

L'initialisation de la priorité se fait de la manière suivante :

```
matask.tc_Node.ln_Pri = 5;
```

et en assembleur :

```
lea.l matask,a0    ;adresse de base de task dans a0
move.l #05,9(a0)   ;Pri initialisé à 5
```

Pour finir, il faut indiquer le nom du noeud :

En C :

```
matask.tc_Node.in_name = "exemple de task";
```

En assembleur :

```
lea.l matask,a0     ;adresse de base de task dans a0
lea.l nom,a1        ;adresse du nom se trouve dans a1
move.l a1,10(a0)    ;pointeur sur le nom à enregistrer
nom:
dc.b "exemple de task",0
```

La chaîne de caractères doit être terminée par un 0. Dans cet exemple, on peut remarquer que seule la position de l'enregistrement est à connaître pour initialiser une telle structure en assembleur.

L'espace compris entre l'adresse de base et la position atteinte correspond à l'offset. Da notre dernier exemple, il a une valeur de 10.

L'Initialisation de *ln_Succ* et de *ln_Pred* sera vue dans le chapitre suivant.

2.2.3 LES LISTES

Une liste est une série de structures noeud, qui sont enchaînées les unes aux autres. Comme cela a déjà été vu, on trouve en premier lieu dans une structure, un pointeur *(ln_Succ)* sur le prochain noeud. Puis en deuxième lieu, un pointeur *(ln_Pred)* sur le noeud précédent.

Pour améliorer la gestion d'une liste, on a introduit une tête de liste, marquant la fin et le début d'une liste. En dehors de cette information, il apparaît aussi dans la structure, la forme d'enregistrement dont il s'agit. Une structure de liste est donc de la forme suivante :

Les nombres se trouvant devant les membres de la structure suivante correspondent aux offsets. On pourra ainsi y accéder également par l'assembleur. Les offsets n'appartiennent pas à la structure C et ne doivent pas être pris en compte. Mais ils seront les bienvenus pour le programmeur en assembleur.

```
    struct Liste {
 0    struct Node *lh_Head;
 4    struct Node *lh_Tail;
 8    struct Node *lh_Tailpred;
12    UBYTE  lh_Type;
13    UBYTE  lh_pad;
  );
```

**lh_Head*

Pointeur sur le premier enregistrement (noeud) d'une liste.

**lh_Tail*

Toujours égal à 0.

Pointeur sur le dernier enregistrement de la liste.

lh_Type

Donne le type des noeuds liés dans une liste. *lh_Type* sera défini comme le type du nom.

lh_pad

lh_pad sert uniquement à ce que la structure ait un nombre pair d'octets.

Le pointeur *ln_Succ* du dernier enregistrement pointe sur *lh_Tail* (deuxième enregistrement de la structure liste). *lh_Tail* est nul (NIL) et indique que le noeud supposé lu n'est plus valide.

Le pointeur *ln_Pred* du premier noeud pointe sur *lh_Succ* (fait aussi partie de la structure liste). Si le pointeur *ln_Pred* supposé est lu, *lh_Tail* deviendra nul, en indiquant ainsi que cet enregistrement était le premier de la liste.

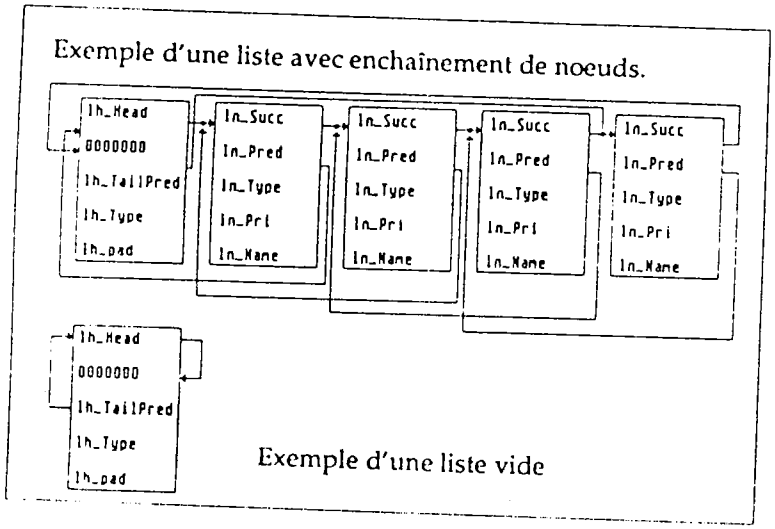


Figure 2.2

Initialisation d'une liste

Après avoir détaillé la structure d'une liste, nous allons voir comment on l'initialise. Dans ce but, il est préférable de générer une structure liste vide. Voici l'initialisation en C :

```
#include "Exec/lists.h"
main ()
{
struct List Liste;
Liste.lh_head = (struct Node *)&Liste.lh_Tail;
Liste.lh_Tail = 0;
Liste.lh_Tailpred = (struct Node *)&Liste.lh_head;
Liste.lh_Type = NT_TASK;
}
```

En assembleur, l'initialisation est de la forme :

```
lea.l  Liste,a0
move.l a0,(a0)
addq.l #04,(a0)
clr.l #04(a0)
move.l a0,8(a0)
move.b #01,12(a0)
```

Une liste générée de cette manière ne peut être que vide. L'emboîtement des noeuds d'une liste sera détaillé plus loin. A la place, on va s'appliquer à mettre en évidence le fait qu'une liste est vide. Il existe deux méthodes de détermination. L'une consiste à chercher si *lh_Head* pointe sur 0 *(NIL)*, l'autre à établir si *lh_TailPred* pointe sur le début de la liste *(lh_Head)*. Si c'est le cas, la liste concernée est vide.

Le test en C est de la forme suivante :

```
if (Liste.lh_TailPred == &Liste) {
   printf ("la liste est vide");
   }
```

2.2.4 ROUTINES EXEC POUR LA GESTION DES LISTES

Pour gérer les listes, *Exec* met à disposition une serie de fonctions utilisables, facilitant le déroulement des opérations.

La première fonction correspond à *Insert()*. Elle sert à emboîter les noeuds d'une liste, avec référence de la position où l'emboîtement à lieu.

Insert

```
Insert(Liste, Noeud, Prédécesseur);
         a0     a1        a2
```

Offset : -234

Paramètres :

Liste Pointeur sur la liste où le noeud doit être emboîté.

Noeud Pointeur sur le noeud dans la liste où a lieu l'emboîtement.

Prédécesseur Pointeur sur le noeud précédant l'emboîtement. Si ce pointeur a une valeur différente de 0, le paramètre Liste n'est plus essentiel. Ce ne sera pas le pointeur du noeud qui sera employé pour déterminer la position de l'emboîtement, mais le paramètre prédécesseur. Si ce dernier est à 0, le noeud sera inséré à la première position. La deuxième possibilité de mettre un noeud en première position est de mettre prédécesseur sur *lh_Head*. Pour la dernière position, il suffit d'initialiser ce paramètre sur *lh_TailPred*. Pour placer un noeud en première ou dernière position, il existe encore un certain nombre d'autres fonctions.

Remove

```
Remove(Noeud);
         a1
```

Offset : -252

Description : Comme le nom l'indique, cette fonction sert à l'élimination des noeuds d'une liste.

Paramètre :

Noeud Pointeur sur le noeud qui doit être éliminé. S'il ne s'agit pas d'un pointeur sur un noeud, Exec ne le reconnaît pas et 'élimine' tout de même, ceci pouvant planter le système.

AddHead

```
AddHead(Liste, Noeud);
          a0     a1
```

Offset : -240

Description : Cette fonction est utilisée pour insérer un noeud en tête de liste.

Paramètres :

Liste Pointeur sur la liste où le noeud doit être inséré.

Noeud Pointeur sur le noeud à insérer.

RemHead

```
RemHead(Liste);
          a0
```

Offset : -258

Description : Elimine le premier noeud de la liste indiquée.

Paramètre :

Liste Pointeur sur la liste où le premier noeud sera éliminé.

AddTail

```
AddTail(Liste, Noeud)
          a0      a1
```

Offset : -246

Description : Insérer un noeud comme dernier membre d'une liste

Paramètres :

Liste Pointeur sur la liste où le noeud doit être inséré.

Noeud Pointeur sur le noeud qui doit être inséré.

RemTail

```
RemTail(Liste);
          a0
```

Offset : -258

Description : RemTail élimine le dernier enregistrement d'une liste

Paramètre :

Liste Pointeur sur la liste, où le dernier enregistrement doit être éliminé.

Enqueure

```
Enqueure(Liste, Noeud);
          a0      a1
```

Offset : -270

Description : Cette fonction est utilisée pour ordonner un enregistrement suivant son niveau de priorité, dans une liste. Comme cela a été détaillé dans le chapitre 2.2.2, le noeud possédant le niveau de priorité le plus important sera mis au début de la liste. Si plusieurs noeuds ont la même priorité, le nouveau sera mis derrière ceux déjà présents.

Paramètres :

Liste Pointeur sur la liste, où le noeud doit être enregistré.

Noeud Pointeur sur le noeud qui sera inséré.

FIndName

```
Enregistrement = FindName(Liste,"Nom");
                                a0      a1
```

Offset : -276

Description : Avec FindName, on peut chercher un noeud avec le nom correspondant dans la liste indiquée.

Paramètres :

Liste Pointeur sur la liste qui sera détaillée.

Nom Pointeur sur le nom qui doit être cherché. Cette chaîne de caractères doit être terminée par un 0. Lors de l'appel d'une fonction C, il ne s'agira pas d'un pointeur sur le nom, mais de la chaîne de caractères propre.

Résultat : Dans l'enregistrement (d0), la fonction renverra un pointeur sur le noeud trouvé. Si aucun enregistrement ne correspond au nom fixé, la fonction renverra un 0.

L'exemple suivant montre comment on peut établir si le nom d'un noeud se trouve deux fois dans une liste. L'exemple ne peut pas être lancé dans sa forme présente. La liste doit être initialisée avant l'appel de *'FindName()'*, sous peine de plantage du système.

```
#include <exec/lists.h>

main()
{
struct node *FindName(),*node;
struct List *liste;
```



```
if ((node = FindName (liste,"testnode"))!=0)
   if ((node = FindName (node,"testnode"))!=0)
      printf ("Le nom testnode a été trouvé deux fois\n");
}
```

Après la liste des fonctions disponibles pour le traitement des listes, leur utilisation doit être détaillée. C'est le rôle de notre exemple, qui montre comment une liste est constituée, gérée, et comment on efface un de ses enregistrements.

Exemple de liste :

```
#include <exec//lists.h>

char *name[] = ("Node1","Node2","NODE3");

struct List liste;
struct Node node[3],*np;

main()

{
 int i;
 char n;
 liste.lh_Head = (struct Node *)&liste.lh_Tail;
 liste.lh_Tail = 0;
 liste.lh_TailPred = (struct Node *)&liste.lh_Head;
 liste.lh_Type = NT_TASK;

 for (i=0;i<=2;i++) {
    node[i].ln_Type = NT_TASK;
    node[i].ln_Name = name[i];

    AddTail (&liste,&node[i]);
 }

émission();
printf ("\n Emission de la liste terminée\n");
```

```
  Remove (np);
  émission();
  printf ("\n 2ème noeud sera éliminé\n");
}

émission ()
{
  for (np = liste.lh_Head;
       np != &liste.lh_Tail;
       np = np->ln_Succ)
         printf ("\n %s \n",np->ln_name);
}
```

2.3 Les librairies

Ce chapitre concerne aussi bien les programmeurs en C que ceux employant l'assembleur. La compréhension de ce chapitre est essentielle si on veut utiliser l'Amiga avec toutes ses possibilités.

Une *librairie* est une bibliothèque de fonctions, qui pourra être utilisée par le programmeur. Elle est constituée d'une série d'instructions auxquelles on accède par des sauts. Ces librairies sont employées par le programmeur, aussi bien en C qu'en assembleur, lorsque certaines fonctions ou instructions ne sont pas disponibles dans ces langages. Par exemple, la fonction *OpenScreen()* sert à établir un écran (Screen). En C, cette instruction n'est pas présente et elle ne pourra être exécutée qu'avec l'aide d'une librairie de l'Amiga.

Une librairie est structurée de la manière suivante :

```
          .
          .
          .
00060A JMP $FC2FD6
000610 JMP $FC0B28
000616 JMP $FC0AC0
          .
          .
          .
```


Les librairies sont partagées en différentes classes. La plupart d'entre elles se trouvent déjà dans la zone ROM du calculateur ; certaines doivent être chargée à partir d'une disquette pour être utilisées. Voici les librairies qui sont disponibles :

- clist.lib
- console.lib
- diskfont.lib
- dos.lib
- expansion.lib
- exec.lib
- graphics.lib
- icon.lib
- intuition.lib
- layers.lib
- mathffp.lib
- mathieeedoubbas.lib
- mathtrans.lib
- potgo.lib
- ram.lib
- timer.lib
- translator.lib

La librairie *Exec* occupe un autre emplacement, afin que ses fonctions soient immédiatement accessibles après un RESET. Lorsqu'on veut utiliser une fonction d'une autre librairie, on doit le communiquer au système, celui-ci ouvrant la librairie correspondante qui sera alors accessible au programmeur. Si cette librairie est déjà ouverte, le programmeur sera prévenu et le système lui donnera le moyen d'y accéder.

Pour chaque librairie, l'adresse de base est connue et les fonctions pourront être sélectionnées au moyen d'un offset négatif.

L'appel d'une librairie se fera de la façon suivante :

```
move.l LibBase,a6        ;adresse de base de la librairie
jsr Offset(a6)           ;appel d'une fonction avec un offset négatif
```

tableau, que vous pourrez trouver à la fin de ce livre, en appendice. Avant la sélection d'une librairie, les paramè , des registres correspondants devront être initialisés.

Comme cela a déjà été dit, les fonctions de la librairie *Exec* sont directement accessibles après un RESET. Pour pouvoir en sélectionner une, l'adresse de base *Exec* doit être connue. Cette dernière correspond à l'emplacement mémoire $04.

En C, on y accède par lecture de la variable standard *'SysBase'*.

La sélection d'une fonction *Exec-Library* en C, est assez simple : voici pour exemple l'appel de la routine *FindName*.

```
#include <exec/execbase.h>
struct ExecBase *SysBase;
struct Library *FindName(), *Library;

main()

{
   library = FindName(&(SysBase->LibList),"dos.library");
   printf ("\n %x \n",Library);
}
```

Ce petit programme C explore la liste des librairies existantes à la recherche de la *dos.library*. Si elle est trouvée, son adresse sera émise. Si elle ne l'est pas, c'est un 0 qui sera émis.

Aucun problème ne doit donc être rencontré, à part peut être le fait de confondre les fonctions *Exec* lors de la sélection. La fonction *FindName* est un pointeur sur la liste et sur le nom du noeud qui doit être communiqué. La façon dont l'affectation de la liste se déroule sera détaillée dans le chapitre concernant *'ExecBase'*.

Nous allons voir maintenant comment le compilateur C traduit ce programme en assembleur. Lorsqu'on se limite à l'essentiel, le programme compilé a l'allure suivante.


```
    global _SysBase,4
    global _Lib    .4

_main:

  pea *Node              ;décalage dans la pile du pointeur sur le noeud
  move.l _SysBase,a6     ;pointeur sur ExecBase (à partir de $4)
  pea 378(a6)            ;décalage dans la pile du pointeur sur la liste
  jsr _FindName          ;sélection de la fonction FindName
  move.l d0,_Library     ;réponse dans Library
  move.l _library,-(a7) ;et dans la pile en
  jsr _printf            ;passant à la routine _printf
  rts

  _FindName
  movem.l 4(sp),a0/a1    ;chargement des paramètres pour FindName, de la
                         pile dans les registres correspondants.
  move.l  -276(a6)       ;appel de la fonction Findname
```

Le listing complet en assembleur est un peu plus long, tout ce qui ne concernait pas l'appel des fonctions de librairies ayant été éliminé.

_SysBase est un pointeur sur la librarie *Exec*, ses fonctions étant accessibles par des offsets négatifs. Lorsqu'on accède à la mémoire avec des offsets positifs, on obtient une valeur d'*ExecBase* (stucture générale du système d'exploitation), laquelle sera détaillée plus loin. Sous les enregistrements d'*ExecBase*, on trouve une liste où les librairies sont indicées.

Le résultat de la compilation est complexe, mais il est nécessaire de le comprendre.

Le programme général met le pointeur de la liste et du noeud dans la pile, et appelle ainsi le sous-programme. C'est par ce dernier que les paramètres seront chargés dans les registres prévus. Puis l'adresse de base de la librairie sera écrite dans a6. Tout sera alors prêt pour la sélection de la fonction *FindName* de la librairie *Exec*. L'offset de cette fonction est -276 (cf chapitre 2.2.4).

L'appel de la fonction se fera par l'instruction 'JMP -276(a6)'. Le paramètre obtenu en réponse sera stocké dans *'Library'* :alé dans la pile et affiché à l'écran avec l'instruction *'printf'*.

Le système connaissant l'emplacement de la librairie *Exec*, une de ses fonctions pourra être sélectionnée sans problème. Si on désire accéder à une autre librairie, soit le compilateur C, soit le système d'exploitation doit savoir où cette librairie se trouve. C'est pour cette raison, que la librairie *Exec* a à sa disposition une fonction permettant de découvrir les adresses de base des autres librairies. Cette fonction se nomme *'OpenLibrary'* et a la forme suivante :

```
LibPtr = OpenLibrary(LibName, Version);
  d0                 a1        d0
```

LibName

Pointeur sur le nom de la librairie (le nom sera toujours terminé par un 0).

Version

Donne la version de la librairie que l'utilisateur veut ouvrir. Si plusieurs des librairies disponibles portent le même nom, elles ne seront différenciées que par leur numéro de version. Si une nouvelle version est exigée, alors qu'elle n'est pas encore disponible, la fonction *'OpenLibrary'* échouera.

LibPtr

Contient, après l'appel de la fonction, l'adresse de base de la librairie, si cette dernière a été trouvée. Sinon, c'est un 0 qui sera renvoyé.

La variable *nom*, où l'adresse de base de la librairie sera mise en mémoire, ne peut pas être choisie n'importe comment. Dès que le compilateur C connaît l'adresse de base d'une librairie, cette dernière doit être stockée dans une variable prévue à cet effet.

Si cette variable est déclarée, sans être initialisée avec la bonne valeur, l'appel de la fonctio soldera par l'effondrement du programme ou du système.

Listes des variables réservées

libraries	variables
clist.lib	CListBase
diskfont.lib	DiskFontBase
dos.lib	DosBase
expansion.lib	ExpansionBase
exec.lib	SysBase
graphics.lib	GfxBase
icon.lib	IconBase
intuition.lib	IntuitionBase
layers.lib	LayersBase
mathffp.lib	MathBase
mathieeedoubbas.lib	MathIeeeDoubBasBase
mathtrans.lib	MathTransBase
potgo.lib	PotgoBase
translator.lib	TranslatorBase

Il existe beaucoup de possibilités pour déclarer ces variables. Le plus simple est de les déclarer 'ULONG' (mot long non indicé), économisant ainsi la modification du pointeur. Evidemment, dans cette déclaration, il ne faut pas oublier le fichier *include 'Exec/types.h'*, sinon le compilateur ne comprendra pas ce que signifie 'ULONG'.

2.3.1 OUVERTURE D'UNE LIBRAIRIE

Nous allons illustrer l'ouverture et l'utilisation d'une librairie par un exemple qui permet d'initialiser la librairie *intuition* et la fonction *Screen*.

```
#include <exec/types.h>
#include <intuition/intuition.h)
```

```
struct NewScreen ns = {
                        0,0,
                        640,256,
                        2,
                        0,1,
                        HIRES,
                        CUSTOMSCREEN,
                        NULL,
                (UBYTE *) "Mon écran"
                        NULL,
                        NULL };

ULONG IntuitionBase;

main()
{
 ULONG screen;

 if(!(IntuitionBase=OpenLibrary("intuition.library")))
   exit(100);

 screen = OpenScreen (&ns);
}
```

Après qu'*Exec/types.h* et la structure *Intuition* aient été reliés, la structure *Screen* est initialisée et *IntuitionBase* déclarée ULONG. *IntuitionBase* doit être déclarée globalement, étant donné qu'elle est utilisée en C, dans la procédure *OpenScreen*. Une fois la fonction *OpenLibrary()* de la librairie *Exec* appelée et la librairie *Intuition* ouverte, l'adresse de base de cette dernière sera stockée dans *IntuitionBase* et la fonction *OpenScreen* sera initialisée.

Voici maintenant la traduction de ce programme par le compilateur. Nous n'avons représenté ici que la partie nous intéressant.

```
_ns                 ;initialisation de la structure Screen
 dc.w 0
 dc.w 0
 dc.w 640
```



```
 dc.w 256
 dc.w 2
 dc.b 0
 dc.b 1
 dc.w -32768
 dc.w 15
 dc.l $0000
 dc.l .1+0
 dc.l $0000
 dc.l $0000
.1:
dc.b "Mon écran",0

global _IntuitionBase,4

_main:

movem.l version,-(sp)       ;version de la librairie dans la pile
pea *name                   ;pointeur sur le nom dans la pile
jsr _OpenLibrary            ;appel de la fonction OpenLibrary
move.l d0,IntuitionBase     ;retour mis en mémoire
tst.l d0                    ;test
bne .5                      ;non nul -> ok
pea 100                     ;numéro de sortie
jsr _exit                   ;sortie

.5:

pea _ns                     ;pointeur sur la structure Screen
jsr _OpenScreen             ;appel de la fonction OpenScreen
rts                         ;renvoi du sous-programme

_OpenScreen:

move.l 4(sp),a0           ;pointeur sur la stucture Screen
move.l _IntuitionBase,a6  ;adresse de base prise dans la
                           variable IntuitionBase.
jmp  -198(a6)             ;appel de la fonction

_OpenLibrary:
```

```
move.l _SysBase,a6        ;adresse de base issue d'Exec
move.l 4(sp),a1           ;pointeur sur le nor
move.l 8(sp),d0           ;version issue de la pile
jmp  -552(a6)             ;appel de la fonction OpenLibrary
```

Vous pouvez voir ici que sans la déclaration d'*IntuitionBase*, le programme ne serait pas compilé, étant donné que cette variable est utilisée dans la routine *OpenScreen*. Si elle est déclarée mais mal initialisée, le programme plantera.

2.3.2 FERMETURE D'UNE LIBRAIRIE

Une librairie doit toujours être fermée lorsqu'elle n'est plus indispensable, afin que le système d'exploitation puisse en être déchargé et que la mémoire disponible augmente.

Une telle fonction se trouve dans la librairie *Exec* :

```
CloseLibrary(library);
             a1
```

library

Pointeur sur la librairie ouverte et qui doit être fermée.

2.3.3 STRUCTURE D'UNE LIBRAIRIE

La fonction *librairie* en C est établie avec le fichier *Include* :

```
#define LIB_VECTSIZE    6L
#define LIB_RESERVED    4L
#define LIB_BASE        (-LIB_VECTSIZE)
#define LIB_USERDEF     (LIB_BASE-
                        (LIB_RESERVED*LIB_VECTSIZE))
#define LIB_NONSTD      (LIB_USERDEF)

#define LIB_OPEN        (-6L)
#define LIB_CLOSE       (-12L)
```



```
#define LIB_EXPUNGE     (-18L)
#define LIB_      IC    (-24L)

#define LIBF_SUMMING (1L<<0)
#define LIBF_CHANGED (1L<<1)
#define LIBF_SUMUSED (1L<<2)
#define LIBF_DELEXP  (1L<<3)

extern struct Library {
 0    struct  Node lib_Node;
14    UBYTE   lib_Flags;
15    UBYTE   lib_pad;
16    UWORD   lib_NegSize;
18    UWORD   lib_PosSize;
20    UWORD   lib_Version;
22    UWORD   lib_Revision;
24    APTR    lib_IdString;
28    ULONG   lib_Sum;
32    UWORD   lib_OpenCnt;
};
```

Explication de la structure :

Lib_Node

Structure de type noeud (cf. 2.2.2). Avec l'aide de cette structure, les librairies sont reliées pour former une liste. Le type de librairie est dans ce cas, naturellement, 'NT_LIBRARY' et le nom du noeud correspond au nom de la librairie.

lib_pad

Octet inutilisé, mais sert à mettre le mot suivant ou le long mot d'une structure à nouveau sur une adresse paire.

lib_NegSize

Indique la taille de la zone pour l'offset négatif.

lib_PosSize

Indique la taille de la zone comprise entre la librairie et l'adresse de base. Cette valeur est intéressante, car une librairie peut contenir plus d'enregistrements que ne le permet une structure C. Le nombre d'enregistrements et leur signification dépendent de la librairie.

lib_Version

Indique la version de la librairie sélectionnée.

lib_Revision

Indique la quantité de remaniements d'une librairie sélectionnée.

lib_IdString

Pointeur sur un texte, renfermant des informations supplémentaires sur la librairie.

lib_Summ

Somme de contrôle d'une librairie. Si cette dernière est modifiée, la somme devra être recalculée.

lib_OpenCnt

Indique le nombre de tâches utilisant cette librairie. Lorsqu'aucune tâche n'accède à cette librairie, la fermeture de cette dernière dépendra du type de la librairie.

Les fonctions d'une librairie sont disponibles pour l'utilisateur à partir de l'offset -30, même si, théoriquement, elles peuvent débuter à -6. Lorsqu'on examine attentivement les premiers offsets, on remarque qu'ils pointent sur des fonctions utilisées par *Exec* pour gérer les librairies.

Il s'agit en fait de fonctions nécessaires à l'ouverture et à la fermeture d'une librairie, auxquelles les fonctions *Exec* accèdent. Chaque librairie possède donc ses propres routines d'ouverture et de fermeture. C'est dans ces dernières que se décide si une librairie non utilisée doit être fermée ou non.

De même que la définition du fichier *Include* a été donnée, voici la signification des quatre offsets suivants :

LIB_OPEN	-6	ouverture d'une librairie
LIB_CLOSE	-12	fermeture d'une librairie
LIB_EXPUNGE	-18	élimination d'une librairie
LIB_EXTFUNC	-24	Ouverte pour extension

Les librairies non éliminées n'utilisent pas LIB_EXPUNGE lorsqu'elles ne sont plus nécessaires. Toutes les fonctions qui ne sont pas employées doivent pointer sur une routine qui désactive d0.

2.3.4 MODIFIER UNE LIBRAIRIE

Pour modifier une librairie existante, on utilise une fonction présente dans la librairie *Exec*, permettant la modification du saut d'offset déterminé. Cette fonction est de la forme suivante :

SetFunction

```
SetFunction(Librairie, offset, saut);
              a1       a0      d0
```

Offset : -420

Description : Modifie le saut d'offset permettant d'appeler une fonction d'une telle façon que cette dernière pointe sur la nouvelle routine. La somme de contrôle de la librairie sera à nouveau calculée.

Library Pointeur sur la librairie à modifie..

Offset Indique l'offset de la fonction, qui doit être modifié.

Saut Pointeur sur la routine.

2.3.5 ETABLISSEMENT D'UNE LIBRAIRIE PARTICULIERE

Avant d'expliquer comment utiliser une librairie, il est nécessaire de voir comment une librairie personnelle peut être établie.

La mise en place d'une telle librairie est nécessaire, lorsqu'on a l'intention d'employer plusieurs tâches en travail parallèle, utilisant un certain nombre de fonctions communes. L'accès à ces dernières en sera facilité.

Pour créer une librairie, *Exec* met à disposition plusieurs fonctions, qui seront détaillées en premier lieu.

InitStruct

```
InitStruct(initTable, Mémoire, Taille);
            A1        A2       D0
```

Offset : -78

Description : La fonction initialise une structure suivant une zone mémoire et une table données.

Paramètres :

InitTable pointeur sur la table, qui aidera à créer la structure.

Mémoire pointeur sur la mémoire allouée.

Taille indique la taille de la structure initialisée. La mémoire ... la stucture sera installée, n'a pas besoin d'être ...ée, car ceci sera pris en charge par la fonction InitStruct.

La table, à partir de laquelle la structure sera installée, a un aspect déconcertant. Elle se compose d'un octet d'instruction suivi des données, qui suivant l'aspect de l'octet, seront traitées de manière différente. Après ces données, on observe à nouveau une succession d'octets et de données. La longueur des données dépend de l'octet d'instruction. La fin de la table est marquée par un octet d'instruction de valeur nulle.

L'octet d'instruction est divisé en deux niveaux, niveau bas et niveau haut. La combinaison de bits du haut niveau (4 bits de plus fort poids) correspond à l'instruction alors que le bas niveau correspond au nombre d'instructions.

Nous allons nous consacrer en premier lieu au niveau haut. Ce dernier est à nouveau divisé en deux bits supérieurs et deux bits inférieurs. Les deux bits de poids fort correspondent à l'instruction propre. Les deux bits de poids faible donnent la taille des données avec lesquelles l'octet d'instruction opérera. Les différentes tailles de données à disposition sont le mot long, le mot et l'octet.

On pourra donc obtenir 4 instructions différentes avec les 2 bits de poids fort.

Combinaison 00

Cette combinaison établit que les données débutant après l'octet d'instruction seront transférées dans la structure mise en place. Le type de données à traiter est défini par les deux bits suivants (Long mot, mot, octet).

Combinaison 01

Cette combinaison indique que la donnée de longueur correspondante sera copiée dans la structure mise en place.

Cette combinaison indique qu'après le mot d'instruction, l'octet présent servira d'offset dans la structure mise en place. L'offset sera additionné à l'adresse de départ de la structure et les données présentes après cet octet seront copiées à partir de l'emplacement ainsi déterminé.

Combinaison 11

Cette combinaison indique que les trois octets après l'instruction seront utilisés comme offset 24 bits. Cette instruction est identique à la précédente.

Les deux bits suivants de l'octet d'instruction correspondent au type des données à traiter (bits 4 et 5).

Combinaison 00 : mot long (adresse paire)
Combinaison 01 : mot (adresse paire)
Combinaison 10 : octet
Combinaison 11 : à ne pas utiliser, car cela chatouille le guru

Le niveau bas de l'octet d'instruction indique le nombre de fois où la fonction déterminée sera exécutée ; elle sert de compteur. Comme ce compteur peut être décrémenté jusqu'à -1, la fonction sera toujours exécutée une fois de plus que le nombre indiqué par le compteur.

L'octet d'instruction doit toujours être à une adresse paire :

```
dc.b %00010010,$00
dc.w $FFFF,$FFFF,$1234
```

L'octet d'instruction correspond à $12 (%00010010) et établit donc que les trois mots suivants seront copiés dans la structure. L'octet nul après celui de l'instruction est nécessaire, car les mots doivent débuter à une adresse paire.

```
dc.b %10000001,$10
dc.l $12341234,$FFFF1111
```


L'octet d'instruction indique que deux mots long doivent être copiés à partir de la position [illegible] dans la structure.

Exemple de programme :

Dans cet exemple, la place mémoire d'une structure librairie sera réservée et partiellement initialisée.

```
AllocMem   = -128
FreeMem    = -210
MemType    = $10001
InitStruct = -78
StructSize = 34
Size       = $300

move.l $4,a6
move.l #MemType,d1
move.l #Size,d0
jsr AllocMem(a6)
tst.l d0
beq erreur
lea.l Table,a1
move.l d0,a2
move.l StructSize,d0
jsr InitStruct(a6)

Erreur: rts

Nom : dc.b "Test",0,0

Table:
dc.b %01000001,$00
dc.l 0                  ;Node, Succ et Pred
dc.b %00100001
dc.b $09,0,0            ;Type et Pri
dc.b %00011000,0
dc.l Nom                ;Nom
dc.w 0,0,0,1,0          ;NegS, PosS, Version, et Revision
dc.l Nom                ;IDString
dc.b %00000000          ;marque de fin
```

MakeLibrary

```
Library = MakeLibrary(Vecteur,Structure,Init,Taille,SegListe);
  d0                  a0        a1        a2   d0       d1
```

Offset : -84

Description : Il est possible, avec cette fonction, de mettre en place une librairie personnelle. La place mémoire de la structure librairie sera déterminée par la fonction lorsque *lib_MemSize* et *lib_PosSize* seront correctement initialisés.

Paramètres :

Vecteur Pointeur sur la table des vecteurs de la librairie. Sur cette table se trouvent soit les pointeurs sur les différentes fonctions, soit les offsets qui seront additionnés à l'adresse de base pour obtenir le saut des fonctions. Si vous avez stocké des offsets sur la table, cette dernière devra débuter à $FFFF (-1). La marque de fin d'une table est à nouveau à -1 avec la même longueur d'enregistrements (mot long pour table d'offsets, mot long comme pointeur de fonctions).

Structure Pointeur sur une table d'initialisation, qui a été détaillée pour la fonction *InitStruct*. La structure librairie sera initialisée avec cette table, à la fin de la fonction *MakeLibrary*. Les enregistrements *lib_NegSize* et *lib_PosSize* ne doivent pas être initialisés avec l'aide de cette table, sinon la valeur prête à être insérée sera recalculée par la fonction. Si le paramètre *Structure* n'est pas activé dans la fonction, aucune table ne sera utilisée pour l'initialisation, la mise en place de la structure se faisant *'à la main'*.

Init Pointeur sur le programme qui sera appelé à la fin de la fonction *MakeLibrary()*, dès que le pointeur sera initialisé. Une routine personnelle pourra donc initialiser une structure librairie, lorsque celle-ci ne le sera pas avec l'aide d'une table. Le pointeur de la structure mise en place sera dans d0, celui de liste segment étant dans a0. Si la routine *Init* doit être modifiée, cette modification sera délivrée en tant que paramètre à la fin de la fonction.

SegListe Pointeur sur une liste segment (utilisée par le DOS).

Library Pointeur de retour sur la structure librairie.

Avec cette fonction, il est possible de mettre en place une librairie personnelle. Celle-ci n'est cependant pas encore insérée dans la *LibListe* de la structure *ExecBase*. La somme de contrôle de la librairie n'est pas encore calculée. Pour cette tâche, la librairie *Exec* met à disposition la fonction suivante :

AddLibrary

```
AddLibrary(Library);
             a1
```

Offset : -396

Paramètre :

Library Pointeur sur la structure *librairie* mise en place précédemment par la fonction *MakeLibrary()*.

Pour mettre en place une librairie personnelle utilisable, voici un programme d'exemple :

```
MakeLib = -84
AddLib = -396
```

```
Blinken = -30
OpenCnt = 32
InitStruct = -78
StructSize = 34

        move.l $4,a6
        lea.l Vecteurs,a0       ;pointeur sur les vecteurs
        move.l #0,a1            ;effacer la structure
        lea.l init,a2           ;routine init
        move.l #StructSize,d0   ;taille de la structure
        clr.l d1                ;SegListe
        jsr MakeLib(a6)
        tst.l d0
        beq Erreur
        move.l d0,a1
        jsr AddLib(a6)
        lea.l nom,a1            ;Nom de Librairie
        move.l #1,d0            ;Version
        jsr OpenLib(a6)
        tst.l d0
        beq Erreur
        move.l d0,LibBase
        move.l d0,a6
        move.l #$20000,d0       ;valeur du clignotement
        jsr blinken(a6)         ;appel de la fonction

Erreur: rts

init:   move.l d0,a0
        move.b #9,8(a0)   ;Prise en compte dy type
        lea.l Nom,a1
        move.l a1,10(a0)    ;Prise en compte du nom
        move.l a1,24(a0)    ;IDString = Name
        move.w #1,20(a0)    ;Version
        rts
open:    move.l a6,d0
        add.w #$01,OpenCnt(a6)
        rts
close:   sub.w #$01,OpenCnt(a6)
```



```
        clr.l d0
        rts
expunge:  clr.l d0
        rts
extfunc:  move.w d0,$dff180
        sub.l #1,d0
        bne blink
        rts

LibBase:  dc.l 0
name:     dc.b 'test.library',0,0

Vecteurs:
dc.l open,close,expunge,extfunc,blink,$ffffffff
```

Le programme crée une librairie du nom de *"test.library"*. On y trouve une fonction accessible à l'offset -30. Cette fonction porte le nom *Clignoter*. Son rôle est de modifier pendant un cours instant la couleur de l'écran. D0 retourne le paramètre correspondant à la durée de cette modification.

Il est absolument indispensable que les offsets librairies -6 à -24 soient utilisés pour les accès aux fonctions internes comme cela a déjà été expliqué.

Etant donné que les structures *Librairies*, *Resource* et *Device* comprennent comme tête standard de structure, une structure *Librairie*, il est possible de créer une structure *Resource* ou *Device* avec la fonction *MakeLibrary()*.

RemLibrary

```
Error = RemLibrary(librairie);
 D0                A1
```

Offset : -402

Description : Cette fonction élimine une structure *Library* dans la liste des librairies de la structure *ExecBase* : celle-ci ne pourra plus être par la suite ouverte avec la fonction *OpenLibrary()*.

Paramètres :

Librairie est un pointeur sur la structure *Librairie*.

Error indique si une erreur est apparue lors de l'utilisation de la fonction. Si c'est le cas, un message d'erreur sera transmis dans D0. Autrement, l'erreur est initialisée à 0.

OldOpenLibrary

```
Library = OldOpenLibrary(NomLib);
 D0                      A1
```

Offset : -408

Description : Cette fonction est issue de la version Kickstart 1.0. Elle sert à ouvrir une Librairie sans test du numéro de version. Cette fonction a été conservée dans la librairie *Exec* afin que les programmes conçus avec la version Kickstart 1.0 puissent aussi tourner avec la version 1.2.

2.4 Le multi-tâches

Le multi-tâches est une des nombreuses possibilités de *Exec*. On entend par ce terme la capacité du système d'exploitation à exécuter plusieurs programmes, chacun d'entre eux représentant alors une tâche.

Comme l'Amiga ne dispose que d'un seul processeur, il ne peut exécuter qu'une seule tâche à la fois. Le 68000 partage son temps de calcul entre les différentes tâches, ce qui nous donne l'impression qu'il les exécute toutes à la fois. Ceci n'est possible qu'en organisant les temps d'accès, ce qui signifie que chaque tâche dispose du processeur pendant un cours laps de temps donné. On peut dire que le processeur partage les tâches suivant un procédé de multiplexage du temps. Toutes les tâches se trouvant en mémoire n'ont pas besoin de tourner en même temps. Beaucoup d'entre elles ne sont activées qu'en cas de besoin lorsqu'une touche déterminée est pressée ou le bouton de la souris activé par exemple. Pour cette raison, les différentes tâches sont classées en différentes catégories *(Task States* en anglais).

Running : Ceci correspond à la tâche exécutée à l'instant même par le processeur. Il n'y a qu'une seule tâche de cette catégorie au même moment.

Ready : Toutes les tâches prêtes à être exécutées se trouvent dans cet état. Le processeur ne leur consacrant à l'instant même aucun temps d'exécution.

Waiting : Les tâches de cette catégorie correspondent à celles qui ne doivent pas être traitées à l'instant mais qui attendent un événement déterminé.

Une tâche peut se trouver de plus dans l'un des trois états suivants :

Added : une telle tâche vient d'être rajoutée au système et ne se trouve pas encore dans l'un des trois états cités ci-dessus.

Removed : cette tâche vient de se terminer. Elle n'appartient plus à l'une des trois catégories citées ci-d us et va être écartée du système.

Exception : une tâche *Exception* est un état dans lequel la tâche peut être déplacée par un événement particulier. Après traitement de cette exception, la tâche retourne à l'un des trois états cités ci-dessus.

Une tâche se compose de deux éléments essentiels : le programme propre et la structure *Task*. Cette dernière comprend toutes les informations concernant la tâche nécessaire à *Exec*. On comprend mieux les multiples possibilités d'une tâche en examinant plus attentivement la structure *Task*.

2.4.1 LA STRUCTURE TASK

Cette structure telle qu'elle apparaît dans le fichier *Include "exec/tasks.h"* d'un compilateur C est de la forme suivante (les nombres entre parenthèses correspondent à l'écart d'un élément avec l'adresse de base de la structure) :

```
extern struct Task {
  UBYTE  tc_Flags;       /* (14) */
  UBYTE  tc_State;       /* (15) état de la tâche */
  BYTE   tc_IDNestCnt;   /* (16) compteur pour Disable() */
  BYTE   tc_TDNestCnt;   /* (17) compteur pour Forbid() */
  ULONG  tc_SigAlloc;    /* (18) bits de signal occupés */
  ULONG  tc_SigWait;     /* (22) attente de ces derniers */
  ULONG  tc_SigRecvd;    /* (26) signal de réception */
  ULONG  tc_SigExcept;   /* (30) exceptions déclenchées */
  UWORD  tc_Trap_Alloc;  /* (34) instruction Trap occupée */
  UWORD  tc_TrapAble;    /* (36) instruction Trap autorisée */
  APTR   tc_ExceptData;  /* (38) données pour exceptions */
  APTR   tc_ExceptCode;  /* (42) code pour exceptions */
  APTR   tc_TrapData;    /* (46) données pour TrapHandler */
  APTR   tc_TrapCode;    /* (50) code pour TrapHandler */
  APTR   tc_SPReg;       /* (54) Mémoire pour SP */
  APTR   tc_SPLower;     /* (58) limite inférieure de la pile */
```



```
  APTR   tc_SPUpper;     /* (62) limite supérieure de la pile +2 */
  VOID   (*tc_Switch     /* (66) perte du CPU */
  VOID   (*tc_Launch...  /* (70) obtention du CPU */
  struct List tc_MemEntry /* (74) mémoire occupée */
  APTR   tc_UserData;    /* (88) pointeur sur les données Task */
);
```

Comme on peut le constater, la tête d'une structure *Task* forme une structure *Node*. La raison essentielle est que *Exec* gère les tâches dans deux listes différentes : l'une correspond aux tâches de type *Ready*, l'autre correspond aux tâches de type *Waiting*. On peut appliquer à une liste *Task*, les principales fonctions telles que *Insert()*, *Remove()* ou *FindName()*.

Il existe évidemment des fonctions de gestion des listes Task. Chaque liste possède une tête de liste connue. Dans le cas des listes Task, ces dernières se trouvent dans la structure *ExecBase*. Il ne sera rien dit de plus sur cette structure ici, celle-ci étant décrite dans l'un des chapitres suivants. Les pages suivantes vont expliquer l'accès aux listes Task.

```
#include <exec/execbase.h>

extern ExecBase *SysBase;

main()
{
struct Task *waiting, *ready, *running;

waiting=(struct Task *) SysBase -> TaskWait.lh_Head;
ready=(struct Task *) SysBase -> TaskReady.lh_Head;
running=(struct Task *) SysBase -> ThisTask;

etc..
}
```

Ce programme transfère à l'état *Waiting* et *Ready* chaque pointeur sur la première structure Task de la liste correspondante. Lors de l'exécution, il s'agit d'un pointeur sur la tâche exécutée.

MemList et TaskReady sont donc des têtes de listes qui ne correspondent à un pointeur que pour *ThisTask*. Comme il ne peut y avoir qu'une seule tâche exécutée à la fois, il n'existe pas ici de liste.

Task-Switching

Voici maintenant quelques mots sur le processus qui permet l'utilisation du processeur par plusieurs tâches : le *Task-Switching*. Il s'agit de la commutation de différentes tâches. Si une tâche se trouve dans l'une des deux listes Task, elle sera automatiquement insérée dans ce processus. La première question que l'on se pose est : *comment se passe la commutation entre tâches ?* Une tâche ne sait pas quand elle aura accès au processeur ni quand son accès prendra fin. Si elle teste le champ *ThisTask d'ExecBase*, elle y trouvera toujours son propre pointeur étant donné que ce champ ne lui est accessible que lorsque le CPU lui est attribué. Une tâche peut être interrompue à tout moment. Le processus est déclenché par une interruption qui apparaît lorsque la tâche a épuisé son temps d'accès ou lorsqu'une autre tâche importante doit être traitée en priorité. La routine *Switch* du système d'exploitation rend possible toutes ces commutations. Contrairement à la tâche qui fonctionne en mode Utilisateur du 68000 la routine Switch sera toujours traitée en mode Superviseur. En premier lieu, les registres processeur seront validés sur la pile Task. Le pointeur de pile *Utilisateur* sera transféré dans le champ *tc_SPReg* de la structure Task, le registre de Status et le compteur programme venant en dernier sur la pile *Utilisateur*. Ils seront transférés par le 68000 lors de l'interruption automatiquement sur la pile Superviseur. Puis la tâche sera prise en charge dans la liste *Ready*. La nouvelle tâche provient soit de la liste *Ready*, soit de la liste *Waiting*. Elle en sera écartée et transférée dans le champ *ThisTask*. Son pointeur de pile et les registres seront alors issus du champ *tc_SPReg*. *Exec* abandonne la routine Switch avec une instruction RTE. Après cette dernière, une nouvelle tâche sera traitée automatiquement. Ceci n'est évidemment qu'un résumé du déroulement de la routine *Switch*. Si la tâche se trouve dans l'état *Exception*, il peut arriver que la routine *Switch* appelle des routines dont les adresses de la structure *Task* correspondent à *tc_Launch* et *tc_Switch*.

Comme cela a été dit, l'ensemble du processus se déroule automatiquement. Il ne faut pas s'en occuper.

Une seconde question concernant le Task-Switching est : *quand se passe la commutatic* On peut formuler la question d'une autre façon: *quelle fraction de temps revient à une tâche déterminée ?* Le partage du temps de calcul se nomme : *Task-Scheduling*. L'élément de base utilisé par Exec est le champ *ln_Pri* de la structure *Node* au début de la structure Task. Plus simplement, on peut dire que plus la priorité d'une tâche est importante, plus elle aura de temps de calcul à disposition et plus vite elle sera traitée. Exec commence toujours avec la tâche de priorité supérieure. Cette dernière se voit attribuer un temps processeur qui diminuera relativement son niveau de priorité par comparaison. A ce moment là, la tâche qui était prioritaire ne le sera plus, et une nouvelle pourra être exécutée par le processeur. Ceci permet à d'autres tâches de priorité inférieure d'être traitées. Le niveau de priorité peut prendre des valeurs comprises entre -128 et +127, une tâche normale (CLI) a la priorité 0. Les tâches du système d'exploitation s'étalent entre des valeurs de -20 à +20. Il n'est donc pas nécessaire de mettre des valeurs extrêmes dans le champ *tc_Node.ln_Pri* d'une structure Task. Un autre champ de la structure Task correspond à *tc_Node.ln_Name*. Il renferme un pointeur sur le nom de la tâche. La recherche de cette dernière est ainsi grandement facilité. Le champ *tc_State* contient l'état de la tâche qui peut être :

non valable	= 0
added	= 1
running	= 2
ready	= 3
waiting	= 4
exception	= 5
removed	= 6

La pile Task

En dehors de la structure Task, une tâche nécessite aussi une pile. Celle-ci est de type Utilisateur. *tc_SPLower* correspond à la limite inférieure de la pile alors que *tc_SPUpper* correspond à la limite supérieure. L'adresse se trouvant dans *tc_SPReg* est utilisée comme mémoire par le pointeur de pile. En temps normal, *tc_SPReg* est égal à *tc_SPUpper*.

On peut aussi mettre *tc_SPReg* à n'importe quelle adresse comprise entre *tc_SPUpper* et *tc_SPLower*. On peut alors uti... la zone entre *tc_SPReg* et *tc_SPUpper* comme mémoire pour les variables globales ou autres.

tc_SPUpper pointe sur la limite supérieure de la pile signalant ainsi la première adresse de la zone pile. Le mot se trouvant à la dernière position de la pile est d'ailleurs à l'adresse *tc_SPUpper* moins 2. Etant donné qu'Exec met en mémoire les registres sur la pile Task lors du *Task-Switching*, cette dernière doit avoir une taille minimale de 70 octets. On pourra y trouver ensuite des variables concernant la tâche et surtout l'adresse de retour. Comme une tâche se compose de différents éléments, structure Task, pile, programme, etc... on doit lui réserver plus ou moins de mémoire. Il y a d'ailleurs la possibilité de rajouter à la structure Task, une liste comportant toute la place mémoire de la tâche. Le champ *tc_MemEntry* comprend la tête de liste correspondante.

Autorisation et interdiction du Task-Switching

Il peut être facheux qu'une tâche perde l'accès du processeur à n'importe quel moment. Vous voulez afficher la liste des tâches de type *Waiting* à l'écran mais pendant que votre programme lit entrée après entrée, une tâche de type *Ready* passe en mode *Waiting* ou inversement. Les valeurs lues seront fausses. C'est pour cela qu'il existe la solution suivante. Lorsqu'une tâche accède à des structures de données qui sont liées au système entier ou à d'autres tâches, le Task-Switching doit être désinitialisé afin que les données ne soient pas modifiées par une autre tâche ou par le système d'exploitation.

Forbid() et Permit()

Ces deux routines représentent le premier échelon de désinitialisation. *Forbid()* désinitiale le Task-Switching, *Permit()* l'autorise à nouveau. Les deux routines sont appelées sans paramètre et ne donnent aucune valeur en retour.

Disable() et Enable()

Souvent, il ne suffit pas de désactiver le Task-Switching. Beaucoup de structures de données du système peuvent être modifiées par Exec pendant les interruptions. On peut ainsi interdire ces dernières avec *Disable()* et les autoriser à nouveau avec *Enable()*. Mais attention : une interdiction du Task-Switching, même pendant un long moment, ne dérange pas ; il en est tout autrement avec les interruptions : leurs apparitions régulières sont nécessaires à Exec. Si on désactive les interruptions pendant un long moment, cela pourra amener un effondrement du système lorsque l'on voudra revenir en mode multi-tâches.

Il est aussi possible d'intercaler plusieurs *Forbid()* ou *Disable()* les uns derrière les autres : il existe deux compteurs : *TDNestCnt* et *IDNestCnt* (*Task Disable Nesting Counter* et *Interrupt Disable Nesting Counter*). Avec chaque appel de *Forbid() TD_NestCnt* est incrémenté de 1 et décrémenté de 1 avec *Permit()*. La commutation des tâches n'est possible que lorsque TDNestCnt < 0. Ceci signifie que le nombre d'appels *Permit()* doit être égal au nombre de *Forbid()* avant que le Task-Switching ne soit autorisé. Le programme suivant montre l'utilisation des fonctions *Enable()* et *Disable()*. Il lit le pointeur de l'ensemble des structures Task des tâches *Ready* et *Waiting* dans un champ pendant que les interruptions sont désactivées au moyen de *Disable()*. Ces dernières sont à nouveau autorisées avec *Enable()* et il est alors affiché à l'écran au moyen du pointeur les champs importants des structures Task tels que le nom, la pile, la priorité. Le programme montre aussi quelle tâche est exécutée. Vous pourrez l'expérimenter avec des instructions CLI telles que NEWCLI, RUN ou SETTASKPRI.

La tâche de type *Running* est aussi affichée. Celle-ci est normalement celle permettant l'exécution du programme.

```
#include <exec/execbase.h>

struct ExecBase *SysBase;

main()
{
register struct Task *a_task;
APTR run, tnodes[50], wtask, ltask;
```

```
register APTR Anode;

Disable();

Anode = tnodes;
run = (APTR)SysBase->ThisTask;

for(a_task=(struct Task *)SysBase->TaskReady.lh_Head;
 a_task->tc_Node.ln_Succ;
 *Anode=(APTR)a_task,Anode++,
 a_task=a_task->tc_Node.ln_Succ);
wtask=Anode;

for(a_task=(struct Task *)SysBase->TaskWait.lh_Head;
 a_task->tc_Node.ln_Succ;
 *Anode=(APTR)a_task,Anode++,
 a_task=a_task->tc_Node.ln_Succ);
ltask=Anode;

Enable();

printf("\nTask in the running state:\n");o1();o2(run);

printf("\nTask(s) in the ready state:\n");o1();
for(Anode=tnodes;Anode!=wtask;o2(*Anode),Anode++);

printf("\nTask(s) in the waiting state:\n");o1();
for(;Anode!=ltask;o2(*Anode),Anode++);

}

void o1()
{
printf
("Stackadress Stacksize Priority  Signals  Name\n");
printf
("---------- --------- -------- --------- -----------\n");
}
```



```
void o2(at)
register struct Ta··  *at;
{
printf("%10lx%10lx%8ld%11lx  %s\n",at->tc_SPLower,
 (ULONG)at->tc_SPUpper - (ULONG)at->tc_SPLower -2L,
 (LONG)at->tc_Node.ln_Pri,
 at->tc_SigWait,at->tc_Node.ln_Name);
}
```

Etant donné que ce programme valide le pointeur sur les structures Task et non le contenu de ces structures, il existe encore une possibilité d'erreur potentielle comme par exemple lorsqu'une tâche est effacée lors de l'affichage des valeurs. Par modification des routines o1() et o2(), on peut afficher d'autres champs des structures Task.

Générer de nouvelles tâches

Après avoir décrit et expliqué les tâches et structures Task, nous allons aborder la création d'une nouvelle tâche. Voici ce qui est nécessaire : en premier lieu, il faut une structure Task, puis une pile, puis un nom de tâche car la structure tâche contient déjà un pointeur sur le nom propre. En dernier lieu, il faut avoir un programme propre à la tâche. Ici apparaissent quelques problèmes : il est nécessaire d'avoir de la mémoire de libre pour la tâche car elle restera en mémoire à la fin du programme. Afin de ne pas trop compliquer ce programme, la structure Task, le nom de la tâche et la pile seront rassemblés dans un nouveau type de structure *"alltask"* pour laquelle un bloc mémoire conséquent est réservé. La tâche proprement dite est écrite comme une fonction C normale portant le nom *"Code"*. Elle ne fait rien d'autre qu'incrémenter un compteur jusqu'à la valeur $FFFFFF. Avant qu'elle y parvienne, il faut plusieurs minutes. Son exécution se remarque à la lenteur soudaine de l'Amiga, car notre tâche utilise une partie du temps de calcul du processeur. On peut aussi utiliser notre programme précédent qui permet de lister toutes les tâches. Celle présente apparaîtra en tant que tâche *Ready* avec le nom *Exemple*. Afin de bien copier la fonction *"code"* à sa position mémoire, vous pourrez employer son nom comme pointeur d'adresses.

code. Comme il n'y a aucune possibilité en C d ommuniquer le besoin de mémoire d'une fonction, il sera calculé par . Jifférence entre l'adresse de départ et celle de fin.

```
/**** générer une tâche ****/

#include <exec/types.h>
#include <exec/Tasks.h>
#include <exec/memory.h>

#define STACK_SIZE 500 /* taille de la pile */

main()
{
void code(),end();
APTR mycode,AllocMem();
Static char Taskname[] = "module d'exemple"
register APTR c1,c2;

struct alltask {
struct Task tc;
char Name[sizeof(Taskname)], Stack[STACK_SIZE];
} *mytask;

mytask = AllocMem((ULONG)sizeof(*mytask),
  MEMF PUBLIC/MEMF_CLEAR);
if(mytask==0)
{printf("Pas de mémoire pour la structure AllTask!\n");
return(0);
}

mycode = AllocMem((ULONG)end-(ULONG)code,MEMF_PUBLIC);
if(mycode==0)
{FreeMem(mytask,(ULONG)sizeof(*mytask));
printf("Pas de mémoire pour le code Task!\n");
return(0);
}

strcpy(mytask->Name,Taskname);
```



```
mytask->tc.tc_SPUpper=my*   sk->Stack+STACK_SIZE;
mytask->tc.tc_SPReg=myt     >tc.tc_SPUpper;

mytask->tc.tc_Node.ln_Type=NT_TASK;
mytask->tc.tc_Node.ln_Name=mytask->Name;

for(c1=code,c2=mycode;c1<=end;*c2++=*c1++);

AddTask(mytask,mycode,0L);
}

/*** La fonction "code" correspond au module ***/

void code()
{
ULONG count;

for(count=0;count>0xFFFFFF;count++);
}

void end()()
```

Comme on peut le remarquer, peu de champs de la structure *Task* sont initialisés :

tc_SPLower avec la limite de pile inférieure

tc_SPUpper et tc_SPReg avec la limite de pile supérieure

tc_Node.ln_Type avec le type de liste NT_TASK

On peut aussi se passer du nom.

Une fonction essentielle est utilisée dans ce programme :

```
AddTask(task,initialPC,finalPc);
```

Cette fonction insère une nouvelle tâche dans le système. Normalement, cette dernière sera insérée directement dans la liste *ready*.

Task Pointeur sur la structure *Task* qui, au minimum, doit être initialisée avec les 4 champs cités plus haut.

InitialPC Adresse où le traitement du programme doit débuter. Dans notre exemple, l'adresse de départ de la fonction *code* a celle stockée en mémoire dans *mycode*.

FinalPC *FinalPC* renferme l'adresse qui sera active lorsque le module rencontrera l'instruction RTS. On peut mettre en place des adresses de routines qui réinitialisent la mémoire, ferment des fichiers ouverts etc... Si *FinalPC* est initialisée avec 0, *Exec* utilisera sa routine *FinalPC* standard. Celle-ci libère la mémoire pointée par le champ *MemEntry*, puis élimine ce module de la liste système.

Fin d'une tâche

Il exite plusieurs possibilités pour finir une tâche :

1. La tâche atteint une instruction RTS, qui ne reproduit pas un saut de retour JSR ou BSR du programme. Le processus sera alors celui déterminé par *FinalPC*.

2. Une exception du 68000 est déclenchée, sans aucun rapport avec la tâche en elle-même, comme par exemple, une erreur de bus ou d'adresse, division par zéro... *Exec* engendre alors une *"software error-Task Held"* ou une méditation du guru. A la fin de ce chapitre, il est expliqué comment éviter ces erreurs.

3. Appel de la fonction *RemTask*, qui élimine une tâche du système.

2.4.1.1 Fonctions TASK

Exec met à disposition plusieurs fonctions pour générer ou éliminer des tâches, ainsi que pour gérer les listes *Task*.

AddTask

```
AddTask(task, initialPC, finalPC);
          a0       a1        a2
```

Offset : -282

Description : Addtask insère un nouveau module dans le système.

Paramètres :

task Pointeur sur une structure *Task*. Les champs *tc_SPUpper*, *tc_SPLower*, *tc_SPReg* et *tc_Node.ln_Type* doivent être initialisés correctement.

InitialPC Adresse de début de traitement du programme du module.

finalPC Adresse de retour qui, au début de la tâche, est mise sur la pile. Si cette tâche rencontre une instruction RTS, cette adresse sera active. Si *finalPC* est à 0, *Exec* met en place sa routine standard *finalPC*.

FindTask

```
Task = FindTask(Nom);
 d0             a1
```

Offset : -294

Description : *FindTask* cherche une tâche portant un nom déterminé dans la liste *Task*. Si cette tâche est trouvée, un pointeur sur cette structure *Task* sera délivré. Si on met le nom à 0, on obtiendra un pointeur sur la structure *Task* de la tâche en cours.

Paramètre :

Nom Pointeur sur le nom de la tâche recherchée ou 0.

Résultat :

Task Pointeur sur la structure *task* de la tâche recherchée.

RemTask

```
RemTask(task);
        A1
```

Offset : -288

Description : *RemTask* élimine une tâche ou module du système. Si le champ *tc_MemEntry* pointe sur une liste *MemEntry*, cette dernière sera libérée.

Paramètre :

Task Pointeur sur la structure *Task* de la tâche à écarter. Si Task = 0, la tâche en cours sera éliminée.

SetTaskPri

```
Prioritéprécédente =SetTaskPri(Task, nouvellepriorité);
        D0                      A1          D0
```

Offset : -300

Description : *SetTaskPri* retourne l'ancienne priorité d'une tâche et retourne à l'ancienne valeur. Il sera donc exécuté un *Rescheduling* ce qui signifie que le temps de calcul de chaque tâche est redécoupé suivant les nouvelles priorités. Si on initialise à haute priorité une tâche avec cette fonction, cette dernière aura accès immédiatement au processeur.

Paramètres :

Task correspond au pointeur sur la structure Task de la tâche.

Nouvellepriorité

correspond à la nouvelle priorité de la tâche (8 bits < à D0).

Prioritéprécédente

correspond à l'ancienne priorité de la tâche (8 bits < à D0).

Forbid

```
Forbid()
```

Offset : -132

Description : Interdit le Task-Switching et incrémente TDNestCnt.

Permit

```
Permit()
```

Offset : -138

Description : Décrémentation de TDNestCnt et Task-Switching autorisé à nouveau lorsque TDNestCnt < 0.

Disable

```
Disable()
```

Offset : -120

Description : Interdiction de toutes les interruptions et incrémentations de IDNestCnt.

Enable

```
Enable()
```

Offset : -126

Description : Décrémentation de IDNestCnt et interruption à nouveau active lorsque IDNestCnt < 0.

2.4.2 COMMUNICATION ENTRE TACHES

Toutes les tâches ne traitent pas que leurs propres données. Elles peuvent communiquer leurs données avec d'autres tâches. La plupart de ces communications sont des processus de type entrée/sortie car les routines qui gèrent les différents outils d'entrée/sortie tels que le clavier, l'écran ou les disquettes peuvent être considérés comme des tâches propres. Il a déjà été dit que certaines tâches attendent un signal d'autres tâches avant de rentrer en action. Ces signaux sont décrits dans le paragraphe suivant.

2.4.2.1 Les signaux Task

Chaque tâche possède 32 bits de signal afin d'avoir la possibilité de différencier les processus. Chaque tâche peut utiliser n'importe quel signal. Un signal déterminé peut avoir par contre une tout autre signification suivant la tâche concernée. Chaque fonction système utilise des signaux déterminés pour ses communications. Si une tâche veut utiliser un de ces signaux, elle doit tout d'abord l'occuper. Ceci se passe avec la fonction *AllocSignal()*. Normalement les 16 bits inférieurs sont réservés aux fonctions du système, ce qui laisse 16 signaux à utiliser librement. On peut avec *AllocSignal()* occuper un signal déterminé lorsque l'on donne le numéro du signal souhaité en argument ou on laisse *AllocSignal* chercher lui-même le prochain signal libre lorsque l'on entre -1.

Comme résultat, nous obtenons toujours le numéro signal souhaité s'il n'a pas encore été occupé, ou -1 si une err s'est produite. *AllocSignal(-1)* retoume la valeur -1 lorsqu'aucun signal n'est libre.

Le programme C suivant occupe le premier signal trouvé avec la fonction *AllocSignal* :

```
Signal=AllocSignal(-1L);
Signal<0 ?
printf("Pas de signaux libres"||):
printf("Le signal numero %ld est occupé!", (long)Signal);
```

Il y a deux façons d'indiquer un signal déterminé : on peut indiquer son numéro qui est un nombre compris entre 0 et 31 correspondant au numéro de bit ; la deuxième façon est d'indiquer le mot Signal en entier. Dans ce masque Signal, on retrouve le signal correspondant, l'avantage ici étant la possibilité d'indiquer plusieurs signaux en une seule fois. *AllocSignal* retourne le numéro du signal. Pour le retrouver par le masque Signal, on doit initialiser le bit au numéro du signal suivant la position du bit. Ceci peut se faire en C de la manière suivante :

```
MasqueSignal=1<<NumeroSignal;
```

En langage machine :

```
MOVE.W NumeroSignal, D0
MOVE.L MasqueSignal, D1
BSET D0, D1
```

ou les variables *MasqueSignal* et *NumeroSignal* reproduisent les valeurs des adresses correspondantes. Les signaux occupés par une tâche sont mis en mémoire dans le champ *tc_SigAlloc* d'une structure Task.

Attente du signal

La fonction courante d'un signal est d'initialiser l'exécution d'une tâche. Examinons le cas où une tâche attend une autre tâche déterminée.

Ceci peut être fait ous la forme d'une boucle mais prend une part de temps de calcul s , autre résultat. Afin d'éviter cette perte, une tâche peut attendre par exemple un événement tel que l'activation d'une touche grâce à l'attente d'un signal Task. Chaque tâche peut attendre son propre signal. Pendant l'attente, la tâche est transférée dans la liste *Waiting*, ne nécessitant ainsi aucun temps de calcul. Pour laisser une tâche attendre son propre signal, il existe la fonction *Wait()*. Cette dernière ne nécessite que quelques paramètres tel qu'un masque signal comprenant tous les signaux qui doivent être attendus. Il est ainsi possible d'attendre plusieurs signaux en même temps. Dès qu'un des signaux indiqués est occupé par une autre tâche, la fonction *Wait()* s'inverse et donne comme résultat un masque signal comprenant le ou les signaux. Au moyen de l'opération ET logique entre les signaux souhaités et le masque signal retourné par la fonction *Wait()*, on peut déterminer quel signal est apparu. Le programme exemple suivant en C vous le démontre :

```
unsigned long Signal;
Signal=Wait(Touche|Bouton souris|Menu);
if(Signal & Touche) ( /* Touche enfoncée */ )
if(Signal & BoutonSouris) ( /* Bouton souris activé */ )
if(Signal & Menu ) ( /* Point menu activé */ )
```

Si un des signaux souhaités est déjà occupé avant l'appel, la fonction *Wait()* retourne de suite au programme.

Les signaux source de l'attente d'une tâche sont mis en mémoire dans le champ *tc_SigWait*, ceux réceptionnés étant mis en mémoire dans *tc_SigRecvd*.

Si avant l'appel d'une fonction *Wait()*, le *Task-Switching* ou les interruptions sont désactivés, la fonction les initialisera à nouveau. Lorsque *Wait()* retourne au programme, l'état d'interdiction est à nouveau mis en place. Il existe cinq routines liées aux signaux des tâches ou *SetSignals()* n'est pas nécessaire à l'utilisateur normal.

AllocSignal

```
NumeroSignal = AllocSignal(NumeroSignal);
     D0                          D0
```

Offset : -330

Description : Au moyen de cette fonction, on peut occuper un des signaux Task. Si on donne comme numéro de signal -1, *AllocSignal()* cherche le premier signal libre et l'occupe. Si le signal souhaité est déjà occupé, *AllocSignal()* retourne la valeur -1.

Cette fonction ne peut être appelée que pour les signaux propres aux tâches. Par contre, elle ne doit pas être appelée à l'intérieur d'une exception.

Paramètre :

NumeroSignal Ceci correspond au numéro du signal à occuper (0-31) ou -1.

Résultat :

NumeroSignal Ceci correspond au numéro du signal occupé ou -1 lorsque le signal demandé est déjà occupé.

FreeSignal

```
FreeSignal(NumeroSignal);
     D0
```

Offset : -336

Description : *FreeSignal()* est la fonction inverse de *AllocSignal()*. Le signal correspondant au numéro est libéré. De la même manière que *AllocSignal()*, *FreeSignal()* ne doit pas être employé à partir d'une exception.

Paramètre :

NumeroSignal ceci correspond au numéro du signal à libérer (0-31).

SetSignals

```
Signalprécédent = SetSignals(NouveauSignal, Masque);
     D0                      D0              D1
```

Offset : -306

Description : *SetSignals()* transfère l'état de chaque signal dont le bit correspondant est initialisé dans le masque en un nouveau signal dans *tc_SigRecvd*. Si un bit de NouveauSignal et de Masque est égal à 1 le signal correspondant sera activé. Si un bit de Nouveau-Signal = 0 alors que celui de Masque est égal à 1, le signal sera libéré. Si un bit du Masque est égal à 0, le signal Task reste inchangé.

Paramètres :

NouveauSignal contient le nouvel état du signal.

Masque détermine le bit Signal à modifier.

Résultat :

Signalprécédent indique l'état précédent du signal Task.

SetSignals(OL, OL)
libère le signal sans le modifier.

Signal

```
Signal(Task, Signal);
        A1     D0
```

Offset : -324

Description : Avec cette fonction, on peut occuper un signal d'une autre tâche. Cette fonction est le centre du système Signal car il est possible avec elle de communiquer entre tâches. Lorsqu'une tâche réceptionne un signal attendu, elle retourne automatiquement dans le mode *Ready* ou *Running*. Cette fonction est utilisée par les messages système qui seront décrits dans le chapitre suivant.

Paramètres :

Task est un pointeur sur la structure Task de la tâche qui réceptionne.

Signal correspond au masque Signal contenant le bit Signal à communiquer.

Wait

```
Signal=Wait(MasqueSignal);
  D0            D0
```

Offset : -318

Description : *Wait* attend un signal du masque signal indiqué. Ceci signifie qu'une tâche restera dans l'état *Waiting* aussi longtemps que le signal ne sera pas occupé par une autre tâche ou interruption. Si un des signaux est activé avant l'appel de la fonction *Wait*, *Wait* retourne au programme. Le résultat est sous la forme d'un masque signal contenant les signaux attendus.

Attention : Cette fonction doit être appelée uniquement en mode utilisateur.

Paramètre :

MasqueSignal ce sont les signaux contenus dans ce masque qui sont attendus par la tâche.

Résultat :

Signal ceci correspond aux signaux réceptionnés issus du masque.

2.4.2.2 Le message système

Les signaux Task forment l'élément de base pour les communications système entre tâches aussi appelées message système. Ceci autorise non seulement le transfert des signaux mais aussi l'envoi et la réception des communications pouvant contenir n'importe quelle donnée.

On trouve aussi le schéma automatique des boucles d'attente dans le cas où le réceptionnaire ne peut réagir assez vite à une information. La base de ce type de communication est le port Message. Ceci est à nouveau une structure de données. En C, la structure est du format suivant :

```
struct MsgPort {
  struct Node mp_Node;
  UBYTE mp_Flags;           /*(14) Flags pour le mode Action
  UBYTE mp_SigBit;          /*(15) Bit Signal de la tâche
  struct Task *mp_SigTask;  /*(16) Pointeur sur la tâche qui réceptionne
  struct List mp_MsgList;   /*(20) Tête de liste de la liste message
```

Un port message est un emplacement pour l'information d'une tâche. Chaque tâche peut envoyer des données à un port message mais seule une tâche peut les réceptionner. Les champs d'une structure Message-Port ont la signification suivante :

mp_Node

Ceci est une structure *Node* telle qu'elle a été vue dans le chapitre 1. Dans le champ *ln_Name*, on trouve un pointeur sur le nom du port message. Ceci facilite la recherche d'un port message déterminé. Le type *Node* dans *ln_Type* est toujours pour un port message NT_MSGPORT.

Les autre champs de la structure *Node* ne seront utilisés que lorsque l'on cherchera à trouver un port message dans une liste. Cette dernière pourra être de type personnel où appartenir à une liste du port public, liste de tous les ports messages connus par *Exec*.

mp_Flags

Les deux bits inférieurs de ce champ déterminent ce qui se passe lorsque le port message réceptionne une information. Il existe plusieurs possibilités :

PA_IGNORE (2)
Cette combinaison de bit détermine qu'il ne se passe rien lorsqu'une information est réceptionnée.

PA_SIGNAL (0)
A chaque fois que le port message réceptionne une information, le signal sera envoyé au champ *mp_SigBit* de la tâche cible.

PA_SOFTINT (1)
A chaque information réceptionnée une interruption du software est libérée.

mp_SigBit

Dans ce champ, on trouve le numéro du bit signal qui sera envoyé à la tâche lorsque mp_Flags = PA_SIGNAL. Il s'agit d'un numéro de signal compris entre 0 et 31.

mp_SigTask

Ce champ doit contenir un pointeur sur la structure Task de la tâche à qui le signal présent dans *mp_SigBit* doit être communiqué. Si le mode PA_SOFTINT est sélectionné, *mp_SigTask* contient un pointeur sur la structure *Interruption* de l'interruption Software correspondante.

mp_MsgList

Ceci correspond à la tête de liste de la liste de toutes les informations (Messages). Chaque information réceptionnée sera rajoutée à la fin de cette dernière, ceci aussi bien dans le cas où l'on a sélectionné le mode PA_IGNORE, PA_SOFTINT ou PA_SIGNAL. Cette tête de liste doit être initialisée de façon correspondante comme par exemple avec *NewList()*.

Structure d'une information

Chaque information se trouve sous la forme d'une structure *Message* dont la longueur maximale doit atteindre 64 Koctets. L'information est rajoutée directement à la structure Message. Cette structure est interrompue dans le fichier *Include "exec/ports.h"* :

```
struct Message {
   struct Node mn_Node;
   struct MsgPort *mn_ReplyPort;/* (14) Port de réponse */
   UWORD mn_Lenght;/* (18) Longueur du message en bits */
}
```

mn_Node

Ceci est une structure *Node* normale. Elle sert à lier le message à la liste des messages réceptionnés du port Message. Le champ *ln_Typ* est à initialiser avec le type Node NT_MESSAGE.

mn_Lenght

Ce champ renferme la longueur de l'information en octets.

Envoi d'une information

Une information est envoyée au moyen de la fonction *PutMsg* à n'importe quel port Message.

```
PutMsg(MessagePort, Message);
```

Cette fonction envoie le message au port Message. Les deux paramètres sont à considérer comme étant des pointeurs sur les structures correspondantes. L'exemple suivant envoie un texte comme message à un port Message hypothétique.


```
{
extern APTR Por     * Pointeur sur le port Message */
static char text[] = "Ceci est un exemple d'information!";
static struct {
struct Message msg;
char contenu[sizeof(text)];
} information;

information.msg.mn_Node.ln_Type=NT_MESSAGE;
strcpy(information.contenu, text);

PutMsg(Port, &information);
}
```

Lors de l'envoi d'un message, ce dernier est rajouté à la liste des messages réceptionnés, *mp_MsgList*. Ceci se passe comme avec les champs *ln_Succ* et *ln_Pred* dans la structure *Node* du message, *mn_Node*. Le message n'est pas copié. Ceci signifie que le message entier correspond à une partie d'une tâche qui a envoyé le message. L'envoi d'un message autorise donc la tâche réceptionnaire à utiliser une zone mémoire de la tâche émettrice.

Réception d'une information

La réception se fait suivant deux opérations. En premier lieu, il s'agit d'attendre un signal du port Message et lorsque ce dernier le signalise, de réceptionner proprement dit. En constatant que le port Message est initialisé de façon correcte, la tâche à deux possibilités d'attendre l'annonce d'un message sur le port Message : il utilise soit la fonction *Wait()*, soit la fonction *WaitPort()*.

```
Message = WaitPort(Port);
```

Cette fonction attend l'annonce d'un message au port Message *"Port"*. Si un ou plusieurs messages sont annoncés, la fonction retourne au programme. Sinon elle installe la tâche dans le mode *Waiting* de la même manière que la fonction *Wait*. Le résultat est un pointeur sur structure Message du premier message, message qui ne sera p écarté du port Message.

Wait() a l'avantage de rendre possible l'attente de plusieurs messages. Si on désire attendre une information à un port message déterminé, *WaitPort()* est plus approprié car cette fonction n'attend que lorsque le port message est vide. *Wait* au contraire est géré par les signaux. Si par exemple le port Message a déjà réceptionné deux messages avant l'emploi de la fonction *Wait()*, il peut apparaître les problèmes suivants : le premier appel de *Wait()* retourne de suite au programme car les deux messages ont initialisé le signal correspondant. Si on veut attendre la prochaine information avec la même fonction, l'attente peut n'avoir aucune fin car le message souhaité est arrivé depuis un moment. Ainsi seul un signal sera initialisé même si plusieurs messages ont été recéptionnés. On est obligé alors avant l'emploi du deuxième *Wait()* de tester si la prochaine information est déjà arrivé. Pour cette raison, il est préférable d'utiliser *WaitPort()* lorsqu'on attend qu'un seul port Message.

La prise en compte d'un message est réalisée par la fonction *GetMsg()*.

```
Message = GetMsg(Port);
```

Cette fonction prend en compte la première information arrivée au port et transfert un pointeur dans sa structure Message. Le message est alors écarté de la liste du port Message. La fonction place le message à la première position de la liste. Etant donné que les nouveaux messages sont liés derrière, la liste reproduit les messages réceptionnés suivant une boucle d'attente de type FIFO. FIFO signifie en anglais *"First In First Out"* ce qui veut dire en français que l'élément prélevé en premier dans la liste sera aussi le premier à être transmis. On peut ainsi avec *GetMsg()* prendre à la suite une série de messages sur le Port. Si aucun message n'apparaît *GetMsg()* retournera la valeur 0. L'exemple suivant utilise *WaitPort()* et *GetMsg()* pour réceptionner une information à un port hypotétique :

```
extern struct MsgPort *Port;
struct Message *GetMsg();
int signal;

if((signal=AllocSignal(-1L))<0)
```



```
(printf("il r    este plus aucun signal libre!");return(0);)

Port->mp_Flags=PA_SIGNAL;
Port->mp_SigBit=signal;
Port->mp_SigTask=FindTask(0); /* recherche de la touche par elle-même */

WaitPort(Port);
message = GetMsg(Port);
```

Réponse à une information

Lorsqu'une tâche envoie une information, cette dernière voudra naturellement savoir si l'information est arrivée. La raison est que le message appartient à la tâche émettrice. En envoyant un message, elle donne au réceptionnaire l'autorisation de lire dans la zone mémoire où se trouve le message. De plus, la tâche cible peut très bien apporter une réponse au message de retour. Comme cette zone peut être très vite utilisée à nouveau par la tâche émettrice, cette dernière doit savoir si les informations mises à disposition ont été ou non réceptionnées et traitées. Elle ne peut les effacer sans savoir si elles ont déjà été ou non lues. Pour cette raison, il existe un port de type *Reply* (réponse). Ce dernier peut être n'importe quel port de la tâche émettrice. Afin de communiquer l'adresse de ce dernier à la tâche cible, il existe un champ dans la structure Message :

mn_ReplyPort

Ce champ contient l'adresse du port *Reply* de la tâche émettrice. Pour répondre à une information, la tâche cible la renvoie à ce port réponse après l'avoir lu ou éventuellement traité. La source sait ainsi que le message a été réceptionné.

```
ReplyMsg(Message);
```

autorise le retour du message.

Créer un nouveau Message-Port

Pour créer un nouveau *Message-Port*, il est nécessaire d'initialiser correctement une structure *Message-Port* et de l'insérer en mémoire. En premier lieu, ceci se fait au moyen d'une structure C avec le type mémoire *static*. Evidemment, la mémoire peut aussi être initialisée avec la routine *AllocMem()* du système.

Lorsqu'on en a terminé avec la structure *Message-Port*, on doit encore savoir si on la rajoute à la liste. Ceci se fera au moyen de la fonction *AddPort*. Cette dernière prend aussi en charge l'initialisation du champ *mp_MsgList* de la structure *Message*, rendant superflu l'appel de la routine *NewList()*. *AddPort* nécessite comme seul paramètre l'adresse de la structure *Message*. Le fait que le *Message-Port* se trouve dans la liste du Port activé, présente l'avantage suivant : une autre tâche de ce port pourra trouver son nom très rapidement. Si on ne le rajoute pas à cette liste, on devra communiquer l'adresse du *Message-Port* à toutes les tâches voulant utiliser ce Port. Pour trouver un Port dans une liste, il existe la fonction *FindPort*.

```
Port = FindPort(nom);
```

Cette fonction cherche un *Message-Port* au moyen du paramètre *nom* et donne son adresse lorsqu'il sera trouvé. Si on insère un Port avec la routine *AddPort()*, on doit tout d'abord tester si un Port ne possède pas déjà ce nom.

Lorsqu'un Port n'est plus nécessaire, on peut l'éliminer assez facilement, après avoir attendu les messages correspondants et en répondant avec la routine *ReplyMsg()*.

Si le *Message-Port* appartient au Port activé, on doit l'écarter du système avec la fonction *RemPort(Port)*, avant de l'éliminer (libérer la mémoire).

Pour simplifier l'installation de nouveaux *Message-Ports*, il y a la fonction *CreatePort()*. Celle-ci n'appartient pas au système d'exploitation mais se trouve dans la librairie du compilateur C Amiga *(amiga.lib)*.

Son code source C est structuré de la manière suivante :

```
#include <exec/exec.h>

extern APTR AllocMem();
extern UBYTE AllocSignal();
extern struct Task *FindTask();

struct MsgPort *CreatePort (Name, Pri)
 char *Name;
 BYTE Pri;
{
BYTE Signal;
struct MsgPort *Port;

if ((Signal=AllocSignal(-1))==-1)
   return((struct MsgPort *)0);

Port=AllocMem((ULONG)sizeof(*Port),MEMF_CLEAR|MEMF_PUBLIC);

if (Port==0) {
   FreeSignal (Signal);
   return((struct MsgPort *)0);
}

Port->mp_Node.ln_Name = Name;
Port->mp_Node.ln_Pri = Pri;
Port->mp_Node.ln_Type = NT_MSGPORT;

Port->mp_Flags = PA_SIGNAL;
Port->mp_SigBit = Signal;
Port->mp_SigTask = FindTask(0);

if (name != 0)
   AddPort(Port);
else
   NewList (&(Port->mp_MsgList));

return(Port);
}
```

Cette fonction crée un *Message-Port* avec un nom et une priorité donnés. Le résultat contient l'adresse du nouveau Port ou 0 lorsqu'il n'y a plus de mémoire ou de signal libre. Si le pointeur sur le nom est différent de zéro, le Port sera inséré dans la liste. Avec cette fonction, il est possible, par exemple, d'établir rapidement et simplement un *ReplyPort*. Il existe aussi dans *Amiga.Lib*, une fonction permettant d'éliminer un Port :

```
DeletePort(Port)
 struct MsgPort *Port;
{
if ((Port->mp_Node.ln_Name) != 0)
   RemPort(Port);

Port->mp_Node.ln_Type = 0xFF;
Port->mp_MsgList.lh_Head=(struct Node *)-1;

FreeSignal(Port->mp_SigBit);

FreeMem(Port,(ULONG) sizeof(*Port));
}
```

DeletePort() élimine le Port donné. Si cette fonction connaît le nom de ce dernier, elle pourra l'effacer de la liste du Port activé.

Task-Exception

Il peut être gênant, lors de l'attente d'un signal, que la tâche ne puisse pas être exécutée. Si on prend, par exemple, un module qui dessine une fonction mathématique. Comme le dessin de cette dernière peut durer assez longtemps, il doit y avoir la possibilité d'interrompre le processus en appuyant sur une touche. Ceci doit être transmis par un *Message-Port*. On doit par contre tester le signal à l'intérieur de la boucle de dessin, pour savoir si le Port a accepté un message. Ce test va, malheureusement, ralentir la boucle. Cet état peut être évité sur l'Amiga au moyen d'une *Task-Exception* (état d'exception d'une tâche).

Comme pour une interruption, la tâche sera interrompue par le déclenchement d'un signal. Ceci peut se passer à n'importe quel endroit. Puis le convertisseur d'exception sera appelé. Celui-ci fait partie de la tâche primaire et a donc accès à ses données. Dans notre exemple, il peut initialiser la variable de boucle avec la dernière valeur, abrégeant ainsi le processus de dessin. En langage machine, il existe beaucoup plus de possibilités. On peut, par exemple, manipuler directement la tâche.

Pour permettre une *Task-Exception*, les étapes suivantes sont nécessaires :

1) L'adresse de départ du convertisseur *Exception* doit être interrompue avec le champ correspondant de la structure *Task,tc_ExceptCode*. Il reste la possibilité d'écrire un pointeur sur les données dans *tc_ExceptData*.

2) Il doit être établi quels signaux peuvent libérer une exception. Le champ *tc_SigExcept* de la structure *Task* le détermine. Chaque signal qui y est activé libère une exception lorsqu'il est accepté par une tâche. Pour simplifier l'initialisation ou l'élimination d'un bit de ce champ, il existe une fonction spéciale : *SetExcept*. La description exacte se trouve dans le récapitulatif, à la fin de ce chapitre.

Si une exception se déclenche, *Exec* met le contenu actuel des registres processeurs (PC, SR, D0-7 et A0-6) sur la pile *Task*, afin de permettre la suite du déroulement de la tâche à la fin de l'exception.

Puis un masque signal est mis dans D0, celui-ci renfermant tous les signaux d'exception permis. L'adresse se trouvant dans *tc_ExceptData* sera copiée dans A1. Enfin, le traitement des codes d'exception débutera à l'adresse présente dans *tc_ExceptCode*.

La fin d'une exception doit être signalée par RTS. A ce moment-là, *Exec* reprend le contenu des anciens registres dans la pile *Task* et continue le déroulement de la tâche.

Lors d'une exception, *Exec* empêche le déclenchement d'autres exceptions.

Si un signal apparaît lors d'une exception et une nouvelle exception est libérée, cette dernière sera exécutée à la fin du déroulement de l'exception présente.

Si un signal est déjà activé, avant qu'on autorise, par *SetExcept*, l'exécution d'une exception, cette dernière se déclenchera automatiquement.

Trap processeur

L'autre état d'exception correspond aux *Traps*. Ces derniers déterminent les exceptions du processeur 68000 qui ne doivent pas être confondues avec les exceptions *Task*, décrites plus haut, même si Motorola leur a attribué le même nom d'exception. Les exceptions 68000 suivantes seront caractérisées par *Exec* en tant que *Trap* :

Trap :	2	Erreur de Bus
	3	Erreur d'adresse
	4	Erreur d'instruction illégale
	5	Division par zéro
	6	Erreur CHK
	7	Erreur TRAPV
	8	Violation de privilège
	9	Trace
	10	Erreur avec 1010 au départ (line 1010 emulator)
	11	Erreur avec 1111 au départ (line 1111 emulator)
	32-37	Erreur TRAP

Un *Trap* est toujours la suite directe d'une déclaration dans un programme. Ceci peut être soit voulu (CHK, TRAPV, TRAP, 1010, 1111 ou TRACE), soit issu d'une erreur de programmation (Erreur de Bus, d'adresse, division par zéro, violation de privilège).

Si un *Trap processeur* se déclenche, *Exec* saute à un convertisseur *Trap*. L'adresse du convertisseur se trouve dans le champ *tc_Trapcode*.

Normalement, on y trouve un pointeur sur le convertisseur *Trap* standard d'*Exec*.

Ceci engendre (malheureusement) le message bien connu *"Software error-Task held"* ou bien une méditation du gourou (guru meditation).

On peut aussi replier l'adresse présente dans *tc_Trapcode* sur un convertisseur personnel, celui-ci pouvant réagir soit à tous les *Traps*, soit à un seul déterminé, puis sauter au convertisseur standard. Un convertisseur *Trap* sera accessible directement. *Exec* met exclusivement les numéros de *Trap* (Cf liste ci-dessus), sur la pile. Ceci a la signification suivante :

En premier lieu, on se trouve en mode superviseur et on travaille avec la pile superviseur. Pendant le déroulement du convertisseur *Trap*, le *Task-Switching* est désactivé.

Deuxièmement, le contenu de la pile peut varier. Le format normal est le suivant :

Stackpointer	(SSP)	numéro de Trap (mot long)
Stackpointer	+4	registre Status (mot)
Stackpointer	+2	adresse de retour (mot long)

Lors d'une erreur d'adresse ou de bus, d'autres informations seront mises sur la pile. Le déroulement sera différent suivant le processeur employé (68000, 68010, 68020). Pour que la compatibilité soit totale, il faudra faire très attention. Pour plus d'informations, reportez-vous à la littérature concernant le 68000.

Pour libérer le convertisseur *Trap*, il suffit de prendre le numéro de trap de la pile (attention ! mot long), et d'y ajouter une instruction RTE. Comme aucun des registres d'*Exec* n'a été sauvegardé sur la pile, il ne faudra pas modifier le contenu des registres processeurs.

L'exemple suivant utilise un convertisseur *Trap* pour intercepter une division par zéro. Comme on peut difficilement écrire en C un tel convertisseur *Trap*, il sera directement intégré dans le code source en langage machine au moyen de *#asm* et de *#endasm*. Etant donné que beaucoup de compilateurs C ne connaissent pas ce genre de déclaration du préprocesseur, il est conseillé de compiler et d'assembler séparément la partie C et la partie en langage machine, et de les linker par la suite. Ce programme a été conçu avec AZTEC C.

```
/****  Interception d'une exception du 68?   ****/

#include <exec/execbase.h>

extern struct ExecBase *SysBase;

main()
{
/*** Rajout du convertisseur Trap ***/

extern APTR Trap;
APTR oldtrap;
USHORT Nombre1,Nombre2;
struct Task *ThisTask;

oldtrap=SysBase->ThisTask->tc_TrapCode;
SysBase->ThisTask->tc_TrapCode=&Trap;
SysBase->ThisTask->tc_TrapData=(APTR)0;

/*** Libération d'un Trap "Division par zéro" ***/

Nombre1 = 10; Nombre2 = 0;
Nombre1 = Nombre1/Nombre2;

if((ULONG)SysBase->ThisTask->tc_TrapData==0)
 printf("Cette place ne sera plus atteinte\n!");
else
 printf ("Exception reconnue, tc_TrapData = Numéro-de-Trap:%ld\n",
 SysBase->ThisTask->tc_TrapData);

/*** Convertisseur Trap désactivé ***/

SysBase->ThisTask->tc_TrapCode=oldtrap;

}

/*** Convertisseur Trap ***/
/** Créé avec Aztec C **/
/** TAB pour Opcode nécessaire! **/
```



```
#asm

_Trap move.l a0,-(sp)
      move.l 4,a0          ;SysBase
      move.l 276(a0),a0    ;SysBase->ThisTask
      move.l 4(sp),46(a0) ;Numéro de Trap dans
                           ;SysBase->ThisTask->tc_TrapData
      move.l (sp),a0
      add.l  #8,sp
      rte

#endasm
```

Sans le convertisseur *Trap*, ce programme se serait "planté" avec un *"Software Error-Task held"*, étant donné qu'il y a division par zéro. La preuve qu'il y a eu libération de *Trap* nous est donné par l'instruction IF. En effet, seul *Trap* peut différencier *tc_TrapData* de 0, après qu'il ait été effacé par la tâche.

Directement après la division illégale, *Exec* saute au convertisseur *Trap*. Celui-ci prend le numéro de *Trap* sur la pile superviseur et l'écrit dans le champ *tc_TrapData*. Puis l'instruction *printf()* affiche à l'écran le nombre 5 qui correspond en fait au numéro de *trap* d'une division par zéro.

Les instructions Trap

Un *Trap* qui sera initialisé par une des 16 instructions *Trap* (numéro 32 à 47) sautera aussi au convertisseur *Trap*. Il y a aussi la possibilité de déterminer à l'avance les numéros de *Trap*, comme pour les signaux, par les fonctions *AllocTrap* et *FreeTrap*.

L'allocation et la libération des *Traps* ne servent, exclusivement, qu'à faire connaître les *Traps* utilisés ou non.

Si une instruction *Trap* se déclenche, elle sera toujours branchée au convertisseur, et cela indépendamment du fait qu'une instruction *Trap* ait été initialisée par *AllocTrap* ou non.

AddPort

```
AddPort(Port);
        a1
```

Offset : - 354

Description : AddPort() insère une structure *message-port* donnée à la liste du port activé. Elle sera classée suivant son niveau de priorité. On peut accéder à la tête de liste par *SysBase -> PortList*. *AddPort* initialise aussi la structure *mp_MsgList* à l'intérieur du *Message-Port*.

Paramètre :

Port Pointeur sur la structure *Message-Port*.

AllocTrap

```
Numtrap = AllocTrap(Numtrap);
 d0                 d0
```

Offset : - 342

Description : AllocTrap permet d'occuper une des instructions *Trap* du 68000. Le numéro *Trap* peut être compris entre 0 et 15, initialisant ainsi l'instruction *Trap* correspondante. Avec le numéro -1, *AllocTrap* initialise la première instruction *Trap* libre.

Paramètre

Numtrap Nombre compris entre 0 et 15 (ou -1 pour la première instruction libre).

Résultat :

Numtrap contient l'instruction *Trap* occupée. Si numtrap = -1, c'est que l'instruction souhaitée n'est pas libre ou qu'il n'y a plus d'instructions libres.

FindPort

```
Port = FindPort(Nom);
 d0             a1
```

Offset : - 390

Description : FindPort() cherche le prochain *Message-Port*, caractérisé par un nom, dans la liste du Port activé. Si ce port existe, cette fonction donne, en retour, un pointeur sur ce port.

Paramètre :

Nom Nom du port cherché.

Résultat :

Port Pointeur sur le *Message-Port*. Lorsque ce dernier n'existe pas, Port = 0.

FreeTrap

```
FreeTrap(Numtrap);
          d0
```

Offset : - 348

Description : *FreeTrap* libère l'instruction *Trap* caractérisée par le numéro donné.

Paramètre :

Numtrap Numéro de l'instruction *Trap* (0-15).

PutMsg

```
PutMsg(Port, message);
        a0      a1
```

Offset : - 366

Description : *PutMsg* envoie une information à un *Message-Port*. Dans ce dernier, elle sera rajoutée à la liste des informations et le contenu du champ *mp_Flag* de l'action correspondante sera libérée.

Paramètres :

Port Adresse de la structure *Message-Port* du Port de destination.

Message Pointeur sur la structure *Message* du message.

RemPort

```
RemPort(Port);
         a1
```

Offset : - 360

Description : Cette fonction élimine un *Message-Port* de la liste des Ports actifs. Il ne sera alors plus possible d'y accéder au moyen de *FindPort*.

Paramètre :

Port Pointeur sur le *Message-Port*.

ReplyMsg

```
ReplyMsg(message);
          a1
```

Offset : - 378

Description : *ReplyMsg()* envoie un message de retour à son *Reply-Port*. Si le champ *mn_ReplyPort* de la structure *Message* est égal à 0, cette fonction sera ignorée.

Paramètre :

Message Pointeur sur la structure *Message*.

SetExcept

```
Présignal = SetExcept(NewSignal, masque);
    d0                    d0         d1
```

Offset : - 312

Description : *SetExcept* détermine quel signal peut être libéré par une exception. Le comportement de cette fonction est identique à celui de *SetSignal()*.

Paramètres :

NewSignal Nouveaux états des signaux *Exception.*

Masque Le masque établit quels signaux *Exception* doivent être influencés.

Résultat :

Présignal Etat des signaux *Exception* avant la modification.

WaitPort

```
Message = WaitPort(Port);
                   a0
```

Offset : - 384

Description : *WaitPort* attend la réception d'un message à un Port déterminé. Si un message y parvient ou s'il y en avait déjà un présent avant l'appel de la fonction, *WaitPort()* retournera l'adresse de ce message. Le message ne sera pas éliminé de la liste des messages réceptionnés. Pour ce faire, on devra utiliser *GetMsg()*.

Paramètre :

Port Adresse du Port.

Résultat :

Message Pointeur sur le premier message de la liste.

2.5 Gestion de la mémoire de l'Amiga

L'Amiga dispose d'une gestion de la mémoire dynamique, pour tout ce qui concerne mémoire d'écran, mémoire disquette, etc...

Ceci se passe d'une telle manière que chaque chargement de programme peut se voir affecter n'importe quelle zone mémoire. Cette gestion dynamique de la mémoire permet le traitement simultané de plusieurs programmes, étant donné qu'aucun d'entre eux ne se verra attribuer une zone mémoire prédéfinie, comme c'est le cas, par exemple, pour le C64.

Le système doit être, par contre, informé de la mémoire nécessaire à un programme, afin d'en attribuer une quantité suffisante à l'utilisateur. Le système ne s'apercevra pas que tel programme occupe telle zone mémoire, mais saura, au contraire, qu'il n'en a plus à disposition, pour telle ou telle tâche.

Lorsqu'une tâche ne nécessite plus une zone mémoire déterminée, elle avertit le système, afin que cette zone puisse être attribuée à une autre tâche. Si cela n'est pas communiqué au système, cette zone restera occupée jusqu'au prochain RESET, provoquant ainsi une diminution importante de la capacité mémoire.

L'occupation de la mémoire n'est possible que par blocs de 8 octets. Autrement, le système arrondira la taille mémoire donnée au prochain multiple de 8 octets. La zone mémoire minimale disponible est donc de 8 octets.

Pour occuper ou libérer une zone mémoire particulière, il existe dans la librairie *Exec* plusieurs fonctions, dont deux sont particulièrement importantes. Il s'agit des fonctions *AllocMem()* et *FreeMem()*.

Pour occuper la mémoire, il faut communiquer au système la quantité à attribuer, ainsi que certains caractères propres à cette zone.

Ces modalités qui peuvent être choisies sont :

MEMF_CHIP

Indique que la mémoire affectée doit être de type CHIP-MEMORY. Cette zone de 512 Koctets est directement accessible par le Blitter. Le graphisme et le son doivent toujours être stockés dans cette partie. Même si, à l'heure actuelle, cette zone ne possède que 512 Koctets et que cette modalité est toujours exécutée, vous devrez, pour des raisons de compatibilité, avec une extension mémoire, reconsidérer l'ensemble. Le code de cette condition est $02 et sera comme toutes les autres modalités, défini dans le fichier *Include "exec/memory.h"* en C ou *"exec/memory.i"* en assembleur.

MEMF_FAST

Indique que la mémoire affectée doit se trouver en dehors de la zone inférieure de 512 koctets (CHIP-MEMORY). L'affectation ne sera possible que si on dispose d'une extension RAM. Le code de cette modalité est $04.

MEMF_PUBLIC

Indique que la mémoire affectée ne doit pas être décalée. Le décalage n'est pas intégré dans la version présente du système d'exploitation, mais il devra être utilisé pour des raisons de compatibilité de gestion de mémoire des tâches, interruptions, Message-Port, etc... Le code de cette modalité est de $01.

MEMF_CLEAR

Indique que la mémoire affectée doit être remplie par des 0. Le code de cette modalité est $10000.

MEMF_LARGEST

Indique que la mémoire affectée doit être un bloc mémoire de la plus grande taille disponible. Le code est $20000.

Si plusieurs modalités doivent être exécutées, comme par exemple CHIP et CLEAR, les codes devront être couplés avec un OU logique.

Si la condition "CHIP OU FAST-MEMORY" doit être réalisée, le système cherchera d'abord à exécuter FAST-MEMORY. En cas d'échec, c'est CHIP-MEMORY qui sera exécuté.

Attention :

On doit éviter d'occuper ou de libérer de la mémoire lors d'une interruption, étant donné que les routines ont été conçues pour ne pas bloquer les interruptions. Si une tâche traite une routine d'affectation de mémoire, lors d'une interruption, qui travaille avec des routines mémoire, le système pourra se trouver en grande difficulté.

AllocMem

```
Mémoire = AllocMem(taillemem, condition);
  d0                d0          d1
```

Offset : - 198

Description : La fonction cherche une zone mémoire libre, qui corresponde aux conditions données, puis la caractérise comme zone occupée. L'adresse de départ de la zone trouvée sera donnée par d0. S'il n'est pas possible d'occuper la zone mémoire souhaitée, le système retournera un 0 dans d0 pour signaler l'erreur.

Paramètres :

Taillemem indique la taille de la mémoire qui doit être attribuée.

Condition correspond aux modalités énoncées précédemment, caractérisant la fonction *AllocMem()*.

Avec la fonction *AllocMem()*, il n'est pas possible d'attribuer une zone mémoire déterminée, mais il est possible d'en disposer d'une.

Comme il existe une fonction permettant d'attribuer et d'occuper de la mémoire, il y a aussi une fonction permettant de libérer de la mémoire.

FreeMem

```
FreeMem(blocmémoire, taille);
          a1          d0
```

Offset : - 210

Description : La fonction rend au système la zone mémoire précédemment occupée, permettant ainsi une nouvelle affectation pour une autre tâche. Les paramètres pris en charge par cette fonction seront arrondis.

Paramètres :

Blocmémoire Pointeur sur le début de la zone mémoire, qui doit être rendue au système. Le pointeur sera arrondi à un multiple de 8.

Taille indique la quantité de mémoire à libérer. La taille sera arrondie à un multiple de 8.

Attention :

Si on cherche à libérer de la mémoire non occupée, le système s'effondrera avec le numéro de guru suivant : 81000009.

Le programme C suivant montre comment on peut disposer de la mémoire, puis à nouveau la libérer.

```
#include <exec/memory.h>
#include <exec/types.h>

#define SIZE 1000

main()
{
```

```
    ULONG Mémoire;

    Mémoire = AllocMem (SIZE,MEMF_CHIP | MEMF_PUBLIC);

    if (Mémoire = 0) {
          printf ("\n  Mémoire non attribuée\n");
          exit (0);
          }
    printf ("\n  Mémoire attribuée\n");

    FreeMem (SIZE,Mémoire);
}
```

Dans "MEMOIRE" sera marquée l'adresse de départ de la mémoire attribuée.

2.5.2 LA STRUCTURE MEMORY-LIST

Il est souvent nécessaire d'occuper plusieurs zones mémoire. Pour cette raison, on doit appeler, pour chaque zone, la fonction *AllocMem*. La librairie *Exec* dispose de deux fonctions pour nous faciliter la tâche. Ces fonctions se nomment *AllocEntry()* et *FreeEntry()*.

Pour pouvoir les appeler, on doit en premier lieu, initialiser une structure, dans laquelle les fonctions prendront leurs valeurs.

Cette structure s'appelle MemList et est de la forme suivante :

```
    struct MemList  {
0     struct Node ml_Node;
14    UWORD ml_NumEntries;
16    struct MemEntry ml_ME [1];
    };
```


ml_Node

C'est une structure de type *Noeud*, capable de coupler plusieurs structures *MemList*.

ml_NumEntries

Indique la quantité de mémoire qu'on pense occuper.

ml_ME[1]

C'est une structure isolée, qui contient les conditions de la mémoire à réserver ainsi que la taille de la zone. La structure est de la forme suivante :

```
struct MemEntry {
  union {
    ULONG meu_Reqs;
    APTR meu_Addr;
  }      me_Un;
  ULONG me_Length;
};

#define me_un me_Un
#define me_Reqs me_Un.meu_Reqs
#define me_Addr me_Un.meu_Addr
```

me_Un

Se partage selon la condition *'union'*. Il peut contenir soit les conditions (Requests) de l'affectation de la mémoire, soit le pointeur sur la mémoire réservée. Lors de l'installation de la structure *MemEntry* pour l'appel de la fonction *AllocEntry()*, on y trouve les conditions d'affectation (par exemple : MEMF_CHIP) et après l'appel de la fonction, l'adresse de départ de la zone mémoire affectée.

me_Length

Donne la longueur de la mémoire réservée.

AllocEntry

```
Liste = AllocEntry(MemList);
 d0                    a0
```

Offset : - 222

Description : Cette fonction délivre un pointeur sur la structure *MemList* dans a0. Elle cherche seulement à occuper toutes les zones mémoire insérées dans la structure. Dès qu'une zone est occupée, le pointeur de cette zone sera mis à la place des conditions (requirements).

Paramètres :

Liste Pointeur sur la structure *MemList*, nouvellement placée. Si une erreur apparaît pendant l'affectation, il sera retourné dans d0, les conditions de la mémoire non attribuée, où le bit de poids fort (bit 31) sera activé. Dans ce cas, la mémoire ne sera pas occupée, même si d'autres *Entries* auraient pu le faire.

FreeEntry

```
FreeEntry(liste);
          a0
```

Offset : - 228

Description : Cette fonction retourne au système toutes les zones mémoire marquées dans la structure *MemList*.

Paramètre

Liste Pointeur sur la structure *MemList* qui doit être délivrée par la fonction *AllocEntry()*.

Il est possible, soit avec la fonction *AllocEntry()*, soit avec la fonction *FreeEntry()*, de traiter plusieurs structures *MemList* liées en même temps.

Vous vous demandez peut-être comment il est possible d'initialiser plusieurs *Entries* avec une structure *MemList*, alors que cette dernière n'est reliée qu'à une structure *MemEntry(ml_ME [1])*. Pour initialiser plusieurs *Entries*, on doit utiliser une astuce. On se représente pour cela une structure isolée, qui est par exemple de la forme suivante :

```
struct {
        struct MemList me_tête;
        struct MemEntry me_rajout [3];
        } MaListe;
```

On ne délivre pas à la fonction *AllocEntry* un pointeur sur la structure *MemList*, mais un pointeur sur sa propre structure initialisée, avec lequel il sera possible d'initialiser plusieurs *Entries*. Dans notre exemple, 4 zones mémoire seront occupées par la fonction *AllocEntry*, étant donné que la structure *MemEntry* dispose encore de sa propre structure.

Voici l'utilisation de la fonction *AllocEntry()* par un exemple :

```
#include <exec/memory.h>
#include <exec/types.h>

   struct MemListe {
      struct MemList me_Tête;
      struct MemEntry me_Rajout[2]; };

main()
{
   struct MemList *Listemémoire, *AllocEntry();
```

```
struct MemListe MaListe;

MaListe.me_Tête.ml_NumEntries = 3;
MaListe.me_Tête.ml_me[0].me_Reqs = MEMF_CLEAR;
MaListe.me_Tête.ml_me[0].me_Length = 100;
MaListe.me_Tête.ml_me[1].me_Reqs = MEMF_CLEAR | MEMF_FAST;
MaListe.me_Tête.ml_me[1].me_Length = 1900;
MaListe.me_Tête.ml_me[2].me_Reqs = MEMF_PUBLIC | MEMF_CHIP;
MaListe.me_Tête.ml_me[2].me_Length = 300;

Listemémoire = AllocEntry (&MaListe);

if ( ((ULONG)Listemémoire) >> 30) {
   printf("\n  Toutes les entrées ne peuvent être attribuées\n");
   exit (0);
   );

)
```

2.5.3 AFFECTATION DE MEMOIRE ET LES TACHES

Quand une partie de la mémoire doit être occupée par une tâche, il est recommandé de le faire avec la fonction *AllocEntry()*.

Dans la structure *Task* est reliée une structure *Liste (tc_MemEntry)*, dans laquelle la mémoire occupée de la forme structure *MemList* peut être couplée à une liste. La mémoire qui est couplée à cette liste peut alors facilement être rendue au système par le biais des routines que la tâche libère. Un autre avantage de l'insertion de la mémoire utilisée dans la liste est que la tâche peut reconnaître à tout moment la zone qu'elle occupe.

Il est naturellement aussi possible d'occuper une zone mémoire avec la fonction *AllocMem()* et de l'insérer dans une telle liste.

2.5.4 LA GESTION INTERNE DE LA MEMOIRE

Puisqu'on connaît maintenant la façon de réserver une place mémoire, nous allons aborder la gestion interne de la mémoire.

Comme on peut le penser, la gestion de la mémoire se fait à nouveau au moyen d'une structure du type :

```
     struct MemHeader {
0      struct Node mh_Node;
14     UWORD mh_Attributes;
16     struct MemChunk *mh_First;
20     APTR mh_Lower;
24     APTR mh_Upper;
28     ULONG mh_Free;
     };

#define MEMF_PUBLIC  (1L<<0)
#define MEMF_CHIP    (1L<<1)
#define MEMF_FAST    (1L<<2)
#define MEMF_CLEAR   (1L<<16)
#define MEMF_LARGEST (1L<<17)
#define MEM_BLOCKSIZE 8L
#define MEM_BLOCKMASK 7L
```

mh_Node

Structure *noeud*, permettant de coupler la structure *MemHeader* à une structure *Liste*.

mh_Attributes

Indique les conditions de la mémoire à gérer (ex. MEMF_FAST).

**mh_First*

Pointeur sur la première structure *MemChunk*. L'architecture et le sens de cette structure seront détaillés plus loin.

mh_Lower

Pointeur sur le début de la mémoire, qui sera gérée par Header.

mh_Upper

Pointeur sur la fin de la mémoire, qui sera gérée par Header.

mh_Free

Indique la quantité de mémoire disponible avec la gestion de Header.

Comme cela a déjà été dit, le système ne connaît pas la quantité de mémoire occupée, mais connaît, au contraire, la quantité disponible. Les zones mémoire inoccupées sont reliées entre elles à l'aide d'une structure *MemChunk.* Par cette dernière, le système détermine sans difficulté la taille et la position des blocs de mémoires libres. Cette structure est de la forme suivante :

```
    struct MemChunk {
0     struct MemChunk *mc_Next;
4     ULONG mc_Bytes;
    };
```

* *mc_Next*

Pointeur sur la prochaine structure *MemChunk.*

mc_Bytes

Indique le nombre d'octets libres dans ce bloc mémoire.

L'entrée *m irst* de la structure *MemHeader* pointe sur la première structure *MemChunk* qui se trouve au début des premiers blocs mémoire. Les 4 premiers octets de ces blocs libres sont les pointeurs sur les prochaines zones mémoire libres. Les 4 octets suivants indiquent la taille de la zone mémoire.

Dans la dernière structure *MemChunk,* le pointeur *mc_Next* est mis à zéro et *mc_Bytes* indique le nombre d'octets entre la position mémoire présente et la valeur qui est marquée dans *mh_Upper.*

Une structure *MemHeader* gère l'entière *Chip-Memory* et, suivant le cas où elle est présente, la *Fast-Memory.* Ces structures sont réunies dans une liste qui est contenue dans la structure *ExecBase.* La liste porte le nom *"MemList"* et se trouve à l'Offset 322.

La priorité *MemHeader* pour la zone *Fast-Memory* est 0, alors que celle de la zone *Chip-Memory* est de -10. C'est la raison pour laquelle le système cherchera d'abord à occuper la *Fast-Memory,* puis seulement après la *Chip-Memory.*

Si la mémoire doit être occupée maintenant, la *MemList* de la structure *ExecBase* sera consultée afin de voir si les conditions données dans la fonction *AllocMem()* correspondent à celles de la structure *MemHeader.* En deuxième lieu, il sera testé si la mémoire libre, qui est indiquée dans *mh_Free,* suffit, afin que la quantité de mémoire donnée puisse être réservée. Si une des deux conditions n'est pas remplie, le système consultera si la prochaine structure *MemHead* est disponible. Si aucune ne l'est, un message négatif sera retourné par la fonction *AllocMem().* Sinon le pointeur *mh_First* sera activé et testera si la première structure *MemChunk,* donnée par la mémoire, suffit. Si ce n'est pas le cas, le pointeur dérivera sur la suivante et le processus continuera jusqu'à ce que la taille de la zone mémoire soit suffisante. Dans ce cas, la quantité de blocs mémoires libres et la position à laquelle la mémoire libre recommence, seront calculées.

Il y sera inséré une structure *MemChunk* qui sera reliée de manière correspondante. La nouvelle zone occupée sera écartée de l'enchainement et le nombre d'octets occupés sera retranché au total de la mémoire libre.

2.5.5 LES FONCTIONS Allocate ET Deallocate

Il reste la possibilité d'installer une structure *MemHead* propre et de gérer des zones mémoire, séparées avec les fonctions *Allocate()* et *Deallocate()*. Ces dernières permettent en tout et pour tout d'occuper ou de libérer une zone mémoire sans faire appel à une ou plusieurs conditions.

Allocate

```
Mémoire = Allocate(MemHeader, Bytesize);
 d0                a0          d0
```

Offset : - 186

Description : Cette fonction occupe la mémoire qui sera gérée par la structure *MemHead* donnée.

Paramètres :

Mem_Header Pointeur sur la structure *MemHead*.

ByteSize Indique la quantité de mémoire à occuper.

Mémoire Pointeur sur la mémoire occupée. Si cette dernière n'est pas trouvée en quantité suffisante, le nombre zéro sera retourné.

Deallocate

```
Deallocate(MemHeader, Memoire, ByteSize);
            a0          a1          d0
```

Offset : -192

Description : Cette fonction retourne la mémoire occupée à la structure *MemHeader*.

Paramètres

MemHeader Pointeur sur la structure *MemHeader*.

Mémoire Pointeur sur le début de la mémoire libre retournée.

BiteSize Indique la quantité de mémoire libre retournée.

Pour éclairer les notions développées par ces instructions, voici un exemple de programme :

```
#include <exec/execbase.h>
#include <exec/memory.h>
#include <exec/types.h>

#define Byte_Size 10000

struct ExecBase *SysBase;

main()
{
   struct MemHeader *header;
   struct MemChunk *chunk;
   APTR Mémoire, AllocMem(), Allocate();

   header = (struct MemHeader *) AllocMem(sizeof(struct MemHeader)
            ,MEMF_PUBLIC | MEMF_CLEAR);
   if (Mémoire == 0) {
      printf("\n AllocMem 1 sans succès\n");
      exit (0);
   }


   Mémoire = AllocMem(Byte_Size,MEMF_PUBLIC | MEMF_CLEAR);
   if (Mémoire == 0) {
      printf("\n AllocMem 2 sans succès\n");
      FreeMem(header,sizeof(struct MemHeader));
      exit (0);
   }
```

```
    chunk = (struct MemChunk *) Mémoire;
    chunk->mc_Next = 0;
    chunk->mc_Bytes = Byte_Size;

    header->mh_Node.ln_Type = NT_MEMORY;
    header->mh_Node.ln_Pri  = -100;
    header->mh_Node.ln_Name = "MemHeader";
    header->mh_Attributes = MEMF_PUBLIC | MEMF_CHIP;
    header->mh_First = chunk;
    header->mh_Lower = Mémoire;
    header->mh_Upper = Mémoire + Byte_Size;
    header->mh_Free  = Byte_Size;

    AddTail(&SysBase ->MemList,header);

    Mémoire = Allocate(header,500);
    if (Mémoire == 0) {
       printf("\n Allocate sans succès\n");
       exit (0);
    }

    Deallocate(header,Mémoire,500);
}
```

2.5.6 DESCRIPTION DES FONCTIONS RESTANTES

AvailMem

```
AvailMem(conditions);
   d1
```

Offset : - 216

Description : Cette fonction indique la taille de la mémoire caractérisée par les conditions (par exemple MEMF_CHIP).

AllocAbs

```
Mémoire = AllocAbs(ByteSize, Position);
  d0              d0          a1
```

Offset : -204

Description : Cette fonction permet l'occupation d'une zone mémoire déterminée, qui ne sera pas cherchée par *Exec*, mais donnée par le programmeur.

Paramètres :

ByteSize Indique la taille de la mémoire à occuper.

Position Pointeur sur la mémoire à occuper.

Mémoire Pointeur sur la mémoire occupée, correspondant à celle donnée. S'il n'est pas possible d'occuper la mémoire souhaitée, la marque d'erreur 0 sera retournée.

2.6 Manipulation IO de l'Amiga

Ce chapitre ne concerne pas l'utilisation des différentes entrées de périphériques, mais s'attache plus particulièrement à détailler la façon dont *Exec* s'occupe de la gestion IO. La direction des entrées/sorties de l'Amiga dans un programme personnel sera détaillée dans la partie DOS de ce manuel. Il est pourtant recommandé d'étudier le chapitre DOS afin de mieux comprendre ce qui va suivre.

2.6.1 ARCHITECTURE D'UNE STRUCTURE IO REQUEST

Afin de permettre un processus d'entrée/sortie, on utilise une structure *IO Request*, dans laquelle on délivrera les instructions pour le périphérique.

Il existe deux sortes de structures *IO Req* qui se nomment : "IOREQUEST" et "IOSTDREQ" (IO - Standard - Request). La structure *IOStdReq* est une extension de la structure *IORequest*. Ces structures sont de la forme suivante :

```
      struct IORequest {
0       struct Message io_Message;
20      struct Device *io_Device;
24      struct Unit *io_Unit;
28      UWORD io_Command;
30      UBYTE io_Flags;
31      BYTE io_Error;
      };
```

io_Message

Ceci est une structure *Message* telle qu'elle a été détaillée dans le chapitre 2.4. Son rôle est de permettre aux périphériques de nous signaler qu'ils en ont terminé avec les commandes IO. La structure *Message* doit être réalisée correctement avant que les entrées/sorties ne fonctionnent.

**io_Device*

Pointeur sur la structure *Device* utilisée ; celle-ci sera décrite plus loin.

**io_Unit*

Pointeur sur une structure *Unit* dont la description suit.

io_ Command

Ceci est un mot qui délivrera l'instruction d'exécution.

io_ Flags

Ceci sera nécessaire pour délivrer une instruction ou un compte rendu de status à un périphérique spécifique. L'octet est divisé en une partie de poids faible et une partie de poids fort. Les 4 bits inférieurs seront gérés par *Exec* de façon interne. Les 4 bits supérieurs pourront être utilisés par le programmeur afin de pouvoir communiquer avec un périphérique.

io_Error

Ceci sera nécessaire, afin de pouvoir délivrer au programmeur des messages d'erreur.

Parfois, cette structure ne suffit pas pour utiliser un périphérique. Dans ce cas, il existe une autre structure qui assure à l'utilisateur plus de possibilités. Elle est de la forme suivante :

```
      struct IOStdReq {
0       struct Message io_Message;
20      struct Device *io_Device;
24      struct Unit   *io_Unit;
28      UWORD io_Command;
30      UBYTE io_Flags;
31      BYTE  io_Error;
32      ULONG io_Actual;
36      ULONG io_Length;
40      APTR  io_Data;
44      ULONG io_Offset;
      };
```

io_Actual

Indique le nombre d'octets transférés. Cette valeur ne pourra être lue qu'à la fin du transfert.

io_Length

Indique le nombre d'octets à transférer. Cette valeur doit être initialisée avant le transfert. Elle est souvent mise à -1, afin de pouvoir transférer un nombre variable d'octets.

io_ Data

Pointeur sur le tampon de données dans lequel les données seront transférées.

io_ Offset

Indique l'Offset qu'un périphérique spécifique utilisera. Le périphérique lecteur de disquette donnera un bloc comme Offset.

2.6.2 ARCHITECTURE D'UN PERIPHERIQUE

La forme de la structure *Device* correspond à celle d'une librairie :

```
struct Device {
  struct Library dd_Library;
};

#define DEV_BEGINIO (-30L)
#define DEV_ABORTIO (-36L)
#define IOB_QUICK    0L
#define IOF_QUICK    (1L<<0)
#define CMD_INVALID 0L
#define CMD_RESET    1L
#define CMD_READ     2L
#define CMD_WRITE    3L
#define CMD_UPDATE   4L
#define CMD_CLEAR    5L
#define CMD_STOP     6L
#define CMD_START    7L
#define CMD_FLUSH    8L
#define CMD_NONSTD   9L
```


Pour pouvoir ...er un périphérique, il faut en premier lieu l'ouvrir. L'instruction d'ouverture est de la forme suivante :

```
Error = OpenDevice(Name, Unit, IORequest, Flags);
 d0                    a0    d0       a1        d1
```

Avant d'utiliser la fonction *OpenDevice()*, il faut tout d'abord initialiser la structure *IORequest*.

De la même façon que les librairies, chaque périphérique dispose d'une table de saut, chaque saut se faisant par Offset négatif. Les fonctions atteintes de cette manière servent à l'ouverture et à la fermeture d'un périphérique, ainsi qu'à l'exécution d'une Entrée/Sortie (IO). Une telle routine est nécessaire afin qu'une fonction comme *OpenDevice()* puisse ouvrir chaque périphérique, même si la prise en charge de l'activité est différente de périphérique à périphérique.

Les fonctions importantes utilisables pour chaque périphérique sont :

Offset	Fonction
-36	Abort IO
-30	Begin IO
-12	Close
-6	Open

Nous allons voir maintenant avec une routine assembleur, ce qui se passe lorsqu'on ouvre un périphérique.

La routine ici présente est une partie importante de la fonction *OpenDevice()*, qui n'a pas été appelée directement de la librairie *Exec*, mais d'une routine de la *Ram Library*.

On accèdera à la routine avec :

d0 = Unit
d1 = Flags
a0 = Pointeur sur le nom du périphérique (DeviceName)
a1 = Pointeur sur IORequest
a6 = Pointeur sur ExecBase

```
fc0666 move.l  A2,-(A7)       A2 sauvegardé
fc0668 move.l  A1,A2          Pointeur IORequest dans A2
fc066a clr.b   31(A1)         Flag Error éliminé
fc066e movem.l D1-D0,-(A7)    D0 et D1 sauvegardés
fc0672 move.l  A0,A1          Pointeur sur le nom dans A1
fc0674 lea     350(A6),A0     Pointeur sur DeviceList dans A0
fc0678 addq.b  #1,295(A6)     Forbid
fc067c bsr.l   $fc165a        Chercher nom dans DeviceList
                              (FindName())
fc0680 move.l  D0,A0          Pointeur sur Device dans A0
fc0682 movem.l (A7)+,D1-D0    Reprise de d0 et d1
fc0686 move.l  A0,20(A2)      Transmettre pointeur sur le périphérique
                              dans IORequest
fc068a beq.s   $fc06ac        Erreur, Pas de périphérique
fc068c clr.l   24(A2)         Pointeur sur Unit effacé
fc0690 move.l  A2,A1          Pointeur sur IORequest dans A1
fc0692 move.l  A6,-(A7)       A6 sauvegardé
fc0694 move.l  A0,A6          Pointeur sur Device dans A6
fc0696 jsr     -6(A6)         Saut à OpenDevice
fc069a move.l  (A7)+,A6       Reprise de A6
fc069c move.b  31(A2),D0      Flag d'erreur dans d0
fc06a0 ext.w   D0             Signe d'erreur
fc06a2 ext.l   D0             Signe d'erreur
fc06a4 jsr     -138(A6)       Permit()
fc06a8 move.l  (A7)+,A2       Reprise de a2
fc06aa rts                    Saut de retour
```

Périphérique absent (erreur) :

```
fc06ac moveq   #$ff,D0        Valeur erreur dans d0
fc06ae move.b  D0,31(A2)      Ecriture dans le flag d'erreur
fc06b2 bra.s   $fc06a4        Saut immédiat
```


La routine [illegible]nte n'est que la partie importante de la fonction *CloseDevice()*.

On accède à cette routine avec :

A1 = pointeur sur *IORequest*
A6 = pointeur sur *ExecBase*

```
fc06b4 addq.b  #1,295(A6)     Forbid
fc06b8 move.l  A6,-(A7)       A6 sauvegardé
fc06ba move.l  20(A1),A6      Pointeur sur Device dans A6
fc06be jsr     -12(A6)        Saut à CloseDevice
fc06c2 move.l  (A7)+,A6       Reprise de A6
fc06c4 jsr     -138(A6)       Permit
fc06c8 rts                    Saut de retour
```

Comme exemple pour la routine *OpenDevice*, qui peut être appelée par l'Offset -6, on va prendre comme exemple la routine du périphérique lecteur de disquette.

Le nombre *Unit* indique le numéro de lecteur de disquette pour le périphérique *TrackDisk*. Pour chacun des quatre lecteurs possibles, il existe un pointeur réservé dans la structure *Device* qui, aussitôt que le lecteur est initialisé, pointe sur un port *Message* correspondant.

Ce port, comme le périphérique correspondant, a de plus aux entrées standard, qui sont déterminées par une structure C, quelques entrées non standard, comme par exemple, un compteur pour le nombre des accès à la structure *Message-Port*. Cette routine sera appelée avec :

D0 = Nombre Unit
D1 = Flags
A1 = Pointeur sur IORequest
A6 = Pointeur sur périphérique

```
fe9f42 movem.l A4/A2/D2,-(A7)    D2, A2, A4 sauvegardés
fe9f46 move.l  A1,A4             Pointeur sur IORequest dans A4
fe9f48 move.l  D0,D2             Numéro unité dans D2
fe9f4a cmpi.l  #$00000004,D0     Test ?
fe9f50 bcs.s   $fe9f56           Continue si test ok
fe9f52 moveq   #$20,D0           Sinon erreur dans D0
fe9f54 bra.s   $fe9f82           Erreur affichée, Fin
fe9f56 lsl.w   #2,D0             Numéro 4 comme Offset
fe9f58 lea     36(A6),A2         Pointeur sur port lecteur
fe9f5c adda.l  D0,A2             Offset additionné
fe9f5e move.l  (A2),A0           Port lecteur dans A0
fe9f60 move.l  A0,D0             Lecteur occupé ?
fe9f62 bne.s   $fe9f70           Non -> OK
fe9f64 bsr.l   $fe9d3e           Sinon déterminer port
fe9f68 tst.l   D0                Port trouvé ?
fe9f6a bne.l   $fe9f82           Continue si non
fe9f6e move.l  A0,(A2)           Port transmis au périphérique
fe9f70 move.l  A0,24(A4)         Port transmis à IORequest
fe9f74 addq.w  #1,32(A6)         Nbre d'accès du périphérique augmenté
fe9f78 addq.w  #1,36(A0)         Nbre d'accès au port lecteur augmenté
fe9f7c movem.l (A7)+,A4/A2/D2    Reprise de D2, A2, A4
fe9f80 rts                       Saut de retour
```

Erreur d'affectation de port lecteur :

```
fe9f82 move.b  D0,31(A4)         Numéro erreur dans Error-Flag
fe9f86 moveq   #$ff,D0           Pointeur sur erreur dans D0
fe9f88 move.l  D0,24(A4)         Pointeur sur Unit effacé
fe9f8c move.l  D0,20(A4)         Pointeur sur périphérique effacé
fe9f90 bra.s   $fe9f7c           Saut immédiat
```

La fonction *CloseDevice()* saute avec l'Offset -12 du périphérique dans une routine propre. Cette routine pour le périphérique *Trackdisk* possède une forme particulière qui est la suivante :

```
fe9f92 movem.l A3-A2,-(A7)       Sauvegarde A2 et A3
fe9f96 move.l  A1,A2             Pointeur sur IORequest dans A2
fe9f98 move.l  24(A2),A3         Pointeur sur port lecteur dans A3
fe9f9c subq.w  #1,36(A3)         Nombre d'accès
fe9fa0 bne.s   $fe9fa8           Continue tant que lecteur utilisé
```



```
fe9fa2 bset    #3,64(A3)         Initialisation de Flag
fe9fa8 subq.w  #1,32(A6)         Nombre d'accès sur le périphérique
fe9fac moveq   #$ff,D0           Valeur d'effacement chargée
fe9fae move.l  D0,24(A2)         Pointeur sur port effacé
fe9fb2 move.l  D0,20(A2)         Pointeur sur périphérique effacé
fe9fb6 movem.l (A7)+,A3-A2       Reprise de A2 et A3
fe9fba moveq   #$00,D0           Retour D0 à 0
fe9fbc rts                       Saut de retour
```

2.6.3 DIRECTION DES ENTREES/SORTIES (IO)

D'APRES LES FONCTIONS

Pointe périphérique : il existe une tâche qui permet de délivrer les instructions.

Ce sont les fonctions suivantes qui permettent cette transmission d'instructions :

DoIO

```
Erreur = DoIO(IORequest);
 D0                A1
```

Offset : -456

Description : Cette fonction sera principalement utilisée pour la direction des entrées/sorties. Elle attend jusqu'à ce que l'instruction transmise soit terminée et retourne dans son programme propre. Pendant cette attente, la tâche sera mise en mode *Wait*.

SendIO

```
SendIO(IORequest);
         A1
```

Offset : -462

Description : Cette fonction sert à envoyer une entrée/sortie à un périphérique correspondant sans attendre que l'instruction transmise soit effectuée.

CheckIO

```
Terminé = CheckIO(IORequest);
  D0              A1
```

Offset : -468

Description : Cette fonction teste si un processus IO déterminé a été traité. Si c'est le cas, il sera retourné dans D0, le pointeur sur la structure correspondante *IORequest*. Si le processus IO n'est pas terminé, il sera retourné à la valeur 0.

WaitIO

```
WaitIO(IORequest);
         A1
```

Offset : -474

Description : Cette fonction attend aussi longtemps que le processus IO est traité.

Pendant ce temps, la tâche est mise dans le mode *Wait* afin que les autres tâches puissent continuer. Les fonctions *SendIO* et *WaitIO* correspondent, ensemble à l'instruction *DoIO*.

AbortIO

```
AbortIO(IORequest);
          A1
```

Offset : -480

Description : Cette fonction termine un processus IO.

Après cette courte description des fonctions, nous allons voir comment ces dernières s'intègrent au système d'exploitation.

La routine assembleur *DoIO* est de la forme suivante :

On trouve dans A1 un pointeur sur la structure *IORequest* qui a été précédemment atteinte.

```
fc06dc move.l  A1,-(A7)      A1 sauvegardé
fc06de move.b  #$01,30(A1)   Quick-Bit activé
fc06e4 move.l  A6,-(A7)      A6 sauvegardé
fc06e6 move.l  20(A1),A6     Pointeur sur Device
fc06ea jsr     -30(A6)       Sauter à IO Execute
fc06ee move.l  (A7)+,A6      Prendre A6
fc06f0 move.l  (A7)+,A1      Prendre A1

Ici commence la fonction 'WaitIO' qui sera utilisée par la fonction
'DoIO'.

fc06f2 btst    #0,30(A1)     Test Quick-Bit
fc06f8 bne.s   $fc0744       Terminé, si activé
fc06fa move.l  A2,-(A7)      A2 sauvegardé
fc06fc move.l  A1,A2         Pointeur sur IORequest dans A2
```

```
fc06fe move.l  14(A2),A0        Pointeur     Reply-Port
fc0702 move.b  15(A0),D1        Bit Signal activé
fc0706 moveq   #$00,D0          D0 effacé
fc0708 bset    D1,D0            Bit pour Signal activé
fc070a move.w  #$4000,$dff09a   Disable-
fc0712 addq.b  #1,294(A6)       Macro
fc0716 cmpi.b  #$07,8(A2)       Type de Msg = Reply MSG ?
fc071c beq.s   $fc0724          Si Type OK -> continue
fc071e jsr     -318(A6)         Sinon attendre Msg ( Wait() )
fc0722 bra.s   $fc0716          Saut indéterminé
fc0724 move.l  A2,A1            IORequest dans A1
fc0726 move.l  (A1),A0
fc0728 move.l  4(A1),A1         Noeud de la liste Reply-Msg
fc072c move.l  A0,(A1)          écarté
fc072e move.l  A1,4(A0)
fc0732 subq.b  #1,294(A6)       Enable-
fc0736 bge.s   $fc0740
fc0738 move.w  #$c000,$dff09a   Macro
fc0740 move.l  A2,A1            Pointeur sur IORequest dans A1
fc0742 move.l  (A7)+,A2         Mise en place de A2
fc0744 move.b  31(A1),D0        Flag erreur dans D0
fc0748 ext.w   D0               Signe
fc074a ext.l   D0               Signe
fc074c rts                      Saut de retour
```

Dans la routine ci-dessus, on initialise en premier lieu, le bit *Quick*. Puis le pointeur sur le périphérique sera mis dans A6 afin de pouvoir sauter à la fonction *BeginIO*, qui sera décrite plus loin. Dans cette routine, la validité de l'instruction d'exécution sera testée et dans le cas positif, délivrée à la tâche *TrackDisk*. Si le programme retourne à cette routine, le type de cette structure *IORequest* sera toujours *Message*. La tâche sera prise comme *Message* dans la routine de la structure *IORequest*.

A cet endroit, la remise des instructions est interrompue. Ceci se traduit par un état d'attente. Si le bit *Quick* n'est pas effacé, la routine sera terminée. Il faudra tester si le processus IO se termine. Ici, il suffit de comparer le type de la structure *Message* à *Reply-Msg*. Si ce n'est pas le cas, la tâche entre dans une période d'attente jusqu'à ce que le message correspondant soit arrivé.

Il est important de détailler pourquoi la structure *Message* de la structure *IORequest* est comparée au type *Reply-Msg*.

De la routine *BeginIO*, un *Message* sera envoyé à la tâche correspondante, qui prend en charge le traitement des instructions.

Le *Message* correspond ici à notre structure *IORequest*. Avec la fonction *PutMsg()*, le type du message envoyé passera automatiquement à *Message* (valeur d'octet 0,5). Notre structure *IORequest* a toujours la même position en mémoire mais elle n'est plus considérée comme structure *noeud* de la liste *Message* de la tâche. La tâche traite alors nos instructions et envoie un *Message-Reply* en retour, afin qu'il puisse terminer son travail. Ce message sera à nouveau la structure *IORequest*. Par la fonction *ReplyMsg()*, le type de message envoyé passera de nouveau automatiquement à *ReplyMsg* (valeur d'octet 0,7) et à nouveau inséré dans la liste *Message* du *ReplyPort*. La fonction *WaitIO()* teste le type de la structure *Message* dans notre structure *IORequestet* et détermine qu'il s'agit bien d'un *Reply-Message*. Ce dernier, inséré dans la liste *ReplyMsg* sera à nouveau éliminé.

Description de la fonction SendIO

Dans A1 on trouve le pointeur sur la structure *IORequest*.

```
fc06ca clr.b   30(A1)           Effacer tous les flags
fc06ce move.l  A6,-(A7)         A6 sauvegardé
fc06d0 move.l  20(A1),A6        Pointeur sur Device
fc06d4 jsr     -30(A6)          Saut à BeginIO
fc06d8 move.l  (A7)+,A6         Reprise de A6
fc06da rts                      Saut de retour
```

Vous pouvez voir que la fonction *SendIO* n'est rien d'autre que la première partie de la fonction *DoIO*.

Description de la fonction **CheckIO**

Dans A1 on trouve le pointeur sur la structure *IORequest*.

```
fc074e btst    #0,30(A1)      Test si le Bit-Quick est activé
fc0754 beq.s   $fc075a        Boucle, sinon
fc0756 move.l  A1,D0          Signal OK délivré
fc0758 rts                    Saut de retour
fc075a cmpi.b  #$07,8(A1)     Test : Type de la structure Message
                              = Replymsg ?
fc0760 beq.s   $fc0766        Oui -> signal retour positif
fc0762 moveq   #$00,D0        Non -> signal négatif
fc0764 rts                    Saut de retour
fc0766 move.l  A1,D0          Ok transmis
fc0768 rts                    Saut de retour
```

La fonction *CheckIO* teste exclusivement s'il s'agit d'un *Message Reply*. Si c'est le cas, un pointeur sur la structure *IORequest* sera délivré dans D0, sinon, une valeur 0 sera retournée dans le même registre.

Vous pouvez voir dans cette routine qu'il est testé s'il est bien transmis un *Message Reply*, ce dernier n'étant pas écarté de la liste.

L'élimination de ce dernier sera exécutée avec l'utilisation de la fonction *CheckIO*. On pourra prendre, par exemple, la fonction *GetMsg()* tant que le bit *Quick* ne sera pas activé.

Description de la fonction **AbortIO**

Dans A1 on trouve un pointeur sur la structure *IORequest*.

```
fc076a move.l  A6,-(A7)       A6 sauvegardé
fc076c move.l  20(A1),A6      Pointeur sur Device dans A6
fc0770 jsr     -36(A6)        Saut à AbortIO
fc0774 move.l  (A7)+,A6       Reprise de A6
fc0776 rts                    Saut de retour
```


Comme cela a déjà été dit, chaque périphérique dispose d'une petite table de saut pour sa gestion. Nous allons l'examiner de plus près, avec le périphérique *TrackDisk*. Elle sera appelée avec l'Offset -30 et utilisée par les fonctions *DoIO* et *SendIO*.

On trouve dans A1 et A6, respectivement un pointeur sur la structure *IORequest* et un pointeur sur le périphérique.

```
fe9fbe clr.b   31(A1)         Flag erreur effacé
fe9fc2 moveq   #$00,D0        D0 effacé
fe9fc4 move.b  29(A1),D0      io_Command dans D0
fe9fc8 cmpi.b  #$16,D0        Test : instruction autorisée ?
fe9fcc bcc.s   $fea016        Saut, si non
fe9fce move.l  24(A1),A0      Pointeur sur Device-Port
fe9fd2 move.l  #$000c61c2,D1  Bits d'instruction
fe9fd8 btst    D0,D1          Instruction exécutée ?
fe9fda bne.s   $fe9ff0        Si oui -> exécution
fe9fdc andi.b  #$7e,30(A1)    Flags effacés
fe9fe2 move.l  A6,-(A7)       A6 sauvegardé
fe9fe4 move.l  52(A6),A6      ExecBase activé
fe9fe8 jsr     -366(A6)       IORequest délivré à la tâche Track
fe9fec move.l  (A7)+,A6       A6 activé
fe9fee bra.s   $fea014        Saut immédiat

fe9ff0 bset    #7,30(A1)      Mise en place des flags d'exécution
fe9ff6 move.b  #$05,8(A1)     Passage IORequest à Message pour
                              activer WaitIO
fe9ffc movem.l A3-A2,-(A7)    A2 et A3 sauvegardés
fea000 move.l  A0,A3          Pointeur sur Port-Drive dans A3
fea002 move.l  A1,A2          Pointeur sur IORequest dans A2
fea004 lea     762(PC)(=$fea300),A0   Pointeur sur la table
fea008 lsl.w   #2,D0          Instruction *4
fea00a move.l  0(A0,D0.W),A0  Calcul de saut
fea00e jsr     (A0)           Saut
fea010 movem.l (A7)+,A3-A2    Reprise de A2 et A3
fea014 rts                    Saut de retour
```

2.7 Manipulations des interruptior ur l'Amiga

Dans ce chapitre, nous allons voir comment l'Amiga utilise ses sept niveaux d'interruption et surtout comment nous allons pouvoir les employer. Pour une meilleure compréhension de ce chapitre, il est préférable d'avoir déjà lu la partie concernant les interruptions au chapitre 1 de ce volume.

L'interruption sera, ici, considérée après que le processeur ait eu le signal correspondant des 4 703 interruptions logiques. Dès que le processeur a reçu un signal d'interruption et surtout s'il n'est pas saturé, l'adresse de l'interruption sera chargée par le pointeur programme suivant le niveau de priorité et inséré à la place correspondante.

Les vecteurs de la suite d'adresses des interruptions commencent à l'adresse $0064 jusqu'à l'adresse $00, pour se poursuivre jusqu'à l'adresse $007F. Les 4 premiers octets du vecteur concernent l'interruption de niveau 1, alors que les octets de $007C à $007F du vecteur, concernent l'interruption de niveau 7.

Toutes les interruptions sont accessibles sur un registre qui permet de gérer 15 interruptions différentes. L'autorisation de ces dernières sera testée à l'aide du registre *Interrupt-Enable*, ceci permettra d'autoriser ou non le passage d'une interruption vers le processeur. Etant donné que pour les 15 interruptions, seules 7 priorités sont à disposition, plusieurs interruptions possèdent le même ordre de priorité. En raison des priorités identiques, certaines interruptions différentes auront le même comportement. Le tableau suivant montre la correspondance entre les interruptions et leur niveau de priorité. La pseudo-priorité permet de gérer des interruptions possédant le même niveau de priorité. Les chiffres correspondent, en fait, au numéro de bit des interruptions dans le registre *Interrupt-Request*.

Pour facilite compréhension, voici un exemple :

L'interruption qui sera libérée, lors du passage à la ligne zéro de l'écran du faisceau du Raster, possède la même priorité processeur que l'interruption signifiant la fin de l'activité du Blitter. Ces deux interruptions sont portées par le bit 5 et 6 du registre *Interrupt-Request*. Au niveau du SoftWare, l'interruption portant le numéro de bit le plus haut sera traitée en premier. Dans ce cas, l'interruption portant sur le Blitter sera exécutée avant celle portant sur le Raster. Le tableau suivant montre les 15 interruptions de l'Amiga avec les pseudo-priorités correspondantes.

Pseudo-priorité	Nom	Priorité-processeur	Fonction
14	INTEN	6	Autorisation d'interruption
13	EXTER	6	Interruption du CIA-B ou du port extension
12	DSKSYN	5	Valeur de synchronisation disquette
11	RBF	5	Tampon d'entrée du port-série plein
10	AUD3	4	Canal audio 3 occupé
9	AUD2	4	Canal audio 2 occupé
8	AUD1	4	Canal audio 1 occupé
7	AUD0	4	Canal audio 0 occupé
6	BLIT	3	Accès blitter terminé
5	VERTB	3	Début du temps mort vertical
4	COPPER	3	Réservé pour interruptions Copper
3	PORTS	2	Interruption du CIA-A ou du port d'extension
2	SOFT	1	Réservé pour les interruptions du Software
1	DSKBLK	1	Transfert disquette DMA terminé
0	TBE	1	Tampon de sortie du port-série vide

2.7.1 ARCHITECTURE D'UNE STRUCT E INTERRUPT

La gestion des interruptions de l'Amiga se fait, ce qui n'est plus une surprise, à nouveau par une structure. Cette structure est de la forme suivante :

```
      struct Interrupt {
0       struct Node is_Node;
14      APTR is_Data;
18      VOID (*is_Code)();
      };
```

is_ Node

Ceci est une structure *Noeud*.

is_ Data

Ceci est un pointeur sur les données en mémoire qui seront utilisées par n'importe laquelle de ces interruptions. La taille de ces données pourra être définie à la mise en place de la structure.

is_(Code) ()*

Pointeur sur le programme interruption qui sera exécuté.

Il existe deux possibilités différentes d'employer une interruption ; avec la première, on peut accéder avec une interruption, à un programme qui n'est autre qu'un convertisseur d'interruption. On peut aussi, avec une interruption, appeler différents programmes qui sont gérés par un serveur d'interruption.

Chaque interruption autorisée est inscrite dans la structure *ExecBase*. Dans cette structure, on trouve pour chacune des quinze interruptions possibles, une petite sous-structure qui joue un rôle important lors de l'initialisation de l'interruption.

Ces sous uctures se nomment *IntVector* et sont de la forme suivante :

```
      struct IntVector {
0       APTR iv_Data;
4       VOID (*iv_Code)();
8       struct Node *iv_Node;
      };
```

L'initialisation de cette structure *IntVector* se différencie suivant la gestion de l'interruption correspondante, par un convertisseur ou serveur d'interruption.

Dans le cas du convertisseur d'interruption, cette initialisation est de la forme suivante :

iv_Data

Pointeur sur les données de la structure *Interrupt*.

*(*iv_Code)()*

Pointeur sur le programme d'interruption qui doit être exécuté.

**iv_Node*

Pointeur sur la structure *Interrupt* qui a été décrite précédemment.

L'initialisation avec le serveur d'interruption est de la forme suivante :

iv_Data

Pointeur sur la structure *ServerList* qui sera détaillée plus loin.

*(*iv_Code)()*

Pointeur sur une routine qui prend en charge la gestion de plusieurs programmes d'interruption.

**iv_Node*

Il est, ici, inutilisé et a la valeur zéro.

Ces structures *Interrupt-Vector* n'ont pas besoin d'être initialisées tant que l'on voudra utiliser le convertisseur d'interruptions. L'initialisation débute à l'appel de la fonction *Exec SetIntVector*. Pour utiliser une interruption avec un convertisseur d'interruptions, la structure *Interrupt* doit être initialisée par l'appel de la fonction *SetIntVector*.

Nous allons voir maintenant comment une interruption est traitée sans être passée par le processeur.

Comme exemple de gestion d'une interruption, voici un programme en assembleur qui traite une interruption de niveau 3 (priorité 3).

$FC0CD8 correspond à l'adresse de la sous-routine où le processeur continue son travail, après avoir reconnu une interruption de niveau 3. Cette valeur d'adresse n'est à prendre en compte qu'avec la version du Kickstart des Amiga 500 et 2000.

Pour l'Amiga 1000, la routine est la même, mais dans cette version du Kickstart, elle a été légèrement décalée.

```
movem.l A6-A5/A1-A0/D1-D0,-(A7)   Registre sauvegardé
lea     $dff000,A0                Début du registre dans A0
move.l  $0004,A6                  SysBase dans A6
move.w  28(A0),D1                 Lecture registre Interrupt-Enable
btst    #14,D1                    Bit Master testé
beq.l   L1                        Pas d'interruption autorisée
and.w   30(A0),D1                 Filtrage des interruptions autorisées
                                   avec le registre Interrupt-Request
btst    #6,D1                     Test Bit pour Blitter done
beq.s   L2                  Si non -> autre bit testé
```



```
movem.  56(A6),A5/A1      Pointeurs sur les données et le
                          programme issus de la structure IntVector
pea     -36(A6)           Adresse de retour mise en place sur ExitInter
jmp     (A5)              Branchement à une interruption

L2:

btst    #5,D1             Test du Bit Raster
beq.s   L3                Non -> test continue
movem.l 144(A6),A5/A1     Pointeurs sur les données et le programme
                          issus de la structure IntVector
pea     -36(A6)           Saut de retour à ExitInter
jmp     (A5)              Embranchement

L3:

btst    #4,D1             Test du Bit Copper
beq.s   L1                Non -> fin
movem.l 132(A6),A5/A1     Pointeurs sur les données et le programme
                          issus de la structure IntVector
pea     -36(A6)           Saut de retour à ExitInter
jmp     (A5)              Embranchement

L1:

movem.l (A7)+,A6-A5/A1-A0/D1-D0   Registres remis en place,
rte                               Saut de retour
```

Au moyen du listage assembleur, nous pouvons déterminer les registres à utiliser avec prudence dans un programme d'interruption personnel ainsi que ceux qui doivent être fixés avec une valeur connue.

Les registres D0, D1, A0, A1, A5 et A6 seront sauvegardés dans la pile avant l'embranchement sur le programme d'interruption. De plus, il y sera marqué l'adresse de retour, afin qu'un programme d'interruption personnel puisse être arrêté par une instruction RTS.

Description des registres :

D0 : ne contient aucune information importante.

D1 : contient le ET logique entre *IntEnaReg* et *IntReqReg*, indiquant quelle interruption est autorisée ou non.

A1 : pointeur sur le début du registre Hardware.

A5 : pointeur sur le code à exécuter.

A6 : pointeur sur *SysBase*.

Aucun de ces registres ne doit reprendre sa valeur précédente avant le retour du programme d'interruption.

Lors de l'utilisation d'un convertisseur d'interruptions, l'embranchement à un programme d'interruption propre se fera avec l'instruction : JMP(A5). Le programme d'interruption doit être terminé avec l'instruction RTS.

Lors de la gestion d'une interruption par un serveur d'interruption, chaque programme d'interruption qui apparaît lors de l'exécution des interruptions correspondantes, est relié pour former une liste avec une structure *Interrupt*.

L'ordre de traitement correspond aux priorités présentes dans la structure *is_Node*.

La structure *Liste* contenant les interruptions est de la forme suivante :

```
    struct ServerList {
0     struct List sl_List;
14    UWORD  sl_IntClr1;
16    UWORD  sl_IntSet;
18    UWORD  sl_IntClr2;
20    UWORD  sl_pad;
    );
```


sl_List

Ceci est une structure liste.

sl_IntClr1 et sl_IntClr2

Mots dans lesquels les bits sont activés suivant l'état de l'interruption se trouvant dans le registre *Interrupt-Request*. Le bit 15 Clr/Set est éliminé ici (son rôle est détaillé au chapitre 1 de ce volume). Si on écrit cette valeur dans le registre *Interrupt-Request*, le bit d'interruption sera éliminé.

sl_IntSet

Mot ayant le même contenu que *sl_IntClr*. La différence réside dans le fait qu'ici, le bit Clr/Set est activé.

sl_pad

Mot qui possède toujours la valeur zéro.

Cette structure ne se retrouve dans aucun fichier *Include* et doit être écrite par le programme.

Etant donné que nous allons maintenant utiliser un serveur d'interruption, l'initialisation de la structure *InterruptVector* se fera de telle manière qu'on trouvera un pointeur sur une routine gérant une liste *Interrupt (*iv_Code)()*.

Cette routine, qui se tient à l'adresse $FC12FC, sera accessible avec l'instruction JMP(A5). Elle est de la forme suivante :

Dans A1, on trouve le pointeur sur la structure *ServerList*.

```
fc12fc move.w  18(A1),-(A7)   Lecture et mi:  n mémoire de sl_IntClr
fc1300 move.l  A2,-(A7)       A2 sauvegardé
fc1302 move.l  (A1),A2        Pointeur sur le 1er Interrupt
fc1304 move.l  (A2),D0        Contrôle autorisé ?
fc1306 beq.s   $fc1316        Si non, saut
fc1308 movem.l 14(A2),A5/A1   a1 = is_Data et
                              A5 = (*is_Code)()
fc130e jsr     (A5)           Accès au programme d'interruption
fc1310 bne.s   $fc1316        Saut si message de retour = 0
fc1312 move.l  (A2),A2        Pointeur sur le prochain emplacement
                              d'interruption
fc1314 bra.s   $fc1304        Saut immédiat
fc1316 move.l  (A7)+,A2       Reprise de A2 précédent
fc1318 move.w  (A7)+,$dff09c  Bit d'interruption effacé dans le registre
                              Interrupt-Request
fc131e rts                    Saut de retour
```

Lorsqu'une interruption est traitée, la suivante est appelée immédiatement tant qu'il y en aura en attente et dès que la précédente aura délivrée un zéro comme message de retour dans D0.

Comme pour le convertisseur d'interruptions, on utilise, lors de l'exécution, les mêmes registres. Leurs attributions sont pourtant différentes.

Description des registres :

D0 : pointeur sur la prochaine interruption

D1 : valeur inutilisée

A1 : pointeur sur la mémoire des données de l'interruption

A5 : pointeur sur le programme d'interruption propre

A6 : valeur inutilisée

Pour mieu :omprendre la gestion des interruptions par un serveur, son utilisation est détaillée dans la partie concernant les interruptions de port du CIA-A.

La structure *Interrupt-Vector* se trouve, pour ces interruptions, à l'Offset 120 de la structure *ExecBase*. Cette structure est initialisée de la façon suivante :

iv_Data

Pointeur sur la structure *Server-List*.

*(*iv_Code)()*

Pointeur sur un sous-programme géré par la *Server-List*. Cette routine se trouve à l'adresse $FC12FC sur Amiga 500 et 2000.

**iv_Node*

Ceci n'est pas initialisé, étant donné que les structures *Interrupt* sont reliées dans la structure *ServerList*.

La structure *ServerList* initialisée se présente comme ceci :

sl_List

Celle-ci est initialisée comme chaque structure *Liste*. Dans cette liste, les structures *Interrupt* sont reliées entre elles.

sl_IntClr1 et sl_IntClr2

Ces deux champs ont la valeur $0008, le bit 3 du registre *Interrupt-Request* étant positionné pour ces interruptions.

sl_IntSet

Ce champ a la valeur $8008.

2.7.2 INTERRUPTIONS-SOFT

Comme le titre peut le laisser entrevoir, ce chapitre va porter sur les interruptions libérées par le SoftWare. Ces interruptions ont une priorité supérieure à celle des tâches, mais inférieure aux interruptions du Hardware. Elles pourront être utilisées dans n'importe quel processus asynchrone.

Pour appeler une telle interruption, on se sert de la fonction *Cause()*. La tâche en cours de traitement sera interrompue, afin que l'interruption puisse être exécutée. Si la fonction *Cause()* est appelée par une interruption du Hardware, cette dernière sera d'abord exécutée avant que la *Soft-Interrupt* soit libérée.

Pour générer une *Soft-Interrupt*, il faut d'abord initialiser sa structure *Interrupt*. Puis la fonction *Cause()* pourra être appelée avec le pointeur sur la structure placée dans A1.

Il ne peut être attribué à une *Soft-Interrupt* que 5 priorités différentes qui sont : -32, -16, 0, +16, +32. Ceci vient du fait que la structure *ExecBase*, lors de l'exécution, installe une structure *SoftIntList* pour chaque priorité et qu'il n'est prévu que 5 places pour les structures. Chacune de ces structures est de la forme suivante :

```
      struct SoftIntList (
0       struct List sh_List;
14      UWORD sh_Pad;
      );
```

sh_List

Ceci est une structure *Liste*.

sh_Pad

C'est un mot qui ne sera utilisé que pour contenir un nombre entier de mots longs dans la structure. Il a toujours la valeur zéro.

Cinq de ces structures font partie de la structure *ExecBase*. Elles se trouvent les unes à la suite des autres et commencent à l'offset 434.

La gestion de ces interruptions par *Exec* est détaillée dans le programme assembleur suivant.

Seule la documentation sur la routine *Cause()* manque :

```
fc1320 move.w  #$4000,$dff09a   Bloquer toutes les interruptions
fc1328 addq.b  #1,294(A6)       IdNestCount incrémenté
fc132c cmpi.b  #$0b,8(A1)       Test = type SoftInt
fc1332 beq.s   $fc1370          Si oui -> saut
fc1334 move.b  #$0b,8(A1)       Si non -> nouvelle
fc133a moveq   #$00,D0          Inscrire
fc133c move.b  9(A1),D0         Priorité dans D0
fc1340 andi.w  #$00f0,D0        Bits non autorisés, effacés
fc1344 ext.w   D0               Continue
fc1346 lea     466(A6),A0       Pointeur sur Int. avec Pri. 0
fc134a adda.w  D0,A0            Position de SoftIntList suivant la
                                priorité déterminée
fc134c lea     4(A0),A0         Pointeur sur lh_Tail
fc1350 move.l  4(A0),D0         Pointeur sur le dernier élément dans D0
fc1354 move.l  A1,4(A0)         Nouvelle interruption inscrite en
                                dernier
fc1358 move.l  A0,(A1)          Int. suivante = 0
fc135a move.l  D0,4(A1)         Pointeur sur précédente
fc135e move.l  D0,A0            Pointeur sur suivante
fc1360 move.l  A1,(A0)          Chaque SoftInt
fc1362 bset    #5,292(A6)       autorisée dans SysFlag
fc1368 move.w  #$8004,$dff09c   occasionne l'interruption
fc1370 subq.b  #1,294(A6)       IdNestCount décrémenté
fc1374 bge.s   $fc137e          Saut si Int. n'est pas encore autorisée
fc1376 move.w  #$c000,$dff09a   Interruption autorisée
fc137e rts                      Saut de retour
```

Ce listing assembleur montre comment *SoftInterrupt* peut être générée. Avant que l'instruction puisse être exécutée, elle doit tout d'abord être placée dans une des cinq structures *SoftIntList*.

La structure dans laquelle elle sera insérée dépend de son niveau de priorité. La manière dont se déroule l'insertion d'une structure permet de comprendre pourquoi seules les priorités -32, -16, 0, +16 , +32 sont autorisées. Chaque structure *SoftIntList* a une longueur de 16 octets. Il sera généré un pointeur sur la structure dont la priorité sera additionnée afin d'obtenir la position souhaitée.

Après avoir déterminé la structure *SoftIntList*, la structure de l'interruption générée sera placée comme dernier élément de la liste et il sera indiqué dans le champ *SysFlags* de la structure *ExecBase* qu'une *SoftInterrupt* est présente. Ensuite, le bit d'exécution d'une *SoftInterrupt* qui se trouve dans le registre *InterruptRequest* sera activé. Ceci sera aussi fait dans la structure *ExecBase* afin de caractériser l'autorisation. Puis les interruptions qui étaient bloquées au début de la routine seront à nouveau actives.

La *SoftInterrupt* sera alors exécutée. Le pointeur du programme correspondant sera issu de la structure *InterruptVector* afin de gérer les structures *SoftIntList*. Le programme débute à l'adresse $FC1380.

```
fc1380 move.w   #$0004,$dff09c  Bit Interrupt-Request effacé
fc1388 bclr     #5,292(A6)      Bit SysFlag effacé
fc138e bne.s    $fc1392         Saut lorsqu'une interruption est
                                autorisée
fc1390 rts                      Retour au programme appelant
fc1392 move.w   #$0004,$dff09a  Bit Interrupt-Enable effacé
fc139a bra.s    $fc13c2         Saut immédiat
fc139c move     #$2700,SR       Toutes les interruptions autorisées
fc13a0 move.l   (A0),A1         Pointeur sur la première
fc13a2 move.l   (A1),D0         Node valide ?
fc13a4 beq.s    $fc13ae         Saut si plus d'interruption
fc13a6 move.l   D0,(A0)         Première interruption effacée de la liste
fc13a8 exg      D0,A1           Permutation entre A1 et D1
fc13aa move.l   A0,4(A1)        In_Pred mis sur la Liste
fc13ae move.l   D0,A1           Pointeur sur la première interruption
fc13b0 move.b   #$02,8(A1)      In_Type mis sur Int.
```



```
fc13b6 ...      #$2000,SR       Bloquer toutes les interruptions
fc13ba movem.l  14(A1),A5/A1    A1 = is_Data, A5 = (*is_Code)()
fc13c0 jsr      (A5)            Branchement
fc13c2 moveq    #$04,D0         Nombre d'Int./Listes -1
fc13c4 lea      498(A6),A0      Pointeur sur Int. Liste avec la priorité
                                supérieure
fc13c8 move.w   #$0004,$dff09c  Bit Int. Request effacé
fc13d0 cmpa.l   8(A0),A0        Liste vide ?
fc13d4 bne.s    $fc139c         Continue si non
fc13d6 lea      -16(A0),A0      Si oui, pointeur sur Int./Liste avec
                                priorité inférieure
fc13da dbf      D0,$fc13d0      Continue si toutes les listes n'ont pas
                                été trouvées
fc13de move     #$2100,SR       Toutes les interruptions bloquées -> Pri.1
fc13e2 move.w   #$8004,$dff09a  Soft-Interrupt à nouveau autorisée
fc13ea rts                      Saut de retour
```

En premier lieu, les routines de même niveau seront testées afin de voir si le bit initialisé indique l'autorisation d'une *SoftInterrupt*, ceci de la même façon que la fonction *Cause()*. Si c'est le cas, la routine sera exécutée. En dernier lieu, le bit *InterruptRequest* sera effacé et les listes *SoftInterrupt* seront recherchées dans leurs structures *Interrupt*. Si une liste est traitée ou vide, c'est la prochaine liste qui sera recherchée. Les listes seront traitées suivant l'ordre décroissant des priorités.

2.7.3 LES INTERRUPTIONS DU CIA

Après avoir détaillé la gestion des interruptions de l'Amiga, nous allons voir les interruptions libérées par les CIA. Les priorités processeur des deux circuits intégrés : au nombre de 6 pour le CIA-B et de 2 pour le CIA-A. Ces deux circuits seront gérés par un serveur d'interruptions. L'élément *iv_Code* de la structure *InterruptVector* pointe sur une structure *ServerList* alors que *(*iv_Code)()* pointe sur la routine de gestion de la liste *Server*. Les structures *InterruptVector* des interruptions se tiennent pour le CIA-A à partir de l'offset 120 et pour le CIA-B à partir de l'offset 240 dans la structure *ExecBase*.

2.7.3.1 La structure CIA-Resource

En temps normal, on ne trouve qu'une structure *Interrupt* dans la liste *Server* de l'interruption. Dans ce cas, cette structure *Interrupt* fait partie intégrante de la structure *CIA-Resource*. *is_Data* pointe à nouveau sur la structure *Resource* alors que *(*is_Code)()* pointe sur une routine qui permet de gérer la structure *Resource*.

Par cette routine, toutes les interruptions du CIA pourront être gérées à l'aide de la structure :

* Interruption Timer-A.

* Interruption Timer-B

* Interruption Horloge temps réel - alarme

* Port série ou Interruption du clavier

* Interruption signaux Flag

La structure *CIA-Resource* peut être comparée à une structure *librairie* où il est toutefois possible de rajouter des extensions.

Cette structure est de la forme suivante :

```
Offset    Signification

0     struct Node lib_Node
          0     Pointeur sur la prochaine Resource
          4     Pointeur sur la Resource précédente
          8     Node-type
          9     Node-pri
          10    Pointeur sur le nom de la Resource
14    UBYTE lib_Flags                \* $00 *\
15    UBYTE lib_pad                  \* $00 *\
16    UWORD lib_NegSize              \* $0018 *\
18    UWORD lib_PosSize              \* $007C *\
20    UWORD lib_Version              \* $0000 *\
22    UWORD lib_Revision             \* $0000 *\
```



```
24    A     lib_IdString             \* $00000000 *\
28    ULONG lib_Sum                  \* $00000000 *\
32    UWORD lib_OpenCnt              \* $0000 *\
34    APTR  CiaStartPtr
38    WORD  IntRequestBit
40    BYTE  IntEnableCia
41    BYTE  IntRequestCia
42    struct Interrupt cia_Interrupt
          42    Pointeur sur la prochaine interruption
          46    Pointeur sur l'interruption précédente
          50    Node-type
          51    Node-pri
          52    Pointeur sur le nom de l'interruption
          56    Pointeur sur le tampon de données (ici Resource)
          60    Pointeur sur le code à exécuter
64    struct IntVector Timer A
          CIAA et CIAB
          64    non initialisé (00000000)
          68    non initialisé (00000000)
          72    non initialisé (00000000)
76    struct IntVector Timer B
          CIAA:
          76    Pointeur sur le tampon de données
          80    Embranchement sur $FE9726
          84    Pointeur sur la structure Interrupt

          CIAB:
          76    non initialisé (00000000)
          80    non initialisé (00000000)
          84    non initialisé (00000000)
88    struct IntVector TOD Alarm
          CIAA:
          88    non initialisé (00000000)
          92    non initialisé (00000000)
          96    non initialisé (00000000)

          CIAB:
          88    Pointeur sur Graphics.library
          92    Saut à $FC6D68
          96    Pointeur sur la structure Interrupt
```

```
100   struct IntVector Serial Data
        CIAA:
        100   Pointeur sur le périphérique clavier
        104   Saut à $FE571C (test clavier)
        108   Pointeur sur une structure Interrupt
        CIAB:
        100   non initialisé (00000000)
        104   non initialisé (00000000)
        108   non initialisé (00000000)
112   struct IntVector Flag Leitung
        CIAA:
        112   non initialisé (00000000)
        114   non initialisé (00000000)
        118   non initialisé (00000000)

        CIAB:
        112   Pointeur sur Disk.Resource
        114   Saut à $FC4AB0
        118   Pointeur sur une structure Interrupt
```

Voici des éclaircissements sur quelques points de la structure :

CiaStartPtr

Pointeur sur le début du registre CIA (CIAA = $BFE001 ; CIAB = $BFD000).

IntRequestBit

Bit du registre *Interrupt-Request* qui sera initialisé lorsqu'une interruption devra être libérée (CIAA = $0008 ; CIAB = $2000).

IntEnableCia

Octet indiquant les interruptions CIA autorisées ou non.

IntRequestC

Octet indiquant l'interruption CIA présente.

Les vecteurs de chaque interruption CIA sont marqués dans la structure *IntVector*. Les structures non initialisées ne seront pas utilisées par le système d'exploitation mais peuvent être utiles, employées dans un autre but.

2.7.3.2 La gestion de la structure Resource

Après avoir détaillé cette structure, nous allons voir les routines permettant de la gérer. Dans A1, on a mis en mémoire le début de la structure *Resource*.

```
Embranchement du CIAB:

fc4610 movem.l A2/D2,-(A7)  D2 et A2 sauvegardés
fc4614 move.b  $bfdd00,D2   Registre Int.-Cont dans D2 (CIAB)
fc461a bra.s   $fc4626      Saut immédiat

Embranchement du CIAA:

fc461c movem.l A2/D2,-(A7)  D2 et A2 sauvegardés
fc4620 move.b  $bfed01,D2   Registre Int.-Cont dans D2 (CIAA)
fc4626 bclr    #7,D2        Bit supérieur effacé
fc462a or.b    41(A1),D2    Bits avec nouvelle liaison
fc462e move.b  D2,41(A1)    Octet Int.-Req écrit
fc4632 and.b   40(A1),D2    Lier octet Enable avec Req.
fc4636 beq.s   $fc4658      Saut si Int. non autorisé
fc4638 move.l  A1,A2        Pointeur sur Resource dans A2
fc463a eor.b   D2,41(A2)    Bit de l'interruption traitée effacé dans
                            registre Req.
fc463e lsr.b   #1,D2        Bit Timer A dans Carry
fc4640 bcs.s   $fc465e      Saut si Int. Timer A
fc4642 lsr.b   #1,D2        Bit Timer B dans Carry
fc4644 bcs.s   $fc4668      Saut si Int. Timer B
fc4646 beq.s   $fc4658      Saut si pas d'interruption
```

```
fc4648 lsr.b   #1,D2          Bit TOD-Alarm    s Carry
fc464a bcs.s   $fc4672        Saut si TOD-Alarm
fc464c beq.s   $fc4658        Saut si pas d'interruption
fc464e lsr.b   #1,D2          Bit données série dans Carry
fc4650 bcs.s   $fc467c        Saut si données série
fc4652 beq.s   $fc4658        Saut si pas d'interruption
fc4654 lsr.b   #1,D2          Bit pour interruption Flag dans Carry
fc4656 bcs.s   $fc4686        Saut si interruption Flag
fc4658 movem.l (A7)+,A2/D2    D2 et A2 remis en place
fc465c rts                    Retour au programme appelant
```

La routine insère chaque interruption dans l'octet *Interrupt-Request* de la structure *Resource*. Ce n'est qu'après qu'il testera si l'interruption est autorisée ou non et qu'il effacera, dans le cas d'une autorisation positive, le bit *Request* correspondant. Les bits des interruptions présentes seront systématiquement décalés et testés dans le registre *Carry* afin de savoir si elles sont autorisées ou non. Dans le cas positif, l'interruption correspondante sera exécutée. Les routines permettant l'exécution d'une interruption sont détaillées ci-dessous.

Dans A2, on trouve le pointeur sur la structure *Resource*.

```
Timer A Interrupt :

fc465e movem.l 64(A2),A5/A1  Saut et prise en compte du pointeur de données
fc4664 jsr     (A5)          Branchement
fc4666 bra.s   $fc4642       Retour de l'interruption

Timer B Interrupt :

fc4668 movem.l 76(A2),A5/A1  Saut et prise en compte du pointeur de données
fc466e jsr     (A5)          Branchement
fc4670 bra.s   $fc4648       Retour de l'interruption

TOD-Alarm Interrupt :

fc4672 movem.l 88(A2),A5/A1  Saut et prise en compte du pointeur de données
fc4678 jsr     (A5)          Branchement
fc467a bra.s   $fc464e       Retour de l'interruption
```



```
Seriell-Dat(   terrupt :

fc467c movem.l 100(A2),A5/A1 Saut et prise en compte du pointeur de données
fc4682 jsr     (A5)          Branchement
fc4684 bra.s   $fc4654       Retour de l'interruption

Flag Interrupt :

fc4686 movem.l 112(A2),A5/A1 Saut et prise en compte du pointeur de données
fc468c jsr     (A5)          Branchement
fc468e bra.s   $fc4658       Retour de l'interruption
```

Comme une librairie, la structure *Resource* dispose de différentes fonctions qui simplifient sa gestion. Ces fonctions sont accessibles avec des offsets négatifs par rapport à l'adresse de base. La structure *CIA-Resource* dispose de quatre fonctions :

Offset	Fonction
-6	SetInterrupt
-12	ClrInterrupt
-18	Clr/Set-EnableBit
-24	ExecuteInterrupt

SetInterrupt

Structure *IntVector* qui, après la mise en place de certaines indications, autorise l'interruption (Enable-Bit). Dans A1, on doit trouver un pointeur sur la structure *Interrupt* et dans D0 le numéro de l'interruption. Le numéro zéro correspond à l'interruption *Timer-A*, alors que le numéro 4 correspond à l'interruption *Flag*. Avec cette fonction, il n'est pas possible de modifier les structures *IntVector*, prêtes à être initialisées. Pour ce faire, il faut tout d'abord effacer la structure avec la fonction *ClrInterrupt*. L'interruption donnée sera autorisée en même temps.

ClrInterrupt

Efface la structure *IntVector* et empêche l'interruption. Dans D0, on trouve le numéro de la structure *IntVector* qui doit être effacée. L'interruption donnée sera bloquée au même moment.

Clr/Set-EnableBit

Met le bit *Interrupt-Enable* dans le registre Hardware de la même façon que la structure *Resource*. Dans D0, les bits à modifier doivent être activés. Le bit 7 indique si ces bits doivent être effacés ou activés. Si le bit 7 est effacé, les bits donnés par le registre *Interrupt-Enable* seront effacés. Lorsque D0=$03, les deux interruptions du Timer seront bloquées. Pour retourner à l'état normal, D0 doit contenir l'ancien état du registre *Hardware-Interrupt-Request*.

ExecuteInterrupt

Avec cette fonction, il est possible de libérer une interruption CIA par une source donnée du SoftWare. Dans D0, les bits doivent être activés suivant un des états déterminés par la source du registre *Cia-Request Interrupt-Request*.

De plus, le bit 7 du registre D0 devra aussi être activé. Pour générer une interruption du Timer-A, on doit appeler la fonction avec D0= $81.

L'appel de la fonction n'est exempte d'erreurs que lorsque le pointeur sur la structure *Resource* se trouve dans A6.

```
move.B # $02,D0
move.L ResourceBase, A6
Jsr -18 (A6)
```

Cette routine permet de bloquer l'interruption Timer-B.

Voici maintenant, à la suite, les listings assembleur de chaque fonction.

SetInterrup

```
fc4690 moveq   #$00,D1          Effacer D1
fc4692 move.b  D0,D1            Numéro d'interruption dans D1
fc4694 mulu    #$000c,D1        Offset réel calculé
fc4698 move.l  $0004,A0         ExecBase dans A0
fc469c move.w  #$4000,$dff09a Disable-
fc46a4 addq.b  #1,294(A0)       Macro
fc46a8 lea     64(A6,D1.W),A0 Début de la structure déterminée
fc46ac move.l  8(A0),D1         Structure déjà initialisée ?
fc46b0 bne.s   $fc46e2          Saut si oui
fc46b2 move.l  A1,8(A0)         *iv-Node activé
fc46b6 move.l  18(A1),4(A0)     (*iv_Code)() activé
fc46bc move.l  14(A1),0(A0)     iv_Data activé
fc46c2 move.w  #$0080,D1        Bit Clr/Set activé sur Set
fc46c6 bset    D0,D1            Ecrire les bits à activer
fc46c8 move.w  D1,D0            Valeur transférée dans D0
fc46ca bsr.s   $fc470a          Fonction Clr/Set-Enablebit
fc46cc moveq   #$00,D0          D0 effacé
fc46ce move.l  $0004,A0         ExecBase dans A0
fc46d2 subq.b  #1,294(A0)
fc46d6 bge.s   $fc46e0          Enable-
fc46d8 move.w  #$c000,$dff09a Macro
fc46e0 rts                      Fin du sous-programme

fc46e2 move.l  D1,D0
fc46e4 bra.s   $fc46ce
```

ClrInterrupt :

```
fc46e6 moveq   #$00,D1          D1 effacé
fc46e8 move.b  D0,D1            Numéro de la structure dans D1
fc46ea mulu    #$000c,D1        Offset calculé
fc46ee move.l  $0004,A0         ExecBase dans A0
fc46f2 move.w  #$4000,$dff09a Disable-
fc46fa addq.b  #1,294(A0)       Macro
fc46fe lea     64(A6,D1.W),A0 Position de la structure déterminée
fc4702 clr.l   8(A0)            Pointeur sur la structure Interrupt
fc4706 moveq   #$00,D1          D1 effacé
fc4708 bra.s   $fc46c6          Enablebit effacé
```

Clr/Set-Enablebits :

```
fc470a move.l  $0004,A0          ExecBase dans A0
fc470e move.w  #$4000,$dff09a    Disable-
fc4716 addq.b  #1,294(A0)        Macro
fc471a move.l  34(A6),A0         CiaStartPtr dans A0
fc471e move.b  D0,3328(A0)       Reg. CIA-Int.-Cont activé
fc4722 lea     40(A6),A1         Pointeur sur IntEnableCia
fc4726 bra.s   $fc474c           Bits activés
```

ExecuteInterrupt :

```
fc4728 move.l  $0004,A0          ExecBase dans A0
fc472c move.w  #$4000,$dff09a    Disable-
fc4734 addq.b  #1,294(A0)        Macro
fc4738 move.l  34(A6),A0         CiaStartPtr dans A0
fc473c move.b  3328(A0),D1       Lire Reg. Cia-Int.-Cont.
fc4740 bclr    #7,D1             Effacer bit supérieur
fc4744 or.b    D1,41(A6)         Lier avec IntRequestCia
fc4748 lea     41(A6),A1         Pointeur sur IntRequestCia activé
Embranchement de Clr/Set-Enablebit à $FC4726

fc474c moveq   #$00,D1           D1 effacé
fc474e move.b  (A1),D1           IntRequestCia dans D1
fc4750 tst.b   D0                D0 activé ?
fc4752 beq.s   $fc4762           Saut si non
fc4754 bclr    #7,D0             Bit supérieur effacé
fc4758 bne.s   $fc4760           Si activé, saut
fc475a not.b   D0                Bits effacés
fc475c and.b   D0,(A1)           Bits correspondants
fc475e bra.s   $fc4762           Saut immédiat

fc4760 or.b    D0,(A1)           Bits correspondants activés
fc4762 move.b  40(A6),D0         Test : si une
fc4766 and.b   41(A6),D0         interruption est autorisée
fc476a beq.s   $fc477a           Fin, si non
fc476c move.w  38(A6),D0         IntRequestBit issue de la structure
fc4770 ori.w   #$8000,D0         Clr/Set Bit positionné
fc4774 move.w  D0,$dff09c        Interruption libérée
fc477a move.l  $0004,A0          Prendre ExecBase
```



```
fc477e subq    #1,294(A0)
fc4782 bge.s   $fc478c           Enable-
fc4784 move.w  #$c000,$dff09a    Macro
fc478c move.l  D1,D0             Délivrer Bits-Request précédents
fc478e rts                       Retour de sous-routine
```

SetIntVector

```
Interrupt = SetIntVector(IntNum, int);
   D0                    D0     A1
```

Offset : -162

Description : Cette fonction initialise la structure *IntVector* pour l'utilisation d'un convertisseur d'interruption. Comme paramètre de retour, on obtient un pointeur sur la structure *Interrupt*, utilisée précédemment pour cette interruption.

Paramètres :

IntNum Indique le numéro de l'interruption à utiliser. Avec le numéro 1 par exemple, le convertisseur d'interruptions initialisera le traitement *Diskblock*.

Int Pointeur sur la structure *Interrupt* initialisée.

Résultat :

Interrupt Dans D0, il sera délivré un pointeur sur la structure *Interrupt* précédemment utilisée.

AddIntServer

```
AddIntServer(IntNum, Int);
                D0     A1
```

Offset : -168

Descripti Cette fonction insère les structures *Interrupt* données dans la liste *Interrupt-Server*. La priorité donnée dans la structure *Node* correspond à la place de l'interruption dans la liste. Une interruption de haute priorité sera donc insérée dans cette liste avant une interruption de moindre priorité.

Paramètres :

IntNum Indique le numéro de l'interruption que l'on va utiliser.

Int Pointeur sur la structure *Interrupt* initialisée.

RemIntServer

```
RemIntServer(IntNum, Int);
                D0     A1
```

Offset : -174

Description : La structure *Interrupt* donnée sera issue de la liste *Interrupt-Server*.

Paramètres :

IntNum Indique le numéro de l'interruption à utiliser.

Int Pointeur sur la structure *Interrupt* initialisée.

Cause

```
Cause(Interrupt);
        A1
```

Offset : -180

Description : On libère une nouvelle interruption par le SoftWare au moyen d'une tâche ou d'une interruption. La priorité de cette interruption est inférieure à celle des interruptions Hardware, mais supérieure à celle d'une tâche, cette dernière pourra donc être gérée par une interruption et prise en charge par un processus asynchrone.

Paramètre :

Interrupt Pointeur sur la structure *Interrupt* initialisée.

2.7.5 EXEMPLE D'UN SERVEUR D'INTERRUPTION

Après avoir détaillé théoriquement la programmation des interruptions, voici un exemple de programme illustrant leur gestion. Ce dernier n'est intéressant que pour les possesseurs d'une extension RAM, puisqu'il sera possible d'occuper l'entière *Fast-Memory* en appuyant sur la touche de fonction F10 et de la vider en appuyant sur la touche F9. F1 sert de commutateur de test du clavier. Ce programme peut être modifié en faisant correspondre à certaines touches d'autres processus.

```
memtype      = $10004
nboctets     = 300
allocmem     = -198
freemem      = -210
addIntServer = -168
RemIntServer = -174
taste        = $bfec01
pri          = 100
Type         = 2
intNum       = 3
is_Data      = 14
is_Code      = 18
ln_Type      = 8
ln_Pri       = 9
ln_Name      = 10

         move.l $4,a6
```



```
           .e.l #nboctets,d0
          ..ove.l #memtype,d1        ;fast, CLR
          jsr allocmem(a6)
          tst.l d0
          beq erreur
          move.l d0,a2
          add.l #32,d0               ;Début de la mémoire données
          move.l d0,is_Data(a2)
          move.b #pri,ln_pri(a2)
          move.b #type,ln_Type(a2)
          move.l #Fin-Début+8,d0     ;Taille code déterminée
          jsr allocmem(a6)           ;Mémoire pour programme
          tst.l d0  ;Erreur?
          bne ok1   ;non
          move.l a2,a1               ;*Interrupt
          move.l #nboctets,d0
          jsr freemem(a6)            ;Mémoire libérée
          jmp erreur
ok1:      move.l d0,a3
          move.l a3,is_Code(a2)
          move.l a2,a1               ;*Interrupt-Struct

          lea.l Début,a2
          move.l #Fin-Début,d0
l1:       move.b (a2)+,(a3)+
          dbf d0,l1

          move.l #intNum,d0          ;Interruption clavier
          jsr addIntServer(a6)
          rts

Erreur:   rts


; Dans A1 se trouve le pointeur sur la mémoire données
; Seuls d0, d1, a1, a5 et a6 seront utilisés

Début:    move.l d0,a5               ;*prochaine interruption
          move.b touche,d0
          not d0
```

```
            ror.b #1,d0
            cmp.b #$59,d0 ;F10
            beq Memoiredésactive
            cmp.b #$58,d0 ;F9
            beq Memoireactive
            cmp.b #$50,d0 ;F1
            beq Test
Erreur1:    clr.l d0
  rts

Mémoiredésactive:
            mem = $20004                ;memory-Type
            availmem = -216

            move.l a1,a5
            tst.l (a5)
            bne erreur1
            move.l #$ffffffff,(a5)+
            move.l $4,a6
l6:         move.l #mem,d1
            jsr availmem(a6)
            tst.l d0
            beq end1
            move.l d0,(a5)+
            jsr allocmem(a6)
            tst.l d0
            beq end1
            move.l d0,(a5)+
            bra l6
fin1:       clr.l (a5)
            bra blink


Mémoireactive:
            move.l a1,a5
            move.l $4,a6
            tst.l (a5)
            beq erreur1
            clr.l (a5)+
l7:         tst.l (a5)
```



```
            beq end2
            move.l (a5)+,d0
            move.l (a5)+,a1
            jsr freemem(a6)
            bra l7
Fin2:       bra blink

Test:
            move.l 4(a5),a1             ;Interruption dans a1
            move.l #intNum,d0
            move.l $4,a6
            jsr RemIntServer(a6)
            bra blink

blink:      move.l #$2000,d0
l5:         move.w d0,$dff180
            sub.l #$01,d0

            bne l5
            rts

Fin:
```

Au moyen de ce programme, on peut ouvrir une structure *Interrupt* et l'insérer dans une interruption clavier. Comme la priorité de cette dernière est inférieure à la priorité de notre structure interruption, qui est en fait un test clavier, ce sera notre interruption qui sera exécutée en premier. Ici, seules certaines touches particulières seront testées. En appuyant sur la touche de fonction F10, l'entière *Fast-Memory* sera occupée, alors que le pointeur sur cette dernière et sa longueur seront placés dans le tampon donnée d'interruption. Si on appuie alors sur F9, le pointeur marqué dans le tampon de donnée d'interruption sera utilisé afin de libérer la mémoire. En appuyant sur F1, la structure *Interrupt* sera écartée de la liste *Interrupt-Server*, éliminant ainsi le test du clavier. Le signal visible de l'activation de la touche et de l'exécution de l'instruction sera le clignotement de l'écran.

2.8 La structure ExecBase

La structure *ExecBase* est la structure fondamentale de Exec, dans laquelle on retrouve tous les paramètres importants nécessaires, par exemple, à la gestion d'une tâche. L'adresse de base de la structure correspond à l'adresse de base de *exec-library* et se laisse facilement adresser en C au moyen de la variable *SysBase*. *Sysbase* est une variable standard permettant de communiquer la position de *exec-library*.

Pour accéder à cette structure en assembleur, il faut tout d'abord déterminer l'adresse de base qui se trouve à l'emplacement mémoire $000004.

L'instruction *Move.l $4,A6* permet de délivrer l'adresse de base de *exec-library* de la structure *ExecBase* dans A6.

Comme cette structure est initialisée dès qu'une routine *Reset* est activée, son adresse de base est toujours la même.

Il n'y a décalage que lorsque la taille ou la place de la mémoire RAM est modifiée. Même après cette modification, sa position sera toujours constante. Un Amiga comportant 512 Koctets de RAM possède une structure *ExecBase* à la position $676 avec le Kickstart 1.2. La carte passerelle PC-XT décale la structure *ExecBase* à l'adresse $C00276 qui n'est à prendre en compte que lorsque l'extension de RAM commence à l'adresse $C00000.

La structure est de la forme suivante :

Offset DEZ	Offset HEX	Adresse pour 512KB	Adresse pour 1MB		
					struct ExecBase {
0	$000	$676	$C00276	struct Library	LibNode;
34	$022	$698	$C00298	UWORD	SoftVer;
36	$024	$69A	$C0029A	WORD	LowMemChkSum;
38	$026	$69C	$C0029C	ULONG	ChkBase;
42	$02A	$6A0	$C002A0	APTR	ColdCapture;
46	$02E	$6A4	$C002A4	APTR	CoolCapture;
50	$032	$6A8	$C002A8	APTR	WarmCapture;

54	$036		$C002AC	APTR	SysStkUpper;
58	$03A	$6B0	$C002B0	APTR	SysStkLower;
62	$03E	$6B4	$C002B4	ULONG	MaxLocMem;
66	$042	$6B8	$C002B8	APTR	DebugEntry;
70	$046	$6BC	$C002BC	APTR	DebugData;
74	$04A	$6C0	$C002C0	APTR	AlertData;
78	$04E	$6C4	$C002C4	APTR	MaxExtMem;
82	$052	$6C8	$C002C8	UWORD	ChkSum;
					struct IntVector IntVects[16];
84	$054	$6CA	$C002CA		Sortie série Pri1
96	$060	$6D6	$C002D6		Bloc disque Pri1
108	$06C	$6E2	$C002E2		Interruption soft Pri1
120	$078	$6EE	$C002EE		CIAA Pri2
132	$084	$6FA	$C002FA		Copper Pri3
144	$090	$706	$C00306		Raster Pri3
156	$09C	$712	$C00312		Blitter prêt Pri3
168	$0A8	$71E	$C0031E		Canal audio 0 Pri4
180	$0B4	$72A	$C0032A		Canal audio 1 Pri4
192	$0C0	$736	$C00336		Canal audio 2 Pri4
204	$0CC	$742	$C00342		Canal audio 3 Pri4
216	$0D8	$74E	$C0034E		Entrée série Pri5
228	$0E4	$75A	$C0035A		Synchronisation disque Pri5
240	$0F0	$766	$C00366		CIAB Pri6
252	$0FC	$772	$C00372		Interruption interne Pri6
264	$108	$77E	$C0037E		NMI Pri7
276	$114	$78A	$C0038A	struct Task	*ThisTask;
280	$118	$78E	$C0038E	ULONG	IdleCount;
284	$11C	$792	$C00392	ULONG	DispCount;
288	$120	$796	$C00396	UWORD	Quantum;
290	$122	$798	$C00398	UWORD	Elapsed;
292	$124	$79A	$C0039A	UWORD	SysFlags;
294	$126	$79C	$C0039C	BYTE	IDNestCnt;
295	$127	$79D	$C0039D	BYTE	TDNestCnt;
296	$128	$79E	$C0039E	UWORD	AttnFlags;
298	$12A	$7A0	$C003A0	UWORD	AttnResched;
300	$12C	$7A2	$C003A2	APTR	ResModules;
304	$130	$7A6	$C003A6	APTR	TaskTrapCode;
308	$134	$7AA	$C003AA	APTR	TaskExceptCode;
312	$138	$7AE	$C003AE	APTR	TaskExitCode;
316	$13C	$7B2	$C003B2	ULONG	TaskSigAlloc;

```
320   $140   $7B6    $C003B6  UWORD  TaskTr:   loc;
322   $142   $7B8    $C003B8  struct List MemList;
336   $150   $7C6    $C003C6  struct List ResourceList;
350   $15E   $7D4    $C003D4  struct List DeviceList;
364   $16C   $7E2    $C003E2  struct List IntrList;
378   $17A   $7F0    $C003F0  struct List LibList;
392   $188   $7FE    $C003FE  struct List PortList;
406   $196   $80C    $C0040C  struct List TaskReady;
420   $1A4   $81A    $C0041A  struct List TaskWait;
434   $1B2   $828    $C00428  struct SoftIntList
                               SoftInts[5];
514   $202   $878    $C00478  LONG   LastAlert[4];
530   $212   $888    $C00488  UBYTE  VBlankFrequency;
531   $213   $889    $C00489  UBYTE  PowerSupplyFrequency;
532   $214   $88A    $C0048A  struct List SemaphoreList;
546   $222   $898    $C00498  APTR   KickMemPtr;
550   $226   $89C    $C0049C  APTR   KickTagPtr;
554   $22A   $8A0    $C004A0  APTR   KickCheckSum;
558   $22E   $8A4    $C004A4  UBYTE  ExecBaseReserved[10];
568   $22F   $8AE    $C004AE  UBYTE  ExecBaseNew-
                                     Reserved[20];
                                  );
#define SYSBASESIZE ((long)sizeof(struct ExecBase))

#define AFB_68010 0L
#define AFB_68020 1L
#define AFB_68881 4L
#define AFF_68010 (1L<<0)
#define AFF_68020 (1L<<1)
#define AFF_68881 (1L<<4)
#define AFB_RESERVED8 8L
#define AFB_RESERVED9 9L
```


LibNode - et 0

Ceci est la structure *librairie* de *Exec-Library*, avec une taille positive de $24C octets et une taille négative de $276 octets. Au moyen de la taille positive, on peut voir que la structure *ExecBase* n'est en réalité qu'une structure *Library*.

SoftVer - Offset 34

LowMemChkSum - Offset 36

Peut être utilisé par le programmeur pour comparer la somme de contrôle qui sera calculée dans la zone comprise entre l'offset 34 et l'offset 78 dans le cas d'une insertion d'un vecteur. La façon dont cette somme est calculée peut être observée à l'offset 82 (ChkSum).

ChkBase - Offset 38

Sera utilisé pour tester la position de *ExecBase* lors d'un *Reset*. La position de *ExecBase* sera additionnée avec *ChkBase*, le résultat devant atteindre $FFFFFFFF. Si ce n'est pas le cas, c'est que l'on se trouve en présence d'une erreur importante et il est préférable de réinstaller la structure *ExecBase* ; sinon la structure ne sera pas réinitialisée, le temps étant ainsi économisé.

ColdCapture - Offset 42

Vecteur qui pourra être utilisé par le programmeur lors d'un *Reset* pour se brancher sur une routine personnelle. Un vecteur non utilisé sera mis à 0. La routine *Reset* reconnaîtra l'initialisation de ce vecteur à une routine donnée. L'adresse de retour sera retournée dans A5.

Avant le branchement à la sous-routine indiquée par le vecteur, ce dernier sera automatiquement mis à 0. Jusqu'à ce moment-là, il n'y a pas encore eu de blocage d'interruption ou d'accès DMA. La routine

personnelle adressée ne doit comporter une opération qui travaille avec la pile étant donné que celle-ci n'est pas encore correctement initialisée.

C'est la raison pour laquelle le retour de sous-routine se passe dans A5 et que la routine n'est pas appelée avec une instruction JSR.

CoolCapture - Offset 46

Peut aussi être utilisé pour se brancher à une routine personnelle lors d'un *Reset*. La différence entre *ColdCapture* et *CoolCapture* réside dans le fait que cette dernière n'appelle la routine personnelle qu'un peu plus tard. Il n'y a pas de retour possible par la routine *Reset* avec *CoolCapture*. Etant donné que la pile, la mémoire, la table d'exception et la *librairie Exec* sont ici déjà initialisées, ce vecteur se prête mieux à la plupart des utilisations que celui de *ColdCapture*. Pour sortir de la routine personnelle, on pourra utiliser l'instruction RTS.

WarmCapture - Offset 50

Ceci est un vecteur *Reset* qui ne nous est pas accessible.

SysStkUpper - Offset 54

Indique la limite supérieure de la pile *Superviseur*.

SysStkLower - Offset 58

Indique la limite inférieure de la pile *Superviseur*.

MaxLocMem - Offset 62

Indique la zone maximum *Chip-Memory* accessible (512 Koctets ou $80000 octets).

DebugEr Offset 66

Pointeur sur la sous-routine du Debogueur de l'AMIGA.

DebugData - Offset 70

Pointeur sur le tampon de données du Debogueur (0).

AlertData - Offset 74

MaxExtMem - Offset 78

Donne la limite supérieure de la mémoire de données mise à disposition. Avec un élargissement de la mémoire sur 1Mo, celle-ci est de $C80000.

ChkSum - Offset 82

Résultat de la somme de contrôle sur la zone comprise entre l'offset 34 et l'offset 78. Elle sera testée avant le branchement à *ColdCapture*. Quand des vecteurs particuliers sont reliés à cette zone, la somme de contrôle doit être recalculée ou validée dans *LowChkSum*. La somme de contrôle se calcule comme suit :

```
fc0440 lea     34(A6),A0     Pointeur sur début
fc0444 move.w  #$0016,D0     Nombre de mot -1 dans compteur
fc0448 add.w   (A0)+,D1      Mots additionnés
fc044a dbf     D0,$fc0448    Compteur décrémenté,
fc044e not.w   D1            Opération not et
fc0450 move.w  D1,82(A6)     en mémoire dans ChkSum
```

IntVects[0] - Offset 84

Interruption de la sortie *Série*. Ne sera pas initialisée après un *Reset*.

IntVects[1] - Offset 96

Interruption de la fin du transfert des blocs disquettes. Après un *Reset*, ceci correspondra à un convertisseur d'interruption.

IntVects[2] - Offset 108

Soft-Interrupt : pour une plus ample description, se reporter au chapitre 2.6.

IntVects[3] - Offset 120

CIAA-Interrupt. Après un *Reset*, cette interruption sert de test clavier *(Interrupt-Server)*.

IntVects[4] - Offset 132

Interruption Copper.

IntVects[5] - Offset 144

Cette interruption sera libérée quand le faisceau d'électrons du *Raster* passera à la ligne Raster 0. Cette interruption correspond aussi au signal d'horloge du *Task-Switching (Interrupt-Server)*.

IntVects[6] - Offset 156

Cette interruption sera libérée lorsque le *Blitter* aura fini son travail (convertisseur d'interruption).

IntVects[7] - Offset 168

Canal audio 0 (convertisseur d'interruption).

IntVects[8] - Offset 180

Canal audio 1 (convertisseur d'interruption).

IntVects[9] - Offset 192

Canal audio 2 (convertisseur d'interruption).

IntVects[10] - Offset 204

Canal audio 3 (convertisseur d'interruption).

IntVects[11] - Offset 216

Cette interruption sera libérée lors d'une entrée série (ne sera pas initialisée après un *Reset*).

IntVects[12] - Offset 228

Cette interruption sera libérée par la synchronisation de la disquette (convertisseur d'interruption).

IntVects[13] - Offset 240

Cette interruption sera libérée en même temps qu'une interruption du CIA-B *(Interrupt-Server)*.

IntVects[14] - Offset 252

Cette interruption ne pourra être libérée que par le SoftWare (ne sera pas initialisée après un *Reset*).

IntVects[15] - Offset 264

Interruption non marquée. Cette interruption ne sera pas utilisée mais initialisée comme *Serveur d'interruption*.

**ThisTask - Offset 276*

Pointeur sur une structure *Task* qui est en train d'être traitée. On ne peut pas, avec un programme qui se déroule dans une tâche, lire ou intégrer une valeur dans le pointeur de cette structure *Task*, étant donné que le pointeur qui est à notre disposition correspond à celui de la structure *Task* qui est en train d'être exécutée. La seule possibilité qui nous reste d'intégrer une valeur est de lire la valeur d'une interruption. Ce rôle peut être joué par l'interruption 5 (faisceau du Raster).

idleCount - Offset 280

DispCount - Offset 284

Quantum - Offset 288

Elapsed - Offset 290

SysFlags - Offset 292

Dans ce drapeau, on trouve les bits importants de gestion du système.

Exemple : bit 5 à 0 correspond à une *Soft-Interrupt* non autorisée. Le même bit mis à 1 correspond à une *Soft-Interrupt* autorisée.

IDNestCnt - Offset 294

Indique si les interruptions sont autorisées. Si *IDNestCnt* se trouve à la valeur $FF (-1), elles seront bloquées avec la fonction *Disable*.

A chaque fois que cette fonction sera appelée, *IDNestCnt* sera incrémenté de 1. Avec la fonction *Enable*, *IDNestCnt* sera à nouveau décrémenté. Lorsque ce dernier se trouvera à la valeur -1, les interruptions seront réactivées (le bit *Master* sera positionné).

TDNestCnt - Offset 295

Indique si la fonction *Forbid* a été appelée. Si c'est le cas, *TDNestCnt* sera incrémenté de 1. La commutation vers une autre tâche sera autorisée lorsque *TDNestCnt* ne renfermera pas la valeur -1. Par la fonction *Permit*, *TDNestCnt* sera à nouveau décrémenté afin de permettre l'exécution d'une autre tâche aussitôt que *TDNestCnt* sera à nouveau à -1.

AttnFlags - Offset 296

Indique le type de processeur relié :

```
#define AFF_68010 (1L<<0)
#define AFF_68020 (1L<<1)
#define AFF_68881 (1L<<4)
```

AttnResched - Offset 298

Resmodules - Offset 300

Pointeur sur un module résident. Ce sont des structures portant sur des routines appelées par une routine commençant à l'adresse $FCAF. Ce module sera appelé par un *Reset*. On dispose d'une fonction avec laquelle on peut chercher ce module par son nom. Cette fonction se nomme *FindResident*.

TaskTrapCode - Offset 304

TaskExceptCode - Offset 308

TaskExitCode - Offset 312

TaskSigAlloc - Offset 316

TaskTrapAlloc - Offset 320

MemList - Offset 321

Pointeur sur une liste mémoire qui caractérise les zones occupées et les zones libres.

ResourceList - Offset 336

Structure-Liste dans laquelle sont reliées les structures *Resource*.

DeviceList - Offset 350

Structure-Liste dans laquelle sont reliées les structures *Device*.

IntList - Offset 364

Non utilisé.

LibList - Offset 378

Structure-Liste dans laquelle sont reliées les structures *Library*.

PortList - Offset 392

Structure-Liste dans laquelle sont reliées les structures *Port*.

TaskRe[illegible]Offset 406

Structure-Liste dans laquelle sont reliées les structures *Task* lorsqu'elles correspondent au mode *Ready*.

TaskWait - Offset 420

Structure-Liste dans laquelle sont reliées les structures *Task* lorsqu'elles correspondent au mode *Wait*.

SoftInts[0] - Offset 434

Structure-liste dans laquelle sont reliées les *Soft-Interrupts* de priorité -32 qui attendent d'être traitées.

SoftInts[1] - Offset 450

Structure-Liste dans laquelle sont reliées les *Soft-Interrupts* de priorité -16 qui attendent d'être traitées.

SoftInts[2] - Offset 466

Structure-Liste dans laquelle sont reliées les *Soft-Interrupts* de priorité 0 qui attendent d'être traitées.

SoftInts[3] - Offset 482

Structure-Liste dans laquelle sont reliées les *Soft-Interrupts* de priorité 16 qui attendent d'être traitées.

SoftInts[4] - Offset 498

Structure-Liste dans laquelle sont reliées les *Soft-Interrupts* de priorité 32 qui attendent d'être traitées.

LastAlert[4] - Offset 514

C'est ici que seront stockées les données lors d'une alerte après avoir été prélevées sur la routine *Reset*.

VBlankFrequency - Offset 530

Indique avec quelle fréquence le faisceau *Raster* affiche une image (50 Hertz).

PowerSupplyFrequency - Offset 531

Indique la fréquence de la tension d'alimentation de l'Amiga (50 Hertz).

SemaphoreList - Offset 532

Structure-Liste dans laquelle sont reliées toutes les structures *Semaphore* dès qu'elles sont utilisées.

KickMemPtr - Offset 546

Pointeur sur une structure *MemList ;* cette mémoire est à nouveau occupée après un *Reset*.

KickTagPtr - Offset 550

Pointeur sur une table résidente qui sera reliée lors de l'installation de la table résidente générale.

KickCheckSum - Offset 554

Somme de tests qui sera calculée par la fonction *SumKickData()*.

ExecBase rved[10] - Offset 558

Ceci correspond à 10 octets réservés à la structure *ExecBase* afin de donner à *Exec* la possibilité d'y mettre en mémoire des valeurs déterminées.

ExecBaseNewReserved[20] - Offset 568

Ceci correspond à 20 octets qui sont réservés à la structure *ExecBase* afin qu'*Exec* ait la possibilité d'y mettre en mémoire des valeurs particulières.

2.9 Routine Reset et programme Reset

Ce chapitre détaille la façon dont une routine *Reset* se déroule, comment la taille mémoire est déterminée et s'il est possible d'écrire un programme personnel *Reset*.

2.9.1 DOCUMENTATION SUR LA ROUTINE RESET

```
fc00d2 lea     $040000,A7              Pointeur de pile activé
fc00d8 move.l  #$00020000,D0           Valeur de la boucle d'attente
fc00de subq.l  #1,D0                   Valeur décrémentée
fc00e0 bgt.s   $fc00de                 Saut
fc00e2 lea     -228(PC)(=$fc0000),A0   Pointeur sur la ROM KickStart
fc00e6 lea     $f00000,A1              Valeur de comparaison chargée
fc00ec cmpa.l  A1,A0                   Reset à $F00000 ?
fc00ee beq.s   $fc00fe                 Saut si oui
fc00f0 lea     12(PC)(=$fc00fe),A5     Pointeur sur la suite du programme
fc00f4 cmpi.w  #$1111,(A1)             Module à partir de $F00000 ?
fc00f8 bne.s   $fc00fe                 Saut si non
fc00fa jmp     2(A1)                   Si oui branchement vers sous-routine
fc00fe move.b  #$03,$bfe201            Port commuté en sortie
fc0106 move.b  #$02,$bfe001            LED éteinte
fc010e lea     $dff000,A4              Pointeur sur l'adresse CHIP
fc0114 move.w  #$7fff,D0               Valeur chargée
fc0118 move.w  D0,154(A4)              Toutes les interruptions bloquées
```

```
fc011c move.w  D0,156(A4)              Interrup    ıs effacées
fc0120 move.w  D0,150(A4)              Accès DMA bloqué
fc0124 move.w  #$0200,256(A4)
fc012a move.w  #$0000,272(A4)
fc0130 move.w  #$0444,384(A4)          Couleur activée
fc0136 move.w  #$0008,A0               Pointeur sur Exceptions activé
fc013a move.w  #$002d,D1               Compteur du nombre de vecteurs
fc013e lea     1140(PC)(=$fc05b4),A1   Pointeur sur routine Harderror activé
fc0142 move.l  A1,(A0)+                Exceptions insérées
fc0144 dbf     D1,$fc0142              Saut si non achevé
fc0148 bra.l   $fc30c4                 Test Guru
```

Ceci teste si le *Reset* est originaire du gourou. Si c'est le cas, il sera délivré dans D7 le numéro de gourou et dans D6, le pointeur mémoire. Sinon, il sera chargé dans D6 la valeur $FFFFFFFF. Puis la routine *Reset* continuera à être exécutée.

```
fc014c move.l  $0004,D0          ExecBase initialisé
fc0150 btst    #0,D0             ExecBase à adresse paire ?
fc0154 bne.s   $fc01ce           Erreur si impaire
fc0156 move.l  D0,A6             Execbase dans D0
fc0158 add.l   38(A6),D0         Addition à ChkBase
fc015c not.l   D0                Résultat inversé
fc015e bne.s   $fc01ce           Saut, si erreur
```

Une erreur apparaît quand le pointeur sur la structure *ExecBase* est faux.

```
fc0160 moveq   #$00,D1           D1 effacé pour somme de test
fc0162 lea     34(A6),A0         Mise en place de ChkBase
fc0166 moveq   #$18,D0           Valeur compteur de boucle
fc0168 add.w   (A0)+,D1          Somme de test calculée
fc016a dbf     D0,$fc0168        Boucle jusqu'à fin de calcul
fc016e not.w   D1                Résultat inversé
fc0170 bne.s   $fc01ce           Erreur si non nul
fc0172 move.l  42(A6),D0         ColdCapture dans D0
fc0176 beq.s   $fc0184           Saut si non activé
fc0178 move.l  D0,A0             Pointeur dans A0
fc017a lea     8(PC)(=$fc0184),A5 Pointeur sur suite
fc017e clr.l   42(A6)            ColdCapture chargé
```



```
fc0182 jmp     (A0)                    Branchement
fc0184 bchg    #1,$bfe001              LED allumée
fc018c move.l  -382(PC)(=$fc0010),D0   Kick. Version comparée
fc0190 cmp.l   20(A6),D0               avec Execlib
fc0194 bne.s   $fc01ce                 Erreur si les versions diffèrent
fc0196 move.l  62(A6),A3               limite supérieure de la mémoire
fc019a cmpa.l  #$00080000,A3           512KB de Chipmemory ?
fc01a0 bhi.s   $fc01ce                 Erreur si supérieur
fc01a2 cmpa.l  #$00040000,A3           256KB de Chipmemory ?
fc01a8 bcs.s   $fc01ce                 Erreur si inférieur
fc01aa move.l  78(A6),A4               MaxExtMem dans A4
fc01ae move.l  A4,D0                   Test expansion
fc01b0 beq.l   $fc0240
fc01b4 cmpa.l  #$00dc0000,A4           Limite supérieure = $DC0000 ?
fc01ba bhi.s   $fc01ce                 Erreur si supérieur
fc01bc cmpa.l  #$00c40000,A4           Limite inférieure = $C40000 ?
fc01c2 bcs.s   $fc01ce                 Erreur si inférieur
fc01c4 move.l  A4,D0                   Non-sens, car A4 = D0
fc01c6 andi.l  #$0003ffff,D0           Test adresse limite
fc01cc beq.s   $fc0240                 Oui, sinon erreur
```

Ici commence la partie de routine qui sera accessible lorsque l'initialisation de la structure *ExecBase* fera apparaître un défaut. Elle sera alors à nouveau initialisée.

```
fc01ce lea     $0400,A6          Zone mémoire disponible
fc01d2 suba.w  #$fd8a,A6         Les adresses d'ExecBase déterminent
                                  le cas sans Fast-Mem
fc01d6 lea     $c00000,A0        Zone Fast-Mem
fc01dc lea     $dc0000,A1        Limite supérieure RAM
fc01e2 lea     6(PC)(=$fc01ea),A5 Pointeur sur suite
fc01e6 bra.l   $fc061a           Limite mémoire prise en compte
```

A l'aide de cette routine, la limite supérieure de la *Fast-RAM* sera connue.

Elle délivre le pointeur sur la fin de la RAM dans A4. S'il n'y a aucune *Fast-Ram* à disposition, il sera retourné un 0 dans A4. Cette autoreconnaissance de la mémoire *Fast* ne fonctionne que si la RAM se trouve à partir de l'adresse $C00000.

Les possesseurs d'un Amiga 1000 au. ι la possibilité d'auto-configurer une extension RAM ne débutant pas à l'adresse $C00000 à l'aide de la disquette *Kickstart*.

```
fc01ea move.l  A4,D0                Fast-RAM ?
fc01ec beq.s   $fc0208              Saut, si non
fc01ee move.l  #$00c00000,A6        Limite inférieure dans A6
fc01f4 suba.w  #$fd8a,A6            Position d'ExecBase communiquée
fc01f8 move.l  A4,D0                Limite supérieure dans D0
fc01fa lea     $c00000,A0           Limite inférieure dans A0
fc0200 lea     6(PC)(=$fc0208),A5   Pointeur sur suite
fc0204 bra.l   $fc0602              Mémoire effacée
```

Dans cette routine, la zone *Fast-Memory* sera remplie de zéro.

```
fc0208 lea     $0000,A0             Limite RAM inférieure
fc020c lea     $200000,A1           Limite supérieure
fc0212 lea     6(PC)(=$fc021a),A5   Pointeur sur suite
fc0216 bra.l   $fc0592              Taille mémoire inférieure prise en compte
```

La zone mémoire inférieure sera comprise de l'adresse $0000 à l'adresse $200000. Il sera testé dans la routine la taille de la mémoire inférieure ; celle-ci sera retournée dans A3.

```
fc021a cmpa.l  #$00040000,A3        Zone inférieure à 256 Ko ?
fc0220 bcs.s   $fc0238              Si oui, alors erreur, Hard-Reset
fc0222 move.l  #$00000000,$0000     $0000 effacé
fc022a move.l  A3,D0                Limite supérieure dans D0
fc022c lea     $00c0,A0             Limite inférieure déterminée
fc0230 lea     14(PC)(=$fc0240),A5  Pointeur sur suite
fc0234 bra.l   $fc0602              Mémoire inférieure effacée
```

La routine sera alors prête à l'effacement de la *Fast-RAM* utilisée, c'est-à-dire de la zone comprise entre $00C0 à la limite supérieure de la mémoire.

```
fc0238 move.w  #$00c0,D0            Couleur de l'écran lors du Reset
fc023c bra.l   $fc05b8              Hard-Reset (LED clignote 11 fois)
```


Lors de l ɔcution de la routine *Reset*, le témoin lumineux clignotera onze fois, indiquant par là le passage à la *Boot-ROM* et le démarrage du *Reset*. L'accès à cette routine pourra aussi se faire lorsqu'une condition *Exception* apparaîtra lors de la première partie du *Reset*.

```
fc0240 lea     $dff000,A0           Pointeur sur adresses CHIP
fc0246 move.w  #$7fff,150(A0)       Accès DMA bloqués
fc024c move.w  #$0200,256(A0)
fc0252 move.w  #$0000,272(A0)
fc0258 move.w  #$0888,384(A0)       Couleur écran activée
fc025e lea     84(A6),A0            Pointeur sur le premier IntVector
fc0262 movem.l 546(A6),D4-D2        KickMemPtr, KickTagPtr, KickCheckSum
                                    en mémoire
fc0268 moveq   #$00,D0              D0 effacé
fc026a move.w  #$007d,D1            Compteur activé
fc026e move.l  D0,(A0)+             Effacer de A0 à ExecBase
fc0270 dbf     D1,$fc026e           Continue , si pas terminé
fc0274 movem.l D4-D2,546(A6)        KickMemPtr, KickTagPtr et
                                    KickCheckSum mémorisés
fc027a move.l  A6,$0004             Pointeur sur ExecBase activé
fc027e move.l  A6,D0                Pointeur dans D0
fc0280 not.l   D0                   ChkBase calculé
fc0282 move.l  D0,38(A6)            ChkBase inséré
fc0286 move.l  A4,D0                Limite supérieure RAM dans D0
fc0288 bne.s   $fc028c              Saut lorsque Fast-RAM est à disposition
fc028a move.l  A3,D0                Sinon, limite supérieure Chip-RAM
fc028c move.l  D0,A7                initialisée en pile système
fc028e move.l  D0,54(A6)            et insérée
fc0292 subi.l  #$00001800,D0        Limite de la pile rectifiée
fc0298 move.l  D0,58(A6)            et activée en limite inférieure
fc029c move.l  A3,62(A6)            Limite Chip-RAM activée
fc02a0 move.l  A4,78(A6)            Limite Fast-RAM activée
fc02a4 bsr.l   $fc30e4              Dernière alerte enregistrée
```

La valeur issue de $FC0148 est mise en mémoire dans D6 et D7 et sera écrite à la place prévue (*LastAlert* (offset 514)).

```
fc02a8 bsr.l   $fc0546              Test du processeur
```

La routine teste la présence des processeurs 68000, 68010, 68020 et 68881. Les bits se trouvant dans D0 seront initialisés suivant le processus reconnu.

```
fc02ac or.w     D0,296(A6)              Bits de AttnFlags positionnés
fc02b0 lea      32(PC)(=$fc02d2),A1     Pointeur sur le tableau

fc02b4 move.w   (A1)+,D0                Offset dans D0
fc02b6 beq.l    $fc033e                 Fin, si plus d'Offset
fc02ba lea      0(A6,D0.W),A0           Pointeur sur position placé
fc02be move.l   A0,(A0)                 Tête de liste enregistrée
fc02c0 addq.l   #4,(A0)                 Pointeur sur lh_Tail
fc02c2 clr.l    4(A0)                   lh_Tail effacé
fc02c6 move.l   A0,8(A0)                lh_TailPred activé
fc02ca move.w   (A1)+,D0                lh_Type pris en compte
fc02cc move.b   D0,12(A0)               et activé
fc02d0 bra.s    $fc02b4                 Saut immédiat
```

Cette partie de routine insère à nouveau dans *ExecBase* la *structure-Liste* suivante :

MemList
ResourceList
DeviceList
LibList
PortList
TaskReady
TaskWait
InterList
SoftIntList (tous 5)
SemaphoreList

```
                fc02d2
                  .
                  .
                  .
Tableau pour l'installation des listes
                  .
                  .
```



```
                fc030c
                  .
                  .
Valeur pour l'installation de la structure ExecLibrary
                  .
                  .
                fc0326
                  .
                  .
Texte ASCII : Chip Memory
                  .
                  .
                fc0332
                  .
                  .
Texte ASCII : Fast Memory
                  .
                  .
                fc033e
```

```
fc033e lea      11380(PC)(=$fc2fb4),A0  Pointeur sur le module Trap-Code
fc0342 move.l   A0,304(A6)              Enregistrer dans TaskTrapCode
fc0346 move.l   A0,308(A6)              Enregistrer dans TaskExceptCode
fc034a move.l   #$00fc1cec,312(A6)      Enregistrer TaskExitCode
fc0352 move.l   #$0000ffff,316(A6)      Enregistrer TaskSigAlloc
fc035a move.w   #$8000,320(A6)          Enregistrer TaskTarpAlloc
fc0360 lea      8(A6),A1                Pointeur sur LibNode.ln_Type
fc0364 lea      -90(PC)(=$fc030c),A0    Pointeur sur table
fc0368 moveq    #$0c,D0                 Compteur activé
fc036a move.w   (A0)+,(A1)+             Structure Execlibrary mise en place
fc036c dbf      D0,$fc036a              Saut, si pas terminé
fc0370 move.l   A6,A0                   Pointeur sur ExecBase dans A0
fc0372 lea      5836(PC)(=$fc1a40),A1   Pointeur sur table
fc0376 move.l   A1,A2
fc0378 bsr.l    $fc1576                 Fonction: MakeFunction()
```

Cette routine permet l'installation de *ExecLibrary*. La table débute à l'adresse $FC1A40 et la longueur de la librairie sera retournée dans D0.

```
fc037c move.w  D0,16(A6)                  Longueur      a librairie enregistrée
fc0380 move.l  A4,D0                      Fast-RAM p    sente ?
fc0382 beq.s   $fc03a8                    Saut, si pas de Fast-RAM
fc0384 lea     588(A6),A0                 Pointeur sur la fin d'ExecBase
fc0388 lea     -88(PC)(=$fc0332),A1       Chaine Fast Mem.
fc038c moveq   #$00,D2                    Priorité à MemHeader
fc038e move.w  #$0005,D1                  Memory-Atribute (Public, Fast)
fc0392 move.l  A4,D0                      Pointeur sur la fin de Fast-RAM
fc0394 sub.l   A0,D0                      Structure ExecBase rectifiée
fc0396 subi.l  #$00001800,D0              SysStack rectifié
fc039c bsr.l   $fc19ea                    Structure MemHeader mise en place
```

La routine débutant à l'adresse $FC19EA met en place une structure *MemHeader* avec les données correspondantes. On trouvera dans D0 la taille de la zone mémoire mise à disposition.

```
fc03a0 lea     $0400,A0                   Début Chip-RAM
fc03a4 moveq   #$00,D0                    D0 activé
fc03a6 bra.s   $fc03b2                    Saut immédiat
fc03a8 lea     588(A6),A0                 Pointeur sur la fin d'ExecBase
fc03ac move.l  #$fffe800,D0
fc03b2 move.w  #$0003,D1                  Memory-Atribute (Public, Chip)
fc03b6 move.l  A0,A2                      Pointeur sur le début de la RAM
fc03b8 lea     -148(PC)(=$fc0326),A1      Chaine Chip Mem.
fc03bc moveq   #$f6,D2                    Priorité MemHeader
fc03be add.l   A3,D0                      Calcul de la zone
fc03c0 sub.l   A0,D0                      Mémoire effective
fc03c2 bsr.l   $fc19ea                    Mise en place de MemHeader
fc03c6 move.l  A6,A1                      ExecBase dans A1
fc03c8 bsr.l   $fc140c                    Calcul de la somme test librairie
```

ExecLibrary sera relié à *LibList* et sa somme de tests sera calculée.

```
fc03cc lea     938(PC)(=$fc0778),A0       Pointeur sur Exceptions
fc03d0 move.l  A0,A1                      Pointeur sur exceptions dans A1
fc03d2 move.w  #$0008,A2                  Pointeur sur destination dans A2
fc03d6 bra.s   $fc03de                    Saut immédiat
fc03d8 lea     0(A0,D0.W),A3              Calcul de l'adresse
fc03dc move.l  A3,(A2)+                   et enregistrement
fc03de move.w  (A1)+,D0                   Prise en compte Offset
```



```
fc03e0 bne.s     c03d8                    Saut, si pas terminé
fc03e2 move.w    ,96(A6),D0               Prise en compte de AttnFlags
fc03e6 btst      #0,D0                    Rajout 68010 ?
fc03ea beq.s     $fc041e                  Saut, si pas présent
fc03ec lea       1166(PC)(=$fc087c),A0     Pointeurs sur nouveaux Traps
fc03f0 move.w    #$0008,A1                Pointeur sur destination
fc03f4 move.l    A0,(A1)+                 Exceptions enregistrées
fc03f6 move.l    A0,(A1)+                 Exceptions enregistrées
fc03f8 move.l    #$00fc08ba,-28(A6)       Enregistrement Extension
fc0400 move.l    #$42c04e75,-528(A6)      Enregistrement Extension
fc0408 btst      #4,D0                    68881 présent ?
fc040c beq.s     $fc041e                  Saut, sinon
fc040e move.l    #$00fc108a,-52(A6)       Enregistrement Extension
fc0416 move.l    #$00fc10e8,-58(A6)       Enregistrement Extension
fc041e bsr.l     $fc125c                  Structure Interrupt enregistrée
fc0422 lea       $dff000,A0               Pointeur sur adresses CHIP
fc0428 move.w    #$8200,150(A0)           Autorisation DMA-Blitter
fc042e move.w    #$c000,154(A0)           Interruptions autorisées
fc0434 move.w    #$ffff,294(A6)           IDNestCnt effacé
fc043a bsr.l     $fc22fa                  Installation du Debugger
fc043e moveq     #$00,D1                  D1 effacé
fc0440 lea       34(A6),A0                Pointeur sur SoftVer
fc0444 move.w    #$0016,D0                Compteur dans D0
fc0448 add.w     (A0)+,D1                 Somme de test calculée
fc044a dbf       D0,$fc0448               Saut, si pas terminé
fc044e not.w     D1                       Valeur inversée
fc0450 move.w    D1,82(A6)                et sauvegardée dans ChkSum
fc0454 lea       118(PC)(=$fc04cc),A0     Ptr sur la structure MemList
fc0458 bsr.l     $fc191e                  AllocEntry()
```

Une structure *MemList* sera mise en place dans la routine et $1024 octets seront réservés pour la tâche et sa pile.

```
fc045c move.l  D0,A2                      Pointeur sur la structure MemList
fc045e lea     4112(A2),A0                Pointeur sur le début de la tâche
fc0462 lea     8(A0),A1                   Calcul de MemEntry
fc0466 addi.l  #$00000010,D0              MeMList additionné
fc046c move.l  D0,58(A1)                  SpLower activé
fc0470 move.l  A0,62(A1)                  SpUpper activé
fc0474 move.l  A0,54(A1)                  SpReg activé
```

```
fc0478 move    A0,USP              SPReg act:  comme pile
fc047a clr.b   9(A1)               Pri effacé
fc047e move.b  $0001,8(A1)         Valeur de tc_Type pour Task
fc0484 move.l  #$00fc00a8,10(A1)   Pointeur sur Nom
```

Le nom de la tâche est *exec.library*. Il sera éliminé plus tard.

```
fc048c lea     74(A1),A0           Pointeur sur tc_MemEntry
fc0490 move.l  A0,(A0)             Liste
fc0492 addq.l  #4,(A0)             pour MemEntries
fc0494 clr.l   4(A0)               installée
fc0498 move.l  A0,8(A0)
fc049c exg     A2,A1               Permutation des pointeurs MemList et Task
fc049e bsr.l   $fc15d8             AddHead()
fc04a2 exg     A2,A1               Nouvelle permutation
fc04a4 move.l  A1,276(A6)          Task enregistrée comme étant ThisTask
fc04a8 suba.l  A2,A2               initPc
fc04aa move.l  A2,A3               et finalPC effacés
fc04ac bsr.l   $fc1c48             AddTask()
fc04b0 move.l  276(A6),A1          Pointeur sur Task dans A1
fc04b4 move.b  #$02,15(A1)         tc_State activé sur RUN
fc04ba bsr.l   $fc1600             Remove() Task de la Liste
```

La tâche sera mise en mode *Running* et écartée de la liste *TaskReady*, ce qui signifie que le programme se déroulant à ce moment-là sera traité en tant que tâche.

```
fc04be andi.w  #$0000,SR           Bloquer toutes les Interruptions
fc04c2 addq.b  #1,295(A6)          SysFlag activé
fc04c6 jsr     -138(A6)            Permit() (Task débloquées)
fc04ca bra.s   $fc0500             Saut immédiat

                fc04cc
                  .
                  .
    Données de la structure MemoryList
                  .
                  .
                fc04fe
```



```
fc0500 lea     -30(PC)(=$fc04e4),A0
fc0504 bsr.l   $fc0900             Rechercher de la structure résidente
```

Dans cette routine, toutes les structures résidentes se trouvant en ROM seront recherchées et leurs pointeurs stockés les uns derrière les autres dans un tableau. De plus, si les pointeurs *KickMemPtr*, *KickTagPtr* et *KickcheckSum* ont été activés dans la structure *ExecBase*, la zone mémoire des données sera occupée et la structure résidente des données sera prise en charge dans le tableau.

L'ordre des entrées du tableau correspond à la priorité des structures résidentes. Le pointeur sur ce tableau sera stocké dans *ResModules*.

```
fc0508 move.l  D0,300(A6)
fc050c bclr    #1,$bfe001          LED-Allumée
fc0514 move.l  46(A6),D0           CoolCapture pris en compte
fc0518 beq.s   $fc051e             Saut, si non activé
fc051a move.l  D0,A0               CoolCapture dans A0
fc051c jsr     (A0)                Branchement
fc051e moveq   #$01,D0             startClass setzen
fc0520 moveq   #$00,D1             Version activée
fc0522 bsr.l   $fc0af0             InitCode(), traitement des structures
                                   résidentes
```

On n'entre plus dans la partie de programme suivante.

```
fc0526 move.l  50(A6),D0           WarmCapture pris en charge
fc052a beq.s   $fc0530             Saut, si non activé
fc052c move.l  D0,A0               WarmCapture dans A0
fc052e jsr     (A0)                Branchement
fc0530 moveq   #$0d,D0             Valeur pour compteur activée
fc0532 clr.l   -(A7)               Stack effacée
fc0534 dbf     D0,$fc0532          Boucle, si non terminé
fc0538 movem.l (A7)+,A5-A0/D7-D0
fc053c jsr     -114(A6)            Accès au Debugger
fc0540 move.l  $0004,A6            ExecBase dans A6
fc0544 bra.s   $fc053c             Saut immédiat
```

2.9.2 STRUCTURES RESIDENTES

Afin de mieux comprendre comment il est possible de construire un module personnel *Reset*, il faut détailler d'abord ce qu'une structure résidente signifie et comment elle sera traitée.

Les structures résidentes sont des structures qui se trouvent dans le système d'exploitation et qui seront recherchées lors d'un *Reset*. Cette recherche se fera grâce à un code de reconnaissance qui se trouve marqué au début de la structure. Les positions de toutes les structures résidentes trouvées seront rassemblées dans un tableau et un pointeur sur ce dernier sera initialisé dans *ResModules* (dans la structure *ExecBase*).

Lors du déroulement du *reset*, le pointeur sera pris en compte dans la fonction *InitCode()*.

Avec l'aide du pointeur de ce tableau, les structures résidentes seront à nouveau trouvées.

Dans une telle structure, les pointeurs sont stockés les uns derrière les autres et indépendamment des Flags qui sont marqués dans la structure résidente, pointent sur une table pour registre, permettant l'appel de la fonction *MakeLib()* ou l'exécution d'un programme.

Autrement dit, il est possible, avec une structure résidente, d'accéder à un programme personnel ou d'appeler la fonction *MakeLib()*.

Lorsque l'on désire appeler la fonction *MakeLib()*, on a plusieurs possibilités. On peut différencier par la fonction *MakeLib()* si, dans la structure mise en place, il est possible d'insérer dans la *List-Library*, avec *AddLibrary()*, dans la *List-Device*, avec *AddDevice()* ou dans la *List-Resource* avec *AddResource()*. Ceci est autorisé grâce à la possibilité de mettre en place des structures *Library*, *Device* et *Resource* avec la fonction *MakeLib()*.

Une structure résidente est de la forme suivante :

```
      struct Resident {
0       UWORD rt_MatchWord;
2       struct Resident *rt_MatchTag;
6       APTR  rt_EndSkip;
10      UBYTE rt_Flags;
11      UBYTE rt_Version;
12      UBYTE rt_Type;
13      BYTE  rt_Pri;
14      char  *rt_Name;
18      char  *rt_IdString
22      APTR  rt_Init;
      );

#define RTC_MATCHWORD 0x4AFCL
#define RTF_AUTOINIT (1L<<7)
#define RTF_COLDSTART (1L<<0)
#define RTM_WHEN 3L
#define RTW_NEVER 0L
#define RTW_COLDSTART 1L
```

rt_MatchWord

est un mot grâce auquel la structure est reconnaissable en mémoire. Lors d'un *Reset*, la structure résidente sera recherchée et reconnue par ce mot. Le mot doit avoir la valeur $4AFC pour que la structure puisse être trouvée.

rt_MatchTag

est un pointeur sur la structure elle-même et employé pour la reconnaissance de la structure en mémoire.

Après que le *MatchWord* ait été trouvé, il sera testé si le mot long suivant est un pointeur sur la structure propre. Si c'est le cas, une structure résidente sera reconnue.

rt_EndSkip

Pointeur sur la fin d'une structure. Avec ce pointeur, il est possible de rallonger la structure pour ajouter des données importantes.

rt_Flags

Indique, suivant les bits activés, si la structure résidente doit être traitée et dans l'affirmative, si l'instruction ou le saut du programme peuvent être exécutés. Si le bit supérieur (bit 7) est effacé, cela signifie qu'on sautera à un programme indiqué dans la structure. S'il est activé, *rt_Init* pointera sur une table de registre nécessaire à l'exécution de la fonction *MakeLib()*.

rt_Version

Indique la version de la structure.

rt_Type

Indique quelle instruction doit être exécutée.

rt_Name

Pointeur sur le nom de la structure.

rt_IdString

Pointeur sur une chaîne de caractères qui donne des indications supplémentaires sur la structure.

rt_Init

Pointeur sur le programme à exécuter ou pointeur sur le tableau du contenu registre à charger lors de l'appel de la fonction *MakeLib()*.

Si cette dernière fonction est appelée, les registres suivants doivent être enregistrés dans le tableau selon l'ordre donné :

D0 = DataSize
D1 = CodeSize
A0 = FuncInt
A1 = StructInt
A2 = LibInt

Le tableau du pointeur sur la structure résidente est détaillé dans les lignes suivantes. La marque de fin de ce dernier correspond au dernier mot long qui comporte la valeur 0.

Ce tableau sera normalement mis en place par la routine *Reset*.

Si le bit supérieur (bit 31) d'un mot long dans le tableau est activé, ceci signifie que le mot long correspond à un pointeur sur la suite du tableau sans prise en compte des bits supérieurs.

```
00fc 00b6 00fc 4afc 00fe 4880 00fe 4fe4
00fc 450c 00fc 4794 00fe 4774 00fe 49cc
00fc 5378 00fe 502e 00fe 507a 00fe 90ec
00fc 34cc 00fe 50c6 00fe 0d90 00fe 510e
00fe 98e4 00fd 3f5c 00fc 323a 00fe 424c
00fe b400 00ff 425a 00fe 8884 0000 0000
```

La deuxième structure résidente est de la forme suivante :

```
fc4afc 4afc 00fc 4afc 00fc 516c 8121 096e 00fc ................
fc4b0c 4b48 00fc 4b16 00fc 4b38 6578 7061 6e73 .........expans
fc4b1c 696f 6e20 3333 2e31 3231 2028 3420 4d61 ion 33.121 (4 Ma
fc4b2c 7920 3139 3836 290d 0a00 0000 0000 01c8 y 1986).........
fc4b3c 00fc 4b86 00fc 4b5a 00fc 4bee 6578 7061 ..........expa
fc4b4c 6e73 696f 6e2e 6c69 6272 6172 7900 e000 nsion.library...
```

Afin d'améliorer l'initialisation de la structure, les valeurs correspondantes sont réintroduites dans la structure suivante.

```
      struct Resident {
 0     UWORD rt_MatchWord;              $4AFC
 2     struct Resident *rt_MatchTag;    $FC4AFC
 6     APTR rt_EndSkip;                 $FC516C
10     UBYTE rt_Flags;                  %10000001
11     UBYTE rt_Version;                33
12     UBYTE rt_Type;                   Library
13     BYTE rt_Pri;                     110
14     char *rt_Name;                   expansion.library
18     char *rt_IdString;               $FC4B16
22     APTR rt_Init;                    $FC4B38
      );
```

L'octet *rt_Flag* est initialisé à %10000001. Le bit supérieur est ainsi activé, impliquant que *rt_Init* pointe sur les données d'un registre qui sera utilisé lors de l'appel de la fonction *AddLibrary()*. Ici, cette dernière fonction sera appelée, étant donné que le type de la structure résidente est *Library* (NT_LIBRARY = 09).

Il existe plusieurs possibilités à disposition :

Type	Appel de fonction
03 = Device	AddDevice()
08 = Resource	AddResource()
09 = Library	AddLibrary()

Pour trouver la structure résidente, on dispose de la fonction *InitCode()*. Lors de l'appel, deux paramètres *StartClass* et *Version* seront délivrés.

Version détermine que seules les structures résidentes dont les versions sont identiques ou supérieures, seront exécutées. La valeur délivrée dans *StartClass* sera reliée avec un ET logique à *rt_Flags*. Si le résultat est différent de 0, la structure sera exécutée et la fonction *InitResident* sera appelée.

Ces deux paramètres déterminent donc le choix dans l'exécution des structures résidentes lors de l'appel de la fonction *InitCode*.

Lors d'un *Reset*, la fonction *InitClass* sera exécutée avec les paramètres *Startclass = 01* et *Version = 00*. Seules les structures résidentes possédant un bit 0 activé dans *rt_Flag* seront exécutées.

Afin de mieux comprendre comment les routines se déroulent, voici les listages assembleur des fonctions.

InitCode

```
                InitCode(StartClass, Version);
                              D0          D1


fc0af0 movem.l A2/D3-D2,-(A7)  Registre sauvegardé
fc0af4 move.l  300(A6),A2      Pointeur sur ResModules
fc0af8 move.b  D0,D2           StartClass dans D2
fc0afa move.b  D1,D3           Version dans D3
fc0afc move.l  (A2)+,D0        Pointeur sur tableau
fc0afe beq.s   $fc0b22         Saut si marque de fin
fc0b00 bgt.s   $fc0b0a        Saut si bit supérieur non activé (bit 31)
fc0b02 bclr    #31,D0          Sinon effacer le Bit 31
fc0b06 move.l  D0,A2           Pointeur mis en mémoire dans le tableau
fc0b08 bra.s   $fc0afc         Saut immédiat
fc0b0a move.l  D0,A1           Pointeur sur Resident dans A1
fc0b0c cmp.b   11(A1),D3       Version comparée
fc0b10 bgt.s   $fc0afc         Version résidente dans les
fc0b12 move.b  10(A1),D0       rt_Flags précédents
fc0b16 and.b   D2,D0           ET logique avec StartClass
fc0b18 beq.s   $fc0afc         Saut, si ne démarre pas
fc0b1a moveq   #$00,D1         segList à Null
fc0b1c jsr     -102(A6)        InitResident()
fc0b20 bra.s   $fc0afc         Saut immédiat
fc0b22 movem.l (A7)+,A2/D3-D2  Reprise des registres
fc0b26 rts                     Saut de retour
```

InitResident

```
InitResident(Resident, SegList);
                A1        D1
```

```
fc0b28 btst    #7,10(A1)       Bit 7 de rt_Flags testés
fc0b2e bne.s   $fc0b3c         Si pas activés, alors instruction testée
fc0b30 move.l  22(A1),A1       Sinon saut
fc0b34 moveq   #$00,D0         D0 effacé
fc0b36 move.l  D1,A0           segList dans A0
fc0b38 jsr     (A1)            Branchement
fc0b3a bra.s   $fc0b7e         Saut immédiat
fc0b3c movem.l A2-A1,-(A7)     Registres sauvegardés
fc0b40 move.l  22(A1),A1       Pointeur sur registre
fc0b44 movem.l (A1),A2-A0/D0   Registre fonction
fc0b48 jsr     -84(A6)         MakeLib()
fc0b4c movem.l (A7)+,A2/A0     Registre retourné
fc0b50 move.l  D0,-(A7)        Signal de retour
fc0b52 beq.s   $fc0b7c         Si Erreur, alors Fin
fc0b54 move.l  D0,A1           Pointeur sur Library dans A1
fc0b56 move.b  12(A0),D0       rt_type dans D0
fc0b5a cmpi.b  #$03,D0         rt_type = Device?
fc0b5e bne.s   $fc0b66         Saut, sinon
fc0b60 jsr     -432(A6)        autrement AddDevice()
fc0b64 bra.s   $fc0b7c         Saut immédiat
fc0b66 cmpi.b  #$09,D0         rt_type = Library?
fc0b6a bne.s   $fc0b72         Saut, sinon
fc0b6c jsr     -396(A6)        autrement AddLibrary()
fc0b70 bra.s   $fc0b7c         Saut immédiat
fc0b72 cmpi.b  #$08,D0         rt_type = Resource?
fc0b76 bne.s   $fc0b7c         Saut, sinon
fc0b78 jsr     -486(A6)        autrement AddResource()
fc0b7c move.l  (A7)+,D0        Reprise D0
fc0b7e rts                     Retour
```


2.9.3 PR[illegible] AMME RESET PROPRE ET STRUCTURES

Après avoir détaillé toutes les conditions, voici comment on peut programmer des structures et un programme *Reset* personnel.

On a en fait deux possibilités. La première consiste à insérer dans son programme personnel les vecteurs *ColdCapture* et *CoolCapture*, la deuxième à réoccuper la zone mémoire avec l'aide des vecteurs *KickMemPtr* et *KickTagPtr* et d'insérer une structure résidente dans le tableau des vecteurs résidents. Tous ces vecteurs se trouvent dans la structure *ExecBase*.

Il faut évidemment choisir entre les deux possibilités. Il est préférable de faire une routine *reset* à l'aide de *ColdCapture*. Il faudra mettre le vecteur sur une routine qui renouvellera le vecteur *ColdCapture* afin de permettre une boucle de fin.

Lors de l'initialisation des vecteurs *ColdCapture*, il faudra, de plus, recalculer la somme de tests qui est formée à partir des premiers vecteurs *ExecBase*. Une routine bloquant le *Reset* aurait l'allure suivante :

```
run:
allocMem    = -198
require     = 1
ColdCapture = 42

            move.l $4,a6
            move.l #Fin-Debut,d0
            move.l #require,d1
            jsr allocMem(a6)
            tst.l d0
            beq Erreur
            move.l d0,a1
            move.l a1,ColdCapture(a6)
            move.w #Fin-Debut,d0
            lea.l Debut,a0
l1:         move.b (a0)+,(a1)+
            dbf d0,l1
```

```
; Calcul somme de tests

            clr.l d1
            lea.l 34(a6),a0
            move.w #$16,d0
l2:         add.w (a0)+,d1
            dbf d0,l2
            not.w d1
            move.w d1,82(a6)
Erreur:     rts

; Routine exécutée par le Reset

Début:      lea.l Debut(pc),a0
            move.l $4,a6
            move.l a0,ColdCapture(a6)
l3:         jmp l3(pc)
Fin:
```

Le programme occupe en premier lieu la mémoire nécessaire, puis copie le programme qui démarrera lors d'un *Reset* dans cette zone. L'adresse de départ du programme sera enregistrée dans *ColdCapture* et la somme de tests sera recalculée.

La réactivation des vecteurs *ColdCapture* est nécessaire étant donné que la routine *Reset* les effacera.

Lorsque le programme sera démarré à la suite d'un *Reset*, seul l'arrêt du calculateur sera sauvegardé.

Avec *ColdCapture*, on pourra sortir très tôt de la routine *Reset* pour passer dans le programme personnel. Vous pouvez vous reporter à la documentation de la routine *Reset* à partir de l'adresse $FC0172. Jusqu'à ce moment, la couleur d'écran sera commutée sur noir, toutes les interruptions et accès DMA seront bloqués et la somme de tests comparée dans la structure *ExecBase*.

Lorsque *CoolCapture* sera activé, les vecteurs de la routine *Reset* seront effacés, les adresses de la suite de la routine *Reset* seront transmises dans A5. Il ne doit pas y avoir de travail sur la pile dans la routine personnelle, ni aucun appel de sous-programme, la pile n'étant pas encore initialisée.

Avec l'instruction JMP(A5), la routine *Reset* sera à nouveau exécutée.

L'autre vecteur utilisé lors du déroulement d'un *Reset*, en dehors de la routine *Reset* dans un programme personnel, se nomme *CoolCapture*. Lorsqu'on accède au programme personnel, la mémoire est prête à être à nouveau organisée, les structures *ExecBase* et *ExecLibrary* sont prêtes à être mises en place, les interruptions sont prêtes à être ordonnées et les exceptions à être remises dans leurs valeurs de départ.

CoolCapture sera accessible par une instruction JSR(A0). Le vecteur pourra être utilisé pour contrôler une extension mémoire qui ne le sera pas automatiquement. Le saut à la routine *Reset* se fait à l'adresse $FC051C.

Le déroulement est identique à celui de *ColdCapture*. Le vecteur sera enregistré, puis la somme de tests de la structure *ExecBase* sera à nouveau calculée. La façon dont cette somme de tests est calculée est détaillée dans le programme d'exemples.

Utilisation de la fonction **KickSumData() :**

La meilleure façon d'obtenir un programme *Reset* personnel et des structures est de générer, avec l'aide des enregistrements d'*ExecBase*, *KickMemPtr*, *KickTagPtr* et *KickCheckSum*.

Avec l'utilisation de ces entrées, on peut occuper n'importe quelle zone mémoire et insérer quelques structures résidentes dans le tableau des pointeurs. Pour ce faire, *KickMemPtr* doit pointer sur une structure *MemList*, ces entrées étant à nouveau occupées lors d'un *Reset*. *KickTagPtr* est un pointeur sur un tableau résident et pointe sur le *ResModules* de la structure *ExecBase*.

Les structures résidentes marquées dans tableau ont une priorité indépendante du tableau mis en place c toutes les structures résidentes.

Ces deux pointeurs ne sont utilisés que lorsque *KickCheckSum* est juste (somme de tests sur le tableau résident et sur les structures *MemList*). Pour faire le calcul de la somme de tests, on utilise la fonction *KickSumData()* de *exec-library*.

KickSumData()

```
Somme = KickSumData();
 D0
```

Offset : -612

Description : La fonction calcule la somme de tests entre la structure *MemList* donnée dans *KickMemPtr* et le tableau résident donné dans *KickTagPtr*. Le résultat de ce calcul sera retourné dans D0.

Si on veut à nouveau occuper une zone mémoire déterminée, après un *Reset*, sans perte de données, on doit marquer cette zone dans une structure *MemList* et l'occuper avec *AllocEntry()*. Chaque structure *MemList* doit être également occupée. Il est aussi possible de chainer ensemble plusieurs structures *MemList*.

Si, lors d'un *Reset*, on veut encore exécuter quelques programmes, il faudra mettre en place des structures résidentes qui permettront d'appeler le programme désiré.

Les pointeurs sur les structures résidentes devront être rassemblés dans un tableau qui se terminera par un 0. La zone mémoire des structures résidentes des programmes exécutés, de même que le tableau résident, devront toujours être occupés avec une structure *MemList* et insérés dans la liste d'occupation de la mémoire lors d'un *Reset*. Le pointeur sur le tableau résident sera enregistré dans *KickTagPtr*.

Puis la fonction *KickSumData()* sera appelée et la somme de tests calculée dans *KickCheckSum* sera mise en mémoire. Les zones mémoire seront alors protégées du *Reset* et les programmes marqués dans les structures résidentes seront exécutés lors du *Reset*.

Voici un tableau qui montre quelles structures résidentes doivent être exécutées lors d'un *Reset* avec les priorités correspondantes.

Pri.	Resident-Pos.	Description
120	$FC00B6	exec.lib. mis en place (non autorisé)
110	$FC4AFC	exception.lib. mis en place
100	$FE4880	potgo.lib mis en place
80	$FC450C	cia.resources mis en place
70	$FC4794	disk.resource mis en place
70	$FE4774	misc.resource mis en place
70	$FE49CC	ram.lib. mis en place (non autorisé)
65	$FC5378	graphics.lib. mis en place
60	$FE502E	keyboard.device mis en place
60	$FE507A	gameport.device mis en place
50	$FE90EC	timer.device mis en place
40	$FC34CC	audio.device mis en place
40	$FE50C6	input.device mis en place
31	$FE0D90	layers.lib. mis en place
20	$FE510E	console.device mis en place
20	$FE98E4	trackdisk.device mis en place
10	$FD3F5C	intuition.lib. mis en place
5	$FC323A	gurus, si présents, activés
0	$FE424C	math.lib mis en place
0	$FEB400	workbench.task, (non autorisé)
0	$FF425A	dos.lib mis en place (non autorisé)
-60	$FE8884	branchement au processus de démarrage

Après la dernière structure résidente, aucune autre ne pourra plus être exécutée, étant donné qu'après l'appel, cette structure ne sera plus retournée avec la fonction *InitCode()*. Une priorité inférieure à -60 n'a donc aucun sens.

2.9.4 ROUTINE NOFASTMEM

Pour terminer ce chapitre, nous allons encore montrer une variante de désactivation de la *Fast-Memory*.

La routine suivante : *NoFastMem* occupe tout simplement l'entière *Fast-Memory* afin que les programmes suivants soient exclusivement chargés dans la *Chip-Memory*. On établira alors que toutes les structures importantes, comme par exemple la structure *ExecBase* ou les interruptions prêtes à être initialisées, lors d'un *Reset*, se tiendront toujours dans la *Fast-Memory*. Ceci peut être la raison du non-fonctionnement de plusieurs programmes, en dépit de la désactivation de la *Fast-RAM*. Il existe malgré tout une autre alternative avec l'utilisation de la routine *Reset*. Un seul programme est alors utilisé, employant toutes les démarches importantes qui apparaissent lors du *Reset*, puis retournant à une place correspondante dans la routine *Reset*. Dans une routine personnelle, on peut désactiver la *Fast-RAM* de la manière suivante : il suffit de sauter à l'emplacement, dans la routine *Reset*, où toutes les structures sont réinitialisées, tout en communiquant au *Reset* qu'il n'y a pas de *Fast-RAM* connectée. Toutes ces structures seront alors initialisées dans la Chip-RAM.

Cette illusion restera présente aussi longtemps que la structure *ExecBase* ne sera pas détruite par un programme, moment où il sera à nouveau constaté combien de mémoire est disponible. Sinon, il restera marqué qu'aucune *Fast-RAM* n'est présente et ceci pour tous les *Reset* suivants qui ne nécessitent pas d'initialisation de la structure *ExecBase* et autres structures.

Un tel programme est de la forme suivante :

```
move.b #$03,$bfe201       ;Port sur sortie
move.b #$02,$bfe001       ;LED éteinte
lea.l $dff000,a4
move.w #$7fff,d0
move.w d0,154(a4)
move.w d0,156(a4)         Accès DMA bloqués
move.w d0,150(a4)
```



```
move.w #   .00,256(a4)
move.l     .0000,272(a4)
move.w #$0444,384(a4)

lea.l $400,a6
suba.w #$fd8a,a6          ;Adresse de base d'ExecBase communiquée
lea.l $c00000,a0
lea.l $dc0000,a1
move.l #$00,a4            ;Pas de Fast-RAM présente
move.l a4,d0
move.l #$40000,a7         ;Pointeur de pile dans A7
move.l #$ffffffff,d6      ;Valeur d'absence d'alerte
jmp $fc0208               ;Saut dans la routine Reset
```

3. L'AMIGADOS

Pour l'utilisateur, la partie essentielle du système d'exploitation de l'Amiga est le DOS, abréviation de *"Disk Operating System"*. Sa tâche est de s'occuper de tous les problèmes d'entrée et de sortie, par exemple des opérations sur disquettes ou des saisies au clavier. Les multiples fonctions nécessaires à cet effet sont mises à la disposition de l'utilisateur sous la forme d'une bibliothèque, de structure analogue à la bibliothèque Exec.

3.1 La bibliothèque DOS

Comme c'était déjà le cas pour la bibliothèque Exec, la bibliothèque DOS ne se trouve pas sur la disquette en tant que fichier, contrairement par exemple à la bibliothèque *Intuition*. Elle doit cependant être une fois ouverte avant l'utilisation, pour rendre possible l'accès au DOS. Grâce à ce processus, le programme qui ouvre la bibliothèque a accès aux fonctions du DOS par l'intermédiaire d'un tableau de pointeurs.

3.1.1 CHARGEMENT DE DOS.LIBRARY

Lorsqu'un programme quelconque veut utiliser une fonction de la bibliothèque DOS, il doit donc tout d'abord ouvrir cette bibliothèque. On emploie pour cela une fonction de Exec, qui porte le nom de *OldOpenLibrary*. Cette fonction reçoit un pointeur sur le nom de la bibliothèque, qui doit être écrit en minuscules et se terminer par un octet 0. On peut aussi utiliser la fonction *OpenLibrary*, à laquelle il faut cependant communiquer un paramètre supplémentaire, la version désirée de la bibliothèque. Si le numéro de version est plus grand ou égal à ce numéro, la bibliothèque s'ouvre. C'est aussi pourquoi on place en général un 0 en ce point, pour que la version n'intervienne pas.

En langage C, on y parvient très simplement.

La ligne

```
DOSBase = OpenLibrary;("dos.library",0);
```

permet d'obtenir d'Exec en DOSBase un pointeur sur la bibliothèque DOS, pointeur grâce auquel on peut ensuite appeler les fonctions DOS. Le pointeur n'a pas à être réutilisé, car le compilateur C se charge de la suite. On peut malgré tout vérifier que le DOS a été correctement ouvert, en examinant la valeur de retour ; cette valeur est 0 s'il s'est produit une erreur. Voici l'instruction correspondante :

```
if (DOSBase == 0) exit(DOS_OPEN_ERROR);
```

En langage machine par contre, les instructions ne sont pas aussi simples, mais néanmoins pas trop longues. L'ouverture de la bibliothèque DOS se programme de la façon suivante :

```
Exec_Base      = 4
OldOpenLibrary = -408


        move.l  Exec_Base,a6        ;Pointeur sur base Exec en A6
        lea     DOS_Name,a1         ;Pointeur sur le nom de bibliothèque
        jsr     OldOpenLibrary(a6) ;Ouvrir la bibliothèque
        move.l  d0,DOS_Base         ;Sauver le pointeur sur base DOS
        beq     error               ;Une erreur est survenue
        ...

      error:                        ;Traitement de l'erreur
        ...


DOS_Base:    dc.l  0              ;Place pour la base DOS
DOS_Name:    dc.b  "dos.library",0
```

Le pointeur obtenu dans D0 est nécessaire pour tous les appels ultérieurs d'une fonction DOS. Si l'ouverture n'a pas bien fonctionné, on reçoit en retour un 0 en D0, et le programme passe à la routine de traitement des erreurs *'error'*.

La bibliothèque DOS est donc disponible par utilisation du programme ci-dessus. Nous avons dit qu'elle avait la même structure que la bibliothèque Exec, et son traitement est donc identique. Les adresses d'entrée des différentes fonctions se trouvent sous l'adresse de base dans DOS_Base, et elles sont donc appelées avec des offsets négatifs.

3.1.2 APPEL DE FONCTION ET TRANSMISSION DE PARAMETRE

Pour l'appel d'une fonction DOS, on a souvent besoin, outre l'adresse de la fonction, de quelques autres paramètres, qui doivent donc être transmis. Ces paramètres sont communiqués dans les registres de données D1 à D4 du processeur.

Un exemple : pour ouvrir une simple fenêtre, on utilise la fonction DOS *Open()*. Les paramètres nécessaires sont les suivants :

* Un pointeur sur le nom du fichier à ouvrir, se terminant par un octet nul, dans le registre D1. Pour notre exemple, nous adopterons comme nom la définition de la fenêtre CON:, suivi des paramètres adéquats.

* En D2 est exigée l'indication du mode d'accès. Ce mode dit s'il s'agit d'un fichier à installer ou d'un fichier déjà existant. Pour la fenêtre à ouvrir dans notre exemple, nous transmettrons *'ancien'*, pour que l'on puisse aussi lire dans la fenêtre.

Le programme en langage machine pour ce exemple serait donc approximativement le suivant:

```
Open      = -30
Mode_old  = 1005


        ...
        move.l  #FileName,d1     ;Pointeur sur la définition de fichier
        move    #Mode_old,d2     ;Mode: ancien
        move.l  DOS_Base,a6      ;adresse de la base DOS en A6
        jsr     Open(a6)         ;Ouvrir fichier (fenêtre)
        move.l  d0,ConHandle     ;Sauver pointeur sur handle fichier
```

```
        beq     error           ;Une erreur     survenue!
        ...

ConHandle:  dc.l 0              ;Place pour le handle du fichier
FileName:   dc.b "CON:10/10/620/200/** Fenêtre de test **",0
```

Nous reviendrons dans un chapitre ultérieur sur l'utilisation précise du canal standard CON:.

3.1.3 LES FONCTIONS DOS

Dans ce chapitre, nous allons maintenant présenter une à une toutes les fonctions DOS. Nous indiquons aussi bien les offsets que les registres dans lesquels les différents paramètres doivent être transmis.

1) Fonctions générales d'entrée/sortie

Open

```
Handle = Open(Nom, Mode);
 D0       -30   D1    D2
```

Ouvrir un fichier

Ouvre le fichier dont la définition est donnée par un nom conclu d'un octet 0 ; D1 pointe sur ce nom.

Le mode est en D2 ; il peut être *Mode_readwrite* (1004 en DOS 1.2) pour permettre les entrées et les sorties ; *Mode_old* (1005) uniquement pour les entrées dans le fichier, ou *Mode_new* (1006) uniquement pour les sorties.

En D0 est retourné un pointeur sur la structure *Filehandle*, ou bien un 0 si la fonction ne peut pas être exécutée. Le *Filehandle* a la forme suivante :

Offset	Nom	Signification
0	Link	Non utilisé
4	Interact	Si <>0, le fichier est interactif
8	ID	Numéro d'identification du fichier
12	Buffer	Pointeur sur la mémoire interne nécessaire
16	CharPos	Pointeur qui donne la position actuelle dans le tampon
20	BufEnd	Pointeur qui indique la fin du tampon
24	ReadFunc	Pointeur sur routine appelée si le tampon est vide
28	WriteFunc	Pointeur sur routine appelée lorsque le tampon est plein
32	CloseFunc	Pointeur sur routine appelée lorsqu'on ferme le fichier
36	Argument1	
40	Argument2	Arguments dépendant du type de fichier

La plupart des entrées sont prévues uniquement pour l'usage interne de *l'AmigaDOS*. Elles ne doivent donc pas être manipulées.

Close

```
Close(Handle);
-36     D1
```

Fermer un fichier

Ferme le fichier ouvert avec *Open*. Le pointeur transmis en D1 est celui qui a été obtenu par la fonction *Open* et qui pointe sur le *Filehandle*.

Read

```
Nombre = Read(Handle, Buffer, Longueur);
  D0      -42    D1      D2      D3
```

Lire des données

Lit dans le fichier spécifié par *'handle'* un nombre d'octets indiqué par *'longueur'*, à partir de l'adresse *'buffer'*.

La valeur retournée en D0 indique le nombre d'octets réellement lus. Si cette valeur est 0, la fin du fichier est déjà atteinte. S'il se produit une erreur, la valeur retournée est -1.

Write

```
Nombre = Write(Handle, Buffer, Longueur);
  D0      -48    D1      D2      D3
```

Ecrire des données

Ecrit dans le fichier spécifié par *'handle'* un nombre d'octets indiqué par *'longueur'*, à partir de l'adresse *'buffer'*.

La valeur retournée en D0 indique le nombre d'octets réellement écrits. Si cette valeur est 0, la fin du fichier est déjà atteinte. S'il se produit une erreur, la valeur retournée est -1.

Seek

```
Position = Seek(Handle, Intervalle, Mode);
  D0        -66    D1      D2      D3
```

Modifier un pointeur de fichier

Cette fonction modifie le pointeur interne dans le fichier spécifié par *'handle'*. Le *'mode'* détermine si la valeur portée dans *'distance'* modifiera le pointeur relativement au début du fichier, à la position actuelle ou à la fin du fichier. Cette valeur est calculée à partir de là avec son signe exact, de telle façon que les déplacements vers l'arrière soient également possibles.

Les modes possibles sont :

OFFSET_BEGINNING	-1
OFFSET_CURRENT	0
OFFSET_END	1

La valeur en retour indique la position actuelle du pointeur après que la fonction ait été exécutée. Pour déterminer la position momentanée du pointeur, on peut prendre simplement le mode *'relatif à la position actuelle'* (OFFSET_CURRENT), et effectuer un déplacement de 0 octets : la position retournée sera alors égale à l'ancienne position.

Input

```
Handle = Input();
  D0      -54
```

Transmettre le canal d'entrée standard

Cette fonction transmet le *handle* du canal, à partir duquel les entrées standard peuvent être lues. Dans le cas où le programme a été appelé à partir du CLI, il s'agit du *handle* de la fenêtre CLI.

Si dans la commande CLI qui a appelé le prc nme, il a été fait usage de la possibilité de redirection des données, c'est le *handle* du canal sélectionné qui est communiqué. Un exemple :

```
>Nom_Programme <DF0:Nom_fichier
```

a pour conséquence que les entrées obtenues au moyen de *Read* sont effectuées à l'intérieur du programme appelé à partir du fichier *'Nom_fichier'*.

Output

```
Handle = Output();
```

Transmettre le canal de sortie standard

Cette fonction transmet le *handle* du canal, dans lequel les sorties standard peuvent être effectuées. Dans le cas où le programme a été appelé à partir du CLI, il s'agit du *handle* de la fenêtre CLI. Ici aussi, il existe la possibilité de modifier la sortie standard, par exemple :

```
>Nom_programme >PRT:
```

envoie les sorties standard du programme appelé à l'imprimante.

WaitForChar

```
Status = WaitForChar(Handle, Timeout);
 D0       -204            D1       D2
```

Attendre la réception d'un caractère

Cette fonction attend du canal spécifié par *'handle'* la réception d'un caractère pendant un temps indiqué par le nombre de microsecondes donné en *'timeout'* (par exemple fenêtre RAW:, attendre la pression d'une touche au clavier).

Si aucun actère n'est reçu dans ce laps de temps, la valeur 0 est retournée dans 'status', sinon la valeur -1. Le caractère peut alors être lu avec la fonction *Read*.

La fonction n'est disponible que s'il s'agit d'un canal interactif (terminal virtuel), par exemple une fenêtre RAW:, dans laquelle on peut effectuer aussi bien des entrées que des sorties, et où les données ne sont pas immédiatement exigibles.

IsInteractive

```
Status = IsInteractive(Handle);
 D0       -216            D1
```

Transmettre le type de canal

La valeur TRUE(-1) est retournée en *'status'*, si le canal spécifié par *'handle'* est un terminal virtuel, où on peut effectuer aussi bien des entrées que des sorties. Sinon la valeur retournée est FALSE (0).

IoErr

```
Error = IoErr ()
 D0      -132
```

Transmettre les erreurs d'entrée/sortie

Si une erreur est signalée après l'appel d'une fonction, la plupart du temps par un 0 en D0 comme valeur retournée, l'appel de *IoErr()* permet de communiquer le message d'erreur exact. D0 contient alors le numéro de l'erreur survenue précédemment (comparer avec la commande WHY du CLI).

Vous trouverez dans le paragraphe suivant la liste des erreurs avec leurs numéros.

2) Opérations sur les disquettes

CreateDir

```
Lock = CreateDir(Nom);
D0       -120       D1
```

Créer un sous-répertoire

Le sous-répertoire *'Nom'* est créé dans le répertoire actuel.

La valeur retournée représente un pointeur sur une structure de fichier *(Lock)*, de la forme suivante :

Offset	Nom	Signification
0	NextBlock	Pointeur sur le Lock suivant de la chaîne ou 0
4	DiskBlock	Numéro de bloc du répertoire ou du file-header
8	AccessType	Type d'accès : -1 = accès exclusif, -2 = accès général
12	ProcessID	Numéro d'identification
16	VolNode	Pointeur sur Info disquettes

Cette structure représente pour ainsi dire la clé d'accès à ce fichier ou à ce sous-répertoire (cf. MAKEDIR du CLI).

Lock

```
lock = Lock(Nom, Mode);
 D0     -84    D1     D2
```

Transmettre la clé du fichier

Un fichier ou un sous-répertoire nommé *'Nom'* est recherché sur la disquette et une structure est engendrée en conséquence. Le mode détermine le type d'accès au fichier. Si c'est le mode *Lecture* (-2), plusieurs tâches élémentaires *(tasks)* peuvent lire dans le fichier ; si c'est le mode *Ecriture* (-1), seul ce programme peut écrire dans le fichier.

CurrentDir

```
oldLock = CurrentDir(Lock);
  D0        -126        D1
```

Transformer un sous-répertoire en répertoire actuel

Le sous-répertoire spécifié par *'Lock'* est transformé en répertoire actuel (cf. la commande CD du CLI).

La valeur retournée représente le pointeur sur le répertoire qui était précédemment le répertoire actuel, ou sur son *'Lock'*.

ParentDir

```
Lock_new = ParentDir(Lock);
  D0         -210      D1
```

Transmettre le répertoire immédiatement supérieur

Retourne en D0 le *Lock* du répertoire immédiatement supérieur à celui qui est indiqué en paramètre. Si le *'Lock'* introduit en paramètre est déjà celui de la racine, on obtient en retour un 0 en D0.

DeleteFile

```
Status = DeleteFile(Nom);
  D0        -72        D1
```

Supprimer un fichier

Cette fonction sert à supprimer le fichier dont le nom est indiqué en paramètre. Le nom transmis doit se terminer par un octet 0. Si la fonction ne peut pas être exécutée (fichier inexistant, fichier protégé contre l'écriture, répertoire non vide, etc...), alors on obtient en retour un message d'erreur.

S'il s'agit de supprimer un sous-répertoire, il ne doit contenir aucun fichier.

Rename

```
Status = Rename(Nom_ancien, Nom_nouveau);
  D0       -78         D1          D2
```

Changer le nom d'un fichier

La fonction permet de changer le nom du fichier indiqué par *'ancien-nom'*. S'il existe déjà un fichier portant le nouveau nom que l'on veut introduire, le processus est interrompu et on obtient en retour un message d'erreur.

Les deux indications de nom peuvent aussi être constituées par des noms de chemins. Dans ce cas, le fichier est transporté de l'ancien répertoire dans le nouveau tout en recevant son nouveau nom. Ceci ne marche néanmoins que si l'on reste sur la même unité logique.

DupLock

```
newLock = DupLock(Lock);
  D0        -96      D1
```

Copier un Lock

Permet de copier l'ancienne structure *Lock* dans la nouvelle. DO pointe alors sur la nouvelle structure. On peut appliquer cette fonction lorsqu'il y a plusieurs processus qui veulent accéder à ce fichier. Mais si un *Lock* n'est admis qu'à l'écriture, il ne peut pas être copié, étant donné que l'accès en écriture est exclusif de toute autre opération.

UnLock

```
UnLock (Lock);
 -90     D1
```

Supprimer un Lock

La structure *Lock*, créée avec *Lock()*, *DupLock()* ou *CreateDir()*, est supprimée grâce à cette fonction, et la mémoire est libérée en conséquence.

Examine

```
Status = Examine(Lock, InfoBlock);
  D0       -102    D1       D2
```

Rechercher des informations dans un fichier

La structure sur laquelle est pointé D2 est remplie grâce à cette fonction par les informations sur le fichier spécifié par son *Lock*. Cette structure est nommée *FileInfoBlock*, et elle est constituée de la façon suivante :

Offset	Nom	Signification
0	DiskKey.L	Numéro de disquette
4	DirEntryType.L	Type d'entrée (+=Directory, -=Fichier)
8	FileName	108 Octets avec le nom de fichier
116	Protection.L	Fichier protégé ?
120	EntryType.L	Type d'entrée
124	Size.L	Longueur fichier en octets
128	NumBlocks.L	Nombre des blocs occupés
132	Days.L	Date de création
136	Minutes.L	Heure de création (minutes)
140	Ticks.L	heure de création (secondes)
144	Comment	116 octets avec commentaire

D0 contir un 0 dans le cas où la fonction ne peut pas être exécutée.

ExNext

```
Status = ExNext(Lock, InfoBlock);
  D0       -108    D1       D2
```

Retourne la mention du répertoire suivant

Permet d'obtenir l'*InfoBlock* rempli par *Examine()*, et le *Lock* du répertoire sélectionné. Les informations du premier fichier ou sous-répertoire rencontré dans le répertoire sélectionné et correspondant à la donnée transmise sont alors reportées dans le *FileInfoBlock*. Lors de tout appel ultérieur de *ExNext()*, c'est la mention suivante convenant à la donnée dans le même répertoire qui est retournée. Si une telle mention n'existe pas, ou s'il s'est produit une erreur, on obtient en retour un 0 en D0.

Grâce à ces commandes *Lock()*, *Examine()* et *ExNext()*, on peut lire le catalogue d'une disquette. On s'y prend pour cela de la façon suivante :

1. Avec *Lock()*, on crée la clé d'accès au répertoire sélectionné.

2. *Examine()* permet alors d'obtenir le nom du répertoire ou de la disquette. On crée en même temps le *FileInfoBlock* nécessaire à l'étape suivante.

3. En appelant plusieurs fois la fonction *ExNext()*, on lit les différentes mentions portées dans le répertoire et ces informations sont reportées dans le *FileInfoBlock*. Cette dernière opération est répétée jusqu'à ce que la fonction *ExNext()* retourne un 0 : il n'existe plus de mentions inscrites dans le répertoire !

Voici maintenant un petit programme en langage machine, exécutant ces diverses étapes.

La routine *Print* appelée n'est pas mentio e ici, mais elle pourrait le cas échéant afficher à l'écran le nom et la longueur du fichier qu'on est en train de lire.

Avant l'appel de cette routine, il faut ouvrir la bibliothèque DOS et déposer l'adresse de base de DOS dans *"dosbase"*.

```
Lock      = -84
Examine   = -102
ExNext    = -108
IoErr     = -132

     ...
directory:                    ;* Catalogue de DF0:
     move.l  dosbase,a6       ;Adresse de base DOS en A6
     move.l  #name,d1         ;Pointeur sur nom de fichier/de chemin
     move.l  #-2,d2           ;Mode 'Lecture'
     jsr     Lock(a6)         ;Rechercher fichier
     tst.l   d0               ;trouvé?
     beq     error            ;non!
     move.l  d0,locksav       ;sinon sauver la clé

     move.l  dosbase,a6       ;Adresse de base DOS
     move.l  locksav,d1       ;Clé en D1
     move.l  #fileinfo,d2     ;Pointeur sur FileInfoBlock
     jsr     Examine(a6)      ;Rechercher nom de la disquette
     tst.l   d0               ;OK?
     beq     error            ;non (se produit très rarement)
     bra     affichage        ;sinon afficher le nom

loop:                         ;* Nom de fichier à rechercher
     move.l  dosbase,a6       ;Adresse de base DOS
     move.l  locksav,d1       ;Clé en D1
     move.l  #fileinfo,d2     ;Pointeur sur FileInfoBlock
     jsr     ExNext(a6)       ;chercher le fichier suivant
     tst.l   d0               ;trouvé?
     beq     error            ;non: fin

affichage:                    ;* afficher le nom
     bsr     Print            ;afficher/traiter le nom
```



```
     br      loop             ;et poursuivre...

error:                        ;* communiquer statut I/O
     move.l  dosbase,a6       ;Adresse de base DOS en A6
     jsr     IoErr(a6)        ;Status holen
     rts                      ;Fin...

name:     dc.b  'DF0:',0
even
locksav:  blk.l   0
fileinfo: blk.l 260
```

Une fois cette routine terminée, on obtient en D0 le code d'erreur transmis par la fonction *IoErr()*. Ce code doit être 232 *(no_more_entries)*, sinon cela signifie que quelque chose a marché de travers.

Info

```
Status = Info(Lock, InfoData);
 D0      -104  D1      D2
```

Rechercher des informations sur une disquette

Le bloc de paramètres pointé par D2 est rempli, grâce à cette fonction, d'informations sur la disquette utilisée. Ce bloc doit commencer à une adresse divisible par 4 *(Longword aligned)*.

Lock doit convenir à cette disquette ou du moins à un fichier ou un sous-répertoire de cette disquette.

Le bloc de paramètres *InfoData* est constitué comme suit :

Offset	Nom	Signification
0	NumSoftErrors	Nombre d'erreurs sur la disquette
4	UnitNumber	Unité de disquettes installée
8	DiskState	Etat disquette (voir ci-dessous)
12	NumBlocks	Nombre des blocs sur la disquette
16	NumBlocksUsed	Nombre des blocs utilisés
20	BytesPerBlock	Nombre d'octets par bloc
24	DiskType	Type de la disquette (voir ci-dessous)
28	VolumeNode	Pointeur sur le nom de la disquette
32	InUse	<>0, lorsque la disquette est active

DiskState indique l'état de la disquette. Les différentes possibilités sont ici les suivantes :

80 Disquette protégée contre l'écriture

81 Disquette qui vient d'être réparée (validating)

82 Disquette OK, on peut écrire dessus

DiskType contient le type de la disquette sous forme de texte, lorsqu'il y en a une. Les valeurs possibles sont les suivantes :

-1 Pas de disquette introduite

BAD Disquette illisible (formatage incorrect)

DOS Disquette DOS

NDOS Le format est correct, mais ce n'est pas une disquette DOS

KICK Disquette Kickstart

SetComment

```
Status = SetComment(Nom, Commentaire);
 D0        -180        D1       D2
```

Placer un commentaire sur un fichier

Le fichier ou le sous-répertoire indiqué par *'nom'* est pourvu d'un commentaire.

Le commentaire peut avoir une longueur d'au plus 80 caractères et il doit se terminer par un octet nul.

SetProtection

```
Status = SetProtection(Nom, Masque);
 D0        -186          D1     D2
```

Fixer l'état d'un fichier

Cette commande permet de déterminer le statut du fichier indiqué, en ce qui concerne l'écriture et la lecture. Les 4 bits inférieurs du masque ont la signification suivante :

Bit	Signification, dans le cas où le bit est positionné
0	On ne peut pas supprimer le fichier
1	On ne peut pas exécuter la commande
2	On ne peut pas écrire dans le fichier
3	On ne peut pas lire dans le fichier

CreateProc

```
Processus = CreateProc(Nom, Pri, Segment, pile);
   D0         -138       D1   D2    D3      D4
```

Créer un nouveau processus

La commande permet de créer un nouveau processus sous le nom pointé par D1. Ce processus aura la priorité définie dans *'Pri'*, et il recevra en partage une pile dont la taille est fixée par *'pile'*.

Dans *'segment'*, on transmet un pointeur sur une liste de segments (cf. aussi *LoadSeg*), dans laquelle est défini le code du programme à démarrer. Le programme doit débuter dans le premier segment de la liste.

Le résultat de la fonction est le nouveau ID du processus, ou un 0 s'il y a eu une erreur.

DateStamp

```
DateStamp(Vecteur);
  -192       D1
```

Communiquer la date et l'heure

Retourne en D1 un pointeur sur un tableau constitué de trois mots longs. Si l'on n'a pas déterminé l'heure dans l'Amiga, les trois mots longs ont pour contenu 0. Sinon, le premier mot long contient le nombre de jours écoulés depuis le 1er janvier 1978, le second contient le nombre de minutes écoulées depuis minuit, et le troisième le nombre de 1/50 de seconde écoulés dans la minute présente.

Cette dernière valeur est cependant toujours un multiple de 50, ce qui fait que l'on n'obtient en réalité que le nombre de secondes multiplié par 50.

Delay

```
Delay(durée);
-198    D1
```

Arrêter momentanément le processus en cours

Le processus est arrêté pendant le nombre de 1/50 de seconde indiqué dans *'durée'*.

DeviceProc

```
Processus = DeviceProc(Nom)
   D0         -174       D1
```

Obtenir le processus utilisant l'I/O

Retourne le numéro d'identification du processus qui utilise en ce moment le canal d'entrée/sortie indiqué en paramètre ; 0 si aucun processus n'a été trouvé.

Si le nom se rapporte à un canal qui se trouve sur une disquette, on peut obtenir en retour, au moyen de la fonction *IoErr()*, un pointeur sur la structure *Lock* du répertoire correspondant.

Exit

```
Exit(Parameter);
-144      D1
```

Mettre fin au programme

Le programme en cours est définitivement arrêté grâce à cette commande. Si le programme avait été appelé à partir du CLI, le contrôle lui est restitué, et la valeur entière transmise en tant que paramètre est interprétée comme valeur de retour. Si le programme avait été démarré en tant que processus, ce processus est supprimé par *Exit()*, et les domaines utilisés par lui comme pile, comme segment et dans mémoire des processus sont tous libérés.

Execute

```
Status = Execute(Commande, Input, Output);
  D0      -222     D1        D2     D3
```

Appeler une commande CLI

Exécute la commande CLI qui figure en paramètre et sur laquelle pointe D1. *'Input'* et *'Output'* indiquent les canaux dans lesquels on veut dériver l'entrée et la sortie de la commande CLI ; ces canaux doivent être précisés ici par leurs *handles*. Si l'on donne la valeur 0 pour l'un des deux, c'est le canal standard qui est utilisé.

Cette commande vous permettra de créer facilement votre propre CLI, ouvrant par exemple une fenêtre en propre, puis appelant *Execute()* avec le *handle* de la fenêtre comme *Input* et un texte vide de commande. Les commandes seront alors saisies dans la fenêtre, et les sorties seront elles aussi retournées dans la même fenêtre.

On pourra également mettre fin à ce CLI par la commande ENDCLI, à condition que le programme RUN se trouve dans le répertoire C:.

LoadSeg

```
Segment = LoadSeg(Nom);
  D0        -150    D1
```

Charger un fichier programme

Permet de charger en mémoire le fichier programme indiqué par son nom. Le programme est éventuellement distribué sur plusieurs modules en mémoire, s'il n'y a pas assez de place pour le recevoir de façon compacte. Les segments qui en résultent sont chaînés entre eux, grâce au premier élément de chaque segment, constitué par un pointeur sur le segment suivant de la liste. Si ce pointeur a la valeur 0, cela signifie qu'il correspond au dernier segment.

S'il se produit une erreur dans cette opération, tous les segments chargés en mémoire sont libérés, et on obtient en retour un 0 en D0. Sinon D0 contient un pointeur sur le premier segment.

Le programme que l'on vient de charger peut ensuite être démarré avec *CreateProc()* et de nouveau arrêté avec *UnLoadSeg()*.

UnLoadSeg

```
UnLoadSeg(Segment);
  -156      D1
```

Supprimer un fichier programme chargé en mémoire

Permet de supprimer le fichier programme chargé grâce à *LoadSeg()*, après quoi la mémoire utilisée est libérée. Le pointeur en D1 pointe sur le premier segment de la liste (cf. *LoadSeg*).

GetPacket

```
Status = GetPacket(Waitflag);
 D0        -162        D1
```

Rechercher un paquet

Permet de recevoir un paquet de données envoyé par un autre processus. Si le *Waitflag* est TRUE (-1), on attend l'arrivée du paquet ; dans le cas contraire, on n'attend pas, et la fonction retourne un 0 s'il n'y a pas de paquet.

QueuePacket

```
Status = QueuePacket(Paquet);
 D0        -168        D1
```

Envoyer un paquet de données

Permet d'envoyer le paquet de données dont la structure est pointée par D1. Si l'opération s'est déroulée sans encombre, on obtient en retour une valeur <>0 en D0.

3.1.4 MESSAGES D'ERREUR DU DOS

Vous trouverez dans la liste suivante les codes d'erreur transmis par *IoErr()* ou par la commande WHY du CLI, suivis par leur noms et leurs significations.

103 *insufficient free store*

Il n'y a pas assez de place libre en mémoire.

104 *task table full*

Il y a déjà 20 processus actifs, c'est le maximum.

120 *argument line invalid or too long*

La liste des arguments pour cette commande n'est pas correcte, ou elle contient trop de données.

121 *file is not an object module*

Le fichier appelé ne peut pas être manipulé.

122 *invalid resident library during load*

La bibliothèque résidente que vous appelez n'est pas valide.

202 *object in use*

Le fichier ou le répertoire indiqué est utilisé en ce moment par un autre programme, et il n'est pas disponible pour d'autres applications.

203 *object already exists*

Ce nom de fichier existe déjà.

204 *directory not found*

Le répertoire sélectionné n'existe pas.

205 *object not found*

Il n'existe pas de canal de ce nom.

206 *invalid window*

Le paramètre indiqué pour la fenêtre à ouvrir n'est pas correct.

209 *packet requested type unknown*

La fonction souhaitée ne peut pas être exécutée sur le périphérique indiqué.

210 *invalid stream component name*

Le nom de fichier n'est pas valide (trop long ou comportant des caractères non autorisés).

211 *invalid object lock*

La structure *Lock* indiquée n'est pas valide.

212 *object not of required type*

Confusion entre nom de fichier et nom de répertoire.

213 *disk not validated*

La disquette n'a pas encore été reconnue par le système, ou alors elle est défectueuse.

214 *dis[illegible]rite-protected*

La disquette est protégée contre l'écriture.

215 *rename across devices attempted*

La fonction RENAME n'est possible qu'à l'intérieur d'une disquette.

216 *directory not empty*

Le répertoire dont on demande la suppression n'est pas vide.

218 *device not mounted*

La disquette sélectionnée ne se trouve pas dans le lecteur.

219 *seek error*

La fonction *Seek()* est pourvue de paramètres non autorisés.

220 *comment too big*

Le commentaire est trop long.

221 *disk full*

La disquette est pleine, ou alors elle ne contient pas assez de place pour cette application.

222 *file is protected from deletion*

Le fichier ne peut pas être supprimé, ou bien il est protégé contre la suppression.

223 *file is protected from writing*

Le fichier est protégé contre l'écriture.

224 *file is protected from reading*

Le fichier est protégé contre la lecture. Pour les trois derniers messages d'erreur, vous pouvez examiner le statut des fichiers en question à l'aide de la commande LIST.

225 *not a DOS disk*

Cette disquette n'a pas été formatée en format *AmigaDOS*.

226 *no disk in drive*

Il n'y a pas de disquette dans le lecteur.

232 *no more entries in directory*

Le dernier appel de la fonction *ExNext()* n'a pas trouvé dans le répertoire de mention correspondant à ce qui est recherché.

3.2 Les disquettes

Le fonctionnement de l'Amiga est fortement orienté vers les disquettes, ce qui veut dire qu'il lui arrive souvent de charger ce dont il a besoin. C'est pourquoi il importe que les informations soient placées en sécurité sur une disquette, et qu'elles puissent être retrouvées assez rapidement. Nous allons ici nous occuper de la division de la disquette, et de l'interprétation des données qu'elle porte.

La division fondamentale d'une disquette est la suivante :

* Page ou numéro de tête (0 ou 1)

* Sillon ou cylindre (0 à 79)

* Secteur (0-10)

Les disquettes Amiga sont toujours traitées sur les deux faces, ce qui fait que l'on dispose de deux pages.

Chaque page de la disquette est à son tour divisée en 80 sillons, ordonnés en anneaux concentriques. Le sillon extérieur porte le numéro 0, le sillon intérieur le numéro 79. Les sillons sont aussi appelés cylindres.

Chaque sillon, enfin, est lui-même partagé en 11 secteurs, numérotés eux aussi à partir de 0. Ces secteurs sont parfois désignés comme des blocs. Au total les secteurs sont numérotés de 0 à 10, mais les blocs le sont de 0 à 1759, cette dernière numérotation correspondant aux numéros logiques des secteurs de la disquette.

Chacun des secteurs contient 512 octets d'informations disponibles, ce qui fait que chaque disquette peut porter 512*11*80*2 octets. Tous ne sont cependant pas à la disposition de vos données, étant donné que la gestion de ces données exige un peu de place pour elle-même.

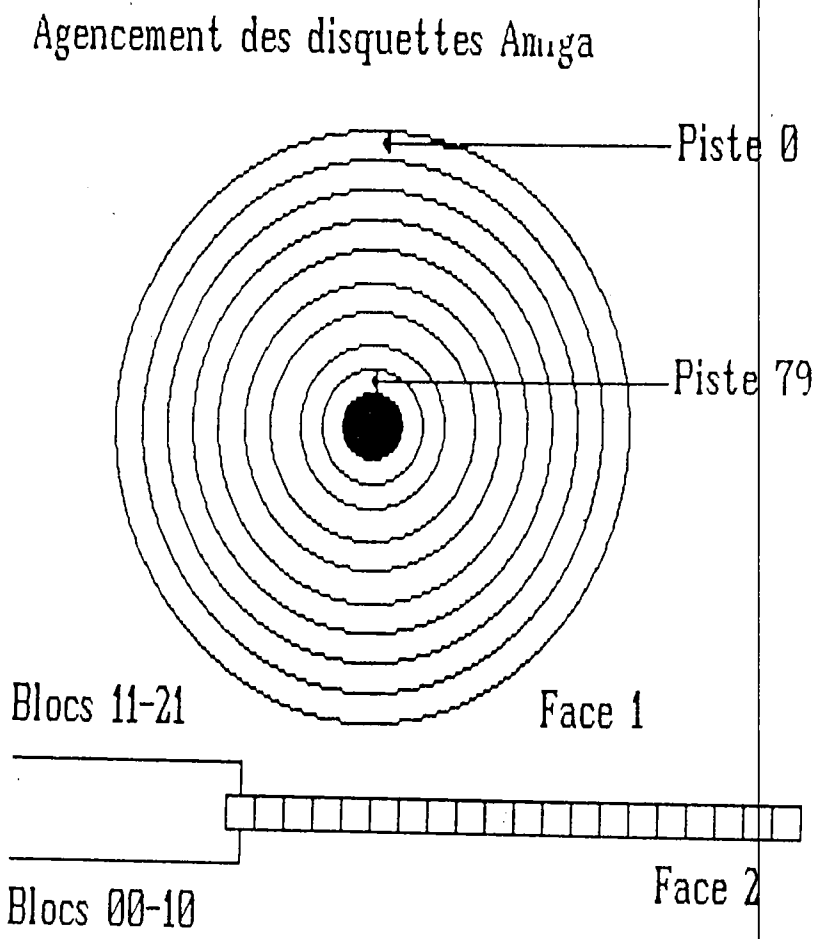


Figure 3.1

Le premier secteur logique, donc le premier bloc sur la disquette, se trouve à la page 0, sillon 0, secteur 0. Le bloc suivant est le secteur suivant de ce sillon, et ainsi de suite. Le bloc 11, par contre, n'est pas le premier secteur du deuxième sillon, mais le premier secteur du premier sillon sur l'autre face (page 1) de la disquette. De cette manière, la disquette est lue ou écrite alternativement sur l'une et l'autre face.

3.2.1 L BOOTAGE

Le premier contact avec la disquette a lieu dès que l'on allume l'Amiga. Un certain nombre d'opérations d'initialisation de l'ordinateur sont d'abord effectuées sans passer par le logiciel, et aussitôt après le lecteur 0 se met en marche. Que se passe-t-il là ?

Que le *Kickstart* soit intégré ou non dans l'Amiga, quelque chose est chargé à partir de la disquette qui se trouve dans ce lecteur. Si le *Kickstart* n'est pas intégré, l'ordinateur le charge à partir de la disquette. Autrement, il recherche la disquette *Workbench*. Ce chargement au moment de la mise en marche de l'ordinateur est appelé *'bootage'* (prononcer 'boutage').

Ce qui est chargé en premier lieu de la disquette insérée, ce sont les blocs de bootage, qui occupent les deux premiers secteurs (0 et 1) de la disquette. C'est là que se trouve l'information disant de quel type de disquette il s'agit. On peut avoir les types suivants :

* Kisckstart
* DOS, éventuellement une disquette DOS chargeable (Workbench disk)
* Disquette non formatée, ou formatée dans un format étranger

Les quatre premiers octets du premier bloc de la disquette indiquent donc de quel type il s'agit. C'est là que se trouvent en effet les lettres DOS conclues par un 0 si l'on a affaire à une disquette DOS, ou les lettres KICK pour une disquette *Kickstart*. S'il n'y a ni l'un ni l'autre, il s'agit d'une disquette étrangère (BAD).

Les quatre octets suivants constituent un mot long, et représentent la somme du bloc de bootage, émise pour vérification. Si la somme est correcte, l'Amiga conclut qu'il s'agit bien d'une disquette *Workbench*.

Le mot long suivant contient le numéro du bloc de la disquette, nommé *bloc racine*, et placé normalement dans le bloc $370.

Nous allons parler tout de suite après de signification de ce bloc. Restons-en pour l'instant au bloc de bootage.

A partir du 7ème mot, on trouve un programme. Si la somme servant de contrôle est correcte, ce programme démarre. Il reçoit en A6 un pointeur sur l'adresse de base Exec, et il peut donc exécuter correctement une commande Exec.

Bloc de bootage (secteur 0)

Mot Long	Nom	Contenu	Signification
0	Disk type	DOS, KICK	Type de la disquette sur 4 caractères
1	Contrôle	???	Somme de contrôle du bloc
2	Bloc racine	$370	Numéro de bloc du bloc racine
3-127	Données		Programme de bootage

Le programme qui se trouve habituellement ici teste au moyen de la commande *FindResident()* d'Exec, si la bibliothèque DOS est résidente. Si ce n'est pas le cas, la valeur -1 est retournée dans le registre D0. Si c'est le cas, un 0 est retourné en D0 et en A0 un pointeur sur la routine d'initialisation du DOS.

Avec un programme approprié, on peut configurer ici tout le processus de bootage et donc l'initialisation de l'Amiga. Vous avez donc la possibilité de configurer une *'disquette Workbench'* concoctée par vos propres soins, étant donné qu'il y a des programmes moniteurs de disquettes qui peuvent se charger de créer la somme de contrôle. Cette somme de tous les mots du bloc est essentielle, car c'est grâce à elle que l'Amiga reconnaît ce bloc comme bloc racine.

3.2.2 ST. CTURES DES FICHIERS ET DISTRIBUTION DES DONNEES

Pour entreposer des données sur une disquette dont la capacité est d'environ 800 Koctets, de telle façon que l'on puisse ensuite retrouver ces données, il existe quelques règles concernant leur distribution. Ces règles sont évidemment connues de l'Amiga DOS, si bien que l'on n'a pas vraiment besoin d'en prendre connaissance. Mais s'il se produit une erreur sur une disquette, encore faut-il pouvoir sauver les données restantes.

Le DISKDOCTOR est justement prévu à cet effet, et l'Amiga le recommande même dans un *Requester* lorsqu'une erreur se produit. Pour comprendre en cas de besoin comment fonctionne ce sauveur, il faut en passer par quelques remarques approfondies sur les formats de disquettes.

L'un des points importants est la distribution des fichiers sur la disquette, ainsi que la technique du catalogue.

3.2.2.1 Division de la disquette

Au contraire de se qui passe pour la plupart des formats de disquette, le catalogue des disquettes Amiga ne se trouve pas dans des secteurs qui se font suite. C'est pourquoi l'affichage du répertoire (par exemple avec la commande DIR du CLI) dure aussi longtemps.

Cette méthode a ses avantages et ses inconvénients. L'inconvénient principal est que l'accès au catalogue peut durer assez longtemps, ce qui peut à la longue porter sur les nerfs de l'utilisateur. Cet inconvénient est cependant contrebalancé par un grand avantage : la possibilité de *'réparer'* une disquette endommagée.

Si, pour une disquette provenant d'un autre système, par exemple ATARI ST, il se produit une erreur en particulier dans le secteur 0, là où se trouve l'ensemble du catalogue de la disquette, on a à faire face à de gros problèmes.

En effet, dans ce cas, la position des dor es et la connexion des secteurs ne peuvent pas être retrouvées, et il faut beaucoup de travail pour sauver les fichiers.

Ce n'est pas le cas avec Amiga. Comme le laisse entendre l'existence de DISKDOCTOR, les fichiers sont assez faciles à retrouver, sans que l'on ait besoin d'un emplacement central contrôlant leur distribution. On y arrive grâce à des redondances importantes, qui prennent certes de la place sur les disquettes, mais qui procurent en revanche une grande sécurité d'emploi.

Comment est-ce que cela fonctionne ? Pour avoir une idée générale de la distribution des données, il nous faut prendre en compte la structure des différents secteurs de la disquette.

Bloc racine (root block)

Indépendamment des secteurs de bootage, il existe sur la disquette, à un endroit déterminé, le *bloc racine.* Celui-ci se trouve habituellement à la page 0, track 40, secteur 0, et il porte donc le numéro 880 ($370). Le troisième mot long du secteur de bootage contient ce même numéro.

C'est dans ce bloc que se trouve la racine de la disquette en son entier. On trouve ici le répertoire principal et sa date de création. Ce bloc est constitué de la façon suivante (toutes les valeurs correspondent à des mots longs, donc à des ensembles de 4 octets) :

Bloc racine

Mot	Nom	Contenu	Signification
0	Type	2	Type 2 (T.SHORT) signifie que ce bloc est le bloc initial d'une structure.
1	Header Key	0	N'a pas de signification.
2	High Seq	0	N'a pas de signification.
3	HT-Size	$48	Taille du tableau (hashtable) portant mention des blocs initiaux des fichiers ou des sous-répertoires reliés par une chaîne.

4	r rvé	0	N'a pas de signification.
5	somme contr.	???	Contient une valeur qui annule la somme de tous les mots du bloc.
6	hashtable		Ici commence la table où se trouvent les blocs initiaux des fichiers ou des sous-répertoires.
78	BM-Flag	-1	Ce flag contient -1 (TRUE) si le bitmap de la disquette est valide.
79	BM-Pages		La table suivante contient des pointeurs sur les blocs contenant le bitmap. Il s'agit le plus souvent d'un bloc unique, ce qui fait que les autres pointeurs sont nuls.
105	days		Contient la date à laquelle la disquette a été modifiée pour la dernière fois.
106	mins		Heure de la modification.
107	ticks		Secondes de la modification.
108	disk name		Contient le nom de la disquette comme chaine BCPL: le premier octet représente le nombre de caractères du nom (max. 30).
121	create days		Date de la création de la disquette.
122	create mins		Heure de création.
123	create ticks		Seconde de création.
124	next hash	0	Toujours nul.
125	parent dir	0	Pointeur sur le répertoire immédiatement supérieur : toujours nul.
126	extension	0	Toujours nul.
127	sec.type	1	Ce mot représente le type secondaire du bloc. 1 pour le bloc racine.

Les valeurs portées dans le *hashtable* indiquent les blocs où commencent les chaînes de fichiers ou de sous-répertoires à l'intérieur du répertoire *racine.* Etant donné qu'il n'y a pas assez de place dans la table pour que toutes les valeurs puissent y entrer, les fichiers et les sous-répertoires sont constitués en chaînes, leurs noms étant ordonnés d'une certaine façon.

A partir de ces noms, on calcule une valeur entre 6 et 77. Cette valeur permet alors d'avoir accès au mot long du tableau *hash,* là où la chaîne commence.

La fonction qui permet de calculer cette valeur est la suivante :

```
hash = longueur du nom
pour chaque caractère du nom:
Hash=Hash *13
      Hash=Hash +valeur ASCII du caractère (toujours majuscule)
      Hash=Hash & $7FF (ET logique)
Hash=Hash modulo 72
Hash=Hash +6
```

Vous verrez dans le chapitre sur le *'trackdisk-device'* comment on programme ceci. Vous y trouverez un programme en langage machine pour déterminer les valeurs de la table *hash*.

Le début de la chaîne se trouve donc là où pointe le pointeur que l'on vient de déterminer ainsi dans la table *hash*. On a de cette façon le bloc, dans lequel se trouve, au 124ème mot long, le numéro de la prochaine mention de la chaîne ; et l'on continue ainsi jusqu'à ce que l'on ait atteint la fin de la chaîne, signalée par un pointeur de valeur 0.

Ces blocs, qui constituent le début d'une structure de fichier ou de répertoire, ont eux aussi une structure particulière. Commençons par le premier bloc d'un fichier, le *file-header-block*.

File-header-block

Ce bloc contient les informations sur le fichier correspondant. Ces informations sont : le nom du fichier, la date de création, un commentaire ainsi que la taille et la distribution du fichier sur la disquette. Le bloc est constitué de la façon suivante :

File-header-block

Mot	Nom	Contenu	Signification
0	Type	2	Type 2 (T.SHORT) signifie que ce bloc est le bloc initial d'une structure.
1	header Key		Numéro du bloc.

2	h[illegible]equ		Contient le nombre total de blocs de ce fichier.
3	data size	0	
4	first data	0	Contient le numéro du premier bloc de données du fichier. Cette valeur se trouve aussi dans le mot n° 77.
5	somme contr.	???	Contient une valeur qui annule la somme de tous les mots du bloc.
6	data blocks		Ici commence la table où sont reportés les blocs de données du fichier. La table commence au mot 77 et la numérotation se fait ensuite vers l'arrière.
78	réservé	0	
79	réservé	0	
80	protect		Ce mot, dans ses 4 bits inférieurs, contient les informations sur l'état du fichier :

Bit	Protégé contre
0	l'effacement
1	la modification
2	l'écriture
3	la lecture

81	byte size		Longueur du fichier en octets.
82	comment		Début du commentaire sur le fichier, en string BCPL (max. 22 caractères).
105	days		Contient la date de création du fichier.
106	mins		Heure de la création.
107	tricks		Seconde de la création.
108	nom fichier		Nom du fichier, en string BCPL : le premier octet représente le nombre de caractères du nom (max. 30).
124	hash chain		N° de bloc du fichier suivant de la chaine, ou 0.
125	parent		Pointeur sur le répertoire, dans lequel ce fichier apparait.
126	extension	0	Pointeur sur le bloc d'extension, ou 0.
127	sec.type	-3	Représente le type secondaire du bloc. -3 ($FFFD) pour le File-header-block.

L'entrée n° 126 est toujours différente de 0 lorsque la table du bloc des données n'est pas assez longue pour faire figurer tous les blocs de ce fichier. Si c'est le cas, on trouve en cet endroit un pointeur sur un bloc où la liste se poursuit.

File-List-Block

Ce bloc, où la liste se poursuit, est appelée *bloc d'extension (file list block)*, et il est constitué de la façon suivante :

File list block

Mot	Nom	Contenu	Signification
0	Type	$10	(T.LIST) signifie que ce bloc est le bloc d'extension d'une structure de fichier.
1	header key		N° de bloc.
2	high seq		Contient le nombre total des entrées dans la table des blocs de données.
3	data size	0	
4	first data	0	N° du premier bloc de données du fichier. Se trouve aussi dans le mot n° 77 du File header block.
5	somme contr.	???	Contient une valeur qui annule la somme de tous les mots du bloc.
6	data blocks		Ici commence la table où sont reportés les blocs de données du fichier. La table commence au mot 77 et la numérotation se fait ensuite vers l'arrière.
78	info	0	Réservé.
124	hash chain	0	N° de bloc du prochain fichier de la chaîne (toujours 0).
125	parent		Pointeur sur le file header block.
126	extension	0	Pointeur sur le bloc d'extension ou 0.
127	sec.type	-3	Représente le type secondaire du bloc.

L'autre possibilité d'un *start-block* est le bloc répertoire de l'utilisateur *(user directory block)*, qui se trouve au début d'une structure de répertoire.

User directory block

Chaque sous-répertoire commence par un bloc de ce type, constitué de manière semblable au bloc racine :

Mot	Nom	Contenu	Signification
0	type	2	Type 2 (T.SHORT) signifie que ce bloc est le bloc initial d'une structure.
1	header key		N° de bloc.
2	high seq	0	N'a pas de signification.
3	HT size	0	N'a pas de signification.
4	réservé	0	N'a pas de signification.
5	somme contr.	???	Contient une valeur qui annule la somme de tous les mots du bloc.
6	hashtable		Ici commence la table où se trouvent les blocs initiaux des fichiers ou des sous-répertoires.
78	réservé	0	N'a pas de signification.
80	protect		Ce mot, dans ses 4 bits inférieurs, contient les informations sur l'état du fichier :
			Bit — Protégé contre
			0 — l'effacement
			1 — la modification
			2 — l'écriture
			3 — la lecture
81	réservé	0	Sans signification.
82	comment		Ici commence le commentaire sur ce sous-répertoire, comme string BCPL (max. 22 caractères).

105	days		Contient la à laquelle la disquette a été modifiée pour la dernière fois.
106	mins		Heure de la modification.
107	ticks		Secondes de la modification.
108	dir.name		Contient le nom de la disquette comme string BCPL: le premier octet représente le nombre de caractères du nom (max. 30).
124	next hash		Toujours nul.
125	parent dir		Pointeur sur le répertoire immédiatement supérieur : toujours nul.
126	extension	0	Toujours nul.
127	sec.type	2	Ce mot représente le type secondaire du bloc. 1 pour le bloc racine.

En dehors de ces blocs de structure, il y a évidemment aussi des blocs de données sur la disquette. Ces blocs de données ont une forme extrêmement simple :

Data-block

Mot	Nom	Contenu	Signification
0	type	8	Type 8 (T.DATA) signifie qu'il s'agit d'un bloc de données.
1	header key		N° de bloc.
2	seq num		N° du bloc de données dans ce fichier.
3	data size	$1E8	Mots valides de ce bloc de données ($1E8 ou moins).
4	next data		N° du bloc de données suivant de ce fichier.
5	somme contr.	???	Contient une valeur qui annule la somme de tous les mots de ce bloc.
6	data		Les données proprement dites commencent ici.

Il ne manque plus que le bloc contenant le bitmap mentionné dans le bloc racine. Il y a ici pour chacun des blocs de la disquette un bit indiquant si le bloc correspondant est libre ou s'il est occupé. Il est constitué de façon très simple :

Bitmap block

Mot	Nom	Contenu	Signification
0	somme contrôle	???	Somme de contrôle du bloc.
1-55	motif de bits		Tableau des bits occupés pour tous les blocs. Le bit 0 du premier mot long est mis pour le bloc, et ainsi de suite. Un bit posé signifie que le bloc est libre.

C'est tout pour la distribution des fichiers sur la disquette. Nous allons maintenant envisager la structure générale d'un tel fichier.

3.2.2.2 Structure d'un programme

Un programme n'est en réalité rien d'autre qu'une série de mots-données en binaire, qui constituent dans leur ensemble un programme en langage machine. Sur les bons vieux ordinateurs à 8 bits, la mémorisation de ces programmes ne posait aucun problème : il suffisait de copier sur la disquette le programme se trouvant en mémoire centrale et le tour était joué.

Sur une machine comme l'Amiga, cette méthode est impraticable. Le premier problème qui surgirait concernerait en effet la division de la mémoire. Si la partie de la mémoire où le programme a fonctionné la première fois est occupée lors de l'appel suivant, un simple chargement du programme sans autre forme de procès n'est plus possible. Le programme doit être chargé en un autre point de la mémoire, ce qui signifie que rien ne va plus, puisqu'un programme en langage machine qui se respecte n'utilise que des adresses absolues.

Le problème est résolu par Amiga de la fa[illegible] suivante : un programme est emmagasiné sur la disquette de telle f[illegible]on que toutes les adresses absolues qui y sont contenues soient décalées (elles sont en fait interprétées comme adresses relatives par rapport à une adresse de base $00000). Ainsi, si le programme est chargé à partir de $20000, il faut que toutes les adresses qu'il contient soient corrigées avant le démarrage, c'est-à-dire augmentées de $20000. Pour que le DOS, qui se charge de cette opération, puisse retrouver dans le programme toutes les adresses à modifier, on mémorise une table en même temps que le programme proprement dit. Cette table contient les offsets pointant sur les mots longs à modifier.

Il est donc clair qu'une seule section ne saurait suffire à emmagasiner un programme sur une disquette. Jusqu'à présent, nous sommes en présence de deux sections : le programme lui-même et la table de redisposition *(relocation table)*. Mais il y a toute une série d'autres sections utilisées par l'Amiga. Une partie de programme composée de ces différentes sections est appelée *"hunk"*. Un ou plusieurs *"hunks"* constituent ensemble une unité de programme *(program unit)*, et plusieurs de ces unités forment un *"object file"*, un *fichier objet*. En dernier lieu, un ou plusieurs fichiers objets constituent un *"load file"*, un *fichier de chargement*, qui représente un programme en état de marche.

La différence entre ces deux types de fichiers (*"object file"* et *"load file"*) est la suivante : un fichier objet contient un programme qui n'est pas encore en état de fonctionner, tel qu'il a été créé par exemple par un compilateur ou un assembleur. Si l'on veut transformer un ou plusieurs de ces fichiers objets en un programme qui soit en mesure de fonctionner, il faut appeler l'éditeur de liens *(linker)*. C'est un programme qui rassemble les fichiers objets pour en faire un programme unique, qui est lui-même ensuite emmagasiné en tant que fichier de chargement, ou *load file*. Le résultat peut alors être utilisé, si on le fait démarrer par exemple par la saisie de son nom dans le CLI.

L'avantage de ce détour est que des parties de programmes qui risquent de s'appeler réciproquement sont créées et traduites indépendamment l'une de l'autre. Le programme principal, écrit par exemple en C et contenant la routine *"main"*, peut simplement appeler les fonctions ou les sous-programmes contenus dans les autres fichiers.

Le fait [illegible] les fonctions soient entreposées hors du programme principal [illegible]améliore la vue d'ensemble que l'on peut avoir du programme, puisque celui-ci est nettement plus court qu'un programme contenant toutes les composantes développées l'une à la suite de l'autre.

On comprend maintenant pourquoi les programmes pris un à un ne peuvent pas fonctionner. On doit appeler des parties de programme qui ne sont pas explicitement contenues dans ces programmes. Ce n'est qu'à l'aide de l'éditeur de liens que toutes les fonctions et tous les sous-programmes sont réunis dans un fichier programme unique.

Nous allons cependant commencer par les plus petites sections qui constituent les *hunks*. Quelques-unes de ces parties de fichiers programmes apparaissent seulement dans les fichiers objets, quelques autres seulement dans les fichiers de chargement. Elles commencent toutes par un mot long déterminé, indiqué entre parenthèses dans le tableau qui suit en hexadécimal.

Voici cette liste de toutes les parties (*hunks*) possibles :

hunk_unit ($3E7)

Début d'une unité de programme dans les fichiers objets. La caractérisation $3E7 est suivie de la longueur du nom de cette unité, puis du nom lui-même, qui doit se terminer sur une fin de mot long.

hunk_name ($3E8)

Ici se trouve le nom du *hunk*. La caractérisation $3E8 est suivie de la longueur du nom, puis du nom lui-même, qui doit se terminer sur une fin de mot long.

hunk_code ($3E9)

Contient une partie de programme pouvant fonctionner après la correction des adresses absolues.

Ici encore, la caractérisation $3E9 est suivie nombre de mots longs dans le programme, puis de ces mots longs ux-mêmes.

hunk_data ($3EA)

Ici aussi commence une partie de programme, ne contenant cependant que des données avec un contenu déterminé (data). Quelques-unes d'entre elles peuvent exiger une correction d'adresse. La caractérisation est suivie du nombre de données puis des données elles-mêmes.

hunk_bss ($3EB)

Les données de cette partie appartiennent certes elles aussi au programme lui-même, mais elles n'ont pas cette fois de contenu déterminé. C'est pourquoi la caractérisation n'est suivie que du nombre des mots longs nécessaires, mais sans que les données soient répertoriées.

hunk_reloc32 ($3EC)

Ce bloc contient les offsets qui pointent sur les mots longs contenant les adresses à corriger à l'intérieur du programme. Ces offsets sont valables pour l'ensemble du programme. La distribution du bloc est la suivante :

La caractérisation $3EC est suivie du nombre d'offsets contenus dans le premier tableau. Le mot long suivant désigne le numéro du *hunk* auquel ces offsets se réfèrent ; puis viennent les offsets eux-mêmes. Le mot long suivant est de nouveau un nombre ; puis vient le numéro de *hunk* de ce tableau, et ainsi de suite, jusqu'à ce qu'il apparaisse le nombre 0, qui conclut cette partie de *hunk*.

Voici le dir nme d'ensemble :

* $3EC (hunk_reloc32)

* Nombre d'Offsets
* Numéro de Hunk
* Offsets...

* Nombre d'Offsets (ou 0: fin)
* Numéro de Hunk
* Offsets...

.
.
.

* 0: Fin de hunk_reloc32

De cette manière, ce tableau recouvre tous les *hunks* qui constituent le programme contenu finalement en mémoire et prêt à fonctionner.

hunk_reloc16 ($3ED)

Ce tableau est constitué comme *hunk_reloc32*, à ceci près que ces offsets se réfèrent à des adresses 16 bits. Ce type d'adresses apparaît dans les adressages relatifs aux PC.

hunk_reloc8 ($3EE)

A également le même format que *hunk_reloc32*. Les offsets contenus dans celui-ci sont utilisés pour des adresses 8 bits, qui apparaissent également dans les adressages relatifs aux PC.

hunk_ext ($3EF)

Dans ce bloc sont reportés les noms des références externes. De telles références n'apparaissent que dans les fichiers objets.

Il s'agit des adresses de fonctions ou de sou· ·utines, qui ne sont pas connues du programme et qui doivent être introduites par le *linker*.

La caractérisation est suivie de plusieurs *"symbol data units"*, conclues par un mot contenu la valeur 0. Ces définitions de symboles ont la structure suivante :

1 octet : type du symbole.

Les différentes possibilités sont les suivantes :

Nom	Valeur	Type de symbole
ext_symb	0	Table de symboles pour recherche d'erreurs
ext_def	1	Définition à corriger
ext_abs	2	Définition absolue
ext_res	3	Relation à la bibliothèque résidente
ext_ref32	129	Correction 32 bits
ext_common	130	Correction générale 32 bits
ext_ref16	131	Correction 16 bits
ext_ref8	132	Correction 8 bits

Une valeur en 3 octets pour la longueur du nom (en mots longs).

Nom du symbole.

Valeur du symbole et éventuellement d'autres données.

hunk_symbol ($3F0)

Dans ce bloc, on trouve encore des symboles avec leur nom et leur valeur. Mais ceux-ci n'intéressent pas le *linker*, mais un *débogueur*, c'est-à-dire un programme servant à rechercher les erreurs dans les programmes. On peut ainsi tester une routine dont l'adresse n'est pas disponible en tant que nombre mais en tant que nom, et aussi les emplacements en mémoire des variables. La caractérisation $3F0 est suivie à nouveau des *symbol-data-units*, et est conclue par un 0.

hunk_debu .3F1)

La structure de ce bloc n'est pas prescrite entièrement. On peut inscrire ici des informations sur le programme, et le programme les aura à sa disposition lorsqu'il s'agira de rechercher des erreurs. La seule prescription est que le bloc doit commencer par la caractérisation $3F1, à la suite de quoi doit venir le nombre de mots longs constituant le bloc.

hunk_end ($3F2)

Celui-ci est le seul bloc, parmi tous les *hunks*, qui soit absolument nécessaire. Il est constitué de sa seule caractérisation, qui est également le dernier mot long d'un programme qui se trouve sur la disquette.

hunk_header ($3F3)

Ce bloc est le début d'un *load file*. On indique ici le nombre de *hunks* qui composent le programme à charger, et la dimension de chacun d'eux. Le bloc contient en outre les noms des bibliothèques résidentes, qui doivent être chargées en même temps que ce programme. Le bloc est constitué de la façon suivante :

* hunk_header ($3F3)
* longueur (nombre de mots longs) du nom du premier hunk
* nom du hunk
* longueur du nom du deuxième hunk (ou 0: fin)
* nom du hunk

.
.
.

* 0: fin de la liste des noms
* numéro de hunk le plus élevé+1: longueur de la table

* numéro du hunk à charger en ..mier
* numéro du hunk à charger en dernier

* ici commencent les hunks du programme

hunk_overlay ($3F5)

Ce bloc est nécessaire lorsque l'on veut travailler en *overlay*. Cela signifie que, dans un domaine de la mémoire déjà occupé par un programme, on veut charger un autre segment de programme ou de données. Après la caractérisation, la table contient les indications sur sa longueur, sur le niveau le plus élevé des recouvrements (le nombre de processus de recouvrement), et sur les données à charger.

hunk_break ($3F6)

Ce numéro d'identification indique à lui seul la fin d'une partie de programme en *overlay*.

Pour que nous apprenions à nous y retrouver dans ce dédale de la distribution des programmes, voici maintenant un petit exemple. Dans le chapitre sur le CLI, je vous ai présenté un petit programme en langage machine, qui correspondait à la commande FONT en CLI. Vous vous rendrez facilement compte de la façon dont ce programme (à partir de l'assembleur SEKA) est amené sur la disquette, en demandant l'affichage du contenu du fichier *'Font'* à l'aide de la commande TYPE en CLI. L'instruction est la suivante :

```
>type Font opt h
```

pour obtenir le résultat à l'écran, ou :

```
>type Font PRT: opt h
```

pour l'obtenir sur l'imprimante. A la sortie écran ou imprimante, voici ce que vous aurez :


```
00(    000003F3  00000000  00000002  00000000
0010:  00000001  00000024  00000001 / 000003E9
0020:  00000024 / 53406700  00180C18  00206600
0030:  000A51C8  FFF66000  000813E0  0000007F
0040:  2C790000  000443F9  00000072  70004EAE
0050:  FE6823C0  00000088  67000028  2C790000
0060:  00884EAE  FFC42200  243C0000  007E263C
0070:  00000009  2C790000  00884EAE  FFD06000
0080:  0008203C  FFFFFFFF  22002C79  00000088
0090:  4EAEFF70  4E75646F  732E6C69  62726172
00A0:  79009B30  3B333133  34306D00  00000000
00B0:  3B4F7065 / 000003EC  00000007  00000000
00C0:  00000018  00000024  00000030  0000003A
00D0:  00000046  00000052  00000068  00000000 /
00E0:  000003F2 / 000003EB  00000001 / 000003F2
```

Les barres obliques ne sont affichés ou imprimés ; elles ne servent ici qu'à séparer les différentes sections, c'est-à-dire les parties de *hunks*. Considérons-les plus en détail :

Au début, on trouve le numéro caractéristique $3F3, *hunk_header*. Le $0 qui suit signifie qu'il n'y a pas de nom de *hunk*. Puis le $2 indique que ce fichier programme ne se compose que de 2 *hunks* (comme nous l'avons dit, ce n'était qu'un petit exemple). Le premier *hunk* à charger est le numéro $0, le dernier est le numéro $1. Les tailles respectives de ces *hunks* sont $24 et $1.

Avec le numéro caractéristique $3E9 *(hunk_code)* commence ensuite la partie dans laquelle se trouvent les données des programmes. La longueur est indiquée par $24. Puis viennent $24 (36) mots-longs, qui constituent le code programme.

En $3EC *(hunk_reloc32)* commence le domaine contenant les offsets destinés à la correction des adresses. Le nombre d'adresses à corriger est indiqué par $7, et ces offsets se rapportent au *hunk* numéro $0. Puis viennent les 7 offsets. Le premier d'entre eux, $18, porte la valeur $0000007F, le 24ème mot du code. C'est donc à ce mot long que sera ajoutée, après le chargement du programme, son adresse de début, de

telle sorte qu'on y trouve l'adresse effec en mémoire de l'octet adressé. Le procédé est le même pour les autres offsets de la liste *reloc_32*.

La liste est suivie d'un $0, qui indique la fin de cette liste.

La caractérisation suivante, $3F2 *(hunk_end)* note la fin du premier *hunk*. Puis vient le second, dont la longueur était d'une unité.

Vient ensuite le numéro $3F2 *(hunk_bss)*, suivi par le nombre $1. Ceci signifie qu'il faut réserver un mot long supplémentaire, dans lequel le programme placera l'adresse de base du DOS.

L'ensemble est conclu par le numéro $3F2 *(hunk_end)*, qui représente la fin du second *hunk* et donc également du programme tout entier.

3.2.2.3 Le format IFF

IFF signifie *Interchange File Format*. C'est un format dans lequel différents fichiers de données sont moulés, pour pouvoir être lus et traités comme il se doit par n'importe quel programme.

En fait, l'IFF ne fait pas vraiment partie des sujets à traiter dans une **"Bible de l'Amiga"**, étant donné qu'il n'appartient pas aux composantes d'Amiga. Il a été développé par la société *Electronic Arts*, et il s'est imposé en tant que standard, ce qui fait qu'on le retrouve maintenant partout. C'est aussi pourquoi il faut dire ici quelques mots de sa structure.

Pourquoi a-t-on besoin d'un format standardisé ? Imaginez donc une image, peinte sur l'écran de l'Amiga à l'aide d'un programme quelconque. Cette image est constituée par un certain nombre de données, qui font apparaître à l'écran les différentes couleurs.

Si vous entreposez simplement ces données sur la disquette, vous aurez des problèmes déjà pour les recharger : quelle est donc la taille de l'image sur laquelle ces données doivent être distribuées ? Quelles sont les teintes qui doivent être utilisées ?

Comme vc .e voyez, les données constitutives de l'image ne sont pas tout. Il vous faut entreposer quelques autres informations dans le fichier, et vous devez le faire de telle façon qu'un autre programme puisse les retrouver.

Ce problème a été résolu grâce à l'introduction de l'IFF. Celui-ci permet de distribuer les données et de les mémoriser de façon uniformisée. Chacun des différents blocs de données reçoit à cet effet un *header*, plus un préambule constitué d'un mot de 4 caractères et d'un mot-données contenant la longueur du bloc.

Le début d'un fichier IFF est constitué par le mot FORM, ce qui signifie qu'ici commence un secteur de données correspondant à une forme d'application déterminée (texte, image, etc...). Le mot long qui suit indique la longueur de la forme à venir. Cette *"forme"* est une combinaison de quelques blocs de données, nommés *"chunks"*. Théoriquement, un fichier IFF peut être constitué de plusieurs formes, lorsqu'on combine par exemple un fichier texte et un fichier image. Mais habituellement, un tel fichier n'est constitué que d'une seule forme, car il ne s'agit le plus souvent que d'un fichier texte, d'un fichier image ou d'un fichier son.

Le mot FORM est donc suivi d'un mot long contenant la longueur de cette forme en octets, correspondant le plus souvent à une longueur de fichier de 8 octets. Puis vient un autre mot de 4 caractères, qui indique le type du fichier (ILBM, WORD, etc...).

Le type est immédiatement suivi de premier *chunk*, puis de la longueur de celui-ci. Ensuite vient le *chunk* suivant, reporté éventuellement à une adresse à l'aide d'un octet entier ; et ainsi de suite jusqu'à ce que l'on atteigne la fin de la forme, et la plupart du temps aussi la fin du fichier.

En résumé :

'FORM'	:	début d'une forme IFF
longueur	:	longueur de la forme en octets
type	:	caractérisation du fichier, ILBH, WORD, SMUS ou 8SVX

nom chunk : par exemple FORM, NAME, AUTH, BODY
longueur chunk : longueur du chunk indiquée en octets

etc...

Il existe à ce jour un très grand nombre de *chunks* possibles. Voici une vue d'ensemble des principaux, avec leur caractérisation et leur signification :

Type de fichier : fichier texte (WORD)

BODY	partie principale des données pour images
COLR	couleurs du texte
DOC	type de texte
FOOT	ligne de bas de page
FONT	polices de caractères utilisées
FSCC	information sur la couleur du texte
HEAD	ligne d'en-tête
PARA	informations sur le gabarit (marges, etc.)
PCTS	informations sur les images à intégrer
PINF	informations sur les images elles-mêmes
TABS	informations sur les tabulateurs
TEXT	texte proprement dit

Type de fichier : fichier graphique (ILBM)

BMHD	données de contrôle des graphiques
CMAP	table des couleurs
BODY	données graphiques

Type de fichier : fichier musique (SMUS)

SHDR	données de contrôle du son (rythme, volume, canal)
NAME	nom du morceau
(c)	copyright
AUTH	auteur

ANNO	remarques sur le morceau
TRAK	indication du canal

Type de fichier : fichier son 8 bits digitalisé (8SVX)

VHDR	données de contrôle (type, rythme, octave, volume)
NAME	nom du morceau
(c)	copyright
AUTH	remarques
ANNO	données du morceau
ATAK	informations d'attaque
RLSE	informations de release

Si vous voulez sauvegarder un fichier IFF sur une disquette, il peut être utile de connaître sa distribution sur la disquette. C'est pourquoi vous allez trouver maintenant un petit programme en langage machine, qui affiche dans une fenêtre toutes les caractéristiques avec leurs longueurs. Le nom du fichier est inscrit directement dans le programme.

```
;*****  Programme Démo IFF  6/87  S.D.  *****

OpenLib  = -408
closelib = -414
ExecBase = 4

Open     = -30
Close    = -36
Seek     = -66
Read     = -42
Write    = -48
mode_old = 1005

key = $bfec01                ;Statut des touches spéciales

run:
     move.l execbase,a6      ;Pointeur sur la bibliothèque Exec
     lea    dosname(pc),a1
```

```
        moveq  #0,d0
        jsr    openlib(a6)      ;Ouvrir Library uOS
        move.l d0,dosbase
        beq    error

        move.l #consolname,d1   ;Définition de Consol
        move.l #mode_old,d2
        move.l dosbase,a6
        jsr open(a6)            ;Ouvrir la fenêtre CON:
        beq error
        move.l d0,conhandle

        move.l #filename,d1
        move.l #mode_old,d2
        move.l dosbase,a6
        jsr    open(a6)         ;Ouvrir le fichier
        beq    error
        move.l d0,filehandle

loop:
        cmp.b  #$37,key         ;Touche Alternate pressée ?
        beq    qu               ;oui : interruption

        move.l #-1,d0
del:
        dbra   d0,del           ;petite pause pour lecture
        bsr    read4            ;charger le déclarateur
        beq    qu               ;EOF
        bmi    error            ;Error

        move.l conhandle,d1
        move.l #buffer,d2       ;Adresse du Buffer
        move.l #6,d3            ;4 caractères
        jsr    write(a6)        ;afficher le déclarateur
        beq    error

        move.l buffer,d5        ;Sauvegarder le déclarateur
        cmp.l  #'FORM',d5       ;FORM?
        bne    noform           ;non
        st     flag             ;sinon sauvegarder le flag
```



```
        bra    rm               ;et poursuivre

noform:
        tst    flag             ;Caractérisation ?
        beq    form             ;non
        clr    flag             ;sinon effacer le flag
        move.l #'----',outpuff  ;caractériser
        bsr    print
        bra    loop             ;et poursuivre

form:
        bsr    read4            ;Lire la longueur
        beq    qu               ;EOF
        bmi    error            ;Error
        move.l buffer,d0        ;Valeur en D0
        bsr    phex             ;et afficher

        cmp.l  #'FORM',d5       ;FORM?
        beq    loop             ;oui : suivant

        move.l filehandle,d1
        move.l buffer,d2
        addq.l #1,d2
        bclr   #0,d2            ;sur adresse paire
        move.l #0,d3            ;Mode: OFFSET_CURRENT
        jsr    Seek(a6)         ;chercher partie suivante
        bra    loop             ;poursuivre

qu:
        move.l conhandle,d1
        move.l #endtext,d2      ;fin du texte
        move.l #25,d3
        jsr    Write(a6)        ;afficher

        move.l conhandle,d1
        move.l #buffer,d2       ;adresse du tampon
        move.l #1,d3            ;1 caractère
        jsr    Read(a6)         ;lire
        bra    fin
```

```
error:
fin:
      move.l conhandle,d1     ;fermer la fenêtre
      move.l dosbase,a6
      jsr    Close(a6)

      move.l filehandle,d1    ;fermer le fichier
      jsr    Close(a6)

      move.l dosbase,a1       ;fermer DOS.Lib
      move.l execbase,a6
      jsr    CloseLib(a6)

      rts                     ;Fin!

read4:                        ;lire 4 caractères
      move.l filehandle,d1
      move.l #buffer,d2       ;adresse du tampon
      move.l #4,d3            ;lire 4 caractères
      jmp    Read(a6)

phex:                         ;afficher D0 en hexadéc.
      lea    outpuff,a0
      move   d0,d2
      move   #3,d3            ;4 chiffres
niblop:
      rol    #4,d2            ;Nibble de gauche vers le bas
      move   d2,d1
      and    #$f,d1           ;noter
      add    #$30,d1          ;transformer en ASCII
      cmp    #'9',d1          ;Chiffre?
      bls    nibok            ;oui
      add    #7,d1            ;sinon corriger
nibok:
      move.b d1,(a0)+         ;Caractères dans le tampon de sortie
      dbra   d3,niblop        ;Continuer la boucle
      move.b #$a,(a0)         ;retour à la fin

print:
      move.l dosbase,a6
```



```
      move.l conhandle,d1
      move.l #outpuff,d2      ;tampon de sortie
      move.l #5,d3            ;5 caractères
      jmp    Write(a6)        ;afficher


dosbase:    dc.l 0
conhandle:  dc.l 0
filehandle: dc.l 0
flag:       dc.w 0
outpuff:    dc.b '      '
buffer:     dc.b '      '
consolname: dc.b 'RAW:0/10/400/240/** Format IFF',0
dosname:    dc.b 'dos.library',0
filename:   dc.b 'Fichier IFF',0
endtext:    dc.b '** Veuillez presser une touche **'
even
```

Le programme ouvre une fenêtre et y donne les caractérisations et les longueurs des différents *chunks* intervenant dans le fichier. Le nom du fichier doit être porté dans le programme, en face de *'filename:'*. Nous avons placé une petite boucle d'attente lors de l'affichage, pour que vous puissiez lire facilement les résultats à l'écran. Si vous trouvez cependant que ça dure trop longtemps, ou encore si le fichier n'est pas un fichier IFF, vous pouvez mettre fin au programme en pressant sur la touche *Alternate*. Il ne reste plus ensuite qu'à presser une touche quelconque pour fermer la fenêtre.

3.3 Programmes

Un programme créé par un éditeur de liens, ou directement par un assembleur, peut être démarré par la simple saisie de son nom dans le CLI. Si l'on veut effectuer le démarrage à partir du *Workbench*, il faut en plus créer un fichier *.info*, qui contient l'icône du programme dans la fenêtre *Workbench*. On peut alors cliquer sur cette icône, et le programme se met en route.

3.3.1 MISE EN ROUTE D'UN PROGRAMM :T PARAMETRES

Comme on le sait déjà du fait des commandes CLI, il existe dans Amiga la possibilité d'indiquer quelques paramètres dans la ligne d'appel du programme, ces paramètres pouvant être adoptés et traités par le programme. Cette ligne d'appel ne peut pas être transmise ainsi, lorsque l'on démarre à partir du *Workbench*. C'est pourquoi la transmission des paramètres se fait différemment, selon que l'on travaille avec le CLI ou *Workbench*.

Le programme appelé doit donc d'abord déterminer de quelle interface il a été mis en route, et rechercher alors éventuellement ses paramètres de la manière appropriée. Considérons d'abord le cas le plus simple, c'est-à-dire le démarrage d'un programme à partir du CLI.

3.3.1.1 Appel avec le CLI

Lorsqu'il est appelé à partir du CLI, le programme reçoit les informations à l'intérieur de deux registres, les informations dont il a besoin sur les paramètres transmis le cas échéant après le nom du programme.

Dans le registre d'adresses A0 est transmise l'adresse de l'emplacement en mémoire où se trouve le texte qui suit le nom du programme, sur la ligne saisie dans le CLI. De plus, on transmet en D0 le nombre des caractères qui viennent à la suite du nom du programme.

Grâce à deux informations, le programme n'a aucune difficulté pour lire et traiter les paramètres comme il se doit. Nous allons donner un petit programme de démonstration en langage machine, que l'on peut appeler avec et sans paramètres.

Il s'agit d'une commande CLI, que vous pouvez également copier dans le répertoire C. Elle a pour tâche de déterminer le mode de présentation du texte qui suit l'appel. Vous pouvez l'utiliser par exemple dans la *Startup.Sequence*, lorsque vous voulez souligner un message ou l'écrire en italique.

Si vous avez .treposé le programme dans le répertoire C sous le nom de *'Font'*, v\ pouvez ensuite l'appeler avec l'instruction

```
>Font n
```

Le paramètre *n* peut être mis de côté, auquel cas le programme revient à la présentation normale du texte.

Mais si vous ajoutez ce paramètre *n*, il faut que ce soit un chiffre entre 0 et 7. Le choix de l'un de ces chiffres a pour effet ce qui suit :

0 Présentation normale
1 gras
3 italique
4 souligné
5 Inversé

Vous pouvez aussi demander une écriture qui soit à la fois en gras et en italique, à condition d'appeler *Font 1* et *Font 4*, l'un après l'autre.

Voici le programme :

```
;*****  Commande FONT *****

; Offsets Exec

OpenLib  = -30-378
ExecBase = 4

; Offsets AmigaDOS

Write    = -30-18
Output   = -30-30
Exit     = -30-114

run:
     subq    #1,d0           ;Nombre d'octets -1
     beq     normal          ;pas de paramètre?
search:
```

```
        cmp.b   #$20,(a0)+      ;chercher l'ar    nt
        bne     found           ;trouvé
        dbra    d0,search
        bra     normal          ;mise en place du font normal

found:
        move.b  -(a0),text+1    ;mise en place du style

normal:
        move.l  execbase,a6     ; Pointeur sur la bibliothèque Exec
        lea     dosname,a1
        moveq   #0,d0
        jsr     OpenLib(a6)     ;Open DOS-Library
        move.l  d0,dosbase
        beq     error           ;pas réussi

        move.l  dosbase,a6
        jsr     Output(a6)      ;Chercher Standard-Output-Handle

        move.l  d0,d1           ;Output-Handle en D1
        move.l  #text,d2        ;adresse du texte en D2
        move.l  #tfin-text,d3   ;Longueur en D3
        move.l  dosbase,a6
        jsr     Write(a6)       ;affichage du texte
        bra     fin             ;OK: fin

error:
        move.l  #1,d0           ;Error-Status
fin:
        move.l  d0,d1           ;Paramètre de retour
        move.l  dosbase,a6
        jsr     exit(a6)        ;Fin du programme

        rts                     ;ne revient pas

dosbase:  dc.l 0
```



```
dosname:  d    'dos.library',0
text:     dc.b $9b,'0;31;40m'
tfin:

even
```

Si vous voulez écrire un programme en C faisant intervenir ces paramètres, voici comment vous devrez vous y prendre. Il faut mettre en place le fichier *'startup.o'* comme premier élément dans l'instruction du *linker*, ce que l'on fait d'ailleurs en règle générale. La ligne des paramètres se trouve alors dans la variable *'argv'*, et le nombre de caractères dans *'argc'*.

La partie du programme correspondant au *startup* a d'autres capacités encore : elle permet d'ouvrir la bibliothèque DOS, et elle détermine le canal standard d'entrée/sortie, grâce aux fonctions *DOS Input()* et *Output()*. Les *handles* de ces canaux se trouvent alors dans *'stdin'* et *'stdout'*. La routine met alors en route la routine *'main'* de votre programme C.

Une autre information transmise par le CLI au programme est la taille du secteur réservé sur la pile. Ce secteur se situe sur la pile derrière l'adresse de retour au CLI et peut être lu et chargé par exemple au moyen de la commande

```
MOVE.L 4(SP),D0.
```

De cette manière, le programme peut vérifier s'il y a assez de place ou non sur la pile pour ses besoins particuliers.

En dehors de ceux-ci, il y a encore quelques autres paramètres qui sont transmis à partir du CLI. Ils permettent de simplifier un programme CLI de multiples façons. Vous trouverez là-dessus d'autres détails dans le chapitre se rapportant aux commandes CLI externes.

Nous venons donc de voir quel était la façon de faire pour initialiser un programme mis en route à partir du CLI. Nous allons maintenant passer à l'autre cas : le démarrage à partir du *Workbench*.

3.3.1.2 Démarrage à partir du Workbench

Si vous cliquez deux fois sur l'icône d'un programme représenté dans la fenêtre *Workbench*, le programme en question sera mis en route. Il recevra des paramètres de la même façon que précédemment, mais pas sous la forme d'une ligne de texte : il les recevra par l'intermédiaire de messages.

Si le programme est écrit en C, et précédé du programme *Startup*, lié à lui à l'aide du *linker*, vous n'avez plus besoin de vous préoccuper de ce message que l'on appelle le message *Startup*. Le programme *Startup* se charge lui-même des tâches suivantes, lorsqu'il constate que le démarrage a été effectué à partir du *Workbench* :

1. Il commence par ouvrir la bibliothèque DOS.
2. Il attend le message *Startup (WaitPort)*.
3. Il va chercher le message *(GetMsg)*.
4. Il teste le nombre des arguments à l'intérieur du message. Si ce nombre est 0, il passe à l'étape suivante.
5. Les arguments transmis sont interprétés comme des structures *Lock*, et le répertoire correspondant est transformé en conséquence en répertoire actuel *(CurrentDir)*.
6. L'argument *sm_ToolWindow* est vérifié ; s'il est différent de 0, la fenêtre indiquée est ouverte, et son *handle* est considéré comme entrée standard.

Quel sera l'aspect d'un programme qui ne dispose pas de ce confortable programme *Startup* ? Comment s'écrira par exemple un programme en langage machine ?

Même si vous n'avez pas besoin du message que le *Workbench* envoie à votre programme, il vous faut le rechercher malgré tout. Si vous ne le faites pas, le gourou entrera de nouveau dans de profondes méditations, autant dire dans une profonde perplexité, étant donné que lors de

l'appel suivant d'une fonction I/O, par exemple l'ouverture d'une fenêtre, il arrivera un message dans le *Message-Port*, et ce message n'aura aucun rapport avec la fonction en question.

Vous êtes donc obligé de faire exécuter par votre programme les mêmes fonctions que dans le programme *Startup*. Appelez d'abord la fonction *FindTask()* de l'Exec, pour obtenir un pointeur sur la structure du processus, c'est-à-dire de votre programme. Vous placerez par contre un 0 en A1 :

```
execbase = 4
FindTask = -294
WaitPort = -384
GetMsg   = -372


        move.l  execbase,a6     ;Adresse de base Exec en A6

        suba.l  a1,a1           ;Effacer argument A1
        jsr     FindTask(a6)    ;rechercher le pointeur
```

En D0, vous obtiendrez ainsi le pointeur sur votre structure de processus. Dans cette structure, vous trouverez l'information qui vous apprendra si ce processus a été mis en route à partir du CLI ou à partir du *Workbench* :

```
move.l  d0,a4           ;Pointeur sur processus en A4
tst.l   $ac(a4)         ;pr_CLI: CLI ou Workbench?
bne     fromCLI         ;c'était CLI!
```

Si l'argument testé est 0, le programme a été mis en route à partir du *Workbench*.

Si c'est bien le cas, la prochaine étape consiste à attendre la réception du message *Startup*. On y parvient à l'aide de la fonction *WaitPort()* :

```
lea     $5c(a4),a0      ;pr_MsgPort: MessagePort en A0
jsr     WaitPort(a6)    ;attendre le message
```

Cette fonction attend la réception d'un message en *MessagePort*.

Dans notre cas, ce sera simplement le *Startup_Message* du *Workbench*. Ce message doit être relevé, pour qu'il ne figure plus dans la file d'attente des messages. On utilise pour cela la fonction *GetMsg()*:

```
lea     $5c(a4),a0      ;Adresse RastPort en A0
jsr     GetMsg(a6)      ;Aller cherche le message
```

Vous pouvez faire usage de ce message, si vous en avez besoin. En D0, vous obtenez un pointeur sur la structure de message, par la fonction *GetMsg()* ; cette structure porte le nom de *WBStartup*.

Le message contient les éléments suivants :

On trouve au début une structure de message normale. Puis viennent les éléments de la structure *Startup* proprement dite :

Offset	Nom	Signification
$14	sm_Process	Descripteur de processsus
$18	sm_Segment	Descripteur de segment programme
$1C	sm_NumArgs	Nombre d'arguments transmis
$20	sm_ToolWindow	Description de la fenêtre à ouvrir
$24	sm_ArgList	Pointeur sur les arguments

sm_Arglist pointe sur les éléments des arguments transmis. Ces arguments contiennent les informations sur l'icône du *Workbench* activée au moment de la mise en route du programme. Il y a des programmes qui utilisent ceci de telle façon qu'un fichier texte activé en supplément par *Shift-clic* au moment de l'appel du programme soit chargé et affiché par les soins du programme. Les arguments de la liste sur laquelle pointe *sm_ArgList* sont constitués des pointeurs suivants :

wa_Lock	Lock fichier (description de répertoire)
wa_Name	Pointeurs sur les noms de fichiers

Pour donner un exemple qui montre comment on applique et comment on programme ce traitement des messages, nous allons écrire un programme qui transmet et qui affiche les *tool-types*. Ceux-ci sont les mentions qui peuvent être portées dans le programme *Workbench INFO*, à l'intérieur des différents fichiers. Vous sélectionnez pour cela un fichier (cliquez un fois), puis le point *Info* dans le menu *Workbench*. Vous voyez s'ouvrir une fenêtre de dialogue, dans laquelle vous pouvez introduire des données à l'intérieur du masque de saisie TOOL TYPES, en cliquant sur *Add*. Ces données sont utilisées par certains programmes en tant que réglages préalables (par exemple par *Notepad*).

Les données en question sont mémorisées dans le fichier *.info* qui appartient au programme. Dans ce fichier se trouvent en outre les données pour l'icône, sa position dans la fenêtre, et bien d'autres choses encore. Pour reporter ces données dans un programme, on dispose d'une autre bibliothèque sur la disquette *Workbench*, dans le répertoire LIBS: la bibliothèque *Icon*.

Cette bibliothèque contient des fonctions servant à traiter les fichiers *.info*. L'une de ces fonctions est *GetDiskObject()*, qui charge le fichier *.info*, et retourne un pointeur sur sa structure. Cette fonction utilise aussi notre programme. Avant d'entrer dans les détails de la bibliothèque *Icon* et de la structure *DiskObject*, je vais vous présenter ce programme, pour que les choses soient plus claires :

```
;** Message Workbench et traitement de l'info  S.D. **

execbase = 4            ;Adresse de base Exec
FindTask = -294         ;chercher Task
WaitPort = -384         ;attendre le message
GetMsg   = -372         ;aller prendre le message

OpenLib  = -408         ;ouvrir Library
CloseLib = -414         ;fermer Library
Open     = -30          ;ouvrir canal
Close    = -36          ;fermer canal
Read     = -42          ;lire les données
Write    = -48          ;afficher les données
CurrentDir = -126       ;directory actuel
mode_old = 1005         ;mode d'ouverture
```

```
GetDiskObject = -78     ;Charger le diskobje


run:
        move.l execbase,a6      ;Adresse de base Exec

        suba.l a1,a1
        jsr    FindTask(a6)     ;chercher le task propre

        move.l d0,a4            ;Pointeur en A4
        tst.l  $ac(a4)          ;pr_CLI: CLI ou Workbench?
        bne    fromCLI          ;CLI! fin...

        lea    $5c(a4),a0       ;MessageWBench
        jsr    WaitPort(a6)     ;attendre

        jsr    GetMsg(a6)       ;aller chercher le message
        move.l d0,message       ;sauvegarder le pointeur

; **** Ouvrir bibliothèques et fenêtres ****

        lea    iconname,a1      ;"icon.library"
        clr.l  d0
        jsr    OpenLib(a6)      ;ouvrir ICON.library
        move.l d0,iconbase      ;sauvegarder la base
        beq    fin3             ;Une erreur est survenue!

        lea    dosname,a1       ;"dos.library"
        clr.l  d0
        jsr    OpenLib(a6)      ;ouvrir DOS
        move.l d0,dosbase
        beq    fin2             ;Une erreur est survenue!

        move.l d0,a6
        move.l #conname,d1
        move.l #mode_old,d2
        jsr    Open(a6)         ;Ouvrir la fenêtre CON:
        move.l d0,conbase
        beq    fin1             ;Une erreur est survenue!
```



```
;**** Met    en place le répertoire actuel, si nécessaire ****

        move.l message,a0       ;Pointeur sur WBMessage
        move.l $24(a0),a0       ;sm_ArgList: Pointeur sur arguments

        beq    fin              ;pas d'arguments!

        move.l (a0),d1          ;D1 => Lock
        move.l dosbase,a6
        jsr    CurrentDir(a6)   ;mettre en place le directory actuel

; **** Charger le disk-object (fichier .info) ****

        move.l message,a0
        move.l $24(a0),a0        ;Pointeur sm_ArgLis
        move.l 4(a0),a0          ;wa_Name: Pointeur sur nom
        move.l iconbase,a6
        jsr    GetDiskObject(a6) ;charger le disk-object

; **** Afficher dans la fenêtre les entrées tool-type ****

        move.l d0,a1            ;Pointeur sur structure DiskObject
        move.l $36(a1),a1       ;do_ToolTypes: Pointeur sur
                                 ToolType-Array
        move.l a1,typetext      ;sauvegarder pointeur texte

typesloop:
        move.l typetext,a1      ;charger pointeur texte
        move.l (a1)+,a0         ;Pointeur sur Text en A0
        cmp.l  #0,a0            ;le texte existe?
        beq    nomore           ;non : fin des sorties
        move.l a1,typetext      ;sinon sauvegarder le pointeur

        move.l a0,d2            ;= adresse du texte pour affichage
        move.l a0,d3            ;obtenir encore la longueur
lenlop:
        tst.b  (a0)+            ;rechercher la fin
        bne    lenlop
        sub.l  a0,d3            ;calculer longueur du texte
        not.l  d3               ;et corriger
```

```
        move.l dosbase,a6
        move.l conbase,d1
        jsr    Write(a6)        ;afficher le texte dans la fenêtre

        move.l conbase,d1
        move.l #lf,d2           ;Linefeed:
        move.l #1,d3
        jsr    Write(a6)        ;ligne suivante

        bra    typesloop        ;passer à l'entrée suivante!

; **** C'était tout, maintenant attendre les touches ****

nomore:
        move.l conbase,d1
        move.l #1,d3            ;un caractère
        move.l #buffer,d2       ;dans le buffer
        jsr    Read(a6)         ;lire (attendre Return)

; **** Fin du programme: tout fermer et retour ****

fin:    move.l conbase,d1
        move.l dosbase,a6
        jsr    Close(a6)        ;fermeture de la fenêtre
fin1:
        move.l execbase,a6
        move.l dosbase,a1
        jsr    Closelib(a6)     ;fermer le DOS
fin2:
        move.l iconbase,a1
        jsr    Closelib(a6)     ;fermer ICON.library

fromCLI:
fin3:
        rts                     ;fin du programme

; **** Champs de données ****

dosbase:  blk.l 1       ;adresse de base DOS
conbase:  blk.l 1       ;base de la fenêtre
```



```
iconbase: blk.l 1       ;base d'icon.library
message:  blk.l 1       ;Pointeur sur WBMessage
typetext: blk.l 1       ;Pointeur texte

dosname:  dc.b 'dos.library',0
iconname: dc.b 'icon.library',0
conname:  dc.b 'CON:10/20/300/100/** Affichage du message',0
lf:       dc.b $a
buffer:   blk.b 2
```

Ce programme ne fonctionne que s'il est mis en route à partir du *Workbench.* Sinon, il est tout simplement interrompu *(fromCLI).* Pour le mettre en route, il reste à créer une icône pour ce programme. Vous y parviendrez très facilement à l'aide de l'éditeur d'icônes. Il faut que l'icône soit ensuite sauvegardée sous le même nom que le programme ci-dessus, et l'appendice *.info* est créé automatiquement.

Une fois que vous avez fait cela, vous pouvez cliquer sur l'icône, et sélectionner le point *Info* dans le menu *Workbench.* Dans la fenêtre qui s'ouvre alors, vous pouvez saisir une ou plusieurs données destinées à TOOL TYPES, puis sauvegarder avec SAVE.

Lorsque vous activez ensuite l'icône en cliquant dessus deux fois, le programme correspondant est chargé et mis en route. Le programme exécute les pas nécessaires, pour obtenir et traiter comme il se doit aussi bien le message *Workbench Startup* que la structure *DiskObject.*

La structure du *DiskObject* ou encore du fichier *.info* est constituée de la façon suivante :

Offset	Nom	Contenu
0	do_Magic	"Nombre magique" qui déclare que le fichier est valide ($E310)
2	do_Version	Numéro de version (1)
4	do_Gadget	Début d'une structure gadget, qui
$30	do_Type	fixe l'aspect et la position de l'icône
$32	do_DefaultTool	Type d'objet (Tool, projet, etc.)

$36	do_ToolTypes	Programme ndard de la disquette
$3A	do_CurrentX	
$3E	do_CurrentY	Position de l'icône dans la fenêtre
$42	do_DrawerData	Pointeur sur la structure de la fenêtre du sous-répertoire
$46	do_ToolWindow	Fenêtre standard pour les tools
$4A	do_StackSize	Taille de la pile pour les tools

Le pointeur *do_ToolTypes* pointe sur une liste de pointeurs, qui pointent à leur tour sur les textes des *Tool_Types* inscrits dans la fenêtre *Info* et qui se terminent par un 0. Les pointeurs de cette liste sont utilisés dans le programme pour afficher les textes.

Le programme donné en exemple montre donc comment l'on accède aux *Tool_Types* du programme. On peut y porter des données fondamentales servant à contrôler la fonction du programme. C'est ce qui se passe pour le programme *Notepad* du *Workbench*, où les *Tool_Types* permettent de fixer des paramètres comme la taille de la fenêtre de saisie ou le type de caractères adoptés.

La transmission de ces données se fait habituellement sous la forme

```
NOM=<Paramètre>[|<Paramètre>]
```

C'est ce que l'on doit faire également pour *Notepad*. L'avantage de cette forme réside simplement dans le fait qu'il y a deux fonctions prévues dans la bibliothèque d'icônes et pouvant tester ces lignes.

La première de ces fonctions est *FindToolType()*, offset -96 ; elle parcourt les données des *Tool_Types* à la recherche d'un nom déterminé. Dans l'exemple de *Notepad*, c'est une ligne avec le nom WINDOW qui est recherchée. On obtient en retour un pointeur sur les paramètres correspondant au signe d'égalité, ou alors un 0 lorsqu'aucune ligne n'apparaît avec ce nom.

Ce point est alors transmis à une autre fonction, *MatchToolType()*, offset -102, en même temps qu'un nouveau pointeur sur un paramètre de comparaison. La valeur qui en résulte permet de savoir si le paramètre de comparaison intervient ou non dans la ligne.

On utilise ce procédé par exemple lorsqu'un programme est en mesure de lire des fichiers, mais n'a le droit de lire que des fichiers d'un type déterminé. On reporte les critères correspondants dans *Tool_Types*, et ils seront ensuite comparés par le programme avec le type du fichier à charger, pour que le programme sache s'il a le droit de les charger ou non.

3.3.2 STRUCTURE DES COMMANDES CLI EXTERNES

Comme vous le savez, toutes les commandes du CLI normal sont externes, ce qui veut dire qu'elles sont mémorisées en tant que programmes sur la disquette dans le sous-répertoire C. Si vous introduisez donc quelque chose dans le CLI, il y aura examen, pour savoir s'il s'agit d'un nom de fichier dans le répertoire actuel, ou d'une commande dont le nom se trouve dans ce répertoire C. Dans le second cas, le programme correspondant est appelé.

Presque toutes les commandes ont besoin d'avoir accès à la bibliothèque DOS, pour pouvoir exécuter la fonction voulue. Mais pour que chacun de ces programmes ne soit pas obligé d'ouvrir à chaque fois la bibliothèque DOS, quelques paramètres leur sont transmis dans les registres du processeur.

Les registres D0 et A0 contiennent la longueur et l'adresse du texte des paramètres, saisi derrière la commande. Nous avons déjà expliqué ce point sur l'exemple du programme FONTS.

Les autres registres contiennent des valeurs susceptibles de nous intéresser :

Registre	Conten
D0	nombre de caractères constituant les paramètres
A0	adresse du texte des paramètres
A1	pointeur sur le début de la pile
A2	pointeur sur la bibliothèque DOS interne
A3	pointeur sur la taille de la pile
A4	pointeur sur le début du programme
A5	pointeur sur la routine d'appel de la fonction
A6	pointeur sur la routine de saut en arrière

Considérons plus particulièrement les registres A2, A5 et A6. Ils permettent déjà d'écrire une commande CLI qui peut être exécutée sans avoir à ouvrir elle-même la bibliothèque DOS.

La convention pour l'appel de ces routines diffère malgré tout quelque peu de l'appel normal des fonctions DOS. A partir de l'adresse pointée par A2, il y a une série d'adresses par sauts, qui pointent sur les différentes routines DOS. Elles ne sont de toutes façons pas appelées directement, mais avec l'adresse en A4, par l'intermédiaire de JSR (A5). Le paramètre de retour n'est pas transmis en D0 mais en D1. Les offsets dans la table sont également différents de ce qu'ils sont lors d'un appel normal.

Ces offsets ne sont pas absolument sûrs, car ils ne figurent pas dans la documentation fournie par Commodore. Mais les offsets que vous trouverez dans la liste ci-dessous sont corrects pour la version actuelle de l'Amiga.

Avant de produire cette liste des offsets de fonctions DOS lors de l'appel direct, il faut expliquer la façon de les utiliser. Nous donnons pour cela un petit programme, qui ne fait rien d'autre qu'ouvrir une petite fenêtre, attendre que la touche *Return* soit actionnée, et refermer alors la fenêtre. Voilà donc trois fonctions DOS appelées sans ouverture de la bibliothèque DOS.

Pour l'appel de fonction, on a utilisé ici une macro. Cette macro est utilisée lors de l'assemblage du programme à chaque fois qu'apparaît son nom *(doscall)*. Le paramètre qui suit est alors mis en place à l'endroit où l'on trouve ?! dans la définition de la macro. Ce paramètre est précisément notre offset.

```
;*****  Fonctions de base du DOS à partir du CLI  6/87  S.D.  *****

Open    =$ff        ;Commandes DOS: Open
Close   =$5d        ;               Close
Read    =$fd        ;               Read

mode_old=1005

; **** Definition de la macro 'doscall' ****

doscall: MACRO          ; ** appel direct du DOS **
          move.b  #?1,d0
          ext     d0          ;Offset en mots longs
          ext.l   d0          ;transformer
          lsl     #2,d0
          move.l  0(a2,d0),a4 ;adresse de la fonction
          moveq   #$c,d0
          jsr     (a5)        ;appel de la fonction
        ENDM

; **** Début du programme ****

run:
     move.l  #consolname,d1   ;Definition Consol
     move.l  #mode_old,d2     ;Mode
     doscall Open             ;ouvrir la fenêtre CON:
     move.l  d1,conhandle

     move.l  conhandle,d1
     move.l  #inbuff,d2       ;adresse Buffer
     move.l  #1,d3            ;1 caractère
     doscall Read             ;lire les caractères (Return)

     move.l  conhandle,d1     ;fermer la fenêtre
```

```
    doscall Close           ;avec Close

    clr.l   d0              ;Status: OK
    jsr     (a6)            ;fin du programme

; **** Champ des données ****

conhandle:  dc.l 0
inbuff:     blk.b 8
consolname: dc.b 'RAW:100/50/300/100/** Fenêtre de test',0
even
```

Vous voyez comme il est simple de programmer une commande CLI. Avec 12 lignes de programme en tout et pour tout, on exécute trois fonctions DOS. Le programme *Fonts* du chapitre sur le CLI aurait pu être aussi court que celui-ci. Essayez donc vous-même.

Les offsets de fonction des fonctions DOS sont, comme vous pouvez le constater, différents des offsets qui ont cours lors d'un appel DOS normal. Voici maintenant cette liste des commandes DOS, avec les offsets correspondants, que l'on peut utiliser dans la programmation comme ci-dessus :

Offset	Nom de la fonction	Offset	Nom de la fonction
$FF	Open	$EF	IoErr
$5D	Close	$EE	CreateProc
$FD	Read	$02	Exit
$FA	Write	$ED	LoadSeg
$41	Input	$52	UnLoadSeg
$42	Output	$EC	GetPacket
$F8	Seek	$EB	QueuePacket
$F7	Delete	$EA	DeviceProc
$F6	Rename	$E9	SetComment
$F5	Lock	$E8	SetProtect
$6D	UnLock	$E7	DateStamp
$71	DupLock	$2F	Delay

$F4	Examine	$57	WaitForChar
$F3	ExNext	$23	ParentDir
$F2	Info	$E6	IsInteractive
$F1	CreateDir	$E5	Execute
$F0	CurrentDir		

En fait, le programme donné en exemple montre que ce ne sont pas de véritables offsets, mais plutôt les numéros des vecteurs à utiliser de la table indiquée par A2. Les valeurs au-dessus de $7F y sont négatives, ce qui veut dire que l'adresse utilisée est inférieure à l'adresse en A2.

3.4 Entrée/Sortie

Une composante essentielle d'un programme est l'échange des données avec le monde extérieur, par l'intermédiaire de l'écran, du clavier, des disquettes ou encore des interfaces. Seule cette entrée/sortie, nommée aussi tout simplement *I/O (Input/Output)*, rend un programme capable de tirer pleinement parti de l'ordinateur sur lequel il fonctionne. Il y a trois manières de réaliser l'entrée/sortie.

La première est l'entrée/sortie au moyen des fonctions DOS correspondantes, comme *Open()*, *Close()*, *Read()* et *Write()*. Cette méthode est évidemment la plus simple, car elle n'exige pas de gros efforts de programmation de votre part. L'inconvénient est que la fonction appelée doit être entièrement traitée avant que votre programme ne poursuive son travail.

La seconde méthode ne présente pas cet inconvénient. Le mot magique est ici *Devices*. Grâce à ces *Devices*, vous pouvez rendre le fonctionnement de l'entrée/sortie entièrement indépendant de la suite de votre programme. L'entrée/sortie fonctionne en arrière-plan, donc parallèlement à votre programme, et ne revendique pratiquement pas de temps en propre au processeur. L'inconvénient de cette technique est l'effort bien plus grand exigé de la part du programmeur.

La troisième méthode de l'entrée/sortie est la programmation directe du hardware de l'Amiga. Cela suppose cependant des connaissances assez poussées sur le système et a de plus l'inconvénient de conduire à de grandes complications si on veut un fonctionnement multi-tâches. Vous trouverez d'autres informations là-dessus dans la partie de cet ouvrage consacrée au hardware.

Commençons notre tour d'horizon sur la programmation de l'entrée/sortie par la méthode standard : l'utilisation des fonctions DOS.

3.4.1 I/O STANDARD

Comme nous l'avons déjà mentionné, les fonctions DOS responsables de l'entrée/sortie des données sont *Open()*, *Close()*, *Read()* et *Write()*. Elles permettent d'exécuter la plupart des tâches que nécessite un programme.

Il existe toute une série de canaux d'entrée/sortie que le DOS connaît déjà par leur nom. Ces noms peuvent être utilisés dans une commande *Open*. Voici quels sont ces canaux standard :

DFn : Désigne le lecteur de disquettes numéro n. Ce numéro peut être 0, 1, 2 ou 3.

SYS : Désigne le lecteur de disquettes à partir duquel a été chargé le système.

RAM : Désigne le disque RAM, toujours disponible, et dont la taille est proportionnelle aux données qu'il contient. Il peut être utilisé comme un lecteur de disquettes, à ceci près que les données sont entreposées non pas sur disquette, mais dans la mémoire vive de l'Amiga.

NIL : Ce canal est un véritable cimetière de données : les données qui y sont envoyées sont rejetées et ne servent donc à rien. Ce peut être parfois utile d'en passer par là, lorsque par exemple un programme veut effectuer des sorties dont vous n'avez rien à faire.

SER : Désigne l'interface sérielle (RS232), et permet l'entrée/sortie des données par ce port.

PAR : Désigne l'interface parallèle (port d'imprimante), qui contient en fait 8 conduits d'entrée/sortie. Vous pouvez donc sortir par ce port des données placées en parallèle, ou les charger.

PRT : Désigne également l'interface parallèle, à cette différence près que cette fois vous pouvez adresser une imprimante. Si cependant l'imprimante est définie pour l'interface sérielle, ce sera quand même celle-ci qui sera adressée. Vous pouvez prescrire à l'avance les définitions pour l'imprimante, à l'aide du programme *Preferences*.

CON : Prescrit une fenêtre pour l'entrée/sortie. Cette fenêtre est ouverte automatiquement lors de l'ouverture du canal. Les paramètres de la fenêtre sont indiqués de la façon suivante :

```
CON:x/y/l/h/Nom
```

x et *y* représentent les coordonnées du coin supérieur gauche de la fenêtre à l'écran, *l* et *h* la largeur et la hauteur de la fenêtre en pixels ; *Nom* est le titre de la fenêtre qui apparaît dans la ligne de titre. Ainsi

```
CON:20/10/200/100/Fenêtre-Test
```

désigne une fenêtre qui porte le nom *"Fenêtre-Test"*, qui commence à la position x=20 et y=10, et dont la largeur et la hauteur font respectivement 200 et 100 points.

RAW : Représente également une fenêtre et transmet les entrées et les sorties à cette fenêtre. Au contraire de ce qui se passe avec CON:, il n'y a pas ici de fonction qui soit traitée à l'avance (par exemple l'édition d'une ligne), si bien que la fenêtre en question ne doit être utilisée que dans des cas très particuliers.

***** Désigne la fenêtre actuelle, par exemple la fenêtre CLI.

Commençons maintenant par l'application probablement la plus importante : la saisie au clavier et l'affichage à l'écran.

3.4.1.1 Clavier et écran

La table ci-dessus montre que *l'AmigaDOS* met trois possibilités à votre disposition pour les entrées/sorties à l'écran : *CON:*, *RAW:* et *.

Fenêtre CON:

Pour ouvrir une fenêtre CON:, on utilise la fonction *Open()* du DOS. La fonction attend pour paramètre un pointeur sur le nom du canal à ouvrir, et le mode sous lequel ce canal doit être ouvert. Les modes possibles sont les suivants :

Mode_new pour un canal dans lequel on peut seulement écrire.

Mode_old pour un canal à partir duquel on peut aussi lire.

Mode_readwrite dans la version DOS 1.2, où on peut aussi bien écrire que lire dans le canal.

Pour ouvrir une fenêtre CON: ou RAW:, on utilise le mode *Mode_old*, puisque le canal est connu, et que l'on peut aussi lire dans ce canal.

Voici un petit programme de démonstration en langage machine, qui ouvre une fenêtre CON:, qui affiche un texte dans cette fenêtre, attend une saisie, puis referme la fenêtre :

```
;*****  Entrée/Sortie simple par CON:  *****

OpenLib  = -408
closelib = -414
ExecBase = 4

; offsets Amiga DOS

Open     = -30
```



```
Close    = -36
Read     = -42
Write    =-48
Exit     =-144

Mode_old = 1005

run:
     move.l  execbase,a6      ;Pointeur sur Bibliothèque Exec
     lea     dosname,a1
     moveq   #0,d0
     jsr     openlib(a6)      ;Open DOS-Library
     move.l  d0,dosbase
     beq     error            ;pas réussi

     move.l  dosbase,a6       ;Adresse de base DOS en A6

     move.l  #name,d1         ;Pointeur sur le nom
     move.l  #mode_old,d2     ;Mode
     jsr     Open(a6)         ;ouvrir le fenêtre
     move.l  d0,conhandle     ;sauvegarder le Handle
     beq     error

     move.l  conhandle,d1     ;Handle de la fenêtre en D1
     move.l  #text,d2         ;adresse du texte en D2
     move.l  #tende-text,d3   ;longueur en D3
     jsr     Write(a6)        ;affichage du texte
     move.l  conhandle,d1     ;Handle de la fenêtre
     move.l  #buffer,d2       ;adresse du tampon
     move.l  #80,d3           ;longueur max.
     jsr     Read(a6)         ;attendre saisie

     move.l  conhandle,d1
     jsr     Close(a6)        ;fermer la fenêtre
     bra fin                  ;terminé

error:
     move.l  #-1,d0           ;Error-Status
fin:
     move.l  d0,d1
```

```
        move.l  dosbase,a6
        jsr     Exit(a6)        ;fin du programme

        rts                     ;ne se produit pas

dosname:    dc.b 'dos.library',0
name:       dc.b 'CON:20/10/200/100/** Fenêtre de test',0
text:       dc.b 'Veuillez écrire votre texte !',0
tende:
buffer:     blk.b 80
even
dosbase:    dc.l 0
conhandle:  dc.l 0
```

Fenêtre RAW:

Le programme ci-dessus peut aussi être mis en route avec RAW: au lieu de CON:. Essayez, et vous remarquerez tout de suite la différence ! La version CON: attend en effet que l'on actionne la touche *Return* après la saisie, tandis que le programme avec RAW: reprend les commandes dès que l'on a appuyé sur une touche, quelle qu'elle soit. Ceci est vrai également pour les touches de fonction et les touches de déplacement du curseur, qui ne sont d'ailleurs pas reconnues par la fenêtre CON:.

Une fenêtre CON: offre donc plus de confort pour la saisie de textes entiers ; une fenêtre RAW: met par contre à votre disposition l'ensemble du clavier.

Mais dans les deux cas, la représentation normale des caractères n'est pas la seule possible. Il existe encore la possibilité de représenter d'autres types de caractères, par exemple soulignés et en gras. Il y a encore d'autres fonctions disponibles, grâce auxquelles on peut manipuler les fenêtres. Il est possible par exemple de supprimer l'image, de la descendre ou de la monter, etc... Toutes ces fonctions sont exécutées grâce à des séquences de contrôle, qui doivent être affichées dans la fenêtre, pourvues pour une partie d'entre elles de paramètres.

Voici une liste des signes de contrôle qui déclenchent l'exécution d'une fonction. Ces signes sont présentés sous la forme de nombres en hexadécimal :

Séquence	Fonction
08	Backspace
0A	Linefeed, curseur vers le bas
0B	curseur vers le haut d'une ligne
0C	Supprimer la fenêtre
0D	Carriage Return, curseur dans la première colonne de la ligne
0E	Retour à la présentation normale (OF suppression)
0F	Passage à un style particulier
1B	Escape

Les séquences qui suivent commencent toutes par le signe $9B, c'est-à-dire le CSI *(Control Sequence Introducer)*. Les caractères qui viennent après ce signe déclenchent tous une fonction. Les valeurs indiquées entre crochets peuvent ne pas figurer. La donnée *'n'* doit être indiquée sous forme de caractères ASCII, et doit représenter un nombre décimal à un ou plusieurs chiffres. La valeur adoptée par défaut pour *n* est placée après le *n* entre parenthèses.

Séquence	Fonction
9B[n] 40	Indentation de n caractères
9B[n] 41	Curseur de n (1) lignes vers le haut
9B[n] 42	Curseur de n (1) lignes vers le bas
9B[n] 43	Curseur de n (1) lignes vers la droite
9B[n] 44	Curseur de n (1) lignes vers la gauche
9B[n] 45	Curseur de n (1) lignes vers le bas colonne 1
9B[n] 46	Curseur de n (1) lignes vers le haut colonne 1
9B[n] [3B n] 48	Placer le curseur dans la ligne-colonne ...

9B 4A	Supprimer la fenêtre à partir de la position du curseur
9B 4B	Effacer la ligne à partir de la position du curseur
9B 4C	Insérer une ligne
9B 4D	Effacer la ligne
9B[n] 50	Effacer n caractères à partir de la position du curseur
9B[n] 53	Déplacement vers le haut de n lignes
9B[n] 54	Déplacement vers le bas de n lignes
9B 323068	Désormais : Linefeed => Linefeed+Return
9B 32306C	Désormais : Linefeed => seulement Linefeed
9B 6E	Retourner la position du curseur ! On obtient en retour une chaîne de caractères de la forme :

9B (ligne) 3B (colonne) 52

9B (style);(coul. premier plan);(coul.arrière-plan) 6D

Les trois paramètres sont eux aussi des nombres décimaux en format ASCII. Ils signifient :

Style : 0=normal
1=gras
3=italique
4=souligné
7=inverse-vidéo

Couleur de premier plan :
30-37 : couleurs 0 à 7 pour le texte

Couleur d'arrière plan :
40-47 : couleurs 0 à 7 pour l'arrière plan

9B (Longueur) 74	Fixe le nbre max. de caractères à présenter
9B (Largeur) 75	Fixe la longueur max. de la ligne

9B (Distance) 78	Définit la distance en pixels à partir du bord gauche de la fenêtre jusqu'au point où l'affichage doit commencer.
9B (Distance) 79	Définit la distance en pixels à partir du bord supérieur de la fenêtre jusqu'au point où l'affichage doit commencer.

Les 4 dernières fonctions peuvent placer le point de départ du texte en position normale si l'on ne fait pas figurer le paramètre.

9B 30 20 70	Rendre le curseur invisible
9B 20 70	Rendre le curseur visible
9B 71	Obtenir la taille de la fenêtre en retour ! On obtient en retour une chaîne de paramètres de la forme suivante :

9B 31 3B 31 3B (lignes) 3B (colonnes) 73

Pour voir comment on utilise ces signes de contrôle, amusez-vous à afficher le texte suivant dans votre fenêtre :

```
text:  dc.b $9b,"4;31;40m"
       dc.b "titre"
       dc.b $9b,"3;33;40m",$9b,"5;20H"
       dc.b "** Hello, le monde ! **",0
```

Les paramètres pour les séquences de contrôle sont indiqués ici, entre guillemets, comme chaînes de caractères ASCII. Vous voyez que vous avez ainsi un moyen très efficace d'afficher des textes.

Ces séquences peuvent être envoyées, mais elles peuvent aussi être reçues, lorsqu'une touche de fonction du clavier ou une touche de déplacement du curseur vient d'être pressée. Les caractères que l'on reçoit en un pareil cas sont les suivants (CSI est mis pour $9B) :

Touche	Sans Shift	Avec Shift
F1	<CSI>0~	<CSI>10~
F2	<CSI>1~	<CSI>11~
...		
F9	<CSI>8~	<CSI>18~
F10	<CSI>9~	<CSI>19~
HELP	<CSI>?~	<CSI>?~
haut	<CSI>A	<CSI>T~
bas	<CSI>B	<CSI>S~
gauche	<CSI>C	<CSI> A~
droite	<CSI>D	<CSI> @~

De cette façon, on peut être renseigné sur tout ce qu'entreprend l'utilisateur au clavier. Si ça ne suffit toujours pas, il existe encore une source d'information : le *RAW-Input-Events*. Ce sont des événements qui sont annoncés par une séquence, si vous le désirez. Le désir de recevoir un message qui renseigne sur ces événements est lui-même communiqué au DOS par une séquence, sous la forme suivante :

```
<CSI>n{
```

Le *'n'* représente ici un nombre entre 1 et 16, ce nombre correspondant à un événement qui doit être annoncé. Les événements en question sont les suivants :

1 Touche pressée
2 Touche de la souris pressée
3 Fenêtre activée
4 Souris déplacée
5 Non utilisé
6 Timer
7 Gadget sélectionné
8 Gadget abandonné
9 Requester effacé

10 [illegible]nu sélectionné
11 [illegible]nêtre fermée (cf. Console-Device)
12 Taille de fenêtre modifiée
13 Nouvelle fenêtre
14 Réglages modifiés
15 Disquette enlevée du lecteur
16 Disquette introduite

Pour certains (10, 11), ces événements ne sont pas disponibles, étant donné qu'une fenêtre ouverte avec DOS ne dispose en fait ni de menus, ni du symbole de fermeture. Ces possibilités commencent à devenir réellement intéressantes lorsqu'on a créé soi-même sa *fenêtre-console*. Ceci n'est malgré tout possible que par la combinaison de *Intuition* et de *Devices*, et c'est pourquoi nous en traiterons un peu plus tard, dans le paragraphe concernant le *Console-Device*.

Lorsqu'un de ces événements énumérés se produit (par exemple l'introduction d'une disquette), on obtient alors en retour une séquence de la forme suivante :

```
<CSI><Classe>;<sous-classe>;<touche>;<état>;<x>;<y>;
<secondes>;<microsecondes>
```

La signification des différents termes est la suivante :

CSI Control-sequence-introducer $9B
Classe Numéro de l'événement (cf.ci-dessus)
Touche Code clavier de la dernière touche pressée ou de la touche souris
Etat Etat du clavier :

Bit	Masque	Signification
0	0001	Touche Shift de gauche
1	0002	Touche Shift de droite
2	0004	Touche CapsLock
3	0008	Control
4	0010	Touche Alternate gauche
5	0020	Touche Alternate droite
6	0040	Touche Amiga gauche

7	0080	T[illegible]e Amiga droite
8	0100	Bloc numérique
9	0200	Répétition des touches
10	0400	Interrupt (non utilisé)
11	0800	Fenêtre active
12	1000	Touche souris gauche
13	2000	Touche souris droite
14	4000	Touche souris milieu (non utilisé)
15	8000	Coordonnées relatives souris

X et Y Coordonnées du pointeur de la souris lors d'un événement concernant la souris.

Secondes
Microsecondes Temps système au moment de l'événement.

Fenêtre *

La plupart des commandes CLI utilisent *, car c'est le moyen le plus simple. Etant donné que l'on obtient de cette façon la fenêtre actuelle, évidemment déjà ouverte, on peut ainsi se dispenser même d'ouvrir et de fermer le canal.

Comme les fonctions *Read()* et *Write()* ont tout de même besoin d'un *handle* du canal dans lequel elles doivent lire ou écrire des données, ce *handle* doit tout d'abord être transmis.

3.4.1.2 Fichiers sur disquettes

De la même façon que les fenêtres CON: et RAW:, on peut aussi ouvrir et traiter des fichiers sur disquettes, avec seulement quelques éléments supplémentaires. Ainsi le mode transmis lors de l'ouverture joue un grand rôle. Si l'on a transmis le mode *'ancien'*, c'est un fichier existant sur la disquette qui est recherché, et il n'est possible que d'en effectuer une lecture.

Avec le mode *'nouveau'*, le fichier est mis en place, et un fichier éventuel existant sous le même nom serait supprimé. Dans un fichier ouvert de cette manière, on peut seulement écrire. Enfin avec le mode *'readwrite'*, on peut écrire et lire dans un fichier existant, c'est-à-dire le modifier.

Pour les fonctions *DOS Read()*, *Write()* et *Close()*, rien ne se modifie en ce qui concerne l'entrée/sortie à l'écran. Il y a cependant un certain nombre de fonctions en plus, qui sont très utiles lors de l'utilisation de fichiers-disquettes.

Etant donné que l'on peut lire des données au fur et à mesure sur une disquette, le système doit avoir un moyen pour se souvenir de l'endroit où l'on a accédé pour la dernière fois dans le fichier. Ceci est réalisé à l'aide d'un pointeur, que l'on peut aussi déplacer directement. On dispose à cet effet de la fonction *Seek()*, grâce à laquelle on peut déplacer le pointeur à volonté vers l'avant ou vers l'arrière. Une position absolue est indiquée dans ce cas, comptée soit relativement à la position actuelle, soit encore par rapport au début ou à la fin du fichier.

Une autre fonction du DOS permet de supprimer un fichier sur une disquette : il s'agit de la fonction *DeleteFile()*. Elle permet aussi de supprimer des sous-répertoires, à condition évidemment qu'ils soient vides.

Les noms des fichiers sont eux aussi modifiables, à l'aide de la fonction *Rename()*. On lui transmet simplement l'ancien et le nouveau nom. L'intéressant ici est que l'on peut non seulement changer le nom d'un fichier, mais aussi sa position sur la disquette. Si, en effet, on indique un autre chemin de recherche dans le nom, le fichier est "déplacé" (et non copié) dans ce nouveau sous-répertoire. Ceci ne fonctionne malgré tout que sur une disquette. Si vous cherchez à effectuer le déplacement vers une autre disquette, vous obtenez un message d'erreur.

Un fichier-disquette peut de plus être protégé contre diverses fonctions. On détermine le type de protection à l'aide d'un masque, transmis à la fonction *SetProtection()*. Les 4 premiers bits du masque (0-3) indiquent si le fichier est protégé contre les actions suivantes :

Bit	Signification si le bit est positionné
0	On ne peut pas supprimer le fichier
1	On ne peut pas exécuter le fichier
2	On ne peut pas écrire dans le fichier
3	On ne peut pas lire dans le fichier

3.4.1.3 Interface sérielle

L'interface sérielle peut être traitée à peu près de la même façon que l'entrée/sortie écran. Un canal est ouvert, du nom de SER:, et on peut écrire ou lire dans ce canal. Mais trois problèmes peuvent surgir lors de cette opération :

1) Lors de l'appel de la fonction *Read()*, l'Amiga attend la réception d'un ou plusieurs caractères sur l'interface sérielle. Si ceux-ci ne viennent pas, il continue à attendre en vain. C'est pourquoi, dans un programme qui attend des données de cette interface, sans être absolument sûr qu'il en viendra, il vaut mieux faire précéder *Read()* de la fonction *WaitForChar()*. Grâce à cette fonction, l'attente est limitée à un temps déterminé (indiqué en microsecondes). Si rien ne vient durant ce laps de temps, on obtient un 0 en retour, et le programme peut afficher le cas échéant un message d'erreur avant de passer la main. Au contraire, lorsque les données attendues arrivent, on obtient en retour la valeur -1, et la lecture de ces données peut commencer.

2) Les données sont réceptionnées, sans que l'on ait moyen de savoir quelle était leur quantité. Il peut alors se produire le problème décrit en 1). C'est la raison pour laquelle on ne peut pas prendre en considération à partir du CLI des données venant de l'interface sérielle par exemple avec COPY SER: TO *. Le CLI ne sait pas, en effet, à quel moment commence le flux de données et à quel moment il cesse.

En conséquence de quoi, il se met en grève. Si l'on a lancé une telle commande, on n'a malheureusement qu'un seul moyen d'y mettre fin : le *Reset*.

3) Un programme peut vouloir envoyer ou recevoir par cette interface, des données dont les réglages ne lui correspondent pas. On peut évidemment prévoir ces réglages avec le programme *Preferences*, et effectuer un nouveau démarrage. Ce qui est malgré tout assez pénible. Tout autant que *Preferences*, votre propre programme peut évidemment effectuer lui-même les réglages nécessaires. Cela n'est cependant pas possible à l'aide d'une simple commande DOS : il faut en passer par le *Serial-Device* à l'aide des fonctions I/O. Vous trouverez des commentaires là-dessus dans le chapitre correspondant.

3.4.1.4 Interface parallèle

La programmation de l'interface parallèle (PAR:) n'est pas nécessaire, car la plupart du temps, c'est l'imprimante qui y est connectée. Cette interface est cependant très intéressante, car on peut non seulement sortir par là des données, mais aussi lire des données.

La façon la plus simple de programmer cette interface est le chemin direct par l'intermédiaire des registres du hardware. L'inconvénient de la méthode est qu'il risque alors de surgir des problèmes lors du fonctionnement multi-tâches, lorsqu'un autre programme cherche lui aussi à avoir accès à cette interface. C'est pourquoi il est plus sûr d'y accéder par l'intermédiaire du DOS. Dans ce cas, le format des données est lui aussi prescrit, et l'on ne dispose plus de la possibilité de programmer des bits un par un à l'entrée, et des bits différents à la sortie.

4. DEVICES

Dans les *Devices*, il y a l'une des grandes forces du système d'exploitation de l'Amiga. Les *Devices* sont des paquets de programmes, qui prennent en charge un certain nombre de tâches. Ces tâches leur sont distribuées par un programme courant, qui peut attendre le résultat, mais qui peut aussi continuer à fonctionner.

La structure fondamentale de ces *Devices* a déjà été commentée au chapitre Exec. Nous ne nous occuperons ici que de l'application pratique. Jetons pour cela d'abord un coup d'oeil sur la programmation, dont les étapes sont les suivantes :

1. Etant donné que l'exécution d'une tâche par le *Device* est annoncée au moyen d'un message, le récepteur de ce message (votre programme) doit être connu. On y parvient grâce à la fonction *FindTask()* d'Exec, à laquelle on transmet le paramètre 0. La valeur obtenue est réutilisée à l'étape suivante.

2. Pour le message de *Device*, on réserve un port au moyen de *AddPort()*. Il s'agit d'un port *Reply*, ce qui signifie simplement : *Port Réponse*. Dans cette structure de *Message-Port*, on place à l'entrée *SigTask* (adresse de port +$10) le pointeur obtenu à l'instant, pointant sur la structure *Task* de votre programme.

3. Le *Device* est ouvert par l'intermédiaire de la fonction *OpenDevice()*. Il faut pour cela fournir un pointeur sur le nom du *Device* et un autre sur la structure I/O, nécessaire pour communiquer avec le programme *Device*.

4. Dans la structure I/O, on porte les paramètres nécessaires pour la fonction souhaitée. Le nombre et le type des paramètres à reporter ici dépendent entièrement de la fonction.

5. On met en route le travail du *Devi* avec *Dolo()* ou *Sendlo()*. Si on l'a fait avec le premier, le programme appelant attend le signal OK du port *Reply* ; avec le second, on ne fait que mettre en route le travail du *Device*, et le programme peut continuer à tourner.

On a vu apparaître ici deux structures qui contrôlent la communication entre le programme de l'utilisateur et les *Devices*. Ce sont la structure de port et la structure I/O. Toutes deux ont déjà été décrites. Nous reprenons ici la structure standard des opérations I/O :

Offset	Nom	Signification
STRUCT MsgNode		
0	Succ	Pointeur sur l'entrée suivante
4	Pred	Pointeur sur l'entrée précédente
8	Type	Type d'entrée
9	Pri	Priorité
10	Name	Pointeur sur le nom
14	ReplyPort	Pointeur sur le ReplyPort
18	MNLength	Longueur du Node
STRUCT IOExt		
20	IO_DEVICE	Pointeur sur le Device Node
24	IO_UNIT	Numéro interne d'unité
28	IO_COMMAND	Commande
30	IO_FLAGS	Flags
31	IO_ERROR	Statut de l'erreur
STRUCT IOStdExt		
32	IO_ACTUAL	Nombre d'octets transférés
36	IO_LENGTH	Nombre d'octets à transférer
40	IO_DATA	Pointeur sur le tampon de données
44	IO_OFFSET	Offset (par ex. pour le Trackdisk-Device)
48	Ici commence à chaque fois la structure prolongée	

Les fonctions I/O normales sont déclenchées à l'aide des commandes standard, appartenant aux définitions I/O.

Ces com des sont :

CMD_INVALID	(0)	Commande non valide
CMD_RESET	(1)	Retour du Device à l'état initial
CMD_READ	(2)	Lecture dans le Device
CMD_WRITE	(3)	Ecriture dans le Device
CMD_UPDATE	(4)	Traitement des tampons
CMD_CLEAR	(5)	Effacement de tous les tampons
CMD_STOP	(6)	Mise en place d'une pause
CMD_START	(7)	Poursuivre après la pause
CMD_FLUSH	(8)	Interruption du travail en cours

En plus de ces commandes, il en existe quelques autres pour chacun des *Devices*. Nous allons les aborder dans les exemples suivants.

Sur une disquette *Workbench* normale, les *Devices* se trouvent dans le sous-répertoire DEVS. Quelques autres *Devices* ne sont pas sur cette disquette, tout en étant disponibles, puisqu'ils sont résidents dans l'Amiga.

Nous allons nous occuper de la programmation des *Devices* les plus importants, à l'aide d'exemples que vous pourrez réutiliser dans vos propres programmes.

4.1 Trackdisk Device : Accès aux disquettes

Le *Trackdisk Device* concerne la relation avec les disquettes prévue par le système d'exploitation. Il est également utilisé par le DOS. Il rend possible l'accès direct aux disquettes, sans avoir à passer par les registres du hardware.

La structure I/O étendue contient les deux entrées suivantes (mots longs), qui ne sont de toutes façons nécessaires que pour les commandes étendues :

IOTD_COUNT Nombre de changements de disquettes autorisé.

IOTD_SECLABEL Pointeur sur le champ Sector-Header, qui peut avoir une taille de 16 octets par secteur à lire.

Ce *Device* possède un grand nombre de commandes supplémentaires. On fait, parmi ces commandes, la différence entre les commandes *Trackdisk* normales et étendues. Voici une liste de toutes les commandes *Trackdisk* valides :

Commandes standard :

CMD_READ	(2)	Lecture d'un ou plusieurs secteurs par disquette
CMD_WRITE	(3)	Ecriture sur un ou plusieurs secteurs par disquette
CMD_UPDATE	(4)	Reporter le tampon de track sur la disquette
CMD_CLEAR	(5)	Annoncer que le tampon de track n'est pas valide

Commandes trackdisk :

TD_MOTOR	(9)	Allumer/éteindre un moteur de lecteur de disquettes
TD_SEEK	(10)	Positionner une tête d'écriture/lecture sur un track déterminé
TD_FORMAT	(11)	Initialiser une ou plusieurs pistes
TD_REMOVE	(12)	Installer une routine d'interruption qui sera appelée lors d'un changement de disquettes
TD_CHANGENUM	(13)	Communiquer le nombre de changement de disquettes
TD_CHANGESTATE	(14)	Vérifier si une disquette est enclenchée
TD_PROTSTATUS	(15)	Vérifier si la disquette est protégée contre l'écriture
TD_RAWREAD	(16)	Lire le contenu non encore traité de la disquette
TD_RAWWRITE	(17)	Ecrire sur la disquette les disquettes non traitées
TD_GETDRIVETYPE	(18)	Communiquer le type de lecteur (1=3½, 2=5¼)
TD_GETNUMTRACKS	(19)	Communiquer le nombre total de sillons
TD_ADDCHANGEINT	(20)	Installer une routine d'interruption qui sera appelée lors d'un changement de disquettes

TD_REMCHANGEINT	(21)	Supprimer la routine ci-dessus
TD_LASTCOMM	(22)	Communiquer la dernière commande

Commandes étendues (numéros supérieurs à +32768 ou + $8000) :

ETD_READ	(2)
ETD_WRITE	(3)
ETD_UPDATE	(4)
ETD_CLEAR	(5)
ETD_MOTOR	(9)
ETD_SEEK	(10)
ETD_FORMAT	(11)
ETD_RAWREAD	(16)
ETD_RAWWRITE	(17)

Ce sont les mêmes commandes que précédemment, mais sans changements de disquettes.

Nous allons maintenant examiner un petit programme en langage machine, utilisant le *Trackdisk Device*. Un exemple d'application est le chargement de quelques secteurs de la disquette dans la mémoire. Si vous possédez un paquet de programmes debogueurs/assembleurs, comme par exemple le *Profimat* ou le *K-SEKA*, vous pouvez examiner le résultat directement. Sinon, vous pouvez aussi, à l'aide des commandes *Open()* et *Write()* de *l'AmigaDOS*, écrire sur la disquette les données obtenues en tant que fichiers, puis les afficher à l'écran ou sortir à l'imprimante avec TYPE et l'option H.

```
;***  Demo Trackdisk-Device: lecture des secteurs 6/87  S.D. ***

ExecBase  = 4                ;Adresse de base Exec
FindTask  = -294             ;chercher la structure task
AddPort   = -354             ;créer un port
RemPort   = -360             ;supprimer le port
OpenLib   = -408             ;ouvrir une bibliothèque
CloseLib  = -414             ;fermer la bibliothèque
OpenDev   = -444             ;ouvrir un Device
CloseDev  = -450             ;fermer le Device
DoIo      = -456             ;mettre I/O en route et attendre
```

```
run:
      move.l  execbase,a6       ;Pointeur su. Bibliothèque Exec
      sub.l   a1,a1             ;task propre
      jsr     FindTask(a6)      ;chercher le task
      move.l  d0,readreply+$10  ;SigTask: Task propre

      lea     readreply,a1
      jsr     AddPort(a6)       ;Add Reply-Port

      lea     diskio,a1         ;structure I/O
      move.l  #0,d0             ;Lecteur DF0:
      clr.l   d1                ;pas de flags
      lea     trddevice,a0      ;nom du Device
      jsr     OpenDev(a6)       ;Open trackdisk.device
      tst.l   d0                ;OK?
      bne     error             ;non : une erreur est survenue!

      lea     diskio,a1
      move.l  #readreply,14(a1) ;set Reply-Port
      move    #2,28(a1)         ;Command: READ
      move.l  #diskbuff,40(a1)  ;Buffec
      move.l  #2*512,36(a1)     ;Longueur: 2 Secteurs
      move.l  #880*512,44(a1)   ;Offset: 880 Secteurs (Root)
      move.l  execbase,a6       ;Adresse de base Exec
      jsr     Dolo(a6)          ;Lire les secteurs!

      move.l  diskio+32,d6      ;IO_ACTUAL en D6
      lea     diskio,a1
      move    #9,28(a1)         ;Command: TD_MOTOR
      move.l  #0,36(a1)         ;Arrêter le
      jsr     Dolo(a6)          ;moteur!

      lea     readreply,a1
      jsr     RemPort(a6)       ;Supprimer le port

      lea     diskio,a1
      jsr     closedev(a6)      ;Fermer le Trackdisk-Device

error:
      rts                       ;fin
```



```
trddevi-    dc.b 'trackdisk.device',0
even

diskio:     blk.l 20,0
readreply:  blk.l 8,0
diskbuff:   blk.b 512*2,0
```

Dans cet exemple, les secteurs 880 et 881 sont chargés du lecteur 0 au *"diskbuff"*. Le secteur 880 est le bloc racine, qui contient comme on le sait le nom de la disquette et quelques autres données.

On appelle ensuite *Dolo()*, pour éteindre le moteur du lecteur de disquettes (pour l'allumer, il aurait fallu inscrire un 1 dans 36(A1)).

Sur le déroulement du programme :

L'appel de la fonction *FindTask()*, à laquelle est transmis comme argument un 0 en A1, provoque la communication du pointeur à votre structure *Task*. Ce pointeur est alors porté dans la structure *Port*, pour que le système sache quel *Task* doit être "réveillé" une fois que le I/O sera terminé.

Etape suivante : le port ainsi préparé est installé dans le système.

On ouvre ensuite le *Trackdisk Device*. En D0, vous pouvez alors sélectionner le lecteur auquel cette fonction doit se rapporter. Si vous désirez traiter plusieurs lecteurs simultanément, il vous faut préparer plusieurs structures I/O et effectuer en conséquence des appels de *OpenDevice()*.

Si une erreur apparaît lors de l'ouverture du *Device*, le programme passe au label *'error'*, et il prend fin. Sinon, la structure I/O est pourvue des données nécessaires :

* d'un pointeur sur la structure de port, où le message d'OK du Device doit aboutir.

* de la commande qui d être exécutée lors des opérations I/O suivantes (ici : 2=COM_READ),

* d'un pointeur sur le tampon à remplir ;

* de la longueur de ce tampon,

* pour la lecture des secteurs, de l'indication de l'offset du début de la disquette, correspondant au numéro de secteur ou bloc *512.

La structure I/O ainsi préparée est alors transmise au système avec *Dolo()*, et la fonction sélectionnée est déclenchée. Le programme attend, tant que l'opération I/O n'est pas terminée. Un paramètre éventuel de retour, comme cela se passe pour la commande TD_PROTSTATUS, est transmis dans IO_ACTUAL, et chargé dans le registre de données D6 avec la commande MOVE.L suivante. Dans l'exemple ci-dessus, il s'agit de la valeur $400, correpondant au nombre d'octets qui ont été lus. Il est vrai que le registre de données n'est plus utilisé par la suite, mais il peut être affiché et examiné si l'on fait usage d'un débogueur.

Suit un nouvel appel de la fonction *Dolo()*, cette fois avec la commande TD_MOTOR (9). Le paramètre dans IO_LENGTH (diskio+36) indique ici si le moteur doit être allumé (1) ou éteint (0).

Ceci étant fait, on met fin à l'utilisation du port, et on ferme le *Device*. Et on en a fini.

Si ce programme est mis en route dans *Profimat* ou *K-SEKA*, les registres du processeur sont affichés, une fois le programme terminé. Vous trouverez dans ce cas en D6 le paramètre de retour $400. Vous pouvez aussi demander l'affichage du nom de la disquette, par la saisie de *'q diskio+432'* (SEKA), le premier octet indiquant la longueur du nom.

S'il y avait un ennui pour une fonction donnée, on obtiendrait une valeur indiquant l'état dans l'octet IO_ERROR. Les valeurs possibles sont ici les suivantes :

20	*otSpecified*	erreur inconnue
21	*NoSecHdr*	il n'existe pas de sector-header
22	*BadSecPreamble*	sector-header non valide
23	*BadSecID*	sector-ID non valide
24	*BadHdrSum*	somme de contrôle header fausse
25	*BadSecSum*	somme de contrôle secteur fausse
26	*TooFewSecs*	pas assez de secteurs disponibles
27	*BadSecHdr*	sector-header non valide
28	*WriteProt*	disquette protégée contre l'écriture
29	*DiskChanged*	il y a eu un changement de disquette
30	*SeekError*	track non retrouvé
31	*NoMem*	pas assez de mémoire
32	*BadUnitNum*	numéro de secteur non valide
33	*BadDriveType*	type de lecteur non valide
34	*DriveInUse*	lecteur déjà actif
35	*PostReset*	phase reset

Ce que nous venons de dire correspond à l'une des directions possibles. L'autre est l'écriture des données sur la disquette. Elle fonctionne exactement de la même manière, à ceci près que la commande doit être placée sur 3 (CMD_WRITE). Ne testez cependant cette possibilité que sur des disquettes de moindre importance, car s'il se produisait une erreur, les données seraient irrémédiablement perdues.

Une autre commande est réellement très intéressante : TD_FORMAT. Elle permet d'écrire dans un ou plusieurs sillons sur la disquette. Les données qui doivent être entreposées dans la mémoire, pointées par le pointeur IO_DATA, sont inscrites dans chacun des sillons indiqués, sans qu'il y ait de test concernant entre autres un éventuel changement de disquette. Cette commande permet donc non seulement de formater des disquettes, mais aussi de les copier, en lisant les secteurs qui sont sur la première disquette et en les recopiant ensuite sur la disquette cible avec TD_FORMAT. L'avantage de cette méthode, en comparaison de ce qui se passe avec CMD_WRITE, est simple : on n'a pas à formater au préalable la disquette !

Si vous voulez cependant formater à nouveau toute une disquette sans utiliser la commande FORMAT de CLI, il vous faut réfléchir à une chose: les données destinées aux sillons doivent être préparées dans un

certain format (cf. le chapitre sur les disq es), ce qui exige beaucoup de travail. Il ne paraît donc pas que ce passage par la commande TD_FORMAT soit à recommander.

Nous allons maintenant considérer une application du *Device Trackdisk*, où nous utiliserons ce que nous avons appris sur la distribution des données sur la disquette. Le programme en langage machine qui va être présenté ici peut servir de programme de diagnostic, soit pour satisfaire votre curiosité, soit pour savoir quelles sont les données qui se sont perdues lors d'une erreur advenue sur la disquette.

Le programme est appelé à partir de CLI, et il faut indiquer un nom de fichier comme paramètre. Le programme en déduit le numéro *hash* et l'affiche. Il charge ensuite le secteur racine de la disquette et affiche le nom de cette disquette. A l'aide du numéro *hash* ainsi calculé, le programme assure l'examen de la chaîne *hash*, à la recherche du nom en question. S'il ne le trouve pas, ou si l'entrée correspondante dans la table *hash* est vide, on obtient en retour le message *'-unknown-'*, et le programme s'interrompt.

Si par contre le fichier ou le répertoire recherché est trouvé, on obtient à la sortie son numéro de bloc et le nombre de blocs de données occupés par le fichier.

Tous ces blocs de données sont ensuite chargés dans l'ordre, et leurs numéros sont affichés. On pourrait aussi prendre ces numéros de blocs dans le bloc *File-Header*, ou éventuellement dans son extension de bloc, mais on ne serait pas assuré ainsi d'obtenir les blocs dans l'ordre. Si donc vous voyez apparaître la fenêtre de dialogue avec le message *'Disk Structure Corrupt'*, vous pouvez tester vos fichiers importants grâce à ce programme pour voir s'ils sont bien complets.

Comme nous ne voulions pas trop compliquer le programme, il n'est pas possible de tester des fichiers se trouvant dans les sous-répertoires. Pour y parvenir, il faut modifier le programme de telle sorte que ce soit la table d'un *directory header block* qui soit utilisée, et non pas celle du *root block*.

Voici c c le programme annoncé. Un certain nombre de fonctions intéressantes y apparaissent, que vous pourrez facilement utiliser dans vos propres programmes. Ce programme a été créé sur l'assembleur K-SEKA, mais vous pouvez très facilement l'adapter à un autre assembleur.

```
;***** File-Tracer; 6/87 S.D. *****

ExecBase  = 4                ;Adresse de base Exec
FindTask  = -294             ;chercher la structure task
AddPort   = -354             ;créer un port
RemPort   = -360             ;supprimer le port
OpenLib   = -408             ;ouvrir une bibliothèque
CloseLib  = -414             ;fermer la bibliothèque
OpenDev   = -444             ;ouvrir un Device
CloseDev  = -450             ;fermer le Device
DoIo      = -456             ;mettre I/O en route et attendre

output    = -60     ;communique la sortie standard
write     = -48     ;affichage des données


run:
     move.l  a0,commpnt
     move.l  d0,commlen
     move.l  execbase,a6
     lea     dosname,a1       ;Nom: dos.library
     clr.l   d0
     jsr     openlib(a6)      ;ouvrir le DOS
     move.l  d0,dosbase
     beq     nodos

     move.l  dosbase,a6
     jsr     output(a6)       ;canal de sortie standard
     move.l  d0,outbase

     sub.l   #1,commlen       ;corriger la longueur du nom
     move.l  commpnt,a0       ;* calculer la valeur hash *
     move.l  commlen,d0       ;Hash=longueur
     clr.l   d2
```

```
        move.l  d0,d1
        subq    #1,d1           ;compteur=longueur-1

hashloop:
        mulu    #13,d0          ;Hash=Hash*13
        move.b  (a0)+,d2
        bsr     upper           ;transformer en Upper-Case
        add     d2,d0           ;Hash=Hash+caractères
        and     #$7ff,d0        ;AND $7FF
        dbra    d1,hashloop     ;boucle

        divu    #72,d0
        swap    d0              ;Hash modulo 72
        addq    #6,d0           ;+6
        move    d0,hash         ;Hash calculé!

        move.l  #hashtxt,d2
        bsr     prtxt           ;"Hash:" affichage
        move    hash,d0
        bsr     phex            ;affichage du numéro de hash

        move.l  execbase,a6     ;Pointeur sur Bibliothèque Exec
        sub.l   a1,a1           ;task propre
        jsr     FindTask(a6)    ;chercher le task
        move.l  d0,readreply+$10 ;set SigTask

        lea     readreply,a1
        jsr     AddPort(a6)     ;Add Reply-Port

        lea diskio,a1
        clr.l   d0
        clr.l   d1
        lea     trddevice,a0
        jsr     OpenDev(a6)     ;Open trackdisk.device
        tst.l   d0
        bne     error

        move.l  #880,d0         ;Secteur 880 (Secteur Racine)
        bsr     loadsec         ;charger dans le tampon de la disquette
```



```
        mo      #voltxt,d2
        b       prtxt           ;"Volume:" afficher
        move.l  dosbase,a6
        move.l  outbase,d1
        move.l  #diskbuff+433,d2 ;adresse du nom
        clr.l   d3
        move.b  diskbuff+432,d3 ;longueur du nom
        jsr     write(a6)       ;afficher le nom de la disquette

        lea     diskbuff,a0
        clr.l   d0
        move    hash,d0
        lsl     #2,d0           ;Hash*4=pointeur sur secteur
        move.l  0(a0,d0),d0     ;va chercher le numéro du secteur
        tst.l   d0 ;Pointeur da?
        beq     none            ;non: entrée hash non occupée
loadloop:
        move.l  d0,sector
        bsr     loadsec         ;charger le secteur suivant

        move.l  commpnt,a0
        lea     diskbuff+432,a1 ;longueur du nom depuis le header
        move.l  commlen,d0
        cmp.b   (a1)+,d0        ;la longueur est-elle exacte?
        bne     nextsec         ;non

        subq    #1,d0
namelop:
        move.b  (a1)+,d2
        bsr     upper           ;caractères dans Upper-Case
        move    d2,d1
        move.b  (a0)+,d2
        bsr     upper           ;caractères dans Upper-Case
        cmp.b   d1,d2           ;comparer les caractères
        bne     nextsec         ;faux
        dbra    d0,namelop
        bra     sectorok        ;le nom est exact!

nextsec:
        move.l  diskbuff+496,d0 ;Pointeur sur le secteur suivant
```

```
      tst.l   d0              ;y en a-t-il
      bne     loadloop        ;oui : poursuivre

none:                         ;sinon
      move.l  #unknown,d2
      bsr     prtxt           ;afficher "-unknown-"
      bra     fin             ;et fin

sectorok:
      move.l  #header,d2
      bsr     prtxt           ;afficher "Header:"
      move.l  sector,d0       ;numéro de secteur du header
      bsr     phex            ;afficher

      cmp.l   #2,diskbuff+508 ;Dir-Header?
      bne     nodir           ;non
      move.l  #dirtxt,d2
      bsr     prtxt           ;afficher "Directory"
      bra     fin             ;et fin

nodir:
      move.l  diskbuff+504,d0 ;Extension
      tst.l   d0              ;elle existe?
      beq     noextens        ;non
      move.l  d0,-(sp)        ;sauvegarder D0
      move.l  #extxt,d2
      bsr     prtxt           ;afficher "Extension"
      move.l  (sp)+,d0        ;va chercher le n° du secteur
      bsr     phex            ;et l'affiche

noextens:
      move.l  #crtxt,d2
      bsr     prtxt           ;affiche CR
      move.l  diskbuff+8,d0
      bsr     phex            ;affiche le nombre de secteurs
      move.l  #sectxt,d2
      bsr     prtxt           ;affiches "secteurs"
      clr     counter         ;compteur des colonnes=0
      bra     seclop1         ;affiche les secteurs
```



```
secloop:
      move.l  sector,d0
      bsr     phex            ;affiche le n° du secteur
      add     #1,counter      ;compteur+1
      cmp     #8,counter      ;8 nombres affichés?
      bne     seclop1         ;non
      clr     counter         ;sinon supprimer le compteur
      move.l  #crtxt,d2
      bsr     prtxt           ;et afficher CR
seclop1:
      move.l  diskbuff+16,d0  ;secteur suivant
      tst.l   d0              ;il en existe un?
      beq     fin             ;non : terminé
      move.l  d0,sector
      bsr     loadsec         ;charger le secteur suivant
      bra     secloop         ;etc...

fin:
      move.l  #crtxt,d2
      bsr     prtxt           ;afficher CR

      move.l  execbase,a6
      lea     readreply,a1
      jsr     RemPort(a6)     ;Remove Port
      lea     diskio,a1
      jsr     closedev(a6)    ;Close Trackdisk-Device
error:
      move.l  dosbase,a1
      jsr     closelib(a6)    ;Close DOS
nodos:
      rts                     ;fin

loadsec:                      ;charger secteur D0
      lea     diskio,a1
      move.l  #readreply,14(a1) ;set Reply-Port
      move    #2,28(a1)         ;Command: READ
      move.l  #diskbuff,40(a1)  ;Buffer
      move.l  #512,36(a1)       ;Longueur: 1 secteur
      mulu    #512,d0
      move.l  d0,44(a1)         ;Offset: n° de secteur+512
```

```
        move.l  execbase,a6
        jsr     Dolo(a6)        ;Lecture se(    r

        lea     diskio,a1
        move    #9,28(a1)       ;TD_MOTOR
        move.l  #0,36(a1)       ;Motor éteint
        jsr     Dolo(a6)
        rts

phex:                           ;affiche D0 en hexadéc.
        lea     outpuff,a0
        move    d0,d2
        move    #3,d3           ;4 chiffres
niblop:
        rol     #4,d2           ;Nibble de gauche vers le bas
        move    d2,d1
        and     #$f,d1          ;note
        add     #$30,d1         ;transforme en ASCII
        cmp     #'9',d1         ;chiffre?
        bls     nibok           ;oui
        add     #7,d1           ;sinon corriger
nibok:
        move.b  d1,(a0)+        ;caractères dans le tampon de sortie
        dbra    d3,niblop       ;poursuivre la boucle
        move.b  #$20,(a0)       ;caractères vides à la fin

        move.l  dosbase,a6
        move.l  outbase,d1
        move.l  #outpuff,d2     ;tampon de sortie
        move.l  #5,d3           ;5 caractères
        jmp     Write(a6)       ;afficher

prtxt:                          ;afficher le texte à partir de (D2)
        move.l  dosbase,a6
        move.l  outbase,d1
        move.l  #12,d3          ;longueur 12 caractères
        jmp     Write(a6)       ;afficher le texte

upper:                          ;transformer D2 en Upper-Case
        cmp.b   #'a',d2         ;caractère <'a'?
```



```
        bl      upperx          ;oui: laisser ainsi
        c   b   #'z',d2         ;caractère >'z'?
        bhi     upperx          ;oui: laisser ainsi
        sub     #$20,d2         ;sinon corriger
upperx:
        rts                     ;terminer


trddevice:  dc.b 'trackdisk.device',0
dosname:    dc.b 'dos.library',0

hashtxt:    dc.b $a,'Num.Hash:   '
voltxt:     dc.b $a,'Volume:     '
unknown:    dc.b $a,'-unknown-   '
header:     dc.b $a,'Header:     '
extxt:      dc.b $a,'Extension: '
dirtxt:     dc.b $a,'Directory ',$a
sectxt:     dc.b 'Secteurs:  ',$a
crtxt:      dc.b '           ',$a

data

even
outpuff:    blk.b 6     ;Buffer pour l'affichage en hexadéc.
sector:     blk.l 1     ;mémoire intermédiaire des secteurs
counter:    blk.w 1     ;compteur pour le formatage de l'affichage
dosbase:    blk.l 1     ;adresse de la base DOS
outbase:    blk.l 1     ;handle affichage standard
hash:       blk.w 1     ;numéro du hash
commpnt:    blk.l 1     ;Pointeur sur la ligne de saisie
commlen:    blk.l 1     ;Longueur de la ligne de saisie

diskio:                 ;structure I/O de la disquette
message:    blk.b 20,0
io:         blk.b 12,0
ioreq:      blk.b 16,0

readreply:  blk.l 8,0
diskbuff:   blk.b 512,0
```

De la même manière, on peut aussi écrire un programme qui charge un fichier à partir de la disquette, sans solliciter le DOS. Le seul problème dans ce cas est qu'il faut d'abord copier les données proprement dites en mémoire.

4.2 Console-Device : Fenêtre Editeur

Ce *Device*, à l'aide duquel on peut préparer des fenêtres servant à la saisie au clavier et à l'affichage, n'appartient pas de plein droit au cadre des *Devices standard*. Il ne peut en effet pas être ouvert et utilisé simplement pour lui-même : il doit être en relation avec une fenêtre. C'est cette fenêtre qui sert pour le *console-device* de support aux saisies et à l'affichage.

Avant donc de pouvoir ouvrir le *Device* lui-même, on doit d'abord ouvrir une fenêtre. Il faut pour cela ouvrir successivement la bibliothèque *Intuition*, un écran et la fenêtre en dernier lieu. Le pointeur qu'on obtient en conséquence, pointant sur la structure de la fenêtre, est transmis lors de l'ouverture du *console-device*.

Nous obtenons ainsi une fenêtre sur un écran ouvert expressément, où l'on peut voir un curseur dans le coin supérieur gauche. Ce curseur n'a encore aucune fonction : il faut d'abord programmer la saisie au clavier et l'affichage des caractères dans la fenêtre.

On a donc besoin de deux structures I/O, l'une pour la saisie, l'autre pour l'affichage. A ces deux structures correspondent évidemment deux ports de message, pour que le *Device* sache d'où les données proviendront, ou bien quelle est leur destination.

Avant de poursuivre avec la théorie, un peu rébarbative, il faut bien l'avouer, considérez donc un instant le programme qui suit, reproduisant les étapes décrites plus haut. Il ouvre un *Screen* et une fenêtre, dans laquelle les saisies et les affichages pourront s'effectuer par l'intermédiaire du *console-device*. Les caractères saisis au clavier sont directement affichés dans la fenêtre, et les touches *'Return'* et *'Backspace'* subissent un traitement spécial. Si l'on clique avec la souris sur le rectangle de fermeture de la fenêtre, le programme se termine.

On peut aussi définir d'autres actions avec la souris, étant donné que la reconnaissance d'un clic sur la souris est prévue dans le programme.

Voici le programme :

```
;** Programme Démonstration pour le Console-device  6/87  S.D. **

FindTask    = -294              ;chercher la structure task
AddPort     = -354              ;créer un port
RemPort     = -360              ;supprimer le port
OpenLib     = -408              ;ouvrir une bibliothèque
CloseLib    = -414              ;fermer la bibliothèque
OpenDev     = -444              ;ouvrir un Device
CloseDev    = -450              ;fermer le Device
DoIo        = -456              ;exécuter I/O
execbase    = 4                 ;Adresse de base Exec
GetMsg      = -372              ;Rechercher le message
SendIo      = -462              ;mettre en route l'I/O

;    ** Fonctions d'Intuition **
openscreen  = -198      ;ouvrir le Screen
closescreen = -66       ;fermer le Screen
openwindow  = -204      ;ouvrir la fenêtre
closewindow = -72       ;fermer la fenêtre


run:
      bsr     openint           ;ouvrir Intuition
      bsr     scropen           ;ouvrir le Screen
      bsr     windopen          ;ouvrir la fenêtre

      move.l  execbase,a6       ;Pointeur sur bibliothèque Exec
      sub.l   a1,a1             ;Task propre
      jsr     FindTask(a6)      ;chercher le Task
      move.l  d0,readreply+$10  ;poser SigTask

      lea     readreply,a1
      jsr     AddPort(a6)       ;Add Read-Reply-Port

      lea     writerep,a1
```

```
        jsr     AddPort(a6)       ;Add Write-i   y-Port

        lea     readio,a1'
        move.l  windowhd,readio+$28 ;notre fenêtre
        move.l  #48,readio+$24  ;longueur de la structure
        clr.l   d0
        clr.l   d1
        lea     devicename,a0
        jsr     OpenDev(a6)       ;Open Console-Device
        tst.l   d0
        bne     error

        move.l  readio+$14,writeio+$14 ;DEVICE
        move.l  readio+$18,writeio+$18 ;copier UNIT

go:
        bsr     queueread         ;lecture de la saisie

loop:                             ;* traitement des événements *
        move.l  execbase,a6
        move.l  windowhd,a0
        move.l  86(a0),a0         ;User-Port fenêtre
        jsr     GetMsg(a6)
        tst.l   d0
        bne     wevent            ;événement fenêtre

        lea     readreply,a0
        jsr     GetMsg(a6)        ;événement Console (touche)?
        tst.l   d0
        beq     loop              ;pas d'événement

cevent:                           ;* Traitement de la touche *
        bsr     conout            ;affichage du caractère
        cmp.b   #$d,buffer        ;Return?
        bne     no1               ;non
        move.b  #$a,buffer        ;sinon afficher LF
        bsr     conout

no1:
        cmp.b   #$8,buffer        ;Backspace?
```



```
        bne     no2
        mov     #' ',buffer       ;sinon effacer le caractère
        bsr     conout
        move.b  #8,buffer
        bsr     conout            ;et revenir

no2:
        bra     go                ;et ainsi de suite

wevent:             ;* traiter l'événement fenêtre *
        move.l  d0,a0
        move.l  $16(a0),d6        ;Message en D6
        cmp.l   #$2000000,d6      ;Window-Close?
        beq     fin               ;oui: Fin

        ;* ici peut prendre place un autre traitement: *

        move.l  windowhd,a0
        move.l  12(a0),d5         ;position de la souris en D5

        ;* placer par ex. le curseur à l'emplacement de la souris... *


fin:                ;* Fin du programme : tout fermer *
        lea     readreply,a1
        jsr     RemPort(a6)       ;Remove Port
        lea     readio,a1
        jsr     closedev(a6)      ;Close Device
        lea     writerep,a1
        jsr     RemPort(a6)       ;Remove Port
error:
        bsr     windclose         ;Fermer la fenêtre
        bsr     scrclose          ;Fermer le Screen
        bsr     closeint          ;Fermer Intuition

        rts                       ;* FIN *
```

```
;** Sous-Routines **

queueread:              ;* démarrage de la saisie sur console *
      move.l  execbase,a6
      lea     readio,a1
      move    #2,28(a1)         ;Command: READ
      move.l  #buffer,40(a1)    ;Buffer
      move.l  #1,36(a1)         ;Longueur:
      move.l  #readreply,14(a1) ;set Reply-Port
      jsr     sendIo(a6)        ;lance la fonction
      rts

conout:                 ;* affichage d'un caractère *
      move.l  execbase,a6
      lea     writeio,a1
      move    #3,28(a1)         ;Command: WRITE
      move.l  #buffer,40(a1)    ;Buffer
      move.l  #1,36(a1)         ;Longueur:
      move.l  #writerep,14(a1) ;set Reply-Port
      jsr     DoIo(a6)          ;exécute la fonction
      rts

openint:                ;* Ouvre Intuition  *
      move.l  execbase,a6
      lea     intname,a1        ;nom de la Library
      jsr     openlib(a6)
      move.l  d0,intbase
      rts

closeint:               ;* Fermer Intuition  *
      move.l  execbase,a6
      move.l  intbase,a1
      jsr     closelib(a6)
      rts

scropen:                ;* Ouvrir un Screen *
      move.l  intbase,a6
      lea     screen_defs,a0
      jsr     openscreen(a6)
      move.l  d0,screenhd
```



```
      rts

scrclose:               ;* Fermer le Screen *
      move.l  intbase,a6
      move.l  screenhd,a0
      jsr     closescreen(a6)
      rts

windopen:               ;* Ouvrir une fenêtre *
      move.l  intbase,a6
      lea     windowdef,a0
      jsr     openwindow(a6)
      move.l  d0,windowhd
      rts

windclose:              ;* Fermer la fenêtre *
      move.l  intbase,a6
      move.l  windowhd,a0
      jsr     closewindow(a6)
      rts

screen_defs:          ;* Structure de screen *
      dc.w 0,0        ;Position
      dc.w 640,200    ;taille
      dc.w 4          ;Bitmaps
      dc.b 0,1        ;couleurs
      dc.w $800       ;Mode
      dc.w 15         ;Type
      dc.l 0          ;Police de caractères standard
      dc.l titel      ;Titre de l'écran
      dc.l 0          ;Titre standard
      dc.l 0          ;pas de gadgets

windowdef:            ;* Structure de fenêtre *
      dc.w 10,20      ;Position
      dc.w 300,150    ;taille
      dc.b 0,1        ;couleurs
      dc.l $208       ;Flags IDCMP
      dc.l $100f      ;Window-flags
      dc.l 0          ;pas de gadgets
```

```
        dc.l 0          ;pas de menu-checks
        dc.l windname   ;nom de la fenêtre

screenhd: dc.l 0        ;Pointeur sur la structure d'écran
        dc.l 0          ;pas de Bitmap
        dc.w 100,50     ;taille minimale
        dc.w 300,200    ;hauteur max.
        dc.w $f         ;Type de Screen

titel:      dc.b "Editor-Screen",0
windname:   dc.b "Fenêtre-Console",0
intname:    dc.b "intuition.library",0
devicename: dc.b 'console.device',0
even
windowhd:   blk.l 1
intbase:    blk.l 1
conbase:    blk.l 1

readio:
message:    blk.b 20,0
io:         blk.b 12,0
ioreq:      blk.b 16,0

writeio:
            blk.b 20,0
            blk.b 12,0
            blk.b 16,0

readreply:  blk.l 8,0
writerep:   blk.l 8,0
buffer:     blk.b 80,0
```

Les séquences qui doivent déclencher diverses fonctions dans la fenêtre sont les mêmes que celles que l'on trouve pour les fenêtres RAW: ou CON: ouvertes par les soins du DOS.

4.3 Na ıtor-Device : synthèse de la parole

Narrator permet à l'Amiga de s'exprimer à haute voix, c'est-à-dire tout simplement de parler. C'est tout de même là un point intéressant, et il vaut la peine de tenir compte de cette possibilité dans vos programmes.

Le *Narrator* est un paquet de programmes, conçu comme un *Device*. On peut donc demander la sortie d'un texte sous forme parlée, et continuer pendant ce temps à travailler. La structure I/O étendue du *Narrator Device* est la suivante :

Mot	Nom	Signification
0	RATE	Vitesse de parole en mots/minutes
1	PITCH	Fréquence de base en hertz
2	MODE	Mode (0= avec, 1= sans intonation)
3	SEX	Sexe de la voix (0=masculin, 1=féminin)
4	CHMASKS	Pointeur sur le champ des masques du canal
6	NUMMASKS	Nombre de masques pour le canal
7	VOLUME	Volume
8	SAMPFREQ	Fréquence d'échantillonage
9	MOUTHS	Flag pour la représentation de la bouche (octet)
	CHANMASK	Canal actuel (singification interne seulement)

La programmation du *Narrator device* est semblable à celle des autres *devices*. Ce qui vient ici en plus, c'est l'utilisateur du "traducteur" ou *translator*, qui traduit un texte normal en phonèmes. Ce *Translator* n'est pas un *device*, mais une bibliothèque, qui ne contient qu'une seule fonction.

Voici maintenant un programme en langage machine, qui fait entendre un texte d'exemple. Vous pouvez vous amuser à effectuer vous-même vos expériences avec ce programme, étant donné que vous avez la possibilité de modifier les paramètres à volonté, et de tester les résultats obtenus.

L'utilisation d'un assembleur avec débo[illegible]ur intégré est ici encore un avantage : on peut utiliser le *Profima*[illegible] ou le *K-SEKA*, sur lequel ce programme a été écrit.

```
;*****  Programme Démo Narrator  6/87  S.D.  *****

ExecBase   =4      ;Adresse de base Exec
FindTask   =-294   ;Find Task
AddPort    =-354   ;Add Port
RemPort    =-360   ;Remove Port
OpenLib    =-408   ;Open Library
closelib   =-414   ;Close Library
OpenDev    =-444   ;Open Device
CloseDev   =-450   ;Close Device
DoIo       =-456   ;Do I/O
SendIo     = -462  ;Send I/O
Translate  =-30    ;Translate Text

run:               ;** Initialiser et ouvrir le système **
     move.l  execbase,a6
     lea     transname,a1
     clr.l   d0
     jsr     openlib(a6)       ;Open Translator-Library
     move.l  d0,tranbase
     beq     error

     sub.l   a1,a1             ;n° de Task = 0:  Task propre
     jsr     FindTask(a6)      ;chercher le Task

     move.l  d0,writerep+$10   ;set SigTask

     lea     writerep,a1
     jsr     addport(a6)       ;Add Reply-Port

     lea     talkio,a1
     clr.l   d0
     clr.l   d1
     lea     nardevice,a0
     jsr     opfinv(a6)        ;Open Narrator.device
     tst.l   d0
```



```
     bne     error

     lea     talkio,a1
     move.l  #writerep,14(a1) ;set Reply-Port (*)
     move    #150,48(a1)      ;Rate (40-400)
     move    #110,50(a1)      ;Pitch (65-320)
     move    #0,52(a1)        ;Mode: avec intonation (0/1)
     move    #0,54(a1)        ;sexe: masculin (0/1)
     move.l  #amaps,56(a1)    ;Masques (*)
     move    #4,60(a1)        ;4 Masques (*)
     move    #64,62(a1)       ;volume (0-64)
     move    #22200,64(a1)    ;vitesse d'échantillonage (5000-28000)

sayit:                  ;** Traduire et dire un texte **
     lea     intext,a0        ;texte original
     move.l  #outtext-intext,d0 ;sa longueur
     lea     outtext,a1       ;Buffer pour la traduction
     move.l  #512,d1          ;sa longueur
     move.l  tranbase,a6
     jsr     Translate(a6)    ;traduction du texte

     lea     talkio,a1
     move    #3,28(a1)        ;Command: Write
     move.l  #512,36(a1)      ;Length
     move.l  #outtext,40(a1)  ;Buffer
     move.l  execbase,a6
     jsr     DoIo(a6)         ;Say it!!

qu:                     ;** Fin **
     lea writerep,a1
     jsr RemPort(a6)          ;Remove Port

     lea talkio,a1
     jsr closedev(a6)         ;Close Narrator

     move.l tranbase,a1
     jsr     closelib(a6)     ;Close Translator-Lib

     clr.l   d0
error:
```

```
    rts

transname:   dc.b 'translator.library',0
nardevice:   dc.b 'narrator.device',0
amaps:       dc.b 3,5,10,12
intext:      dc.b 'hello, i am the amiga computer.',0
even
outtext:     blk.l 128,0
tranbase:    blk.l 1,0
narread:     blk.l 20,0
talkio:      blk.l 20,0
writerep:    blk.l 8,0
```

Dans la préparation du secteur I/O pour les différents modes ou les différentes fréquences, les valeurs qu'il faut indiquer sans faute sont celles qui sont indiquées par une astéristique dans le listing ci-dessus. Toutes les autres sont pourvues automatiquement de leur niveau standard *(Default)*. Ces valeurs sont également prescrites dans le programme, et elles peuvent varier dans des limites que nous avons indiquées entre parenthèses.

Le *narrator device* offre en outre la possibilité de transférer des données au programme appelant pendant la sortie au haut-parleur. Ceci n'est évidemment possible que lorsque cette sortie est commandée non par *Dolo()*, mais par *Sendlo()n* pour que les données puissent être aussi reçues durant le temps de la sortie.

Les données reçues représentent un motif de bits, qui constituent une bouche à l'écran ; cette bouche correspond au phonème qui est prononcé à ce moment-là. Nous n'entrerons pas dans les détails sur ce point, car il est lié à l'affichage graphique, et par suite expliqué dans les ouvrages sur le graphisme. Nous en dirons ceci seulement : le graphisme en question est plutôt primitif, puisqu'il représente la bouche sous forme de parallélogramme. La largeur et la hauteur de cette forme sont obtenues à partir du *device* dans l'extension de la structure *Read-Request-I/O*.

L'extension ...onstituée de la façon suivante :

Offset	Nom	Contenu
48	MRB_WIDTH	Largeur de la "bouche"
49	MRB_HEIGHT	Hauteur
50	MRB_SHAPE	Octet interne des données
51	MRB_PAD	Octet complet pour l'adresse paire

La fonction I/O peut évidemment avoir des ratés. Les messages d'erreur que l'on reçoit dans ce cas peuvent prendre les valeurs suivantes :

Messages d'erreur du Narrator :

Numéro	Signification
-2	pas assez de place en mémoire
-3	l'audio-device n'existe pas
-4	la bibliothèque ne peut pas être créée
-5	mauvais numéro d'unité dans la structure I/O (seulement 0)
-6	pas de canaux audio disponibles
-7	commande inconnue
-8	données destinées au haut-parleur lues mais non écrites
-9	pas d'ouverture possible
-20	phonème imprononçable
-21	vitesse non valide
-22	pitch non valide
-23	sexe non valide
-24	mode non valide
-25	fréquence d'échantillonnage non valide
-26	volume non valide

4.4 Serial Device : l'interface RS

Ce *device* est responsable de la communication sérielle avec le monde extérieur. Ces entrées et sorties par le port sériel de l'Amiga peuvent elles aussi être réalisées avec les fonctions DOS normales, à condition que l'on indique comme nom de fichier SER:. Cette méthode présente cependant quelques inconvénients très sérieux.

On sait que l'inconvénient habituel de l'utilisation du DOS tient dans le fait que les entrées-sorties ne fonctionnent pas en tâche de fond, et qu'on est donc obligé d'attendre jusqu'à ce qu'elles soient terminées. On peut néanmoins contourner cette difficulté, lors de la programmation du *device*, au moyen de la fonction *SendIo()*.

Un autre inconvénient, qui se présente dans le cas présent, est le suivant : les paramètres sériels, comme par exemple la vitesse de transfert des données, doivent être réglés à l'avance avec le programme *Preferences*.

Le *Serial-Device* offre à cet égard une fonction qui lui est propre, permettant de régler tous les paramètres. Pour cette fonction comme pour les autres, on dispose d'une structure I/O d'extension, possédant les entrées suivantes (les valeurs standard sont placées entre parenthèses) :

Offset	IO_Name	Contenu
0	CTLCHAR	Signe de contrôle: xON, xOFF, libre, libre ($1113000)
4	RBUFLEN	Longueur du tampon de saisie($200)
8	WBUFLEN	Longueur du tampon de sortie ($200)
12	BAUD	Vitesse en bauds (9600)
16	BRKTIME	Longueur du break en microsecondes (250000)
20	TERMARRAY	Champ des caractères d'interruption (8 octets)
28	READLEN	Bits par caractère à la lecture (8)
29	WRITELEN	Bits par caractère à l'émission (8)
30	STOPBITS	Nombre de stopbits (1)
31	SERFLAGS	Flags sériel (cf.infra) ($20)
32	STATUS	Mot décrivant l'état

Les bits d ot décrivant l'état obtenu en retour sont les suivants :

Bit	Si	Alors
0	0	busy, le transfert est en cours
1	0	paper out, le récepteur n'est pas prêt
2	0	select, sélectionné
3	0	Data Set Ready (DSR)
4	0	Clear To Send (CTS)
5	0	Carrier Detect (CD)
6	0	Ready To Send (RTS)
7	0	Data Terminal Ready (DTR)
8	1	read overrun, dépassement du tampon
9	1	break sent, Break transmis
10	1	break received, Break reçu
11	1	transmit x-OFFed, xOFF transmis
12	1	receive x-OFFed, xOFF reçu
13-15		réservé

Les bits du flag dans IO_SERFLAGS ont la signification suivante :

Bit	Nom	Signification lorsque le bit est positionné
0	PARITY_ON	Bit de parité souhaitée
1	PARITY_ODD	Parité impaire
2		Non utilisé
3	QUEUEDBRK	Break en arrière-plan
4	RAD_BOOGIE	Mode vitesse supérieure enclenché
5	SHARED	Accès général possible
6	EOFMODE	Reconnaissance EOF en service
7	XDISABLED	xON/xOFF hors service

Le *Serial-Device* possède, en dehors des commandes standard, trois commandes supplémentaires :

Nombre	Nom_SDCMD	Fonction
9	QUERY	
10	BREAK	Emission du break
11	SETPARAMS	Réglage des paramètres

Pour faire en particulier une démonstration de la dernière de ces commandes, celle qui porte le nom de SDCMD_SETPARAMS, nous vous donnons à nouveau un programme d'exemple. Dans ce programme, la vitesse de transfert des données est réglée sur 1200, et on lance le célèbre texte *"Hello, le word!"*.

```
;***** Serial-Device-Demonstration 6/87 S.D. *****

ExecBase  = 4                ;Adresse de base Exec
FindTask  = -294             ;chercher la structure task
AddPort   = -354             ;créer un port
RemPort   = -360             ;supprimer le port
OpenLib   = -408             ;ouvrir une bibliothèque
CloseLib  = -414             ;fermer la bibliothèque
OpenDev   = -444             ;ouvrir un Device
CloseDev  = -450             ;fermer le Device
DoIo      = -456             ;mettre I/O en route et attendre

output    = -60     ;communique la sortie standard
write     = -48     ;affichage des données


run:
     move.l  execbase,a6      ;Pointeur sur bibliothèque Exec
     sub.l   a1,a1            ;Task propre
     jsr     FindTask(a6)     ;chercher le Task
     move.l  d0,readreply+$10 ;poser SigTask
```



```
     lea     reply,a1
     jsr     AddPort(a6)      ;Créer Reply-Port

     lea     devio,a1         ;Pointeur sur la structure I/O

     clr.l   d0
     clr.l   d1
     lea     devicename,a0
     jsr     OpenDev(a6)      ;Ouvrir le serial-device
     tst.l   d0               ;OK?
     bne     error            ;non: Fin

     move.l  #reply,14(a1)    ;set Reply-Port

     move    #11,28(a1)       ;Command: SETPARAMS
     move.l  #1200,ioextd+12  ;1200 bauds
     jsr     DoIo(a6)         ;donner les paramètres

     move    #3,28(a1)        ;Command: WRITE
     move.l  #text,40(a1)     ;Buffer
     move.l  #textl,36(a1)    ;Longueur:
     jsr     DoIo(a6)         ;Envoyer le texte

     lea     reply,a1
     jsr     RemPort(a6)      ;Remove Port

     lea     devio,a1
     jsr     CloseDev(a6)     ;Close Device

error:
     rts                      ;Fin

devicename:  dc.b 'serial.device',0
text:        dc.b 'hello, world!'
textl        = *-text
even
devio:
message:  blk.w 10,0
io:       blk.w  6,0
ioreq:    blk.w  8,0
```

```
ioextd:   blk.w 17,0
reply:    blk.w  8,0
```

Selon cette méthode, on peut évidemment régler bien d'autres paramètres de l'interface RS232.

4.5 Printer-Device : Programmation de l'imprimante

On peut adresser l'imprimante autrement que par l'intermédiaire du canal PRT:. On se sert pour cela du *Printer-Device*, qui s'occupe certes de l'impression normale, mais qui possède une autre propriété : la sortie à l'imprimante du contenu d'une fenêtre ou d'un écran.

Pour l'impression normale, on a besoin de l'extension de structure I/O suivante, portant le nom de *IOPrtCmdReq* :

Offset	Nom	Contenu
32	io_PrtCommand	Commande d'impression
34	io_Param0	Paramètre de la commande
35	io_Param1	Paramètre de la commande
36	io_Param2	Paramètre de la commande
37	io_Param2	Paramètre de la commande

Les commandes de contrôle sont transmises ici à l'imprimante, et c'est aussi là que se déterminent les styles, les caractères, etc... La sortie se fait avec la commande PRTCOMMAND.

Les commandes possibles, ajoutées aux commandes standard pour ce *device*, sont les suivantes :

Valeur	Nom	Fonction
9	PRD_RAWWRITE	Sortie sans conversion des signes de contr.
10	PRD_PRTCOMMAND	Envoi de la commande d'impression
11	PRD_DUMPRPORT	Sortie du contenu écran fenêtre

Si l'on n'utilise pas RAWWRITE et si l'on sort les données par la fonction WRITE normale, les signes de contrôle standard de l'Amiga sont convertis, pour correspondre aux signes de contrôle spécifiques de chaque imprimante. De cette façon, un seul et même programme peut envoyer ses données à une imprimante quelconque, sans avoir besoin de connaître les particularités de cette imprimante.

La commande DUMPRPORT offre, nous l'avons dit, la possibilité de sortir à l'imprimante le contenu d'une fenêtre ou d'un écran. La structure I/O supplémentaire nécessaire à cet effet, du nom de IODEPReq, est constituée de la façon suivante :

Offset	Nom	Contenu
32	RastPort	Pointeur sur le rastport à imprimer
36	ColorMap	Pointeur sur la table des couleurs
40	Modes	Mode graphique du Viewport
44	ScrX	Coordonnée en X de la fenêtre/de l'écran
46	ScrY	Coordonnée en Y
48	ScrWidth	Largeur de la fenêtre/de l'écran
50	ScrHeight	Hauteur de la fenêtre/de l'écran
52	DestCols	Largeur de la cible
56	DestRows	Hauteur de la cible
60	Special	Flags pour les fonctions spéciales

4.6 Parallel-Device : Entrées/Sor[illegible] digitales

D'autres périphériques peuvent être connectés au même endroit que l'imprimante, à condition qu'ils disposent des caractéristiques électriques adéquates. Cela permet d'effectuer l'entrée et la sortie de données digitales. De plus, on peut de la sorte programmer des bits isolés des 8 conduits de données pour qu'ils servent aux entrées et les autres aux sorties. Ce n'est possible néanmoins que par programmation directe du registre $BRE301 du hardware.

Nous nous en tiendrons par suite à la programmation du port tout entier comme port d'entrée ou comme port de sortie. Il existe à cet effet un *device* : le *Parallel-Device*. Pour pouvoir l'utiliser, il faut de nouveau réaliser une extension de la structure I/O normale. Cette extension est constituée comme suit :

Offset	Nom	Contenu
48	PWBufLen	Longueur du tampon de sortie
52	ParStatus	Etat du device
53	ParFlags	Flag parallèle
54	PTermArray	Masque d'interruption
-61		

L'octet d'état contient les bits d'état suivants :

Bit	Nom	Signification si le bit est posé
0	PSEL	Imprimante sélectionnée
1	PAPEROUT	Papier manquant
2	PBUSY	Imprimante occupée
3	RWDIR	Direction des données (0=lire, 1=écrire)

Le flag pa[illegible]ie est constitué des bits suivants :

Bit	Nom	Signification si le bit est posé
0	EOFMODE	Mode EOF en service
5	SHARED	Accès possible pour d'autres tâches élémentaires

Lorsqu'on effectue une lecture à partir du port parallèle, il se pose à nouveau la question de savoir à quoi le récepteur reconnaîtra que le transfert est terminé. Il existe pour cela ici la possibilité d'interrompre la réception pour une suite d'octets déterminée. Cette suite est placée dans les deux mots longs de *TermArray*. La séquence d'interruption est activée lorsque le bit 1 de l'octet de flag est positionné (EOFMODE) et lorsqu'on appelle alors la commande SETPARAMS (10).

4.7 Gameport-Device : Souris et Manette de jeu

Ce *device* permet de traiter toutes les entrées à travers ces deux ports, provenant d'une souris ou d'une manette de jeu. La structure I/O de ce *device* ne nécessite pas d'extension, mais on utilise pourtant deux autres structures.

L'une d'elles est la structure *Event*, qui a été présentée précédemment, dans le paragraphe sur la fenêtre RAW:. Son nom est *InputEvent*, et elle est constituée de la façon suivante :

Offset	Nom	Contenu
0	NexEvent	Pointeur éventuel sur la structure suivante
4	Class	Classe de l'événement
5	SubClass	Sous-classe éventuelle de l'événement
6	Code	Code de l'événement
8	Qualifier	Type de l'événement

10	X	Coordonnée X
12	Y	Coordonnée en Y, le plus souvent relative
14	TimeStamp	Secondes, microsecondes

L'autre structure est nécessaire au réglage de l'événement, qui déclenche la transmission des paramètres dans la structure *Event*. Ce peut le fait que l'on presse ou que l'on relâche un bouton, ou le mouvement de la souris ou de la manette. La valeur souhaitée est à reporter dans le mot correspondant de la structure suivante, nommée *GamePortTrigger* :

Offset	Nom	Contenu
0	Keys	Modification de la touche : bit 0 : touche pressée bit 1 : touche relâchée
2	TimeOut	Interruption lorsqu'il s'écoule un certain nombre de 1/50 de seconde, le nombre étant indiqué ici
4	XDelta	mouvement horizontal
6	YDelta	mouvement vertival

Si l'on veut donc attendre 10 secondes, jusqu'à ce qu'une manette surveillée de cette façon soit mise en mouvement ou qu'une touche soit pressée, on écrit dans *Keys* un 1 (touchée pressée), dans *Timeout* 500 (500/50=10 secondes), et dans *XDelta* un 1.

Avant de pouvoir mettre cette surveillance en marche, il faut évidemment effectuer quelques préparatifs. Il faut tout d'abord ouvrir le *GamePort-Device*, puis annoncer le port dans lequel la manette transmet ses données comme étant le port *Joystick*, et ce n'est qu'ensuite que l'on peut attendre l'événement sélectionné.

Les commandes de ce *device* sont les suivantes :

Commande	Nom	Signification
9	READEVENT	Mise en route de la surveillance
10	ASKCTYPE	Interrogation du type de port
11	SETCTYPE	Position du type de port
12	ASKTRIGGER	Communiquer les événements
13	SETTRIGGER	Positionner les événements

Les types de ports possibles, pouvant être réglés à l'aide SETCTYPE, sont les suivants :

Numéro	Nom	Signification
0	NOCONTROLLER	Fermer le port
1	MOUSE	Port de la souris
2	RELJOYSTICK	Port pour les joysticks relatifs
3	ABSJOYSTICK	Port pour les joysticks absolus

et il y a un type supplémentaire :

-1 ALLOCATED

qui peut apparaître avec ASKCTYPE ; cela signifie que le port est déjà occupé par une autre tâche.

La différence entre un *joystick relatif* et un *joystick absolu* tient au fait suivant : dans le cas relatif, les coordonnées en X et Y du joystick se modifient de façon continue lorsqu'on se maintient dans une direction, alors que dans le cas absolu, elles ne sont modifiées qu'à la fin du mouvement.

Pour que vous puissiez avoir une idée plus précise de la programmation du *Gameport-Device*, voici un programme en langage machine, qui surveille un joystick connecté au port droit :

```
;***** Démonstration Gameport-Device: Joystick  6/87  S.D.  *****

ExecBase = 4
FindTask = -294
AddPort  = -354
RemPort  = -360
OpenLib  = -408
CloseLib = -414
OpenDev  = -444
CloseDev = -450
DoIo     = -456
SendIo   = -462

run:
     move.l  execbase,a6      ;Pointeur sur bibliothèque Exec
     sub.l   a1,a1            ;Task propre
     jsr     FindTask(a6)     ;chercher le Task
     move.l  d0,readreply+$10 ;poser SigTask

     lea     reply,a1
     jsr     AddPort(a6)      ;Créer Reply-Port

     lea    devio,a1
     move.l #1,d0             ;Unit 1: port de droite
     clr.l  d1
     lea    devicename,a0
     jsr    OpenDev(a6)       ;ouvrir le Gameport-Device
     tst.l  d0
     bne    error

;*** Détermination du type de port  ***

     move   #11,28(a1)        ;Command: SETCTYPE
     move.l #Event,40(a1)     ;Buffer
     move.l #1,36(a1)         ;Longueur:
```



```
     mov    .3,NextEvent      ;ABSJOYSTICK
     lea    devio,a1
     move.l #readreply,14(a1) ;set Reply-Port
     move.l execbase,a6
     jsr    DoIo(a6)          ;annoncer le joystick

;*** Définir l'exécution ***

     move   #13,28(a1)        ;Command: SETTRIGGER
     move.l #trigger,40(a1)   ;Buffer
     move.l #8,36(a1)         ;Longueur

     move   #3,Keys           ;DOWN & UP
     move   #0,Timeout        ;Timeout
     move   #1,XDelta         ;XDelta
     move   #1,YDelta         ;YDelta

     lea    devio,a1
     move.l #readreply,14(a1) ;set Reply-Port
     move.l execbase,a6
     jsr    DoIo(a6)          ;Exécuter

;*** Mettre en route la surveillance ***

     move   #9,28(a1)         ;Command: READEVENT
     move.l #Event,40(a1)     ;Buffer
     move.l #22,36(a1)        ;Longueur: un Event
     clr.b  30(a1)            ;Flags

     lea    devio,a1
     move.l #readreply,14(a1) ;set Reply-Port
     move.l execbase,a6
     jsr    DoIo(a6)          ;attente d'un événement

;*** Suppression du port ***

     move   #11,28(a1)        ;Command: SETCTYPE
     move.l #Event,40(a1)     ;Buffer
     move.l #1,36(a1)         ;Longueur:
     move.b #0,NextEvent      ;NOCONTROLLER
```

```
        lea     devio,a1
        move.l #readreply,14(a1) ;set Reply-P
        move.l execbase,a6
        jsr     DoIo(a6)          ;suppression du joystick

fin:
        lea     readreply,a1
        jsr     RemPort(a6)       ;suppression du port

        lea     devio,a1
        jsr     closedev(a6)      ;fermer le Device

error:
        rts                       ;* Fin *


devicename: dc.b 'gameport.device',0
even
devio:
message:    blk.b 20,0
io:         blk.b 12,0
ioreq:      blk.b 16,0

readreply:  blk.l 8,0
Event:
NextEvent:  dc.l 0
Class:      dc.b 0
SubClass:   dc.b 0
Code:       dc.w 0
Qualifier:  dc.w 0
ie_X:       dc.w 0
ie_Y:       dc.w 0
TimeStamp:  dc.l 0,0

Trigger:
Keys:       dc.w 0
Timeout:    dc.w 0
XDelta:     dc.w 0
YDelta:     dc.w 0
```


Normalement, il vaut mieux faire démarrer la surveillance avec *SendIo()*, puisqu'ainsi le programme n'est pas obligé d'attendre que le joystick soit actionné, et qu'il peut donc continuer à se dérouler. Pour cet exemple, on recommandera d'attendre, car sinon le *device* serait fermé avant l'événement, ce qui pourrait conduire à de sérieuses difficultés.

ANNEXE

Aperçu général des fonctions

dans les différentes bibliothèques

Le tableau qui suit vous fournira un aperçu général sur toutes les bibliothèques et sur les fonctions qu'elles contiennent. Chaque partie débute avec le nom de la bibliothèque dans laquelle se trouvent les fonctions mentionnées.

Ces fonctions sont présentées par leur offset négatif en hexadécimal, en décimal, avec leur nom et enfin avec leurs paramètres à transmettre. Les noms des paramètres sont placés entre parenthèses le nom de la fonction ; la seconde parenthèse contient dans le même ordre les registres dans lesquels ces paramètres doivent être transmis. Si aucun paramètre n'est requis, on le signale par ().

clist.library

```
-$001E   -30   InitCLPool(cLPool, size) (A0, D0)
-$0024   -36   AllocCList(cLPool) (A1)
-$002A   -42   FreeCList(cList) (A0)
-$0030   -48   FlushCList(cList) (A0)
-$0036   -54   SizeCList(cList) (A0)
-$003C   -60   PutCLChar(cList, byte) (A0, D0)
-$0042   -66   GetCLChar(cList) (A0)
-$0048   -72   UnGetCLChar(cList, byte) (A0, D0)
-$004E   -78   UnPutCLChar(cList) (A0)
-$0054   -84   PutCLWord(cList, word) (A0, D0)
-$005A   -90   GetCLWord(cList) (A0)
-$0060   -96   UnGetCLWord(cList, word) (A0, D0)
-$0066  -102   UnPutCLWord(cList) (A0)
```

```
-$006C  -108   PutCLBuf(cList, buffer, length)     , A1, D1)
-$0072  -114   GetCLBuf(cList, buffer, maxLength) (A0, A1, D1)
-$0078  -120   MarkCList(cList, offset) (A0, D0)
-$007E  -126   IncrCLMark(cList) (A0)
-$0084  -132   PeekCLMark(cList) (A0)
-$008A  -138   SplitCList(cList) (A0)
-$0090  -144   CopyCList(cList) (A0)
-$0096  -150   SubCList(cList, index, length) (A0, D0, D1)
-$009C  -156   ConcatCList(sourceCList, destCList) (A0, A1)
```

console library

```
-$002A   -42 CDInputHandler(events, device) (A0, A1)
-$0030   -48 RawKeyConvert(events, buffer, length, keyMap) (A0, A1, D1, A2)
```

diskfont library

```
-$001E   -30   OpenDiskFont(textAttr) (A0)
-$0024   -36   AvailFonts(buffer, bufBytes, flags) (A0, D0, D1)
```

doslibrary

```
-$001E   -30   Open(name, accessMode) (D1, D2)
-$0024   -36   Close(file) (D1)
-$002A   -42   Read(file, buffer, length) (D1, D2, D3)
-$0030   -48   Write(file, buffer, length) (D1, D2, D3)
-$0036   -54   Input()
-$003C   -60   Output()
-$0042   -66   Seek(file, position, offset) (D1, D2, D3)
-$0048   -72   DeleteFile(name) (D1)
-$004E   -78   Rename(oldName, newName) (D1, D2)
-$0054   -84   Lock(name, type) (D1, D2)
-$005A   -90   UnLock(lock) (D1)
-$0060   -96   DupLock(lock) (D1)
-$0066  -102   Examine(lock, fileInfoBlock) (D1, D2)
-$006C  -108   ExNext(lock, fileInfoBlock) (D1, D2)
-$0072  -114   Info(lock, parameterBlock) (D1, D2)
```



```
-$0078  -120   CreateDir(name) (D1)
-$007E  -126   CurrentDir(lock) (D1)
-$0084  -132   IoErr()
-$008A  -138   CreateProc(name, pri, segList, stackSize) (D1, D2, D3, D4)
-$0090  -144   Exit(returnCode) (D1)
-$0096  -150   LoadSeg(fileName) (D1)
-$009C  -156   UnLoadSeg(segment) (D1)
-$00A2  -162   GetPacket(wait) (D1)
-$00A8  -168   QueuePacket(packet) (D1)
-$00AE  -174   DeviceProc(name) (D1)
-$00B4  -180   SetComment(name, comment) (D1, D2)
-$00BA  -186   SetProtection(name, mask) (D1, D2)
-$00C0  -192   DateStamp(date) (D1)
-$00C6  -198   Delay(timeout) (D1)
-$00CC  -204   WaitForChar(file, timeout) (D1, D2)
-$00D2  -210   ParentDir(lock) (D1)
-$00D8  -216   IsInteractive(file) (D1)
-$00DE  -222   Execute(string, file, file) (D1, D2, D3)
```

exec.library

```
-$001E   -30   Supervisor()
-$0024   -36   ExitIntr()
-$002A   -42   Schedule()
-$0030   -48   Reschedule()
-$0036   -54   Switch()
-$003C   -60   Dispatch()
-$0042   -66   Exception()
-$0048   -72   InitCode(startClass, version) (D0, D1)
-$004E   -78   InitStruct(initTable, memory, size) (A1, A2, D0)
-$0054   -84   MakeLibrary(funcInit, structInit, libInit, dataSize,
               codeSize) (A0, A1, A2, D0, D1)
-$005A   -90   MakeFunctions(target, functionArray, funcDispBase)
               (A0, A1, A2)
-$0060   -96   FindResident(name) (A1)
-$0066  -102   InitResident(resident, segList) (A1, D1)
-$006C  -108   Alert(alertNum, parameters) (D7, A5)
-$0072  -114   Debug()
-$0078  -120   Disable()
```

```
-$007E  -126  Enable()
-$0084  -132  Forbid()
-$008A  -138  Permit()
-$0090  -144  SetSR(newSR, mask) (D0, D1)
-$0096  -150  SuperState()
-$009C  -156  UserState(sysStack) (D0)
-$00A2  -162  SetIntVector(intNumber, interrupt) (D0, A1)
-$00A8  -168  AddIntServer(intNumber, interrupt) (D0, A1)
-$00AE  -174  RemIntServer(intNumber, interrupt) (D0, A1)
-$00B4  -180  Cause(interrupt) (A1)
-$00BA  -186  Allocate(freeList, byteSize) (A0, D0)
-$00C0  -192  Deallocate(freeList, memoryBlock, byteSize) (A0, A1, D0)
-$00C6  -198  AllocMem(byteSize, requirements) (D0, D1)
-$00CC  -204  AllocAbs(byteSize, location) (D0, A1)
-$00D2  -210  FreeMem(memoryBlock, byteSize) (A1, D0)
-$00D8  -216  AvailMem(requirements) (D1)
-$00DE  -222  AllocEntry(entry) (A0)
-$00E4  -228  FreeEntry(entry) (A0)
-$00EA  -234  Insert(list, node, pred) (A0, A1, A2)
-$00F0  -240  AddHead(list, node) (A0, A1)
-$00F6  -246  AddTail(list, node) (A0, A1)
-$00FC  -252  Remove(node) (A1)
-$0102  -258  RemHead(list) (A0)
-$0108  -264  RemTail(list) (A0)
-$010E  -270  Enqueue(list, node) (A0, A1)
-$0114  -276  FindName(list, name) (A0, A1)
-$011A  -282  AddTask(task, initPC, finalPC) (A1, A2, A3)
-$0120  -288  RemTask(task) (A1)
-$0126  -294  FindTask(name) (A1)
-$012C  -300  SetTaskPri(task, priority) (A1, D0)
-$0132  -306  SetSignal(newSignals, signalSet) (D0, D1)
-$0138  -312  SetExcept(newSignals, signalSet) (D0, D1)
-$013E  -318  Wait(signalSet) (D0)
-$0144  -324  Signal(task, signalSet) (A1, D0)
-$014A  -330  AllocSignal(signalNum) (D0)
-$0150  -336  FreeSignal(signalNum) (D0)
-$0156  -342  AllocTrap(trapNum) (D0)
-$015C  -348  FreeTrap(trapNum) (D0)
-$0162  -354  AddPort(port) (A1)
-$0168  -360  RemPort(port) (A1)
```



```
-$016E  -366  .utMsg(port, message) (A0, A1)
-$0174  -3.   GetMsg(port) (A0)
-$017A  -378  ReplyMsg(message) (A1)
-$0180  -384  WaitPort(port) (A0)
-$0186  -390  FindPort(name) (A1)
-$018C  -396  AddLibrary(library) (A1)
-$0192  -402  RemLibrary(library) (A1)
-$0198  -408  OldOpenLibrary(libName) (A1)
-$019E  -414  CloseLibrary(library) (A1)
-$01A4  -420  SetFunction(library, funcOffset, funcEntry) (A1, A0, D0)
-$01AA  -426  SumLibrary(library) (A1)
-$01B0  -432  AddDevice(device) (A1)
-$01B6  -438  RemDevice(device) (A1)
-$01BC  -444  OpenDevice(devName, unit, ioRequest, flags) (A0, D0, A1, D1)
-$01C2  -450  CloseDevice(ioRequest) (A1)
-$01C8  -456  DoIO(ioRequest) (A1)
-$01CE  -462  SendIO(ioRequest) (A1)
-$01D4  -468  CheckIO(ioRequest) (A1)
-$01DA  -474  WaitIO(ioRequest) (A1)
-$01E0  -480  AbortIO(ioRequest) (A1)
-$01E6  -486  AddRescource(rescource) (A1)
-$01EC  -492  RemRescource(rescource) (A1)
-$01F2  -498  OpenRescource(resName, version) (A1, D0)
-$01F8  -504  RawIOInit()
-$01FE  -510  RawMayGetChar()
-$0204  -516  RawPutChar(char) (D0)
-$020A  -522  RawDoFmt() (A0, A1, A2, A3)
-$0210  -528  GetCC()
-$0216  -534  TypeOfMem(address) (A1)
-$021C  -540  Procedure(semaport, bidMsg) (A0, A1)
-$0222  -546  Vacate(semaport) (A0)
-$0228  -552  OpenLibrary(libName, version) (A1, D0)
```

graphics.library

```
-$001E  -30  BltBitMap(srcBitMap, srcX, srcY, destBitMap, destX, destY,
             sizeX, sizeY, minterm, mask, tempA) (A0, D0, D1, A1, D2, D3,
             D4, D5, D6, D7, A2)
```

```
-$0024   -36   BltTemplate(source, srcX, sr     , destRastPort, destX,
               destY, sizeX, sizeY) (A0, D0, ..., A1, D2, D3, D4, D5)
-$002A   -42   ClearEOL(rastPort) (A1)
-$0030   -48   ClearScreen(rastPort) (A1)
-$0036   -54   TextLength(RastPort, string, count) (A1, A0, D0)
-$003C   -60   Text(RastPort, String, count) (A1, A0, D0)
-$0042   -66   SetFont(RAstPortID, textFont) (A1, A0)
-$0048   -72   OpenFont(textAttr) (A0)
-$004E   -78   CloseFont(textFont) (A1)
-$0054   -84   AskSoftStyle(rastPort) (A1)
-$005A   -90   SetSoftStyle(rastPort, style, enable) (A1, D0, D1)
-$0060   -96   AddBob(bob, rastPort) (A0, A1)
-$0066  -102   AddVSprite(vSprite, rastPort) (A0, A1)
-$006C  -108   DoCollision(rastPort) (A1)
-$0072  -114   DrawGList(rastPort, viewPort) (A1, A0)
-$0078  -120   InitGels(dummyHead, dummyTail, GelsInfo) (A0, A1, A2)
-$007E  -126   InitMasks(vSprite) (A0)
-$0084  -132   RemIBob(bob, rastPort, viewPort) (A0, A1, A2)
-$008A  -138   RemVSprite(vSprite) (A0)
-$0090  -144   SetCollision(type, routine, gelsInfo) (D0, A0, A1)
-$0096  -150   SortGList(rastPort) (A1)
-$009C  -156   AddAnimObj(obj, animationKey, rastPort) (A0, A1, A2)
-$00A2  -162   Animate(animationKey, rastPort) (A0, A1)
-$00A8  -168 GetGBuffers(animationObj, rastPort, doubleBuffer) (A0, A1, D0)
-$00AE  -174   InitGMasks(animationObj) (A0)
-$00B4  -180   GelsFuncE()
-$00BA  -186   GelsFuncF()
-$00C0  -192   LoadRGB4(viewPort, colors, count) (A0, A1, D0)
-$00C6  -198   InitRastPort(rastPort) (A1)
-$00CC  -204   InitVPort(viewPort) (A0)
-$00D2  -210   MrgCop(view) (A1)
-$00D8  -216   MakeVPort(view, viewPort) (A0, A1)
-$00DE  -222   LoadView(view) (A1)
-$00E4  -228   WaitBlit()
-$00EA  -234   SetRast(rastPort, color) (A1, D0)
-$00F0  -240   Move(rastPort, x, y) (A1, D0, D1)
-$00F6  -246   Draw(rastPort, x, y) (A1, D0, D1)
-$00FC  -252   AreaMove(rastPort, x, y) (A1, D0, D1)
-$0102  -258   AreaDraw(rastPort, x, y) (A1, D0, D1)
-$0108  -264   AreaEnd(rastPort) (A1)
```



```
-$010E  -270   WaitTOF()
-$0114  -276   QBlit(blit) (A1)
-$011A  -282 InitArea(areaInfo, vectorTable, vectorTableSize) (A0, A1, D0)
-$0120  -288   SetRGB4(viewPort, index, r, g, b) (A0, D0, D1, D2, D3)
-$0126  -294   QBSBlit(blit) (A1)
-$012C  -300   BltClear(memory, size, flags) (A1, D0, D1)
-$0132  -306   RectFill(rastPort, xl, yl, xu, yu) (A1, D0, D1, D2, D3)
-$0138  -312   BltPattern(rastPort, ras, xl, yl, maxX, maxY, fillBytes)
               (A1, A0, D0, D1, D2, D3, D4)
-$013E  -318   ReadPixel(rastPort, x, y) (A1, D0, D1)
-$0144  -324   WritePixel(rastPort, x, y) (A1, D0, D1)
-$014A  -330   Flood(rastPort, mode, x, y) (A1, D2, D0, D1)
-$0150  -336   PolyDraw(rastPort, count, polyTable) (A1, D0, A0)
-$0156  -342   SetAPen(rastPort, pen) (A1, D0)
-$015C  -348   SetBPen(rastPort, pen) (A1, D0)
-$0162  -354   SetDrMd(rastPort, drawMode) (A1, D0)
-$0168  -360   InitView(view) (A1)
-$016E  -366   CBump(copperList) (A1)
-$0174  -372   CMove(copperList, destination, data) (A1, D0, D1)
-$017A  -378   CWait(copperList, x, y) (A1, D0, D1)
-$0180  -384   VBeamPos()
-$0186  -390   InitBitMap(bitMap, depth, width, heigth) (A0, D0, D1, D2)
-$018C  -396   ScrollRaster(rastPort, dX, dY, minx, miny, maxx, maxy)
               (A1, D0, D1, D2, D3, D4, D5)
-$0192  -402   WaitBOVP(viewPort) (A0)
-$0198  -408   GetSprite(simpleSprite, num) (A0, D0)
-$019E  -414   FreeSprite(num) (D0)
-$01A4  -420   ChangeSprite(vp, simpleSprite, data) (A0, A1, A2)
-$01AA  -426   MoveSprite(viewPort, simpleSprite, x, y) (A0, A1, D0, D1)
-$01B0  -432   LockLayerRom(layer) (A5)
-$01B6  -438   UnlockLayerRom(layer) (A5)
-$01BC  -444   SyncSBitMap(1) (A0)
-$01C2  -450   CopySBitMap(11, 12) (A0, A1)
-$01C8  -456   OwnBlitter()
-$01CE  -462   DisownBlitter()
-$01D4  -468   InitTmpRas(tmpras, buff, size) (A0, A1, D0)
-$01DA  -474   AskFont(rastPort, textAttr) (A1, A0)
-$01E0  -480   AddFont(textFont) (A1)
-$01E6  -486   RemFont(textFont) (A1)
-$01EC  -492   AllocRaster(width, heigth) (D0, D1)
```

```
-$01F2   -498   FreeRaster(planeptr, width, he      . (A0, D0, D1)
-$01F8   -504   AndRectRegion(rgn, rect) (A0, A1)
-$01FE   -510   OrRectRegion(rgn, rect) (A0, A1)
-$0204   -516   NewRegion()
-$020A   -522   ** reservé **
-$0210   -528   ClearRegion(rgn) (A0)
-$0216   -534   DisposeRegion(rgn) (A0)
-$021C   -540   FreeVPortCopLists(viewPort) (A0)
-$0222   -546   FreeCopList(coplist) (A0)
-$0228   -552   ClipBlit(srcrp, srcX, srcY, destrp, destX, destY, sizeX,
                sizeY, minterm) (A0, D0, D1, A1, D2, D3, D4, D5, D6)
-$022E   -558   XorRectRegion(rgn, rect) (A0, A1)
-$0234   -564   FreeCprList(cprlist) (A0)
-$023A   -570   GetColorMap(entries) (D0)
-$0240   -576   FreeColorMap(colormap) (A0)
-$0246   -582   GetRGB4(colormap, entry) (A0, D0)
-$024C   -588   ScrollVPort(vp) (A0)
-$0252   -594   UCopperListInit(copperlist, num) (A0, D0)
-$0258   -600   FreeGBuffers(animationObj, rastPort, doubleBuffer)
                (A0, A1, D0)
-$025E   -606   BltBitMapRastPort(srcbm, srcx, srcy, destrp, destX, destY,
                sizeX, sizeY, minter) (A0, D0, D1, A1, D2, D3, D4, D5, D6)
```

icon.library

```
-$001E    -30   GetWBObject(name) (A0)
-$0024    -36   PutWBObject(name, object) (A0, A1)
-$002A    -42   GetIcon(name, icon, freelist) (A0, A1, A2)
-$0030    -48   PutIcon(name, icon) (A0, A1)
-$0036    -54   FreeFreeList(freelist) (A0)
-$003C    -60   FreeWBObject(WBObject) (A0)
-$0042    -66   AllocWBObject()
-$0048    -72   AddFreeList(freelist, mem, size) (A0, A1, A2)
-$004E    -78   GetDiskObject(name) (A0)
-$0054    -84   PutDiskObject(name, diskobj) (A0, A1)
-$005A    -90   FreeDiskObj(diskobj) (A0)
-$0060    -96   FindToolType(toolTypeArray, typeName) (A0, A1)
-$0066   -102   MatchToolValue(typeString, value) (A0, A1)
-$006C   -108   BumbRevision(newname, oldname) (A0, A1)
```


intuition ".ry

```
-$001E    -30   OpenIntuition()
-$0024    -36   Intuition(ievent) (A0)
-$002A    -42   AddGadget(AddPtr, Gadget, Position) (A0, A1, D0)
-$0030    -48   ClearDMRequest(Window) (A0)
-$0036    -54   ClearMenuStrip(Window) (A0)
-$003C    -60   ClearPointer(Window) (A0)
-$0042    -66   CloseScreen(Screen) (A0)
-$0048    -72   CloseWindow(Window) (A0)
-$004E    -78   CloseWorkBench()
-$0054    -84   CurrentTime(Seconds, Micros) (A0, A1)
-$005A    -90   DisplayAlert(AlertNumber, String, Height) (D0, A0, D1)
-$0060    -96   DisplayBeep(Screen) (A0)
-$0066   -102   DoubleClick(sseconds, smicros, cseconds, cmicros)
                (D0, D1, D2, D3)
-$006C   -108   DrawBorder(Rport, Border, LeftOffset, TopOffset) (A0, A1,
                D0, D1)
-$0072   -114   DrawImage(RPort, Image, LeftOffset, TopOffset) (A0, A1, D0,
                D1)
-$0078   -120   EndRequest(requester, window) (A0, A1)
-$007E   -126   GetDefPrefs(preferences, size) (A0, D0)
-$0084   -132   GetPrefs(preferences, size) (A0, D0)
-$008A   -138   InitRequester(req) (A0)
-$0090   -144   ItemAddress(MenuStrip, MenuNumber) (A0, D0)
-$0096   -150   ModifyIDCMP(Window, Flags) (A0, D0)
-$009C   -156   ModifyProp(Gadget, Ptr, Reg, Flags, HPos,
                VPos, HBody, VBody (A0, A1, A2, D0, D1, D2, D3, D4)
-$00A2   -162   MoveScreen(Screen, dx, dy) (A0, D0, D1)
-$00A8   -168   MoveWindow(Window, dx, dy) (A0, D0, D1)
-$00AE   -174   OffGadget(Gadget, Ptr, Req) (A0, A1, A2)
-$00B4   -180   OffMenu(Window, MenuNumber) (A0, D0)
-$00BA   -186   OnGadget(Gadget, Ptr, Req) (A0, A1, A2)
-$00C0   -192   OnMenu(Window, MenuNumber) (A0, D0)
-$00C6   -198   OpenScreen(OSArgs) (A0)
-$00CC   -204   OpenWindow(OWArgs) (A0)
-$00D2   -210   OpenWorkBench()
-$00D8   -216   PrintIText(rp, itext, left, top) (A0, A1, D0, D1)
-$00DE   -222   RefreshGadgets(Gadgets, Ptr, Req) (A0, A1, A2)
-$00E4   -228   RemoveGadgets(RemPtr, Gadget) (A0, A1)
```

```
-$00EA  -234   ReportMouse(Window, Boolean) (A0, D0)
-$00F0  -240   Request(Requester, Window) (A0, A1)
-$00F6  -246   ScreenToBack(Screen) (A0)
-$00FC  -252   SCreenToFront(Screen) (A0)
-$0102  -258   SetDMRequest(Window, req) (A0, A1)
-$0108  -264   SetMenuStrip(Window, Menu) (A0, A1)
-$010E  -270   SetPointer(Window, Pointer, Height, Width, XOffset,
               YOffset) (A0, A1, D0, D1, D2, D3)
-$0114  -276   SetWindowTitles(Window, windowTitle, screenTitle) (A0, A1,
               A2)
-$011A  -282   ShowTitle(Screen, ShowIt) (A0, D0)
-$0120  -288   SizeWindow(Window, dx, dy) (A0, D0, D1)
-$0126  -294   ViewAddress()
-$012C  -300   ViewPortAddress(Window) (A0)
-$0132  -306   WindowToBack(Window) (A0)
-$0138  -312   WindowToFront(Window) (A0)
-$013E  -318   WindowLimits(Window, minWidth, minHeight, maxWidth,
               maxHeight) (A0, D0, D1, D2, D3)
-$0144  -324   SetPrefs(preferences, size, flag) (A0, D0, D1)
-$014A  -330   IntuiTextLength(itext) (A0)
-$0150  -336   WBenchToBack()
-$0156  -342   WBenchToFront()
-$015C  -348   AutoRequest(WIndow, Body, PText, NText, PFlag, NFlag,
               W, H) (A0, A1, A2, A3, D0, D1, D2, D3)
-$0162  -354   BeginRefresh(Window) (A0)
-$0168  -360   BuildSysRequest(Window, Body, PosText, NegText, Flags,
               W, H) (A0, A1, A2, A3, D0, D1, D2)
-$016E  -366   EndRefresh(Window, Complete) (A0, D0)
-$0174  -372   FreeSysRequest(Window) (A0)
-$017A  -378   MakeScreen(Screen) (A0)
-$0180  -384   RemakeDisplay()
-$0186  -390   RethinkDisplay()
-$018C  -396   AllocRemember(RememberKey, Size, Flags) (A0, D0, D1)
-$0192  -402   AlohaWorkbench(wbport) (A0)
-$0198  -408   FreeRemember(RememberKey, ReallyForget) (A0, D0)
-$019E  -414   LockIBase(dontknow) (D0)
-$01A4  -420   UnlockIBase(IBLock) (A0)
```


layers.library

```
-$001E   -30   InitLayers(li) (A0)
-$0024   -36   CreateUpfrontLayer(li, bm, x0, y0, x1, y1, flags, bm2)
               (A0, A1, D0, D1, D2, D3, D4, A2)
-$002A   -42   CreateBehindLayer(li, bm, x0, y0, x1, y1, flags, bm2)
               (A0, A1, D0, D1, D2, D3, D4, A2)
-$0030   -48   UpfrontLayer(li, layer) (A0, A1)
-$0036   -54   BehindLayer(li, layer) (A0, A1)
-$003C   -60   MoveLayer(li, layer, dx, dy) (A0, A1, D0, D1)
-$0042   -66   SizeLayer(li, layer, dx, dy) (A0, A1, D0, D1)
-$0048   -72   ScrollLayer(li, layer, dx, dy) (A0, A1, D0, D1)
-$004E   -78   BeginUpdate(layer) (A0)
-$0054   -84   EndUpdate(layer) (A0)
-$005A   -90   DeleteLayer(li, layer) (A0, A1)
-$0060   -96   LockLayer(li, layer) (A0, A1)
-$0066  -102   UnlockLayer(li, layer) (A0, A1)
-$006C  -108   LockLayers(li) (A0)
-$0072  -114   UnlockLayers(li) (A0)
-$0078  -120   LockLayerInfo(li) (A0)
-$007E  -126   SwapBitsRastPortClipRect(rp, cr) (A0, A1)
-$0084  -132   WhichLayer(li, x, y) (A0, D0, D1)
-$008A  -138   UnlockLayerInfo(li) (A0)
-$0090  -144   NewLayerInfo()
-$0096  -150   DisposeLayerInfo(li) (A0)
-$009C  -156   FattenLayerInfo(li) (A0)
-$00A2  -162   ThinLayerInfo(li) (A0)
-$00A8  -168   MoveLayerInFrontOf(layer_to_move, layer_to_be_in_front_of)
               (A0, A1)
```

mathffp.library

```
-$001E   -30   SPFix(float) (D0)
-$0024   -36   SPFlt(integer) (D0)
-$002A   -42   SPCmp(leftFloat, rightFloat) (D1, D0)
-$0030   -48   SPTst(float) (D1)
-$0036   -54   SPAbs(float) (D0)
-$003C   -60   SPNeg(float) (D0)
-$0042   -66   SPAdd(leftFloat, rightFloat) (D1, D0)
```

```
-$0048   -72   SPSub(leftFloat, rightFloat) (D0, D0)
-$004E   -78   SPMul(leftFloat, rightFloat) (D1, D0)
-$0054   -84   SPDiv(leftFloat, rightFloat) (D1, D0)
```

mathieeedoubbas.library

```
-$001E   -30   IEEEDPFix(integer, integer) (D0, D1)
-$0024   -36   IEEEDPFlt(integer) (D0)
-$002A   -42   IEEEDPCmp(integer, integer, integer, integer) (D0, D1, D2,
               D3)
-$0030   -48   IEEEDPTst(integer, integer) (D0, D1)
-$0036   -54   IEEEDPAbs(integer, integer) (D0, D1)
-$003C   -60   IEEEDPNeg(integer, integer) (D0, D1)
-$0042   -66   IEEEDPAdd(integer, integer, integer, integer) (D0, D1, D2,
               D3)
-$0048   -72   IEEEDPSub(integer, integer, integer, integer) (D0, D1, D2,
               D3)
-$004E   -78   IEEEDPMul(integer, integer, integer, integer) (D0, D1, D2,
               D3)
-$0054   -84   IEEEDPDiv(integer, integer, integer, integer) (D0, D1, D2,
               D3)
```

mathtrans.library

```
-$001E   -30   SPAtan(float) (D0)
-$0024   -36   SPSin(float) (D0)
-$002A   -42   SPCos(float) (D0)
-$0030   -48   SPTan(float) (D0)
-$0036   -54   SPSincos(leftFloat, rightFloat) (D1, D0)
-$003C   -60   SPSinh(float) (D0)
-$0042   -66   SPCosh(float) (D0)
-$0048   -72   SPTanh(float) (D0)
-$004E   -78   SPExp(float) (D0)
-$0054   -84   SPLog(float) (D0)
-$005A   -90   SPPow(leftFloat, rightFloat) (D1, D0)
-$0060   -96   SPSqrt(float) (D0)
-$0066  -102   SPTieee(float) (D0)
-$006C  -108   SPFieee(float) (D0)
```



```
-$0072  -114   SPAsin(float) (D0)
-$0078  -120   SPAcos(float) (D0)
-$007E  -126   SPLog10(float) (D0)
```

potgo.library

```
-$0006    -6   AllocPotBits(bits) (D0)
-$000C   -12   FreePotBits(bits) (D0)
-$0012   -18   WritePotgo(word, mask) (D0, D1)
```

timer.library

```
-$002A   -42   AddTime(dest, src) (A0, A1)
-$0030   -48   SubTime(dest, src) (A0, A1)
-$0036   -54   CmpTime(dest, src) (A0, A1)
```

translator.library

```
-$001E   -30   Translate(inputString, inputLength, outputBuffer,
               bufferSize) (A0, D0, A1, D1) ftg
```

INDEX

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z