SMAKY6

# SAMOS

## SMAKY AND MICRO-FLOPPY

## OPERATING SYSTEM

Novembre 1982

EPSITEC-system sa

# SAMOS: SMAKY AND MICROFLOPPY OPERATING SYSTEM

Révision 1-G et 1-H

A. Capt

# TABLE DES MATIERES

## 8.1 CONFIGURATION HARDWARE FLOPPY DISQUE

L'installation du disque souple sur le SMAKY6 se compose de plusieurs éléments hardware:

- . le contrôleur de drive
- . de 1 jusqu'à trois drives
- . l'alimentation des drives

LE CONTROLEUR

Il est connecté au SMAKY sur la prise périphérique 26 pôles latérale. Il est alimenté en +5V par cette même prise.

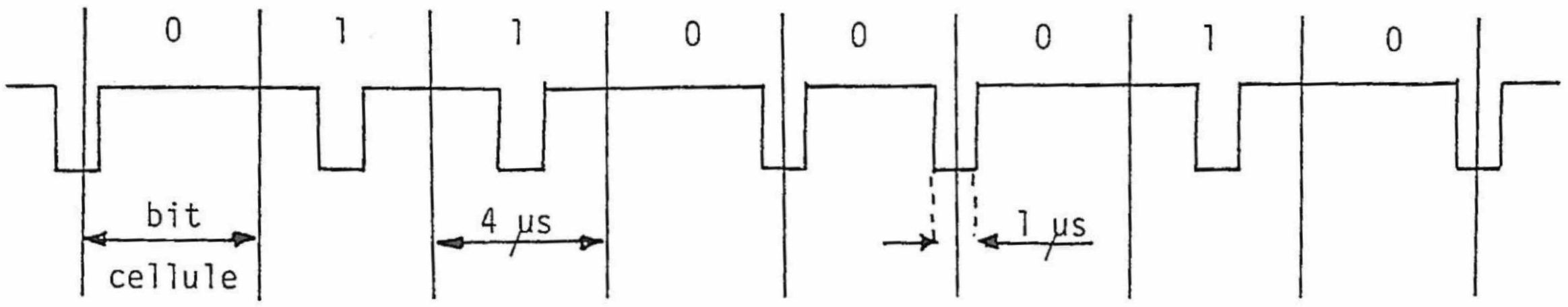
C'est en fait l'interface de commande du ou des drives qui est connecté à ce contrôleur par un câble à prises multiples (Daisy Chain)

Le contrôleur réalise les travaux suivants:

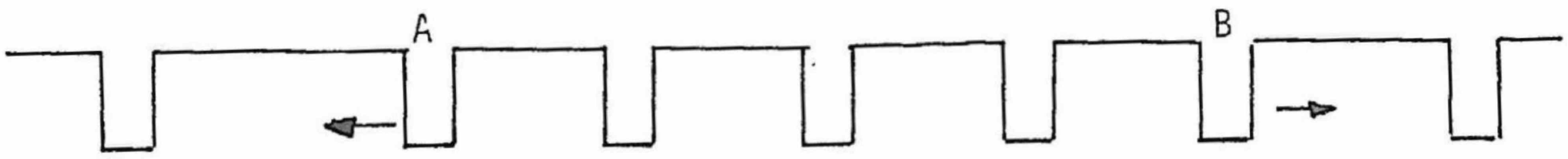
- . codage et décodage des signaux disque
- . data input/output
- . commande fonctionnelle du drive.

CODAGE ET DECODAGE DES SIGNAUX DISQUE

Le système de codage utilisé est le MFM qui permet la double densité sur le disque. Le principe de ce codage est illustré par le dessin qui suit suit et qui montre le diagramme de temps du codage MFM d'un byte $142_8$.



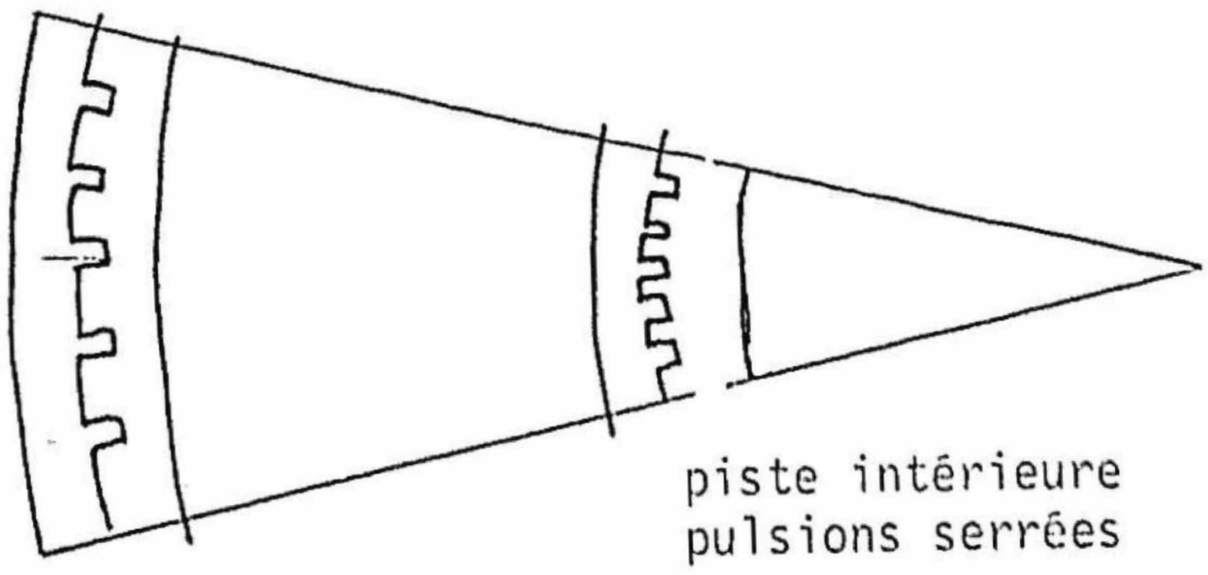

Chacune de ces impulsions est enregistrée sur le disque sous forme d'une transition de flux magnétique. On peut s'imaginer que chacune de ces transitions représente un petit aimant enregistré sur le media. Chacun sait que deux aimants mis en voisinage se contrarient mutuellement. C'est aussi ce qui se passe sur le disque, d'autant plus que les distances entre ces "petits aimants" ne sont pas égales.



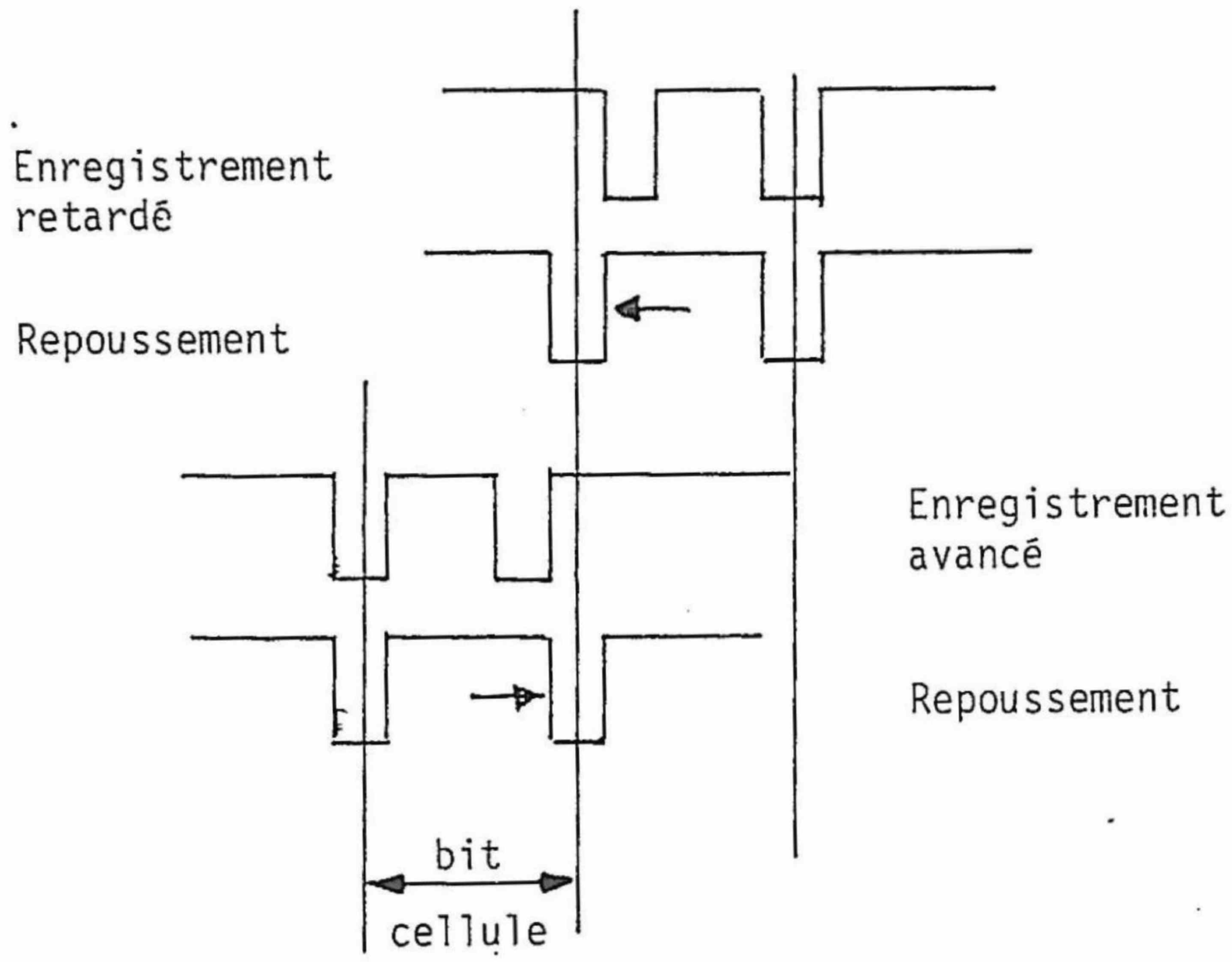

Le petit dessin ci-dessus nous montre que, pour cette configuration de bits, les impulsions A et B ont tendance à se déplacer dans le sens des flèches étant donné leurs positions asymétriques par rapport aux impulsions qui les entourent. Ce phénomène est d'autant plus fort que l'on se rapproche de l'intérieur du disque. En effet la densité est plus grande sur les pistes intérieures.

piste extérieure,
pulsions plus espacées

(256 bytes)



Ce phénomène nécessite la pré-compensation pour garantir une lecture correcte des informations. Ceci consiste à enregistrer les impulsions de type A et B avec un offset avant ou arrière de manière à ce que, une fois repoussées, les impulsions occupent la place correcte.




Le codage avec la pré-compensation décrite ici, ainsi que le décodage de ces signaux lors d'une lecture, sont réalisés entièrement par une unité micro-programmée.

## DATA INPUT OUTPUT

Le codage MFM est du type sériel, il nécessite donc une sérialisation de l'information à l'enregistrement et une désérialisation à la lecture.
Le contrôleur effectue ces tâches par des registres à décalage. La communication avec le SMAKY se fait donc en mode parallèle par l'intermédiaire de deux périphériques pour les data (écriture ou lecture) et de deux périphériques de contrôle de liaison (handshake).

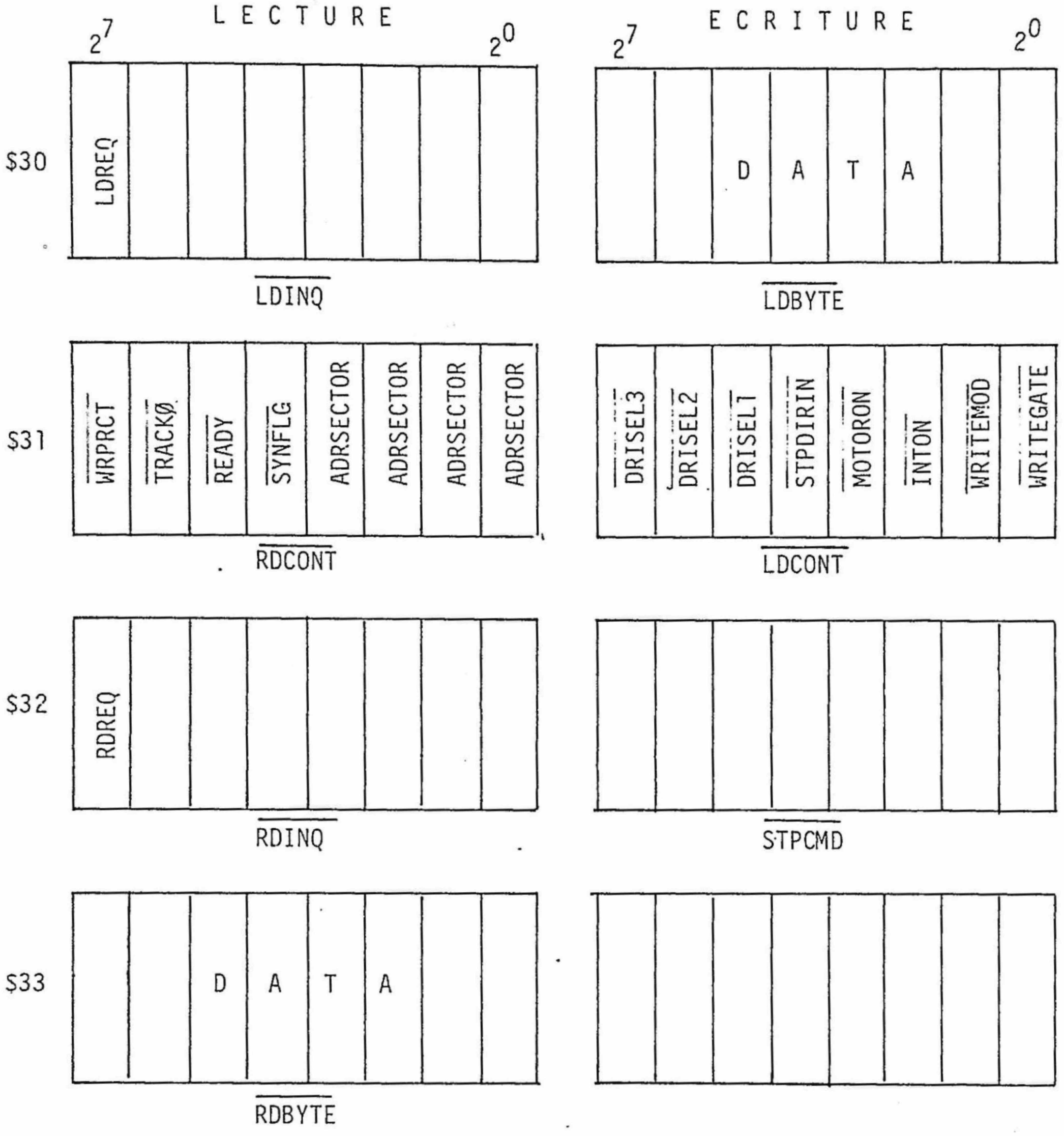
(voir aussi plus loin la description des périphériques).

## COMMANDE FONCTIONNELLE DU DRIVE

En plus du codage et décodage de l'information, il faut naturellement pouvoir commander toutes les fonctions du drive, telles que déplacement de la tête, marche-arrêt du moteur, sélection des modes, etc.

Ces fonctions sont réalisées par le processeur du SMAKY et le software "bas niveau" des appels SAMOS, utilisant le périphérique $CONT du contrôleur. Celui-ci se charge de transmettre les ordres aux différents organes concernés, soit au niveau du contrôleur directement (exemple: activation de l'interruption, sélection du mode écriture), soit au niveau du drive (exemple: mise en marche du moteur). Le sychronisme est obtenu par une interruption, générée par le contrôleur, à chaque début de secteur. Ainsi donc, les routines de plus bas niveau telles que RBBLK (read block), WRBLK (write block) sont des routines d'interruption.

A toutes fins utiles, voici le tableau de définition des périphériques du contrôleur:



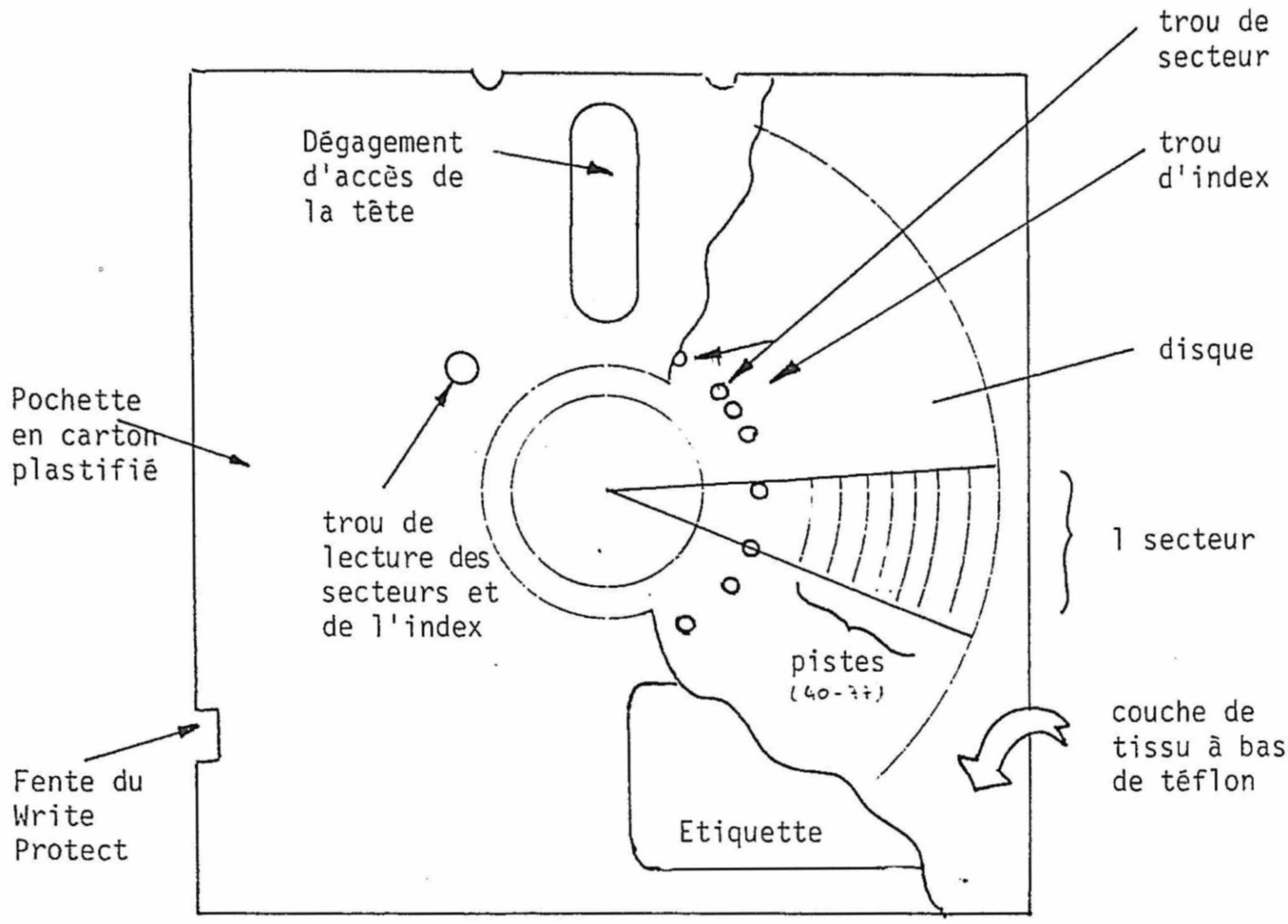

LEXIQUE DU TABLEAU:

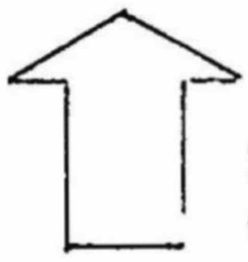
$30 $\overline{\text{LDINQ}}$ = load inquiry, low
LDREQ = load request
$30 $\overline{\text{LDBYTE}}$ = load byte, low

$31 $\overline{\text{RDCONT}}$ = read control, low
$\overline{\text{WRPRCT}}$ = write protect flag, low
$\overline{\text{TRACKØ}}$ = track zero flag, low
$\overline{\text{READY}}$ = ready flag, low
$\overline{\text{SYNFLG}}$ = synchro flag, low
ADRSECTOR = 4 bit sector address
$31 $\overline{\text{LDCONT}}$ = write control, low
$\overline{\text{DRISEL 1 to 3}}$ = drive selection, low
$\overline{\text{STPDIRIN}}$ = step direction, low
$\overline{\text{MOTORON}}$ = motor on, low
$\overline{\text{INTON}}$ = sector interrupt on, low
$\overline{\text{WRITEMOD}}$= controller in write mode, low
$\overline{\text{WRITEGATE}}$= drive in write mode, low

$32 $\overline{\text{RDINQ}}$ = read inquiry, low
RDREQ = read request
$32 $\overline{\text{STPCMD}}$ = step command, low
$33 $\overline{\text{RDBYTE}}$ = read byte, low

# LE DISQUE SOUPLE



Sens d'introduction dans le drive

Comme vous pouvez le voir sur ce dessin, le disque souple est enfermé dans une pochette dont la surface intérieure est recouverte d'un tissu à base de téflon qui donne un frottement minimum, tout en nettoyant le disque.

Le disque est divisé en 16 secteurs égaux, repérés chacun par un trou percé dans le disque souple. Un trou supplémentaire percé entre deux trous de secteur indique l'origine. C'est le trou d'index. La pochette a également un trou qui permet à un système opto-électronique de lire les secteurs et l'index. Une autre ouverture de la disquette permet à la tête de lecture/écriture d'être en contact avec le disque et de se déplacer radialement sur ce dernier. On obtient ainsi une surface subdivisée en pistes (déplacement de la tête), elles-mêmes divisées en secteurs (trou de secteur).

Le disque tourne dans sa fourre à 300 tours/minute. Grâce à ce mouvement, ainsi qu'à celui de la tête, on peut donc atteindre n'importe quel secteur.
L'index, la lecture des trous de secteur, la gestion intelligente du déplacement de la tête, nous permettent de donner à chacun de ces secteurs une adresse bien précise, On peut donc enregistrer, puis retrouver facilement différentes informations à différentes places sur le disque.

La position latérale du trou de lecture des secteurs et de l'index permet de détecter une introduction à l'envers. Si la fente du "write protect" est recouverte d'un élément non transparent, les circuits d'écriture du drive sont inhibés, interdisant ainsi toute écriture accidentelle.

RECOMMENDATIONS: il est vivement recommandé de manipuler les disquettes avec précaution.

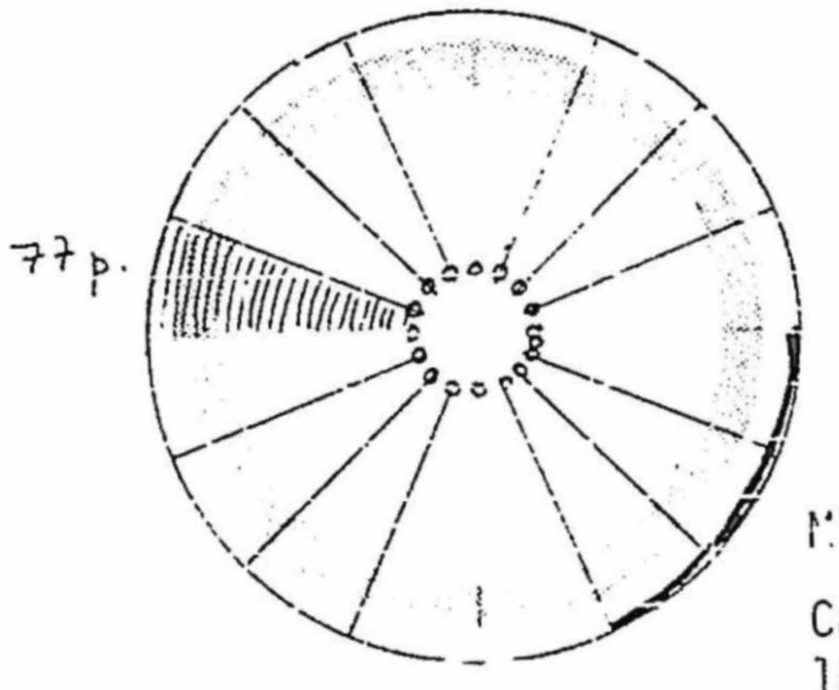
Prendre bien garde à les protéger de la poussière et de la présence d'un champ magnétique élevé (haut-parleur, TV). Ne jamais les tordre. Evitez de les laisser dans le drive, mais rangez-les plutôt dans leur fourre , bien enfermées dans une boîte.

Ecrivez sur vos étiquettes de préférence avant de les coller sur la disquette.

Ces précautions élémentaires vous éviteront bien des ennuis.

ORGANISATION DU DISQUE

Le "directory" occupe les trois permiers secteurs, piste Ø.
Depuis le 4ème secteur, piste 0, jusqu'au 16$^{e}$ secteur , dernière piste (numéro 40 à 77 selon les drives), la place est disponible pour les fichiers.




MICROPOLIS

Capacité:
16 x 77 x 256 bytes

= 315 kbytes

On emploie l'organisation d'écriture contiguë sur le disque.
Admettons que nous écrivons un fichier (de longueur 200 blocs ou secteurs) depuis le secteur 4, piste 8. La tête va écrire sur les secteurs 5 à 16 (piste 8), puis cette piste étant pleine, va passer à la piste 9.

## 8.2 SOFTWARE FLOPPY DISQUE

### DESCRIPTION DES ALGORITHMES

. LECTURE DE BLOCS: La lecture d'un bloc donne lieu à deux vérifications. La première est la correspondance de l'adresse de piste enregistrée sur le disque avec la mémoire de la position de la tête.

Le deuxième est la comparaison du checksum enregistré sur le disque avec celui calculé à la lecture.

Indifféremment pour l'une ou l'autre de ces vérifications, on effectue au maximum dix tentatives de lecture d'un bloc signalant à chaque fois une erreur. Si c'est le cas, on considère que la lecture est impossible et le travail ayant engendré cette lecture est avorté, avec un message d'erreur de lecture.

Lorsque l'on demande la lecture de plusieurs blocs, le hardware permet une lecture consécutive de ces blocs. Donc, lors de la lecture d'un fichier par tranche, on a intérêt à lire des tranches qui soient les plus longues possible.

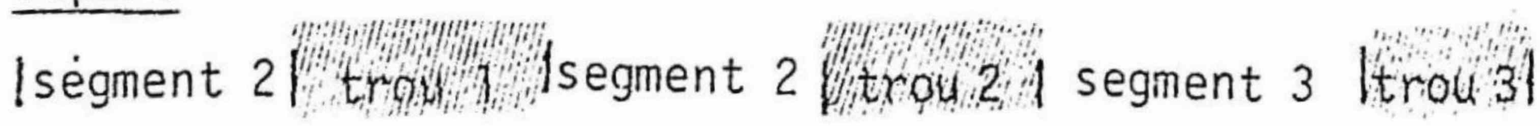
. ECRITURE DE BLOCS: L'écriture de blocs sur le disque s'effectue selon le principe du "read after write" au niveau de la piste. Imaginons qu'un ordre d'écriture nécessite 4 pistes. On effectuera la lecture de contrôle avant de changer de piste, ou si le travail est terminé. Si une erreur est détectée lors de la lecture de vérification, on récrit à nouveau les secteurs que l'on a précédemment écrits sur cette piste. Cette opération peut être répétée au maximum 10 fois. Après quoi on considère que l'écriture est impossible et le travail ayant engendré cette écriture est avorté avec un message d'erreur d'écriture. Dans le cas normal (pas d'erreur !), on obtient avec cette méthode la vitesse d'écriture la plus rapide possible, soit la moitié de la vitesse de lecture.

La remarque quant à la longueur des tranches (haut de la page) est donc aussi valable.

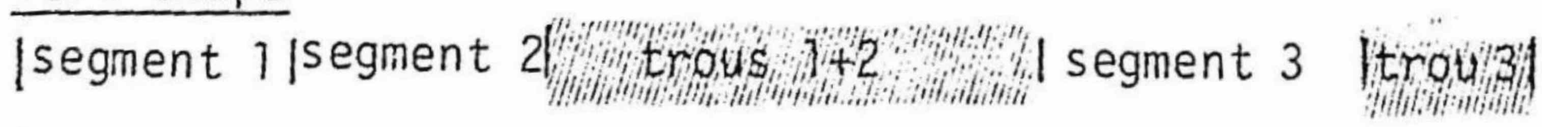
. COMPRESSION D'UN DISQUE: On emploie la méthode suivante: déplacement du segment compris entre deux trous de la valeur du premier trou. Ceci étant recommencé jusqu'à ce que l'on atteigne la fin du disque.

EXEMPLE:

Départ




1ère étape

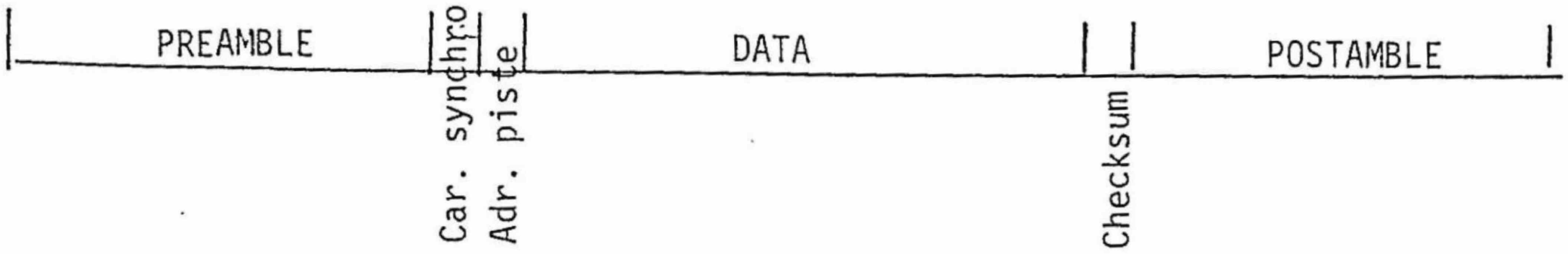



Arrivée

# DESCRIPTION DES APPELS SYSTEME FLOPPY DISQUE

## GENERALITES

FORMAT DES SECTEURS:




- PREAMBLE: cette zone est composée de $50_8$ bytes de synchronisation.
  Elle est utilisée en lecture par la microprogrammation du contrôleur pour se synchroniser.

CARACTERE DE SYNCHRONISATION: il s'agit d'un $377_8$ qui est détecté par le contrôleur pour la synchronisation mot.
Lorsqu'il est lu par le contrôleur, celui-ci active le flag SYNFLG dans le $CONT et commence la lecture des informations écrites sur le disque.

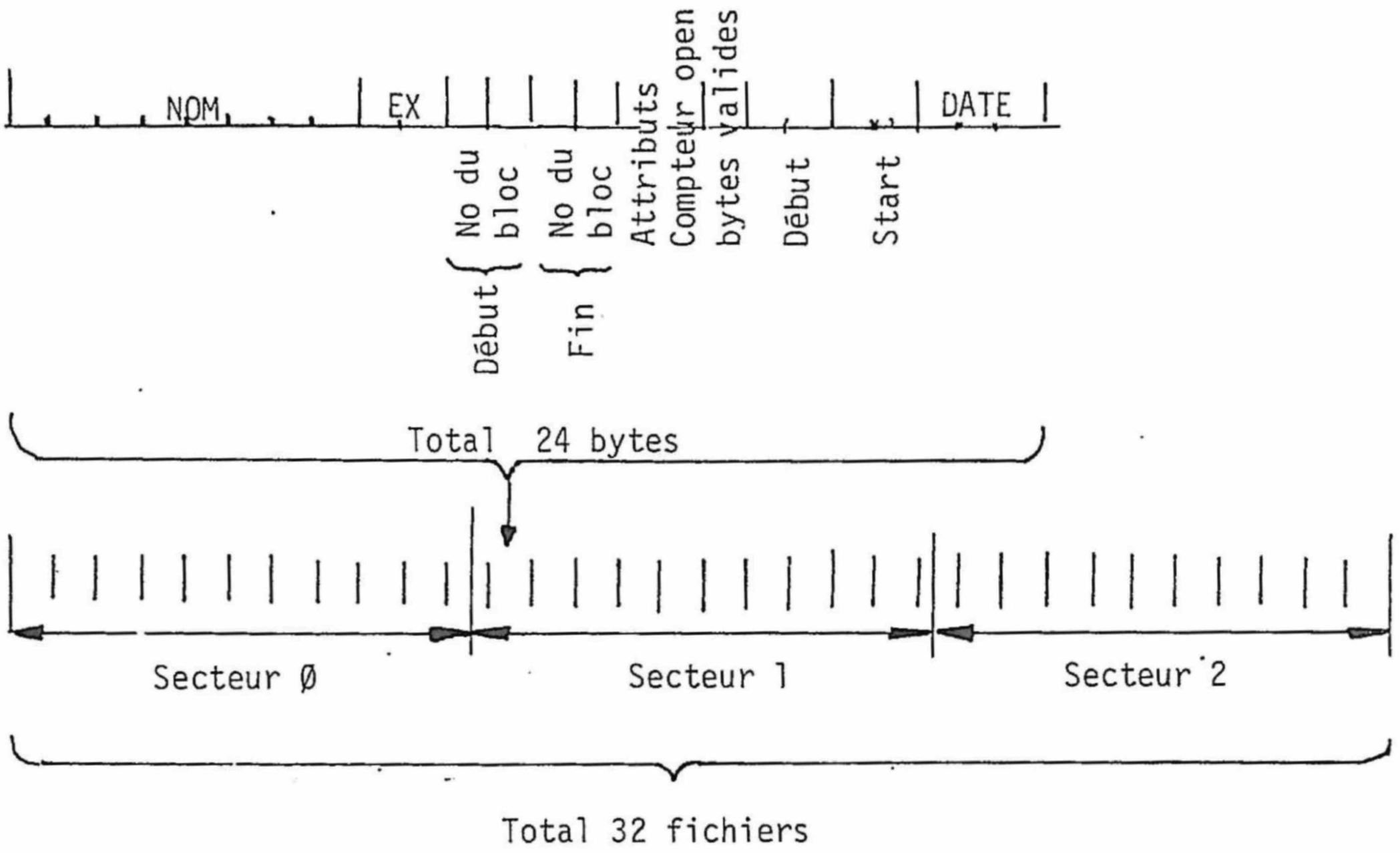
ADRESSE DE LA PISTE: ce premier byte sert, en lecture, à vérifier que l'on est bien sur la piste désirée.

DATA: cette zone contient l'information mémorisée dans ce secteur sous forme de 256 bytes. C'est ce que l'on appellera communément par la suite un bloc.

CHECKSUM: ce byte, enregistré sur le disque lors de l'écriture, est généré par une somme dès 256 bytes de data.
En lecture il permet de détecter une erreur de lecture.

POSTAMBLE: cette zone est générée automatiquement par le contrôleur.
Il s'agit d'environ $50_8$ bytes Ø.
La génération automatique de cette zone par le contrôleur permet l'écriture consécutive de plusieurs secteurs contigus entre eux.

## ORGANISATION DU DIRECTORY




NOM DU FICHIER: 8 caractères ASCII au maximum. Le premier caractère doit obligatoirement être une lettre.

EXTENSION: 2 caractères ASCII au maximum, suivis d'un terminateur reconnaissable.

TERMINATEURS: les terminateurs reconnus sont: espace, retour de chariot, tabulateur, barre oblique à droite, byte Ø et [.

ADRESSE DISQUE DE DEBUT ET FIN: les 4 bytes qui suivent l'extension indiquent les adresses (en blocs) où commence le fichier sur le disque et où il finit.

ATTRIBUTS DU FICHIER: ce byte contient les attributs du fichier et donne en tout temps l'état de ce fichier (ouvert, fermé, protégé, etc.)

| . | . | O | C | P | R | W |

vrai = 1
faux = 0

W = fichier protégé en écriture
R = fichier protégé en lecture.
P = argument W et R protégé
C = fichier ouvert en écriture
O = fichier ouvert en lecture

REMARQUES:

- W et R sont gérés par l'appel ?CHATR.
  P est géré par un appel séparé ?CHATPT, ceci pour une plus grande souplesse d'utilisation de ces attributs.

- C indique que le fichier est ouvert en écriture, tandis que 0 indique qu'il est ouvert en lecture. En fait, 0 est un état virtuel dans le directory. L'image vraie de l'ouverture en lecture est le byte compteur OPEN. Ce bit n'est activé en fonction du compteur OPEN que dans les routines de test des attributs ou dans l'appel ARGS.

COMPTEUR OPEN: Ce byte indique le nombre d'ouvertures de ce fichier en lecture. Zéro signifie qu'il est fermé.

BYTES VALIDES: C'est le nombre de bytes valides dans le dernier bloc du fichier. Il permet de gérer correctement le message "end of file".

DEBUT ET START: ces 4 bytes donnent la position mémoire d'un fichier objet et son adresse de départ. Le début égal à zéro indique qu'il ne s'agit pas d'un fichier objet.

DATE: date de création du fichier

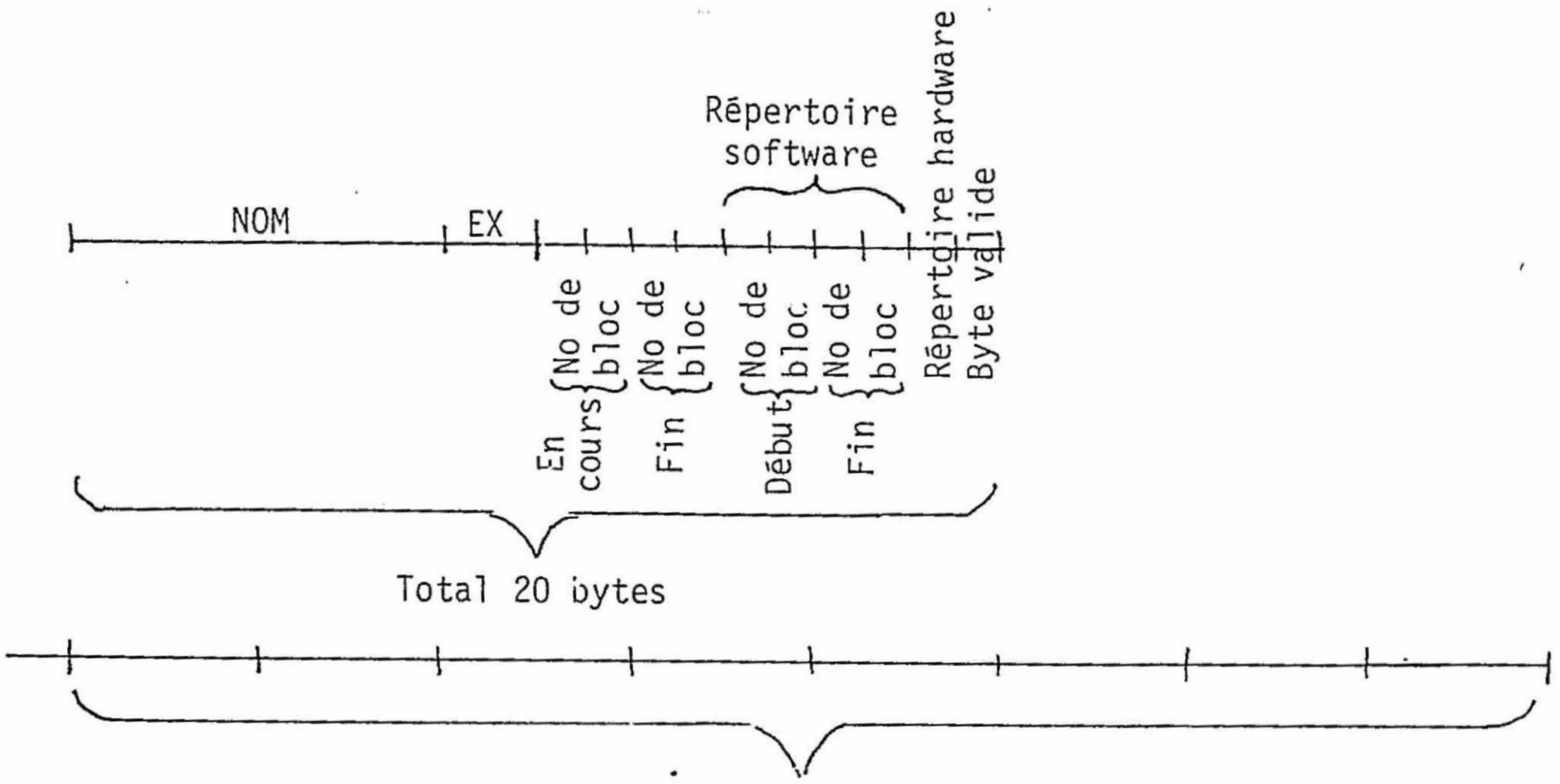
## ORGANISATION DES F.I.T (file information table)

Il existe deux FIT identiques pour les fichiers disque:

. une pour les fichiers ouverts en mode écriture (create),
. une pour les fichiers ouverts en mode lecture (open)

et une FIT spéciale pour les périphériques.

FIT FICHIERS DISQUE




ADRESSE DISQUE EN COURS: les deux bytes suivant l'extension donnent la fin actuelle du fichier, ou, si l'on préfère, la prochaine adresse disque disponible pour ce fichier

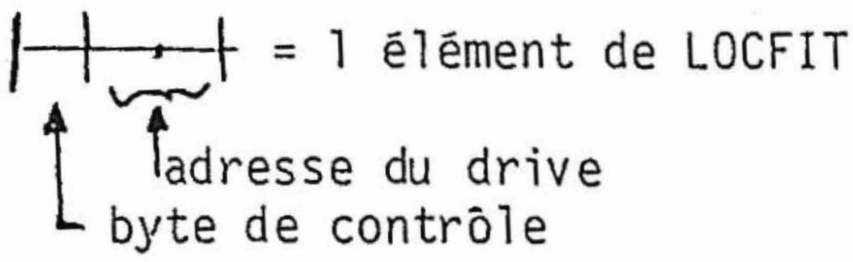
ADRESSE DISQUE DE FIN: ces deux bytes donnent l'adresse disque limite de la réservation pour ce fichier.

REPERTOIRE HARDWARE: c'est l'adresse du drive où se trouve le fichier

REPERTOIRE SOFTWARE: ces deux fois deux bytes donnent l'adresse disque du début et de la fin du répertoire software. Lorsqu'aucun répertoire software n'est utilisé, ces adresses correspondent au début et à la fin du disque

BYTES VALIDES: c'est le nombre de bytes valides dans le dernier bloc dans le cas d'un fichier ouvert en lecture.

FIT FICHIERS PERIPHERIQUES




REMARQUE: La longueur de LOCFIT est fonction du nombre de périphériques traités.

Cette FIT, mémorisée en ROM, est transférée en RAM à l'initialisation.
Elle permet d'effectuer la gestion des périphériques.

BYTE DE CONTROLE: ce byte sert à la fois de No de canal et de byte d'attribut du fichier. Il a l'organisation suivante:

```
          ┌toujours à 1, indique qu'il s'agit d'un canal local
          ↓
         1.D.D.O.C.D.R.W
```

W = fichier protégé à l'écriture
R = fichier protégé à la lecture
C = fichier ouvert en écriture
O = fichier ouvert en lecture
D = bytes utilisés pour différencier les fichiers périphériques entre eux

ADRESSE DRIVER: c'est l'adresse du driver correspondant.
La distinction des travaux à effectuer soit à l'ouverture, soit à la fermeture, ou le travail courant est fait au niveau des drivers de la manière suivante:

| | |
|---|---|
| Entrée dans le driver carry clear | A=Ø travail d'ouverture<br>A≠Ø travail de fermeture |
| Entrée dans le driver carry set | Travail courant |

## 8.3 DESCRIPTION GENERALE

SAMOS est un petit "operating system" d'une taille d'environ 4 kbytes résidant depuis l'adresse 10 000 du SMAKY6. Il permet la gestion de fichiers disques et fichiers périphériques, sous la forme d'appels utilisant le restart 20 du SMAKY6.

Les appels SAMOS auront donc la forme:

?APPEL_SAMOS = No appel * 400 +RST 20

Par exemple:

?CREATE = 23*400+327

Les caractéristiques fondamentales de SAMOS sont: l'autoparamétrisation, l'organisation contiguë, deux types d'accès au fichiers et des indirections sur des routines de base.

### 8.3.1 L'autoparamétrisation

A l'enclenchement du SMAKY6, le programme contenu dans la ROM fantôme charge le software de base (SYSTEM et SAMOS) en mémoire vive et exécute le bootstrap. C'est durant l'exécution de ce bootstrap que SAMOS va s'autoparamétriser. Il va d'abord s'annoncer sur l'écran par SAMOS rév.-vers. Il réserve ensuite une place pour ses différents buffers de travail, ses FIT (file information table) et ses paramètres en RAM. Il va ensuite procéder aux initialisations des RESTART qu'il utilise, soit le 10 pour le hardware et le 20 pour le software.

L'autoparamétrisation proprement dite consistera à reconnaître le nombre de drives actifs sur l'installation. Ceci est fait lors de la phase d'initialisation des drives. Cette initialisation consiste à amener la tête de lecture des drives en piste Ø et d'initialiser le compteur de pistes en mémoire à zéro. Les adresses de drive où ce travail ne peut pas être effectué sont considérés comme invalides et ne seront pas reconnues ultérieurement par le système.
SAMOS signale ces adresses en envoyant sur l'écran la remarque NO DXn, n étant l'adresse non reconnue. On peut donc ainsi changer le nombre de drives sans modifier SAMOS.

### 8.3.2 APPEL DU PROGRAMME CLI.SY

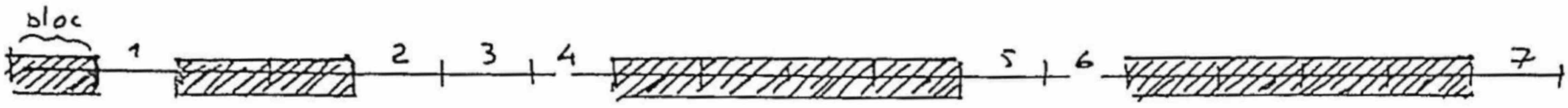
Nous venons de voir tous les travaux d'initialisation du bootstrap. Cependant à la fin de ces initialisations, cet ordre ne se termine pas en revenant au SYSTEME. Il cherche à charger depuis le disque tout d'abord un programme s'appelant ST.SY (programme de start-up). Si ce programme n'existe pas, il tente alors de charger le programme CLI.SY. En cas d'insuccès, il revient alors au SYSTEME en donnant le message ERROR suivi du numéro de l'erreur rencontrée.

Le programme ST.SY permet d'intercaler une fonction supplémentaire avant avant le chargement du CLI.SY, par exemple, chargement de MATPAC.SY ou affichage d'un texte message par exemple.

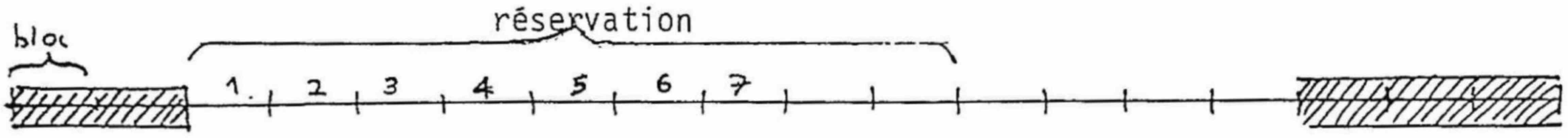
Le programme CLI.SY (command line interpreter) permet de faire effectuer à SAMOS toutes sortes de transactions sous forme de ligne de commande (voir le mode d'emploi de CLI.SY).

## 8.3.3 ORGANISATION CONTIGUE

SAMOS est basé sur une organisation contiguë. Ceia veut dire que les blocs que l'on écrit sur le disque sont contigus entre eux. Contrairement à l'accès aléatoire où l'on saute de bloc vide à bloc vide, ceux-ci n'étant pas forcément contigus, on écrit dans un espace libre réservé à l'avance par tranches de blocs contigus entre eux, les tranches étant elles-mêmes contiguës jusqu'à la fin de l'espace libre réservé.




Organisation aléatoire: on écrira dans 1,2,3 etc. blocs successivement libres.




Organisation contiguë: on écrira dans 1,2,3, etc. blocs contigus libres dans la réservation préalable.

On voit donc que l'organisation choisie a l'inconvénient de nécessiter une réservation préalable. Par contre, ce type d'organisation permet une plus grande rapidité d'accès aux informations. Ce critère fut décisif étant donné la relative lenteur des accès disque avec des micro-floppy.

## 8.3.4 Les accès

Comme les micro-floppy sont la mémoire de masse du système, il est important d'en tirer le meilleur parti possible, du point de vue rapidité.

On peut utiliser deux types d'accès:

. L'accès rapide: cet accès se fait par blocs physiques du disque ($400_8$) directement de disque à mémoire utilisateur ou vice versa.
Cet accès permet de tirer le meilleur prøfit de l'organisation contiguë décrite précédemment, puisque, dans le même passage de lecture (même tour de disque) on arrive à traiter successivement chaque bloc.
La lecture de tous les secteurs d'une piste soit 16. x 256. = 4096. bytes nécessite 200 ms. Cependant cet accès, par le fait qu'il travaille sur des blocs physiques du disque est plus difficile à utiliser du point de vue software.

. l'accès lent: cet accès se fait par byte ou par lignes.
Il utilise des buffers intermédiaires de 1 bloc. Il ne peut donc lire sur le disque dans le même passage (même tour) qu'un bloc à la fois. Il est donc 16x plus lent que l'accès précédent.
Par contre, il est complètement affranchi des contraintes physiques du disque, et permet de travailler totalement en fonction des informations que l'on accède sur le disque.

### 8.3.5 Indirections sur les routines de base

Différentes indirections sur des routines de base permettent, pour des cas bien particuliers, de s'affranchir totalement de la structure et des contraintes de SAMOS et d'accéder directement sur le disque soit en écriture soit en lecture. Ces indirections sont décrites en détail plus loin.

### 8.3.6 Les limites de SAMOS

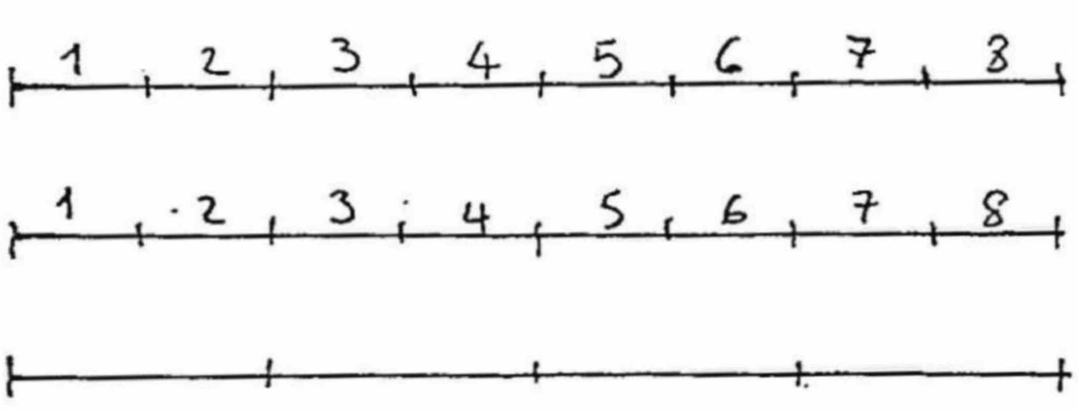
Les limites de SAMOS sont les suivantes.
Il peut reconnaître jusqu'à trois drives "on line".

REMARQUE: il faut cependant noter que dans la version standard actuelle, SAMOS ne reconnaît que deux drives, car la ligne du troisième drive a été utilisée pour gérer le signal HEADLOAD qui permet une optimalisation de l'accès inter-drive pour les drives simple face. Pour les drives double face, cette ligne est utilisée par la ligne de commande de la 2ème tête du drive.

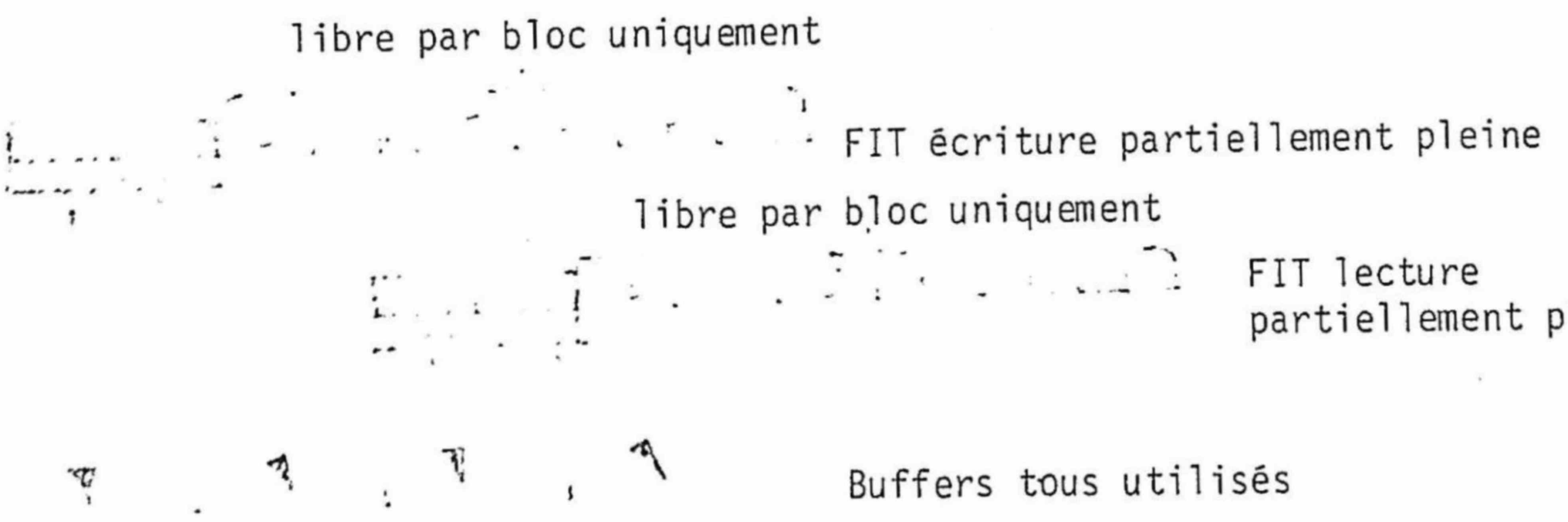
Chaque répertoire peut avoir un maximum de 32 fichiers.

SAMOS dispose de 4 buffers d'entrée/sortie disque; les FIT (file information table) d'écriture ou de lecture peuvent gérer chacune 8 fichiers disque simultanément. Le nombre total de fichiers disque qui peuvent être traités simultanément dépend de la saturation, soit des FIT, soit des buffers.

Par exemple, on peut traiter simultanément 8 fichiers disque en écriture, ouverts en accès par blocs, et 8 fichiers disque en lecture, ouverts en accès par blocs, puisque ce type d'accès n'utilise pas les buffers I/O.




Par contre, si l'on passe à l'accès par byte ou ligne, on ne pourra avoir simultanément que 4 fichiers ouverts dans ce type d'accès. Dans l'exemple ci-dessous nous avons deux fichiers ouverts en écriture et 2 en lecture.




On pourrait par contre rajouter à la configuration ci-dessus encore six fichiers en lecture en accès par blocs et six fichiers en écriture en accès par blocs. Toutes les combinaisons intermédiaires sont évidemment permises.

## 8.3.7 Les fichiers périphériques

Les appels système de SAMOS permettent d'accéder non seulement les fichiers disque, mais également des fichiers périphériques.

On peut accéder à ces fichiers périphériques indifféremment avec l'accès par bloc ou l'accès par byte ou ligne.

Dans les ordres qui permettent de travailler avec les périphériques, on accède à ceux-ci en remplaçant le REP: par $nom_du_périphérique.

Les périphériques décodés sont les suivants:

| | | | |
|---|---|---|---|
| $PR (paper reader) | USART 4 | input | |
| $PP (paper punch) | USART 4 | output | |
| $PI (parallel in) | interface // | input | |
| $PO (parallel out) | interface // | output | |
| $MI (modem in) | USART 6 | input | |
| $MO (modem out) | USART 6 | output | |
| $LP (line printer) | | output | Overlay sur fichier driver LP.SY |
| $KEY(clavier) | | input | |
| $DIS(display) | | output | |

On peut avorter ou terminer la transmission avec un périphérique en pressant sur la touche (KILL)

Le périphérique envoie alors un "end of file" à SAMOS. Dans le cas de transmission avec un périphérique en entrée ($PR et $PI), lorsque la transmission est terminée (plus de son dans le haut-parleur) on signale normalement la fin en pressant (KILL), ceci provoque la fin de l'exécution de l'ordre (fermeture des fichiers). Dans le cas de transmission avec un périphérique en sortie il n'y a normalement pas lieu de presser une touche, puisque le "end of file" sera donné par le fichier que l'on transmet. On peut cependant avorter la transmission en pressant (KILL).

## 8.3.8 Principes généraux des appels SAMOS

Comme pour les appels SYSTEME, on est automatiquement "interrupt on" au retour d'un appel SAMOS.

Les appels SAMOS affectent toujours les registres A et F. Au retour d'un appel SAMOS, le carry est utilisé pour signaler une erreur éventuelle.

Un retour carry set indique qu'une erreur s'est produite, le numéro de cette erreur se trouvant alors dans le registre A. Un retour carry clear indique évidemment un bon déroulement de l'appel.

La plupart des appels SAMOS ont des paramètres en entrée et en sortie. La restitution des paramètres en sortie peut être liée à la condition du bon déroulement ou non de l'appel SAMOS. D'une manière générale, les registres, à part A et F, qui ne sont pas utilisés pour rendre des paramètres en sortie ne sont pas affectés. Les registres prévus pour ne rendre des paramètres qu'en cas de bon déroulement de l'appel ne seront affectés que si l'appel s'est effectué correctement. Dans le cas inverse (paramètres de sortie en cas d'erreurs) ce principe joue également.

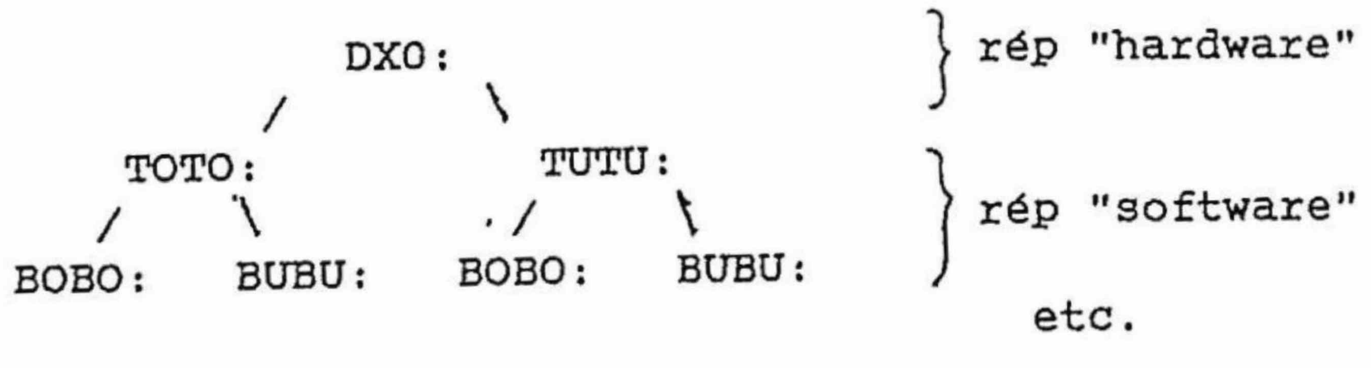
Lors de l'exécution d'un appel SAMOS qui nécessite un accès disque, SAMOS supprime l'interruption 50Hz et se met en mode alpha uniquement.

Ceci veut dire que le clavier n'est plus balayé durant l'accès disque, et l'écran graphique plus visible.

Pour garantir le bon fonctionnement des accès disque, l'utilisateur qui aurait une configuration de système utilisant une autre interruption hardware (exemple: autre prériphérique générant des interruptions) doit supprimer cette autre interruption avant d'effectuer un appel SAMOS nécessitant un accès disque. Ceci afin de garantir la plus haute priorité aux accès disque.

## 8.3.9 Organisation des disquettes

Chaque disquette peut contenir des fichiers répertoires qui peuvent contenir chacun 32 fichiers. Parmi ces fichiers, il peut y avoir également des fichiers répertoires qui peuvent contenir aussi d'autres fichiers répertoires etc..
On peut donc créer une structure arborescente de répertoires et de sous-répertoires.




Traduction du schéma ci-dessus:

sur la diquette en DX0: il y a notamment deux fichiers répertoires TOTO.DR et TUTU.DR qui contiennent chacun entre autres deux fichiers sous-répertoires BOBO.DR et BUBU.DR (l'utilisation du même nom n'est pas interdite).

## 8.4 LES APPELS SAMOS

### 8.4.1 DESCRIPTION GENERALE DES PARAMETRES

#### 8.4.1.1 Pointeur au nom

Par pointeur au nom, on entend un pointeur d'une chaîne de caractères ASCII, qui peut contenir les informations suivantes pour un fichier disque:

- . lien du répertoire
- . nom du fichier
- . extension du nom de fichier disque
- . réservation de blocs

Ou simplement pour un périphérique:

- . nom du périphérique.

Cette chaîne ASCII peut avoir au début des espaces ou des tabulateurs, ils seront automatiquement sautés.

Elle doit par contre être terminée par un des caractères suivants: SPACE, TAB, CR, SLASH, ZERO.

Si la chaîne ASCII contient plusieurs informations, elles ne devront pas être séparées par des caractères non significatifs, mais obligatoirement mises bout à bout. Les lettres minuscules non accentuées sont acceptées mais transformées automatiquement en majuscules par le système. La même information peut donc être indifféremment en minuscules ou en majuscules.

Lien du répertoire: On distingue deux types de répertoires:
au plus bas niveau du système les répertoires hardware qui ont la forme DXn: n étant de numéro du drive (Ø ou 1) que l'on désire accéder. Lorsque cette information est omise, SAMOS prend DXØ: par défaut.
Les répertoires software créés par l'utilisateur, qui sont des fichiers avec l'extension spéciale .DR
L'accès à ces répertoires à la forme REP: REP étant le nom du fichier répertoire. La syntaxe de ce nom est donc celle d'un nom de fichier (voir ci-dessous).

On peut créer des répertoires à l'intérieur d'un répertoire, ce qui permet d'obtenir une structure arborescente (voir § 8.3.9).
Le lien d'un répertoire pourra donc avoir la forme suivante DX1:TOTO:TITI:
On accède dans cet exemple au sous-répertoire TITI.DR dans le répertoire TOTO.DR sur la disquette du drive DX1:

Nom du fichier disque: il est formé d'un maximum de 8 caractères.
Le premier doit obligatoirement être une lettre, les suivants des lettres ou des chiffres.

Extension du nom du fichier disque: elle est formée de deux caractères au maximum qui peuvent être soit des lettres, soit des chiffres.
Elle est précédée obligatoirement par un point.
Elle peut être omise. L'extension sert généralement à caractériser le genre de fichier.

Exemple:
Le fichier source TOTO qui se trouve sur un disque inséré dans le drive d'adresse Ø sera parfaitement caractérisé par les chaînes ASCII suivantes:

```
.ASCII /<TAB><SPACE>DXØ:TOTO.SR<CR>/
.ASCIZ /<TAB>TOTO.SR/
.ASCIZ /toto.sr/
etc.
```

L'extension .MC, réservée aux MACROS, est reconnue pas SAMOS, et elle est "reset protected" à la lecture. En effet, si un programme, appelé par une MACRO effectue un appel RESET, il ne faut pas que le fichier MACRO soit fermé par cet appel, sous peine d'avorter la suite de l'exécution de la MACRO.

Réservation de blocs: pour les noms de fichier en création exclusivement on peut accoller une réservation de blocs entre crochets.

Cette réservation a le même effet que celle passée par le paramètre d'entrée correspondant (voir 8.4.1.2 et appel CREATE ou CRBLK).

Cependant, la réservation entre crochets n'est prise en considération que si le paramètre d'entrée de l'appel de création de ce fichier est égal à zéro.

Exemples de réservations entre crochets:

TOTO.SR[100]

TOTO[28]

TOTO.LS[128|129]
ignoré pris en compte

TOTO.SR[0]
└ réservation par défaut,
soit la plus grande place sur deux

Nom du périphérique: il est simplement formé des lettres caractérisant le périphérique précédé du signe $.

Pour les appels SAMOS qui effectuent un travail au niveau d'un répertoire, et non pas au niveau d'un fichier, il n'est pas nécessaire (mais pas interdit!) de spécifier un nom de fichier. Seule l'information "lien du répertoire" sera prise en compte. Dans le cas du répertoire hardware DXØ:, nous savons que cette information peut être omise. Ainsi, dans ce cas précis, il suffirait que le pointeur au nom soit sur un terminateur (CR, SLASH, ZERO).

## 8.4.1.2 Numéro de canal

La création ou l'ouverture (lecture) d'un fichier nécessite un nom. Si l'on crée un fichier, ce sera le nom que l'on désire attribuer à ce fichier; si l'on ouvre un fichier, ce sera le nom qui avait été donné à la création.

Si l'on crée, ou l'on ouvre un fichier, c'est pour y écrire ou y lire des informations. Pour faire ces opérations, il faudra à nouveau caractériser ce fichier, puisque l'on peut avoir plusieurs fichiers ouverts ou créés en même temps.

Pour ce faire, SAMOS, au moment de la création ou de l'ouverture de ce fichier, attribue un numéro de canal qu'il rend à l'utilisateur. C'est dès lors ce numéro qui caractérisera le fichier et qui sera utilisé pour effectuer les opérations d'écriture, de lecture et l'opération finale de fermeture.

Il est plus facile de manipuler un numéro que de travailler avec un nom. D'autre part, comme un numéro de canal est toujours non nul, on peut utiliser le zéro pour signaler que le canal n'est pas ouvert (mise à zéro préalable et à la fermeture des positions mémoire utilisées pour mémoriser ces numéros de canaux).

Format du No de canal

Le numéro de canal contient d'autre part des informations qui caractérisent le fichier concerné et qui peuvent être testées par l'utilisateur.

Numéro de canal fichier disque | Ø T A x x x x x | (1 byte)

Ø=fichier disque (1=périphérique)
Ø=lecture 1=écriture
Ø=accès par bloc 1=accès par byte ou ligne
numérotation distinctive

Numéro de canal fichier périphérique | 1 D D O C D R W | (1 byte)

W = fichier protégé à l'écriture
R = fichier protégé à la lecture
C = fichier ouvert en écriture
O = fichier ouvert en lecture
D = bit de distinction

8.4.1.3 Nombre de blocs à réserver

C'est le nombre de blocs physiques du disque ($400_8$) en binaire que l'on désire réserver à un fichier que l'on crée. On peut obtenir une réservation par défaut en mettant ce paramètre à zéro. A ce moment SAMOS réservera la moitié de la plus grande place vide.

Dans le cas contraire, SAMOS effectuera la réservation dans le plus petit-trou possible.

8.4.1.4 Pointeur en mémoire

Ce pointeur est sur la première position de l'endroit où, selon que l'on lit ou que l'on écrit, on va délivrer ou chercher l'information qui fait l'objet du transfert.

8.4.1.5 Numéro de canal

Ce paramètre est rendu par SAMOS après l'ouverture ou la création d'un fichier. Il est utilisé par la suite pour communiquer avec le dit fichier.

### 8.4.1.6 Nombre de bytes à lire ou à écrire

Comme nous l'avons vu, SAMOS, peut travailler avec un type d'accès rapide par blocs physiques du disque. Cependant dans ce type d'accès, l'unité de travail est le bloc, soit $400_8$. Il est rare que les fichiers avec lesquels nous allons travailler aient une taille multiple d'un nombre de blocs. Aussi, si nous écrivons ou lisons un fichier par ce type d'accès, il faudra respecter les deux règles suivantes:

. En accès par blocs, on lira toujours un fichier par tranches multiple de $400_8$
. En accès par blocs en écriture, seule la dernière tranche écrite pourra ne pas être multiple de $400_8$.

Pour l'appel UPDATE, spécifiez toujours de préférence un multiple de $400_8$, surtout à l'écriture car on écrira toujours le nombre supérieur de blocs. La taille d'un fichier (nombre de bytes dans le dernier bloc) n'est pas modifiée (et pas modifiable) par l'appel UPDATE. En effet, on accède en UPDATE dans un fichier avec une ouverture en lecture par bloc (OPEBLK). Sa taille a été définie une fois pour toutes lors du CLOSE qui a suivi sa création.

Si l'on utilise l'accès par byte, on est évidemment libre des contraintes décrites ci-dessus, mais la vitesse d'accès sur le disque est plus lente.

### 8.4.1.7 Nombre de bytes lus ou écrits

Ce paramètre est rendu à la fin d'un travail de lecture ou d'écriture. En accès par blocs en écriture, ce paramètre est toujours un multiple entier de $400_8$ correspondant aux blocs effectivement écrits sur le disque. Lorsque la dernière tranche écrite dans le fichier n'est pas multiple de $400_8$, le nombre de bytes valides dans le dernier bloc est mémorisé par SAMOS, et sera écrit dans le directoire au moment de la fermeture de ce fichier. Le nombre de bytes écrits étant alors le multiple supérieur de $400_8$ du nombre de bytes à écrire spécifié en entrée.

Par contre, en accès par blocs en lecture, au moment du "end of file", ce paramètre nous donne le nombre de bytes valides par tranche lue, qui n'est pas forcément un multiple de $400_8$. Il est important de savoir que par ce type d'accès, le nombre de bytes effectivement transférés est le multiple supérieur de $400_8$ au nombre de bytes spécifié. En d'autres mots, le dernier bloc est transféré intégralement et peut contenir des bytes non significatifs.

Si l'on utilise l'accès par byte, ce paramètre correspond simplement au nombre de bytes effectivement lus ou écrits.

Au sujet de ce paramètre de retour, lisez également la description des messages d'erreur "end of file" et "file end overflow".

### 8.4.1.8 Adresse de début, adresse de start

Le fichier que l'on crée sur le disque peut contenir un programme sous forme d'une image mémoire. Dans ce cas, il faut que SAMOS puisse connaître où se situe le programme en mémoire, c'est-à-dire l'adresse de début, et où commence son exécution, c'est-à-dire l'adresse de start.
Ces paramètres lui sont spécifiés au moment de la fermeture du fichier (appel CLOSE) et sont mémorisés dans le directoire. Pour l'appel RESET ces paramètres sont remis à zéro.
Si le fichier que l'on traite n'est pas un programme sous forme d'une image mémoire, la valeur du paramètre "adresse de start" est indifférente; par contre l'adresse de début doit valoir zéro. SAMOS saura ainsi que ce fichier ne contient pas un programme image mémoire exécutable et il pourra ainsi assurer une protection en cas de tentative de chargement et d'exécution de ce fichier.

## 8.4.2 DESCRIPTION DES MESSAGES D'ERREUR

Les messages d'erreur en clair sont contenus dans le fichier ER.SY.
L'appel SAMOS ?ERROR (voir 8.4.28) utilise ce fichier pour traduire le message.

### No 1: Fichier protégé à l'écriture (write protect file)

Ce message signifie que le fichier traité est protégé à l'écriture.
On reçoit ce message si l'on tente de tuer ce fichier, ou de modifier son nom.

### No 2: Fichier protégé à la lecture (read protect file)

Ce message "fichier protégé à la lecture" apparaît si l'on tente d'ouvrir ce fichier, de charger et d'exécuter ce fichier.

### No 4: Fichier permanent (permanent file)

Ce message signifie que les arguments du fichier traité sont protégés.
On reçoit ce message si l'on tente de changer les attributs du fichier.

### No 5: Ligne trop longue (line too long)

Ce message apparaît lorsque la ligne traitée par l'un des appels WRLINE ou RDLINE est plus longue que 256 caractères.

### No 6: Fin du fichier (end of file)

Ce message signifie que l'on a atteint la fin du fichier en lecture.
On reçoit ce message lorsque le nombre de bytes à lire correspond exactement, ou dépasse, le solde de bytes à lire dans le fichier. Le paramètre de retour "nombre de bytes lus" (8.4.1.6) donne alors le nombre de bytes valides.

### No 7: Fichier plein (file end overflow)

Ce message signifie que l'on a tenté d'écrire par dessus la fin d'un fichier en écriture. Le paramètre de retour "nombre de bytes écrits" (8.4.1.6) donne le nombre de bytes effectivement écrits.

No 10: Ouvert en écriture

On reçoit le message "fichier ouvert en écriture" si l'on tente de faire toute autre opération qu'une écriture ou une fermeture de ce fichier.

No 11: Fichier existant (file already exist)

Ce message signifie que le nom du fichier traité existe déjà dans le répertoire. On reçoit ce message si l'on tente de créer un fichier sous un nom qui existe déjà, si l'on tente de renommer un fichier avec un nom qui existe déjà.

No 12: Fichier inexistant (file does not exist)

Ce message signifie que le nom du fichier traité n'existe pas dans le répertoire. On reçoit ce message si l'on tente d'ouvrir, de renommer, de changer les attributs ou de tuer un fichier qui n'existe pas.

No 13: Nom illégal (illegal filename)

Ce message signifie que le nom ou l'extension du fichier traité ne respecte pas la syntaxe (8 caractères au maximum, premier caractère obligatoirement une lettre, pas de signes spéciaux, extension maximum deux caractères, lettres ou chiffres).

No 14: Réservation illégale (Illegal reservation)

Ce message signifie que la réservation donnée entre parenthèses carrées ne respecte pas la syntaxe. Exemples: lettre au lieu de chiffre, oubli de la parenthèse de fermeture, etc.

No 16: Fichier non exécutable (cannot load file)

Ce message signifie que le fichier traité ne contient pas un programme image mémoire exécutable (l'adresse de début dans le directoire égale à zéro).
On reçoit ce message si l'on tente de charger et d'exécuter ce genre de fichier.

No 17: Hors du fichier (out of file)

Ce message concerne uniquement l'appel SAMOS UPDATE. Il signifie que le numéro de bloc spécifié est en dehors des limites du fichier.

No 20: Ouvert en lecture (file in use for reading)

Ce message signifie que le fichier traité est ouvert en lecture.
On reçoit ce message si l'on tente de faire toute autre opération qu'une ouverture, une lecture ou une fermeture de ce fichier.

No 21: Répertoire inconnu (unknown device)

Ce message signifie que l'on tente d'accéder à un répertoire hardware ou software qui n'existe pas.

No 22: Erreur de canal (channel error)

Ce message signifie que le numéro de canal que l'on a spécifié n'existe pas ou que ce canal n'est pas ouvert.

No 23: Fichier(s) ouvert(s) (file(s) in use)

Ce message signifie qu'il y a des fichiers ouverts. On reçoit ce message si l'on tente d'effectuer une compression de disque alors que les fichiers sont encore ouverts.

No 24: Plus de canal libre (all channels in use)

Ce message veut dire que l'on a saturé les FIT (file information table) ou les buffers input/output (voir les limites de SAMOS, § 8.3.6). On reçoit ce message si l'on tente d'ouvrir ou de créer un fichier alors qu'il n'y a pas de canal disponible.

No 25: Répertoire plein (directory full)

Ce message signifie que le répertoire contient déjà 32 fichiers. On reçoit ce message si l'on tente d'en créer un nouveau.

No 26: Disque plein (disk full)

Ce message veut dire que l'on ne peut pas réserver la place demandée sur le disque. Ceci signifie que le nombre de blocs à réserver est supérieur au nombre de blocs du plus grand trou dans le répertoire hardware ou software.

No 30: Floppy hors service (device timeout)

Ce message veut dire que l'adresse de ce drive est reconnue, mais qu'en revanche il ne fonctionne pas.

No 31: Disque protégé (write protect tab set)

Ce message signifie que le disque a un cache de protection à l'écriture. On reçoit ce message si l'on tente, sur ce disque, de faire autre chose qu'une ouverture, une lecture ou une fermeture de fichier.

No 32: Erreur d'écriture (write error)

Ce message veut dire que SAMOS n'a pas réussi à écrire correctement un bloc durant un accès disque en écriture. Le travail a donc été avorté. Cette erreur est fatale lors d'une compression de disquette.

Lors d'une phase d'écriture dans un fichier, les blocs écrits durant cette phase sont irrécupérables par SAMOS.

No 33: Erreur de lecture (read error)

Ce message signifie que SAMOS n'a pas réussi a lire correctement un bloc durant un accès disque en lecture. Le travail a donc été avorté. Cette erreur est fatale lors d'une compression de disque.

No 34: Pas d'exécution (no starting address)

Ce message signifie que le fichier que l'on a chargé n'a pas d'adresse de start (adresse de start égale à 1).

Le fichier ER.SY contient encore dlautres messages utilisés par d'autres programmes. Pour le détail de leur signification, se référer aux notices de ces programmes.

Messages du CLI

No 35: Mauvais chargement (bad load)

No 36: allocation pleine (buffer full)

Messages de MATPAC (appels mathématiques)

No 37: Division par zéro (divide by zero)
No 40: Trop grand (overflow)
No 41: Trop petit (underflow)
No 42: Nombre illégal (illegal number)
No 43: Racine négative (negative square root)
No 44: Log négatif (negative logarithm)

Messages de MATPAC (appels fichier à structure de record)

No 45: Longueur nulle (record length null)
No 46: Pas d'enregistrement (zero record)
No 47: Fichier incompatible (no record file)
No 50: Allocation trop petite (buffer too small)
No 51: Recherche illégale (illegal search parameters)

Messages divers

Mo 110: Ordre illégal (illegal order)
No 114: Erreur système (system error)
No 115: Programme détruit (map error)

## 8.4-3 APPEL Ø ?CREBLK

Ouverture d'un fichier disque ou périphérique en écriture avec accès rapide par blocs.

Paramètres d'entrée DE: pointeur au nom
BC: bloc à réserver
(BC=Ø réservation par défaut)

Paramètres de sortie: A: numéro de canal

Liste des erreurs possibles pour cet appel:

1 Write protect file
10 File in use for writing
11 File already exists
13 Illegal filename
14 Illegal reservation
21 Unknown device
24 All channels in use
25 Directory full
26 Disk full
30 Device timeout
31 Write protect tab set
32 Write error
33 Read error

Remarques:
. une erreur disque à l'écriture pour cet appel est fatale car elle concerne l'écriture du directoire sur le disque
. les erreurs No 1 et 10 concernent les fichiers périphériques: si le périphérique spécifié n'est pas un périphérique de sortie on a l'erreur 1 ou si le périphérique est déjà ouvert, on a l'erreur 10.

## 8.4-4 APPEL 23 ?CREATE

Ouverture d'un fichier disque ou périphérique en écriture avec accès lent par bytes ou par lignes.

Paramètres d'entrée: DE pointeur au nom

BC bloc à réserver
(BC=Ø réservation par défaut)

Paramètre de sortie: A numéro de canal

Liste des erreurs possibles pour cet appel

1 Write protect file
10 File in use for writing
11 File already exist
13 Illegal filename
14 Illegal reservation
21 Unknown device
24 All channels in use
25 Directory full
26 Disk full
30 Device timeout
31 write protect tab set
32 Write error
33 Read error

Remarques:
- une erreur disque à l'écriture pour cet appel est fatale car elle concerne l'écriture du directoire sur le disque
- Les erreurs No 1 et 10 concernent les fichiers périphériques: si le périphérique spécifié n'est pas un périphérique de sortie on a l'erreur 1 ou si le périphérique est déjà ouvert, on a l'erreur 10.

## 8.4-5 APPEL 35 ?CDIR

Création d'un fichier répertoire.

Paramètres d'entrée: DE pointeur au nom

BC bloc à réserver
(BC=Ø réservation par défaut)

Liste des erreurs possibles pour cet appel

11 File already exists
13 Illegal filename
14 Illegal reservation
21 Unknown device
24 All channels in use
25 Directory full
26 Disk full
30 Device timeout
31 Write protect tab set
32 Write error
33 Read error

Remarques:
- pour un fichier répertoire l'extension du nom du fichier sera obligatoirement .DR
  La taille utile sera la réservation moins les trois blocs nécessaires au directoire
- pour d'autres applications, on peut créer avec cet appel des fichiers ayant une autre extension que .DR ou une taille inférieure à 3 blocs. L'appel ne génère pas d'erreur dans ces deux cas là.

### 8.4.6 APPEL 3 ?OPEBLK

Ouverture d'un fichier disque ou périphérique en lecture avec accès rapide par blocs.

Paramètre d'entrée: DE: pointeur au nom

Paramètre de sortie: A: numéro de canal

Liste des erreurs possibles pour cet appel:

2 Read protect file
12 File does not exist
13 Illegal filename
20 File in use for reading
21 Unknown device
24 All channels in use
30 Device timeout
32 Write error
33 Read error

Remarques:
- Une erreur disque à l'écriture pour cet appel est fatale, car elle concerne l'écriture du directoire sur le disque
- Les erreurs No 2 et 20 concernent les fichiers périphériques: si le périphérique spécifié n'est pas un périphérique d'entrée, on a l'erreur 2, si le périphérique est déjà ouvert, on a l'erreur 20.

### 8.4.7 APPEL 24 ?OPEN

Ouverture d'un fichier disque ou périphérique en lecture avec accès lent par byte ou par ligne.

Paramètre d'entrée: DE: pointeur au nom

Paramètre de sortie A: numéro de canal

Liste des erreurs possibles pour cet appel

2 Read protect file
12 File does not exist
13 Illegal filename
20 File in use for reading
21 Unknown device
24 All channels in use
30 Device timeout
32 Write error
33 Read error

Remarques:
- Une erreur disque à l'écriture pour cet appel est fatale, car elle concerne l'écriture du directoire sur le disque.
- Les erreurs No 2 et 20 concernent les fichiers périphériques: si le périphérique spécifié n'est pas un périphérique d'entrée on a l'erreur 2, ou si le périphérique est déjà ouvert, on a l'erreur 20.

## 8.4.8 APPEL 4 ?CLOSE

Fermeture d'un fichier disque ou périphérique ouvert soit en lecture , soit en écriture, et indépendamment du type d'accès.

Paramètres d'entrée: A: Numéro de canal

Pour les fichiers disque uniquement { BC: adresse de début, DE: adresse de start } fichier binaire en écriture

fichier source BC=Ø
DE: quelconque

Liste des erreurs possibles pour cet appel

22 Channel error
30 Device timeout
32 Write error
33 Read error

Remarque: . une erreur disque à l'écriture pour cet appel est fatale, car elle concerne l'écriture du directoire sur le disque.

## 8.4.9 APPEL 6 ?RDBLOC

Lecture d'un fichier disque ou périphérique ouvert en accès rapide par blocs.

Paramètres d'entrée: A: No de canal
BC: nombre de bytes à lire (REMARQUE IMPORTANTE: toujours multiple de $400_8$ (voir 8.4.1.5)
DE: pointeur en mémoire

Paramètres de sortie: BC: nombre de bytes lus

Liste des erreurs possibles pour cet appel

6 End of file
22 Channel error
30 Device timeout
33 Read error

### 8.4.10 APPEL 25 ?RDBYTE

Lecture par byte d'un fichier disque ou périphérique ouvert en accès lent par byte ou par ligne.

Paramètres d'entrée: A: numéro de canal
BC: nombre de bytes à lire
DE: pointeur en mémoire

Paramètre de sortie : BC: nombre de bytes lus

Liste des erreurs possibles pour cet appel

6 End of file
22 Channel error
30 Device timeout
33 Read error

### 8.4.11 APPEL 7 ?RDLINE

Lecture d'une ligne d'un fichier disque ou périphérique ouvert en accès lent par byte ou par ligne.

Paramètres d'entrée: A: numéro de canal
DE: pointeur en mémoire

Paramètres de sortie: BC:longueur de la ligne en bytes

Liste des erreurs possibles pour cet appel:

5 Line too long
6 End of file
22 Channel error
30 Device timeout
33 Read error

Remarque: . La longueur des lignes est limitée à 256 caractères.
Les terminateurs reconnus pour la ligne sont: ZERO, CR, FF.

## 8.4.12 APPEL 1Ø ?WRBLOC

Ecriture dans un fichier disque ou périphérique ouvert en accès rapide par bloc.

Paramètres d'entrée: A: numéro de canal
BC: nombre de bytes à écrire (REMARQUE IMPORTANTE: seule la dernière tranche écrite dans le fichier n'est pas forcément multiple de $400_8$ (voir 8.4.1.5)
DE: pointeur en mémoire

Paramètres de sortie: BC: nombre de bytes écrits

Liste des erreurs possibles pour cet appel:

7 File end overflow
22 Channel error
30 Device timeout
33 Write error

## 8.4.13 APPEL 26 ?WRBYTE

Ecriture par byte dans un fichier disque ou périphérique ouvert en accès lent par byte ou par ligne.

Paramètres d'entrée: A: numéro de canal
BC: nombre de bytes à écrire
DE: pointeur en mémoire

Paramètres de sortie: BC: nombre de bytes écrits

Liste des erreurs possibles pour cet appel:

7 File end overflow
22 Channel error
30 Device timeout
32 Write error

## 8.4.14 APPEL 11 ?WRLINE

Ecriture d'une ligne dans un fichier disque ou périphérique ouvert en accès lent par byte ou par ligne.

Paramètres d'entrée: A: numéro de canal
DE: pointeur en mémoire

Paramètre de sortie: BC: longueur de la ligne en bytes.

Liste des erreurs possibles pour cet appel:

5 Line too long
7 File end overflow
22 Channel error
30 Device timeout
32 Write error

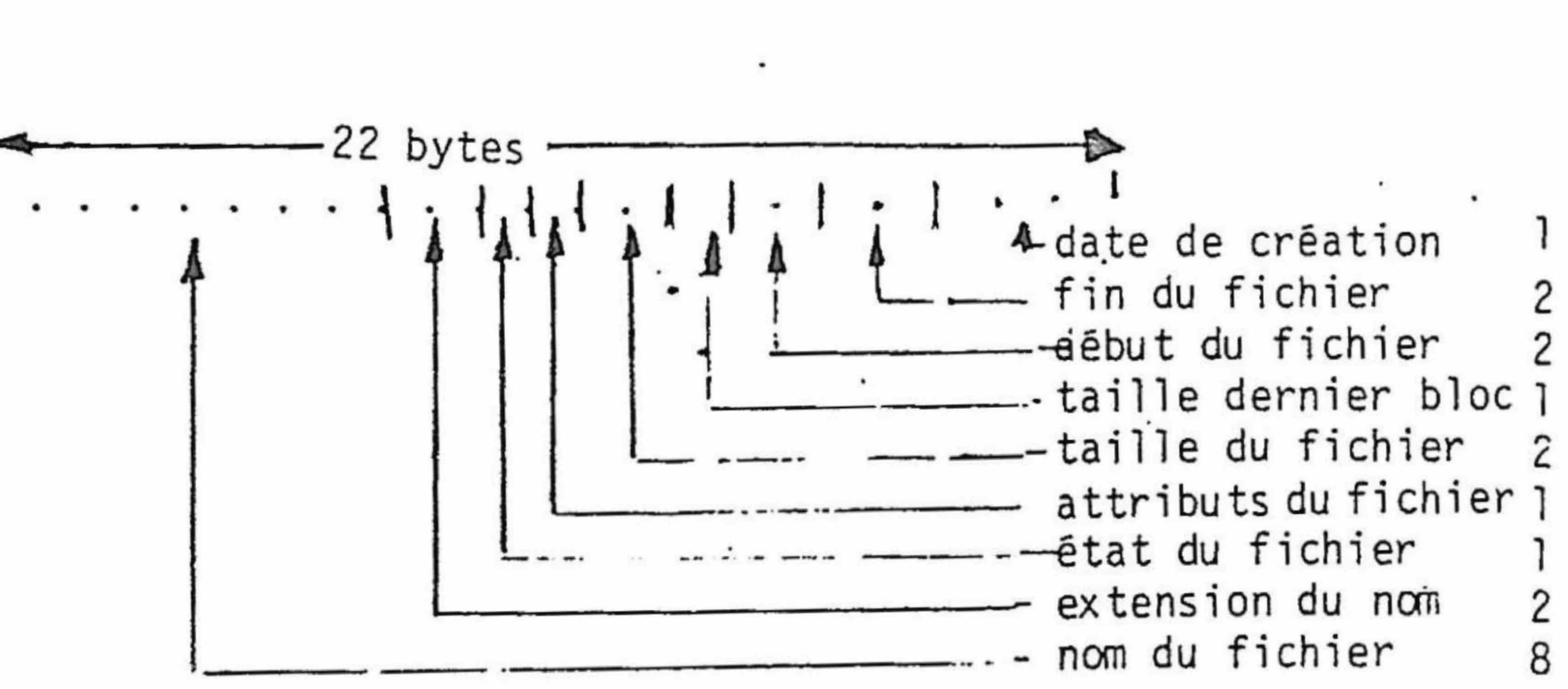
Remarque: La longueur de la ligne est limitée à 256 caractères.
Les terminateurs reconnus sont: ZERO, CR, FF.

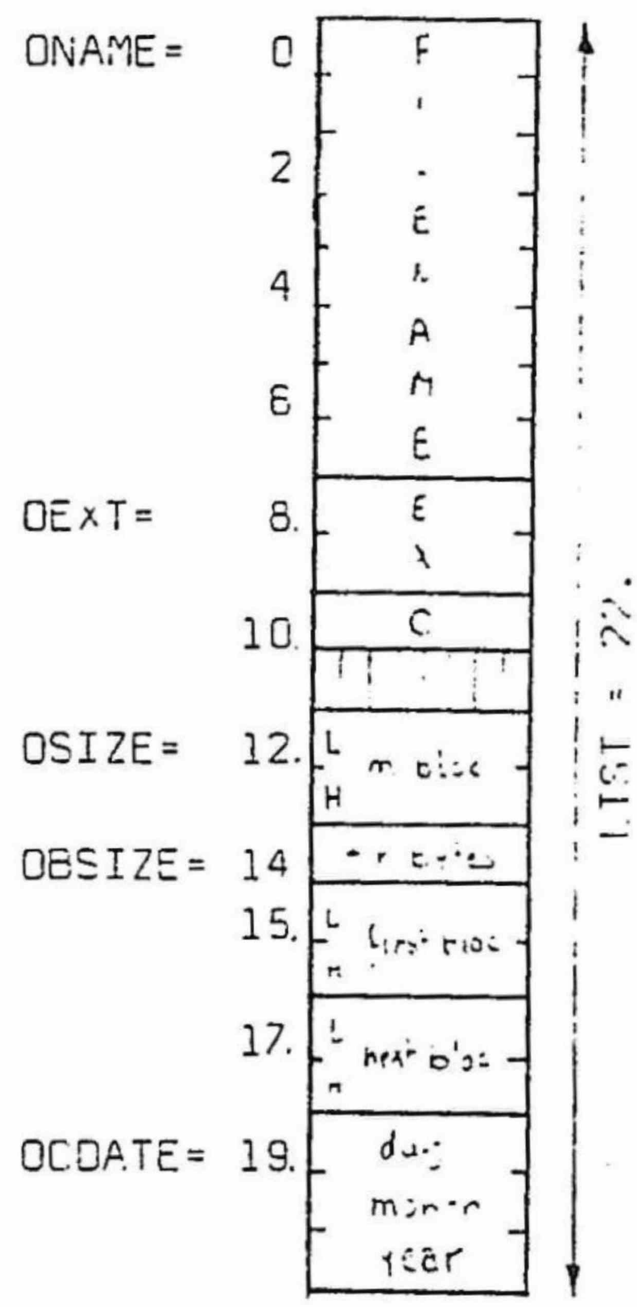
## 8.4.15 APPEL 16 ?LIST

Produit la liste des fichiers d'un répertoire

Paramètres d'entrée: DE: pointeur au nom
BC: pointeur en mémoire

Format d'un élément de la liste




Fichier définition

Date de création: en BCD date de création du fichier dans l'ordre JJMMAA

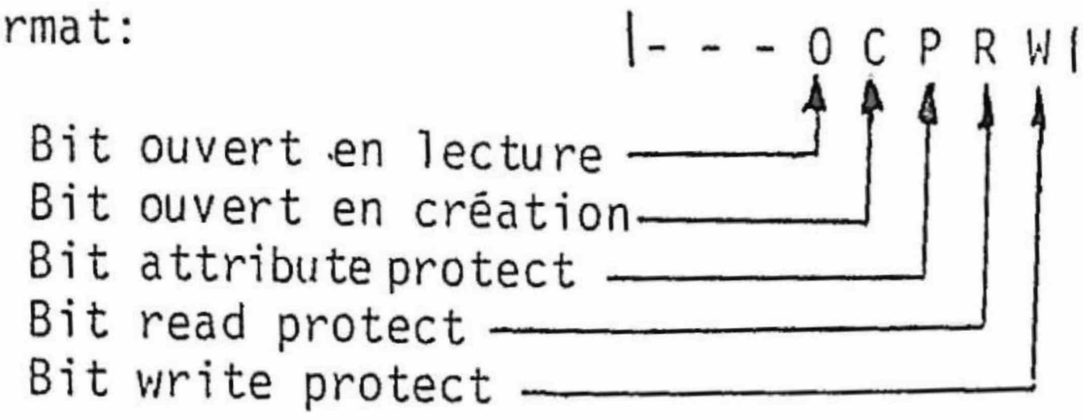
Fin du fichier: en binaire, numéro du dernier bloc + 1

Début du fichier: en binaire, numéro du premier bloc

Taille dernier bloc: en binaire, nombre de bytes valides pour le dernier bloc; 0/ signifiant bloc entièrement valide, soit $400_8$.

Taille du fichier: en binaire, taille du fichier en blocs

Attributs du fichier: 1 byte selon le format:



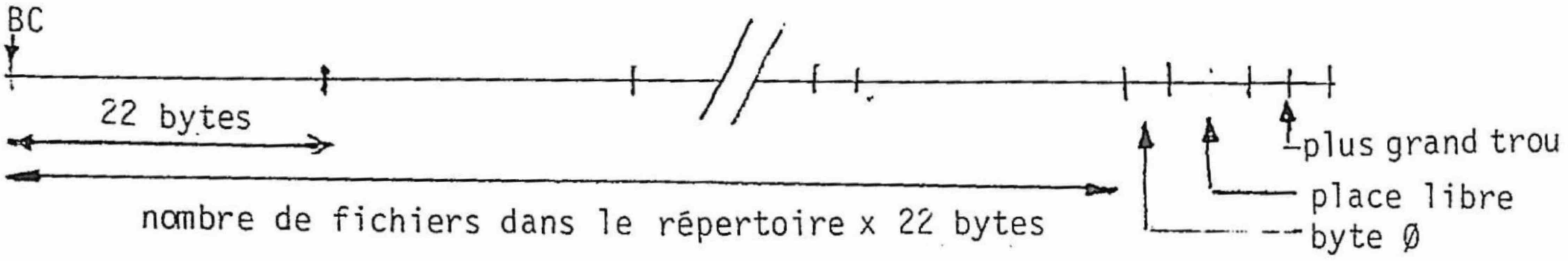

Etat du fichier: 1 caractère ASCII

Ø: fichier fermé
1 à n: nombre d'ouvertures en lecture
C: ouvert en écriture

Extension du nom: 2 caractères ASCII

Nom du fichier: 8 caractères ASCII

Format de la liste:




Place libre: en binaire la somme en blocs de tous les espaces vides sur le dans le répertoire.

Plus grand trou: taille en blocs du plus grand trou dans le répertoire.

Liste des erreurs possibles pour cet appel:

21 Unknown device
30 Device timeout
33 Read error

Remarque: la liste la plus longue possible (32 fichiers) nécessite 707. bytes.

## 8.4.16 APPEL 1 ?DELETE

Suppression d'un fichier dans un répertoire

Paramètre d'entrée DE: pointeur au nom

Liste des erreurs possibles pour cet appel:

1 Write protect file
10 File in use for writing
12 File does not exist
13 Illegal filename
20 File in use for reading
21 Channel error
30 Device timeout
32 Write error
33 Read error

REMARQUE: pour pouvoir être supprimé, le fichier doit être fermé et ne doit pas être protégé à l'écriture.
La disquette ne doit pas être protégée.

## 8.4.17 APPEL 2 ?RENAME

Changement du nom d'un fichier dans un répertoire

Paramètres d'entrée: DE: pointeur à l'ancien nom
BC: pointeur au nouveau nom

Liste des erreurs possibles pour cet appel:

| | |
|---|---|
| 1 | Write protect file |
| 10 | File in use for writing |
| 11 | File already exists |
| 12 | File does not exist |
| 13 | Illegal filename |
| 20 | File in use for reading |
| 21 | Unknown device |
| 30 | Device timeout |
| 31 | Write protect tab set |
| 32 | Write error |
| 33 | Read error |

Remarque: . La chaîne ASCII, pointée par BC, qui contient le nouveau nom, peut contenir un lien de répertoire.
Celui-ci est ignorée pour autant que la syntaxe soit correcte.

REMARQUE: pour pouvoir être renommé, le fichier doit être fermé et ne doit pas être protégé à l'écriture.
La disquette ne doit pas être protégée.

## 8.4.18 APPEL 14 ?COMPRE

Réunit tous les espaces vides d'un répertoire en un seul

Paramètres d'entrée: DE: pointeur au nom

A: nombre de blocs affectés comme buffers de conversion
A=Ø : compression avec les buffers internes de SAMOS.

BC: pointeur du buffer (si A≠Ø)

Liste des erreurs possibles pour cet appel:

| | |
|---|---|
| 21 | Unknown device |
| 23 | File(s) in use |
| 30 | Device timeout |
| 31 | Write protect tab set |
| 32 | Write error |
| 33 | Read error |

Remarques: . Une erreur disque durant COMPRE est fatale

. L'utilisation des buffers de SAMOS pour la compression permet d'éviter d'affecter la mémoire utilisateur; l'opération est cependant très lente. Si l'on spécifie un buffer, on a intérêt à ce qu'il soit le plus grand possible.

. Tous les fichiers du répertoire doivent être fermés

## 8.4.19 APPEL 13 ?LGO

Chargement et exécution d'un fichier sur un disque

Paramètre d'entrée: DE: pointeur au nom

Paramètre de sortie: DE: pointeur du prochain caractère significatif (voir remarques)

Liste des erreurs possibles pour cet appel

| | |
|---|---|
| 2 | Read protect file |
| 10 | File in use for writing |
| 12 | File does not exist |
| 13 | Illegal filename |
| 16 | Cannot load file |
| 20 | File in use for reading |
| 21 | Unknown device |
| 30 | Device timeout |
| 33 | Read error |
| 34 | No starting address |

Remarques:

. Durant l'exécution de cet appel, le stack est déplacé en SPUTIL. En cas d'erreur durant l'exécution de l'appel on distingue deux comportements:

1. Si aucun byte n'a encore été chargé ou s'il s'agit du message "no starting address", on revient de l'appel carry set avec dans A le numéro de l'erreur. Dans ce cas, le stack est remis à sa place initiale.
2. Si le chargement a commencé, on reçoit le message FATAL LOAD ERROR et l'on saute en RESYS de SYSMON, le stack étant mis à MAXMEM. Pour redémarrer dans le CLI, il suffit alors de taper BOOT ou, à la rigueur de faire RESET.

S'il n'y a pas d'erreur, le stack est mis à MAXMEM, les interruptions de la RTC, du timer de l'utilisateur sont désactivées, et le programme est exécuté.

. Cet appel permet un chargement dans n'importe quelle partie de la mémoire. Il n'y a donc pas de protection contre une destruction involontaire du software système.

. Prochain caractère significatif: si l'appel s'est effectué correctement,

on ne revient pas de l'appel et le programme est exécuté. DE pointe alors le premier caractère significatif:

- si l'on a spécifié après le nom du programme une série de "slash", DE pointera le premier "slash"
- sinon les séparateurs seront sautés et DE pointera le premier caractère ASCII rencontré ou le terminateur de ligne.

Exemples:

```
SORT/N/E ADDRESS.AD                SYNTAX ADDRESS.AD
    ↑                                     ↑
   DE                                    DE
```

### 8.4.20 APPEL 15 ?FORMAT

Mise à zéro et test d'écriture d'un répertoire

Paramètres d'entrée DE: pointeur au nom

Liste des erreurs possibles pour cet appel:

21 Unknown device
30 Device timeout
31 Write protect tab set
32 Write error

### 8.4.21 APPEL 5 ?RESET

Fermeture de tous les fichiers ouverts d'un répertoire

Pas de paramètres à spécifier

Remarques:

. Cet appel n'a pas de contrôle d'erreur.
Si une erreur survient durant la fermeture d'un canal, celui-ci est purement et simplement supprimé; cet appel ne doit en principe être utilisé que pour une fermeture d'urgence.

. Les paramètres DEBUT et START sont mis à zéro.

. Les fichiers ayant l'extension .MC sont protégés contre cet appel et restent normalement ouverts.

### 8.4.22 APPEL 12 ?CLR

Suppression des flags d'ouverture de tous les fichiers ouverts dans un répertoire

Paramètre d'entrée: DE: pointeur au nom

Liste des erreurs possibles pour cet appel:

21 Unknown device
30 Device timeout
31 Write protect tab set
32 Write error
33 Read error

Remarques: . Cet appel permet d'éliminer les flags d'ouverture dans le directoire d'un répertoire lorsque l'on a perdu les FIT (file information table)
Ceci est très utile après, par exemple, une panne de courant.

. Une erreur disque à l'écriture est fatale pour cet appel, puisqu'elle concerne l'écriture du directoire.

## 8.4.23 APPEL 20 ?CHATR

Modification des attributs d'un fichier dans un répertoire

Paramètres d'entrée: DE: pointeur au nom
A: attributs W et R

A: |x x x x x x R W|
indifférent — bit de protection à l'écriture
bit de protection à la lecture

état 1: actif état Ø: inactif

Liste des erreurs possibles pour cet appel:

| | |
|---|---|
| 4 | Permanent file |
| 10 | File in use for writing |
| 12 | File does not exist |
| 13 | Illegal filename |
| 20 | File in use for reading |
| 21 | Unknown device |
| 30 | Device timeout |
| 31 | Write protect tab set |
| 32 | Write error |
| 33 | Read error |

Remarques: . Une erreur disque à l'écriture est fatale pour cet appel puisqu'elle concerne l'écriture du directoire
. Le fichier doit être fermé et ne doit pas avoir les attributs protégés.
La disquette ne doit pas être protégée.

## 8.4.24 APPEL 21 ?CHATPT

Modification de la protection des attributs d'un fichier dans un répertoire

Paramètres d'entrée: DE: pointeur au nom
A: attribut de protection P

A: |x x x x x x|P P|
indifférent

=Ø attribut non protégé
≠Ø attribut protégé

Liste des erreurs possibles pour cet appel

| | |
|---|---|
| 10 | File in use for writing |
| 12 | File does not exist |
| 13 | Illegal filename |
| 20 | File in use for reading |
| 21 | Unknown device |
| 30 | DEvice timeout |
| 31 | Write protect tab set |
| 32 | Write error |
| 33 | Read error |

Remarques: une erreur disque à l'écriture est fatale pour cet appel puisqu'elle concerne l'écriture du directoire.

. Le fichier doit être fermé.
La disquette ne doit pas être protégée.

## 8.4.25 APPEL 22 ?ARGS

Renseignements généraux sur un répertoire et éventuellement en plus sur un fichier particulier de ce répertoire.

Paramètres d'entrée: DE: pointeur au nom
BC: pointeur en mémoire

Paramètres de sortie:

BC → 2 bytes: longueur du plus grand trou (en blocs)
1 byte: attributs du fichier

| V | x | x | O | C | P | R | W |

état 1: actif
état Ø: inactif

V: validation du byte. Si V=1, le byte est valide, donc le fichier spécifié dans la chaîne ASCII pointée par DE existe dans ce répertoire. Ce byte d'attribut du fichier ainsi que les suivants, qui se réfèreront au fichier, sont valides.

O: bit d'ouverture à la lecture.
Bit virtuel, non inscrit effectivement dans le directoire, actif si le compteur OPEN du directoire est différent de Ø.

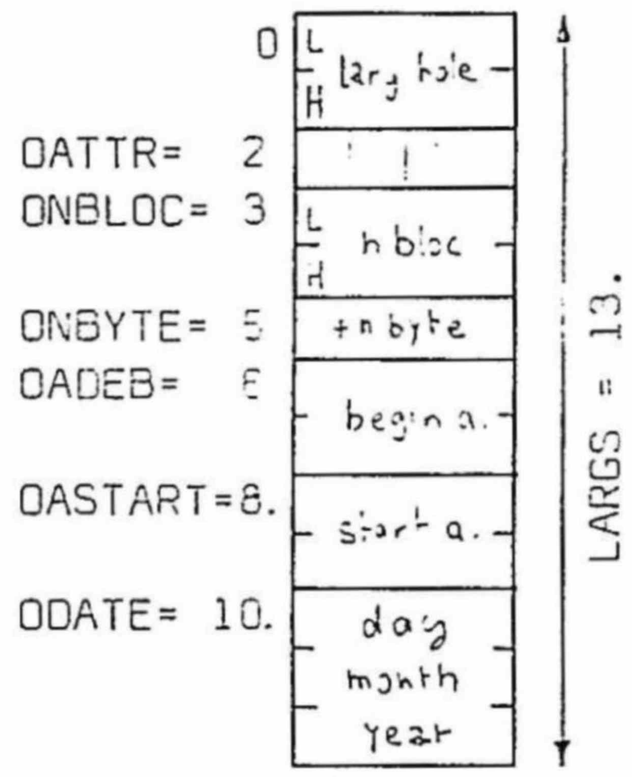
C: bit d'ouverture à l'écriture

P: bit de protection des attributs

R: bit de protection à la lecture } attributs

W: bit de protection à l'écriture }

2 bytes: taille du fichier en blocs
1 byte: nombre de bytes valides dans le dernier bloc
2 bytes: adresse de début
2 bytes: adresse de start
3 bytes: date de création




Fichier définition

Liste des erreurs possibles pour cet appel:

21 Unknown device
30 Device timeout
33 Read error

Remarque: La taille minimum du buffer, pointé par BC, où l'appel délivrera ses informations, doit être de $13_{10}$ bytes.

## 8.4.26 APPEL 17 ?RTN

Effectue un .W ?LGO de CLI.SY en DXØ.

Pas de paramètre d'entrée.

Remarque: On ne revient jamais de cet appel.
Si l'appel s'est effectué correctement, on est dans le programme CLI.SY sinon, on est dans le SYSTEME avec un des messages d'erreur de l'appel ?LGO.

## 8.4.27 APPEL 27 ?UPDATE

Lecture ou écriture de blocs physiques d'un fichier dans un répertoire avec accès aléatoire à l'intérieur du fichier.

Paramètres d'entrée:
- A: numéro de canal
- DE: pointeur en mémoire
- BC: nombre de bytes à lire ou écrire
- HL: numéro du bloc (de Ø à n)
- Carry=1: écriture
- Carry=Ø: lecture

Paramètres de sortie: BC: nombre de bytes lus ou écrits

. Sans message d'erreur (retour carry clear) on rend dans BC le nombre de bytes effectivement lus ou écrits (sauf si le nombre de bytes spécifié en entrée ne respectait pas la règle du multiple de $400_8$; on rend le multiple supérieur de $400_8$).

. En lecture avec le message "end of file" BC donne le nombre de bytes valides (pas forcément multiple de $400_8$).

. En écriture avec le message "file end overflow" BC donne un multiple de $400_8$ correspondant aux blocs effectivement écrits. ATTENTION: ne pas écrire dans la partie non significative du dernier bloc; cette condition n'est pas détectée par SAMOS car le message "file end overflow" est géré au niveau des blocs.

Liste des erreurs possibles pour cet appel:

| | |
|---|---|
| 6 | End of file |
| 7 | File end overflow |
| 17 | Out of file |
| 22 | Channel error |
| 30 | Device timeout |
| 32 | Write error |
| 33 | Read error |

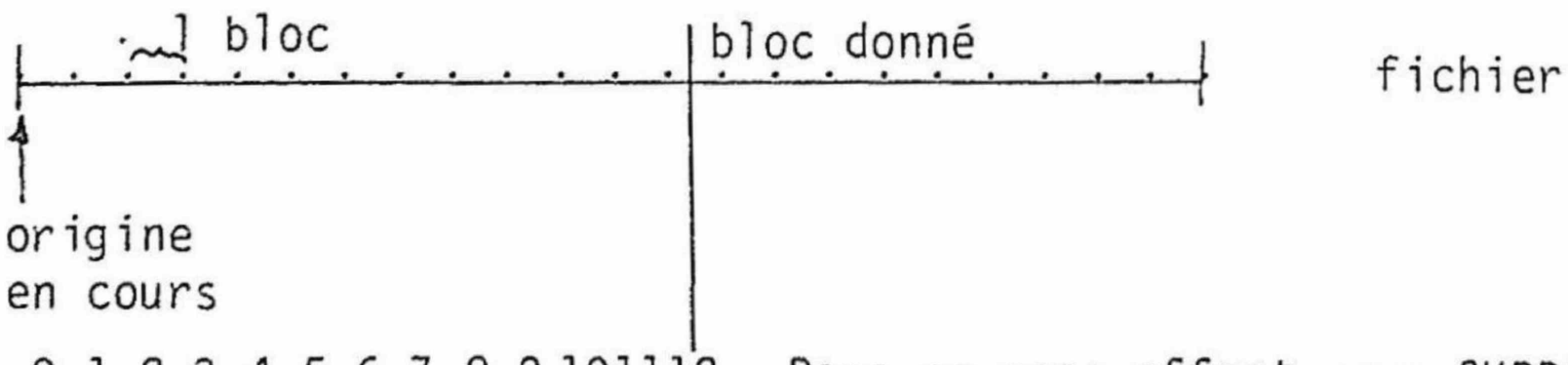
Remarques: . Pour accéder en UPDATE sur un fichier, il faut qu'il ait été ouvert en accès par blocs en lecture (OPEBLK).

On accède aléatoirement à chaque bloc physique en spécifiant dans HL l'offset, en nombre de blocs, par rapport à l'origine en cours et dans BC le nombre de bytes (multiple de $400_8$) à lire ou à écrire depuis là. Juste après l'ouverture, et tant que l'on n'a pas lu ce fichier avec l'appel ?RDBLOC, l'origine en cours coïncide avec le début du fichier.

Après une lecture de ce fichier par l'appel ?RDBLOC l'origine en cours est déplacée sur le bloc suivant le dernier bloc lu par l'accès ?RDBLOC. On ne pourra donc, avec l'appel ?UPDATE, plus accéder aux blocs lus à l'aide de ?RDBLOC.

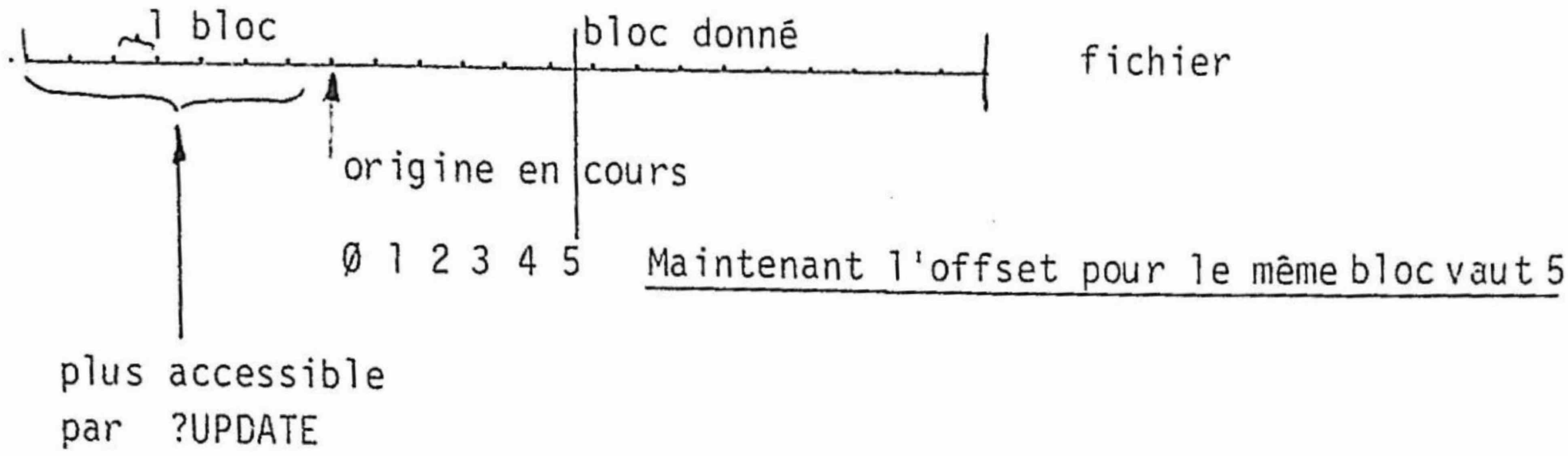
. Si on utilise successivement les appels ?RDBLOC et ?UPDATE, ne pas oublier que chaque appel ?RDBLOC change l'offset d'un bloc donné pour ?UPDATE.

Exemple:




Dans ce cas: offset pour ?UPDATE=12

Lecture de 6 blocs après ?RDBLOC donne:




## 8.4.28 APPEL 30 ?ERROR

Visualisation d'un numéro d'erreur

Paramètre d'entrée: A: No de l'erreur

Fonctionnement: Cherche en DXØ: s'il existe le fichier ER.SY
Si c'est le cas, cherche dans ce fichier s'il s'y trouve ce numéro d'erreur. Si c'est encore le cas, affiche à l'écran la phrase correspondante. Dans les cas contraires affiche à l'écran le message ERROR suivi du numéro de l'erreur que l'on voulait visualiser.

REMARQUE: le fichier ER.SY fait partie du système SAMOS.
Lors de son emploi par l'appel ?ERROR celui-ci est chargé dans un buffer interne. Ce fichier ne doit donc en aucun cas être modifié (exemple ajout de messages personnels!), en effet sa taille ne doit pas dépasser trois blocs.

## 8.4.29 APPEL 31 ?RDERO

Lecture de la somme des erreurs éventuelles durant un accès disque.

Paramètre de sortie: A: somme des erreurs

REMARQUE: SAMOS admet 10 tentatives avant de signaler une erreur de lecture ou d'écriture. Cet appel comptabilise la somme des tentatives supplémentaires.

Exemple: un accès disque lit les blocs 101 à 103 compris.
S'il a fallu pour lire le bloc:

101: 3 tentatives ⇒ 2 erreurs
102: 8 tentatives ⇒ 7 erreurs
103: 4 tentatives ⇒ 3 erreurs

Total et valeur de A:12 erreurs

## 8.4.30 APPEL 33 ?GNBLOC

Lecture du nombre de blocs du répertoire courant.

Paramètre de sortie: BC: nombre de blocs du répertoire courant.

### 8.4.31 Appel 36 ?LOAD

Cet appel permet le chargement direct d'un fichier à un emplacement donné et avec une longueur maximum donnée. Cet appel travaille avec le stack utilisateur. La recherche aura lieu dans le répertoire courant; si la recherche échoue, elle se poursuivra dans les répertoires DX0: et DX1:.

Lorsqu'il charge un fichier binaire (adresse de chargement donnée au CLOSE différente de zéro). il vérifie que l'adresse de chargement spécifiée est bien la même que celle du fichier. Si ce n'est pas le cas, il n'y a pas de chargement et un retour d'appel carry set avec l'erreur 16 (fichier non exécutable). BC vaut alors Ø. S'il s'agit d'un autre fichier (adresse de chargement égale à zéro), ce contrôle n'est pas effectué et le fichier se charge de toutes façons. Pour la longueur du chargement, l'appel ne dépassera jamais le multiple supérieur de 400 de la valeur spécifiée dans BC.

Exemple: BC = 1 400, BC = 400 400, BC = 401 1000.

Si le fichier est trop grand, l'appel revient carry set avec l'erreur 16 et BC donne la longueur effectivement chargée. Si le fichier est plus petit, le retour est carry clear et BC donne la longueur exacte du fichier chargé.

Paramètres d'entrée: DE = pointeur au nom
HL = adresse de chargement
BC = longueur maximale admise (multiple de 400 !!!)

Paramètres de sortie: BC = longueur valide effective
CS en cas d'erreur avec A = No de l'erreur

Modifiés: AF
BC

### 8.4.32 Appel 34 ?DIR

Cet appel permet de se mettre de manière fixe dans un répertoire.

Paramètre d'entrée: DE = pointeur au nom (lien du répertoire)

Paramètre de sortie: ----

Modifiés: AF

### 8.4.33 Appel 37 ?GDIR

Cet appel rend dans le buffer pointé par DE le nom du répertoire courant sans extension et terminé par un zéro. Le buffer doit avoir une longueur de 9 bytes.

Paramètres d'entrée: DE = pointeur au buffer

Paramètres de sortie: nom du répertoire courant dans le buffer

Modifiés: AF

### 8.4.34 APPEL 42 ?SHEAD

Cet appel, uniquement valable dans la version floppy doubles têtes, permet de sélectionner le mode de fonctionnement des drives DX0: et DX1:.

En cas d'utilisation de cet appel dans une configuration sans floppy double tête, on a un retour carry set avec l'erreur 110 (commande illégale).

Paramètres d'entrée: A = xabxxxx
a = drive DX1 b = drive DX0 bit set: 2 têtes

Paramètres de sortie: ---

Modifiés: AF

### 8.4.35 Appel 43 ?GHEAD

Cet appel, uniquement valable dans la version floppy doubles têtes, donne dans A le mode de fonctionnement courant des floppies.

En cas d'utilisation de cet appel dans une configuration sans floppy double têtes, on a un retour carry set avec l'erreur 110 (commande illégale).

Paramètres d'entrée: ---

Paramètres de sortie: A = xabxxxx
a = drive DX1 b = drive DX0 bit set: 2 têtes

Modifiés: AF

### 8.4.36 Appel 40 ?MODAY

Cet appel permet de modifier la date de création d'un fichier.

Paramètres d'entrée: DE = pointeur au nom
ABC = nouvelle date

Paramètres de sortie: ---

Modifiés: AF

### 8.4.37 Appel 41 ?GSAMOS

Cet appel rend dans BC la révision et la version ASCII, ainsi que les paramètres de la configuration utilisée dans A.

Paramètres d'entrée: ---

Paramètres de sortie: B = révision C = version
A = xxxabcde bit set = oui
a = winchester en DX2:
b = winchester en DX1:
c = winchester en DX0:
d = floppy double têtes
e = floppy simple tête

Modifiés: AF, BC

## 8.5 DESCRIPTION DES INDIRECTIONS SUR LES ROUTINES DE BASE

### 8.5.1 DESCRIPTION DES PARAMETRES

Les routines de base disque permettent de lire ou d'écrire n'importe où sur le disque. Il faut cependant spécifier les adresses disque désirées. Ces adresses disque se donnent en numéro de bloc dans un registre 16 bits.

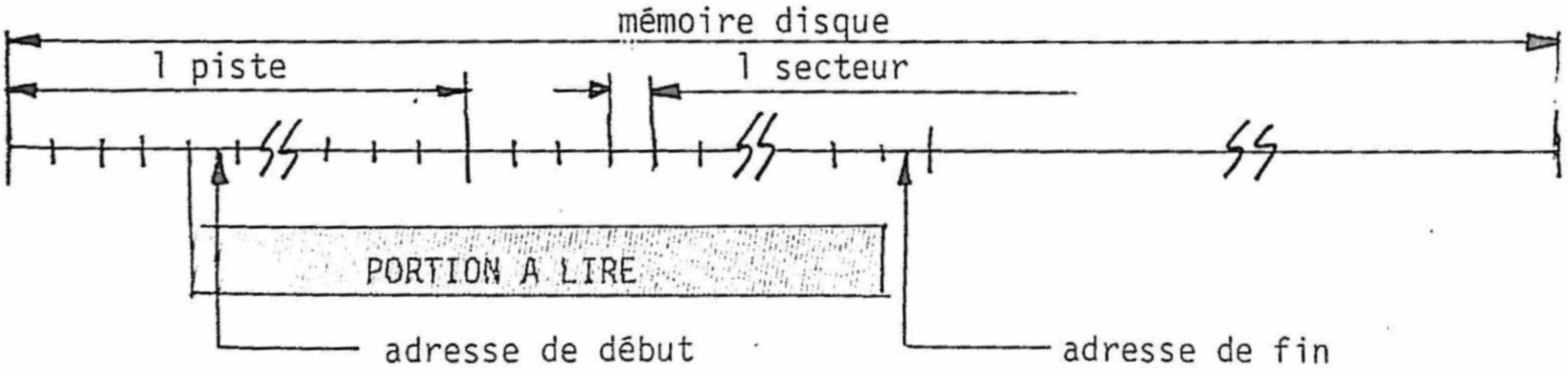
On utilisera, pour faire fonctionner ces routines, également les paramètres pointeur en mémoire et pointeur au nom. Ces paramètres sont identiques à ceux utilisés pour les appels SAMOS (se référer donc à leur description).

REMARQUE IMPORTANTE: des problèmes hardware font que le bon fonctionnement des disques n'est garanti que si l'on n'est pas en mode graphique (dans ce mode, l'augmentation de durée de l'interruption DMA display fait que le processeur n'arrive plus à suivre le contrôleur de disques). Si les appels SAMOS se préoccupent de changer le mode du display et de le restituer en sortie, les routines de base ne le font pas. C'est donc à l'utilisateur de ces routines de se préoccuper de ce problème.

### 8.5.2 RODWIB (Read On Disk Write In Buffer)

Adresse de l'indirection: 10003

Cette routine, comme son nom l'indique, permet de lire une portion du disque et d'écrire son contenu en mémoire. Il faut donc préciser les limites de la portion à lire sur le disque. Ceci est donné par un numéro de bloc de départ et un No de bloc de fin qui est l'adresse du 1er secteur n'appartenant plus à la portion à lire (en d'autres mots, l'adresse du dernier secteur + 1).




Cette information qui est lue sur le disque doit naturellement être écrite dans une zone mémoire spécifiée. C'est le paramètre pointeur en mémoire qui fournit cette information. Il donne l'origine du buffer qui va recevoir l'information. Tout comme un appel SAMOS, une erreur peut intervenir. La routine se comporte de manière similaire aux appels, on reviendra Carry set et avec le numéro d'erreur dans le registre A.

Cette routine ne se suffit pas à elle-même. En effet, il faut pouvoir spécifier préalablement l'adresse du répertoire sur lequel nous allons faire ce travail. Nous pourrons le faire grâce à l'indirection sur la routine TSTDRI décrite plus loin. D'autre part, cette routine ne désélectionne pas le drive. Il faudra le faire avec l'indirection sur la routine STOP , également décrite ci-après. C'est uniquement dans le cas d'un retour avec erreur que le drive est automatiquement désélectionné.

CALL RODWIB

RODWIB = 10003

in: BC = numéro du bloc de départ
HL = numéro du bloc de fin
DE = pointeur en mémoire

out: Carry set si erreur, avec dans A le No de l'erreur

mod: A, BC, DE, HL, F

### 8.5.3 RIBWOD (Read In Buffer, Write On Disk)

Adresse de l'indirection: 10006

Cette routine, comme son nom l'indique, permet d'écrire sur une portion de disque préalablement lue en mémoire. Elle fonctionne exactement à l'inverse de RODWIB qui vient d'être décrit. Se reporter donc au paragraphe précédent pour la compréhension du fonctionnement de RIBWOD.

CALL RIBWOD

RIBWOD = 10006

in: BC = numéro du bloc de départ
HL = numéro du bloc de fin
DE = pointeur en mémoire

out: Carry set si erreur avec dans A le No de l'erreur

mod: A, BC, DE, HL, F.

### 8.5.4 TSTDRI (Test of drive and repertory address)

Adresse de l'indirection: 10011

Cette routine reconnaît dans la chaîne ASCII du pointeur au nom le lien du répertoire spécifié. Les règles de syntaxe décrites pour le pointeur au nom des appels SAMOS sont valables. On utilise cette routine pour spécifier le répertoire que l'on désire accéder. Il n'est pas nécessaire d'appeler cette routine avant chaque accès, mais avant le premïer et par la suite à chaque fois que l'on change de répertoire.

CALL TSTDRI

TSTDRI = 10011

in: DE = pointeur au nom

mod: A, HL, DE, F

### 8.5.5 STOP (disable drive selection)

Adresse de l'indirection: 10000

Cette routine permet de désélectionner un drive qui l'a été par une des routines RODWIB ou RIBWOD. Elle n'est nécessaire qu'en fin de travail. En effet un certain travail peut être composé d'une succession d'accès disque. Il est alors agréable de ne pas désélectionner le drive entre chaque accès. La routine STOP n'est même pas nécessaire si l'on change de répertoire. En effet c'est la routine TSTDRI qui assurera la transition d'un répertoire à l'autre. Il faut remarquer que dès que l'on a exécuté un accès disque (RODWIB ou RIBWOD) l'interruption 50 Hz est inhibée, le clavier n'est donc plus accessible.

On peut naturellement réactiver l'interruption 50 Hz entre deux accès disque à l'aide de l'appel SYSTEME ?ENI5Ø . Il n'est pas nécessaire de le désactiver à nouveau avant le prochain accès disque.

CALL STOP

STOP = 10000

aucun paramètre, n'affecte rien.

## 8.5.6 INIFLO (Init SMAKY6 in Floppy mode)

Adresse de l'indirection: 10017

Cette routine effectue les travaux suivants:

- Affiche sur l'écran SAMOS rév./ version
- Initialise les restarts floppy (restart 10 et 20)
- Initialisation des FIT (file information table)
- Tracking des disques (initialisation en piste Ø) avec reconnaissance des drives où le tracking s'est correctement effectué
- Affichage des adresses de drive non reconnues.

Attention: cette routine ne modifie pas MAXMEM, et, évidemment, n'initialise pas le stack.

CALL INIFLO

INIFLO = 10017

pas de paramètre

mod: A, HL, BC, F

### 8.5.7 INDIRECTION SUR RTN

Adresse de l'indirection: 10025

Effectue un ?LGO de CLI.SY en DXØ: (même effet que l'appel ?RTN)

### 8.5.8 INDIRECTION SUR BOOT

Adresse de la routine: 10022

Exécution du programme de bootstrap

### 8.5.9 NMIRTN

Adresse: 10030

Cette adresse est prévue pour l'initialisation de la touche NMI.
Un retour par cette adresse ferme tout d'abord le fichier MACRO, s'il en existe un d'ouvert, puis effectue un RTN.

### 8.5.10 ROUTINE ADBLK

Adresse de la routine: 10014

Cette routine effectue une addition de blocs entre HL et DE, le résultat étant dans HL. Si le résultat de l'addition dépasse la capacité maximum du répertoire, le retour se fait carry set avec A=erreur 26 (disk full) et HL= le numéro du dernier bloc admissible (soit l'adresse du premier secteur hors capacité).

Cette adresse est relative, le premier bloc d'un répertoire a toujours le numéro zéro. Pour connaître l'adresse absolue, il faut additionner l'adresse absolue du premier bloc du répertoire.

CALL ADBLK

ADBLK = 10014

in: HL et DE: éléments de l'addition
out: A = erreur 26 et carry set si dépassement
HL = résultat addition ou valeur maximum admissible si dépassement
mod: A, DE, HL, F

# 8.6. EXEMPLES D'UTILISATION DES APPELS SAMOS

## 8-6-1 INTRODUCTION

Nous venons de passer en revue tous les appels floppy SAMOS. Vous connaissez donc les définitions de ces outils, mais cela ne veut pas dire que vous sachiez vous en servir. Voici quelques exemples fondamentaux qui devraient vous faciliter la tache dans vos programmes utilisant des fichiers disque ou périphérique.

## 8-6-2 UN PREMIER EXEMPLE SIMPLE ET COMMENTE

Dans cet exemple nous allons tout simplement lire un fichier source sur disque et le transférer sur l'écran grâce au fichier périphérique écran $DIS. Nous ferons ceci 10 bytes à 10 bytes.

Supposons que notre fichier disque s'appelle TOTO.SR, et qu'il se trouve sur la disquette inserée dans le drive DX1:.

Pour pouvoir lire ce fichier il faut tout d'abord l'ouvir. Nous avons besoin pour ce faire des informations suivantes: le nom du fichier et l'adresse du drive où se trouve la disquette contenant notre fichier. Il faut donc définir une chaîne ascii contenant ces deux informations.

```
INFILE: .ASCIZ  /DX1:TOTO.SR/
```

L'étiquette INFILE est l'adresse du premier caractère de cette chaîne ASCII.

Nous pouvons maintenant ouvir notre fichier. Comme nous voulons lire notre fichier par 10 bytes, nous utilisons l'accès par bytes et nous écrivons:

```
        LOAD    DE,#INFILE      ;DE = pointeur au nom
        .W      ?OPEN           ;ouverture en accès par bytes
```

Nous avons à ce stade un problème. Nous savons que nous pouvons revenir d'un appel SAMOS avec une erreur, il est donc nécessaire de prévoir cette éventualité. Comme en cas d'erreur nous avons CARRY SET en sortie d'appel nous écrivons:

```
        JUMP,CS ERROR
```

ERROR étant une portion de notre programme où nous traiterons les erreurs d'entée-sortie disque.

Si notre appel SAMOS d'ouverture de fichier s'est effectué correctement, nous savons que SAMOS nous rend un numéro de CANAL qui sera utilisé pour la suite des transactions. Nous allons donc sauver ce numéro de canal par exemple dans une position mémoire. Nous écrivons donc:

```
        LOAD    INCH,A
```

A ce stade, nous sommes capables de commencer à LIRE notre fichier TOTO.SR. Cependant n'oublions pas que nous voulons ECRIRE les bytes lus sur l'écran. Il faut donc également ouvrir EN ECRITURE le fichier périphérique $DIS.

Pour ouvrir un fichier périphérique en écriture nous n'avons besoin que du nom. Nous définissons donc une autre chaine ascii et nous écrivons:

```
OUTFIL: .ASCIZ  /$DIS/
```

Nous pouvons maintenant ouvir notre fichier et nous écrivons:

```
        LOAD    DE,#OUTFIL
        .W      ?CREATE
```

Comme pour TOTO.SR nous prévoyons une éventuelle erreur:

```
        JUMP,CS ERROR
```

Et nous sauvons également le numéro de canal:

```
        LOAD    OUTCH,A
```

Cette fois, nous somme prêt à effectuer nos transferts de bytes du fichier disque sur le péripherique $DIS.

Nous lisons tout d'abord 10 bytes de TOTO.SR. Pour ce faire, nous avons besoin d'un BUFFER de 10 bytes où l'appel de lecture délivrera son information. Définissons-le:

```
LONGDATA=10.
BUFFER: .BLKB   LONGDATA
```

Lisons maintenant 10 bytes de notre fichier TOTO.SR. Pour ce faire nous initialisons le POINTEUR EN MEMOIRE *DE* au début de notre buffer. Dans *A* nous mettons le numéro de CANAL et dans *BC* le NOMBRE DE BYTES à lire. Ceci donne:

```
        LOAD    DE,#BUFFER
        LOAD    BC,#LONGDATA
DIFI1:
        LOAD    A,INCH
        .W      ?RDBYTE
```

Là, un nouveau problème ce pose en cas d'erreur. Nous ne pouvons pas simplement sauter à ERROR, car nous devons FILTRER l'ERREUR *END OF FILE* qui nous apprend que nous avons lu tous les bytes de TOTO.SR. EREOF étant le numéro de cette erreur, nous pouvons par exemple traiter le problème ainsi:

```
        JUMP,CC DIFI2           ;pas d'erreur = pas de problème
        COMP    A,#EREOF        ;test si END OF FILE
        JUMP,NE ERROR           ;si non, va à ERROR
```

Comme nous lisons par tranche de 10 bytes, il est fort possible que nous ayons tout de même lu quelques bytes. Nous ne pouvons donc pas simplement terminer ici le programme. Il faut VERIFIER QUE LE NOMBRE DE BYTES LUS RENDU DANS *BC* SOIT NUL. Nous écrivons alors:

```
        LOAD    A,B             ;teste si le registre
        OR      A,C             ;BC est nul
        JUMP,EQ FIN             ;si c'est le cas, c'est fini
```

Dans le cas contraire, ou si nous n'avons pas eu d'erreur, il faut maintenant transférer les 10 bytes lus sur l'écran. Comme LE POINTEUR EN MEMOIRE *DE* N'EST PAS MODIFIE nous pointons le début de notre information lue. Comme d'autre part BC = NOMBRE DE BYTES EFFECTIVEMENT LUS, il suffit d'écrire:

```
DIFI2:
        LOAD    A,OUTCH         ;A = numéro de canal de $DIS
        .W      ?WRBYTE
        JUMP,CS ERROR
```

Il suffit maintenant de fermer notre boucle de tranfert, pour exécuter le programme jusqu'au transfert intégral de TOTO.SR.

```
        JUMP    DIFI1
```

Pour que notre programme soit complet nous devons encore voir ce que nous devons faire une fois le transfert terminé. Il faut tout naturellement FERMER LES FICHIERS UTILISES. Pour cela nous pouvons utiliser soit deux fois l'appel CLOSE pour chaque canal ou alors l'appel RESET qui ferme tous les fichiers ouvert. Utilisons donc simplement RESET:

```
FIN:
        .W      ?RESET
        .W      ?RTN
```

Nous avons finalement utilisé l'appel RTN qui termine le programme et revient au CLI.

Il ne reste plus qu'à traiter les cas des erreurs. On utilise généralement l'appel ERROR qui permet la visualisation de l'erreur. On peut par exemple traiter le cas de la manière suivante:

```
ERROR:
        .W      ?RETURN         ;va à la ligne
        .W      ?ERROR          ;visualise l'erreur
        .W      ?RESET          ;ferme tous les fichiers
        .W      ?RTN            ;et retourne au CLI
```

Si l'on ajoute au début une initialisation de l'écran voici finalement notre petit programme DISFILE (display file).

```
        .TITLE  DISFILE PROGRAM
        .PROC   Z80
        .REF    FLO

        LONGDATA=10.

        .LOC    53000

; initialisation du programme:

DISFIL:
        LOAD    C,#LINES        ; initialise une fenêtre
        .W      ?IDIS           ; sur tout l'écran
        LOAD    DE,#INFILE      ; DE = pt au nom fichier entrée
        .W      ?OPEN           ; ouverture en accès par bytes
        JUMP,CS ERROR           ; avorte en cas d'erreur
        LOAD    INCH,A          ; sauve le canal d'entrée
        LOAD    DE,#OUTFIL      ; DE = pt au nom fichier sortie
        .W      ?CREATE         ; création en accès par bytes
        JUMP,CS ERROR           ; avorte en cas d'erreur
        LOAD    OUTCH,A         ; sauve le canal de sortie
        LOAD    DE,#BUFFER      ; DE = pointer le buffer

; Boucle du transfert

DIFI1:  LOAD    BC,#LONGDATA    ; BC = nombre de bytes à opérer
        LOAD    A,INCH          ; A = canal d'entrée
        .W      ?RDBYTE         ; lecture des bytes
        JUMP,CC DIFI2           ; saute si pas d'erreur
        COMP    A,#EREOF        ; test si END OF FILE
        JUMP,NE ERROR           ; si non, va à ERROR
        LOAD    A,B             ; teste si le registre
        OR      A,C             ; BC est nul
        JUMP,EQ FIN             ; si c'est le cas, c'est fini
```

```
DIFI2:

        LOAD    A,OUTCH         ; A = canal de sortie
        .W      ?WRBYTE         ; écriture des bytes
        JUMP,CS ERROR           ; avorte en cas d'erreur
        JUMP    DIFI1           ; continue le transfert

; Fin du programme

FIN:
        .W      ?RESET          ; ferme tous les fichiers
        .W      ?RTN            ; retourne au CLI

; Fin du programme en cas d'erreur

ERROR:
        .W      ?RETURN         ; va à la ligne
        .W      ?ERROR          ; visualise l'erreur
        .W      ?RESET          ; ferme tous les fichiers
        .W      ?RTN            ; et retourne au CLI

; Nom des fichiers, paramètres en RAM et buffer.

INFILE: .ASCIZ  /DX1:TOTO.SR/   ; nom fichier d'entrée
OUTFIL: .ASCIZ  /$DIS/          ; nom fichier sortie
INCH:   .BLKB   1               ; mémoire canal d'entrée
OUTCH:  .BLKB   1               ; mémoire canal de sortie
BUFFER: .BLKB   LONGDATA                ; buffer de transfert


        .END    DISFIL
```

8-6-3 APPRENONS PLUS EN AMELIORANT CE PROGRAMME

Le petit programme que nous venons de décrire a l'avantage d'être très simple pour bien faire comprendre le fonctionnement fondamental des principaux appels SAMOS mais n'est évidemment pas très évolué. Voici ce que nous allons améliorer:

Spécification dans la ligne de commande d'appel du programme des noms du fichier en entrée et du fichier en sortie.

Comme nous allons pouvoir spécifier le fichier en sortie, nous transférerons également les paramètres du fichier: ADRESSE DE DEBUT, ADRESSE DE START pour le cas d'un transfert sur un NOUVEAU FICHIER DISQUE. Nous utiliserons l'ACCES PAR BLOCS et un grand buffer pour être plus rapide.

On voit que ces modifications vont faire de notre programme un PROGRAMME DE TRANFERT DE FICHIER GENERAL, qui peut notemment faire exactement ce que faisait DISFILE mais aussi tous autres types de transferts. Nous l'appellerons donc XFER et pour transférer par exemple le fichier TOTO.TX dans le fichier TITI.SR, il faudra taper XFER TOTO.TX TITI.SR.

```
        .TITLE  XFER PROGRAM
        .PROC   Z80
        .REF    FLO


        .LOC    53000

ERILC   =       110             ; No message d'erreur
                                ; commande illégale
```

```
        ; initialisation du programme:

        ; ouverture du fichier en entrée
        ; après le LGO de notre programme XFER
        ; DE pointe l'argument suivant de la ligne de commande

XFER:
        LOAD    A,(DE)          ; lit un car de la ligne de commande
        OR      A,A             ; test si fin de la ligne
        JUMP,EQ ILLCOM          ; saute à commande illégale si oui
        .W      ?OPEBLK         ; ouverture en accès par blocs
        JUMP,CS ERROR           ; avorte en cas d'erreur
        LOAD    INCH,A          ; sauve le canal d'entrée

        ; lecture des paramètres du fichier d'entrée

        LOAD    BC,#ARGBUF      ; pointe buffer des paramètres
        .W      ?ARGS           ; lit les paramètres

        ; pointeur de la ligne de commande
        ; sur l'argument suivant

XFER0:
        LOAD    A,(DE)          ; lit un car de la ligne de commande
        INC     DE              ; incrémente le pt de la ligne
        OR      A,A             ; test si fin ligne de commande
        JUMP,EQ ILLCOM          ; saute à commande illégale si oui
        COMP    A,#SPACE        ; test si séparateur
        JUMP,EQ XFER1           ; saute si oui
        COMP    A,#TAB          ; test si séparateur
        JUMP,NE XFER0           ; si non => caract. suivant
XFER1:
        ; ouverture du fichier en sortie

        LOAD    BC,#0           ; réservation place par défaut
        .W      ?CREBLK         ; création en accès par blocs
        JUMP,CS ERROR           ; avorte en cas d'erreur
        LOAD    OUTCH,A         ; sauve le canal de sortie
        LOAD    SVOUT,DE        ; sauve pt nom fichier sortie

        ; calcul de la taille du buffer
        ; soit le nombre de bytes maximum que
        ; l'on pourra traiter par tranche du transfert

        .W      ?MEM            ; lit la fin de mémoire
        LOAD    SP,HL           ; met le stack à cet endroit
        DEC     H               ; de la place pour le stack
        LOAD    DE,#BUFFER      ; DE = début du buffer
        OR      A,A             ; clear le carry
        SUBC    HL,DE           ; calcul la taille
        LOAD    B,H             ; BC = nombre maximum de
        LOAD    C,#0            ; bytes MULTIPLE DE 400

        ; Boucle du transfert

XFER2:
        LOAD    A,INCH          ; A = canal d'entrée
        .W      ?RDBLK          ; lecture des blocs
        JUMP,CC XFER3           ; saute si pas d'erreur
        COMP    A,#EREOF        ; test si END OF FILE
        JUMP,NE ERROR           ; si non, va à ERROR
        LOAD    A,B             ; teste si le registre
        OR      A,C             ; BC est nul
        JUMP,EQ FIN             ; si c'est le cas, c'est fini
```

```
XFER3:
        LOAD    A,OUTCH         ; A = canal de sortie
        .W      ?WRBLK          ; écriture des blocs
        JUMP,CS ERROR           ; avorte en cas d'erreur
        JUMP    XFER2           ; continue le transfert

; Fin du programme

        ; fermeture du fichier de sortie avec
        ; les paramètres préalablement lus et sauvés
        ; du fichier en entrée

FIN:
        LOAD    A,OUTCH         ; A = canal de sortie
        LOAD    BC,DEBUT        ; BC = adr de début
        LOAD    DE,START        ; DE = adr de start
        .W      ?CLOSE          ; fermeture fichier sortie
        LOAD    A,INCH
        .W      ?CLOSE          ; fermeture fichier entrée
        .W      ?RTN            ; retourne au CLI

; Fin du programme en cas d'erreur

ILLCOM:
        LOAD    A,#ERILC        ; no erreur commande illégale

ERROR:
        .W      ?ERROR          ; visualise l'erreur
        .W      ?RESET          ; ferme tous les fichiers
        LOAD    DE,SVOUT        ; pointeur nom fichier sortie
        .W      ?DELETE         ; supprime si déjà crée
        .W      ?RTN            ; et retourne au CLI

; paramètres en RAM et buffers.

INCH:   .BLKB   1               ; mémoire canal d'entrée
OUTCH:  .BLKB   1               ; mémoire canal de sortie
SVOUT:  .BLKW   1               ; mémoire pt nom fichier sortie
ARGBUF: .BLKB   6               ; buffer pour appel ARGS
DEBUT:  .BLKW   1               ; paramètre adr début
START:  .BLKW   1               ; paramètre adr de start
BUFFER:                         ; début du buffer de transfert

        .END    XFER
```

Avec ce nouveau programme nous voyons tout d'abord que le passage des arguments de l'ordre par la ligne de commande est relativement facile. Nous voyons également une utilisation de l'appel ARGS qui rend possible le transfert des paramètres du fichier d'entrée. On voit que dans ce cas, on ne peut pas utiliser l'appel RESET pour fermer le fichier de sortie puisqu'il faut spécifier les paramètres de début et de start. Il faut remarquer que pour la création du fichier de sortie, nous avons pris soin de spécifier le paramètre de réservation de la place dans le registre BC, qui pouvait être omis dans DISFILE puisque le fichier de sortie était un péripherique. Finalement on voit une utilisation de l'appel DELETE, qui nous permet de supprimer le fichier de sortie en cas d'erreur et s'il a déja été créé. Remarquons que pour faire ceci, il a fallu sauver le pointeur au nom du fichier et prendre garde d'exécuter l'appel DELETE apres la fermeture du fichier.

## 8-6-4 QUELQUES ROUTINES EXTRAITES DE PROGRAMMES

Vous trouverez ci-après des routines d'entrée-sortie disque extraite de programmes qui montrent comment on peut utiliser avantageusement l'accès par bloc tout en s'affranchissant des contraintes dues à ce type d'accès.

### 8-6-4-1 ROUTINES DE LECTURE ET D'ECRITURE BYTE A BYTE

Voici tout d'abord les paramètres utilisés par ces routines:

```
INCH:      .BLKB     1                    ; input channel
OUTCH:     .BLKB     1                    ; output channel
CNTIN:     .BLKW     1                    ; input bytes counter
CNTOUT:    .BLKW     1                    ; output bytes counter
PTIN:      .BLKW     1                    ; input buffer current pointer
PTOUT:     .BLKW     1                    ; output buffer current pointer
BUFIN:     .BLKW     1                    ; begin adress of input buffer
BUFOUT:    .BLKW     1                    ; begin adress of output buffer
LGHBUF:    .BLKW     1                    ; lengh of one I/O buffer
```

Au départ il faut que les buffers soient définis, il faut donc initialiser les paramètres BUFIN, BUFOUT, LGHBUF. Ces paramètres peuvent évidemment être donnés comme valeurs immédiates dans les routines, si la taille et la position des buffers sont invariables. Il faut également mettre les compteurs CNTIN et CNTOUT à zéro, et initialiser le pointeur courant du buffer de sortie PTOUT au début du buffer. Les fichiers sont également ouverts avec les numéros de canal dans INCH et OUTCH.

ROUTINE DE LECTURE

```
;----------.
;  RDCAR    >
;==========/

;This routine reads a character from the input buffer.
;When the buffer is empty, it fills itself with blocs from
;input file. When the file is empty, there is no return
;from the routine but a short-cicuit branch to ENDJOB.
;When a I/O disk error occurs we have a short-circuit
;branch onto ERROR.

;in:     -
;out:    A = next character from input file
;mod:    A,F

RDCAR:
        PUSH    DE                  ; save on stack
        PUSH    BC
        LOAD    DE,PTIN             ; init DE with buffer pointer
        LOAD    BC,CNTIN            ; init BC with char counter
        LOAD    A,B                 ; test if buffer
        OR      A,C                 ; is empty
        JUMP.,EQ RDCAR1             ; if yes read next file blocs
RDCAR0:
        DEC     BC                  ; else decrement char counter
        LOAD    CNTIN,BC            ; save this new value
        LOAD    A,(DE)              ; read the char from buffer
        INC     DE                  ; increment pointer
        LOAD    PTIN,DE             ; save this new pointer
        POP     BC                  ; restore from stack
        POP     DE
        RET                         ; and return
```

```
RDCAR1:
        LOAD    DE,BUFIN        ; init DE with buffer begin
        LOAD    BC,LGHBUF       ; init BC with buffer length
        LOAD    A,INCH          ; init A with input channel
        .W      ?RDBLK          ; read blocs from file
        JUMP.,CC RDCAR0         ; test if I/O error occurs
        COMP    A,#EREOF        ; if yes, test if end of file
        JUMP,NE ERROR           ; if not, branch to ERROR
        LOAD    A,B             ; else test if 0 char
        OR      A,C
        JUMP,EQ ENDJOB          ; if yes, branch to ENDJOB
        JUMP    RDCAR0          ; else read in buffer
```

ROUTINE D'ECRITURE

```
;---------\
;  WRCAR   >
;=========/

;This routine writes a character into the output buffer.
;When the buffer is full, it writes the buffer into output
;file.
;When a I/O disk error occurs we have a short-circuit
;branch onto ERROR.

;in:    A = character
;out:   -
;mod:   F

WRCAR:
        PUSH    DE              ; save on stack
        PUSH    HL
        LOAD    DE,LGHBUF       ; init DE with buffer length
        LOAD    HL,CNTOUT       ; init HL with char counter
        .W      ?COMPHLDE       ; test if buffer is full
        JUMP.,EQ WRCAR1         ; if yes, write buffer to file
WRCAR0:
        INC     HL              ; else increment char counter
        LOAD    CNTOUT,HL       ; save new counter value
        LOAD    HL,PTOUT        ; init HL with buffer pointer
        LOAD    (HL),A          ; write char into the buffer
        INC     HL              ; increment pointer
        LOAD    PTOUT,HL        ; save new pointer value
        POP     HL
        POP     DE              ; restore from stack
        RET                     ; and return
WRCAR1:
        PUSH    BC              ; save on stack
        LOAD    B,D             ; init BC with buffer length
        LOAD    C,E
        LOAD    DE,BUFOUT       ; init DE with buffer begin
        LOAD    H,A             ; save char in H
        LOAD    A,OUTCH         ; init A with output channel
        .W      ?WRBLK          ; write buffer into file
        JUMP,CS ERROR           ; branch to ERROR if I/O error
        LOAD    A,H             ; else restore char in A
        LOAD    PTOUT,DE        ; save new buffer pointer
        LOAD    HL,#0           ; clear char counter
        POP     BC              ; restore from stack
        JUMP    WRCAR0          ; write char into buffer
```

REMARQUE IMPORTANTE:
Il ne faut pas oublier en fin de programme que le buffer de sortie contient certainement des informations non encore transferées dans le fichier de sortie. Il y a donc lieu de vider ce buffer de la manière suivante:

```
        LOAD    DE,BUFOUT       ; write to output file
        LOAD    BC,CNTOUT       ; the contains of output
        LOAD    A,OUTCH         ; buffer
        .W      ?WRBLK
        JUMP,CS ERROR
```

8-6-4-2 BUFFER DE LECTURE POUR LA GESTION D'ENREGISTREMENT

Il est parfois nécessaire de pouvoir traiter une succession d'informations de longueurs égales ou différentes entre elles, mais certainement presque jamais égales à la longueur d'un bloc ou a un multiple supérieur (par ex: une ligne, un enregistrement). Ainsi, avec l'accès par bloc il est pratiquement certain que le buffer contiendra a la fin UNE INFORMATION TRONQUEE.
Il sera donc nécessaire de ne pas considérer cette partie du buffer durant le traitement de l'information et de la reporter au début du buffer avant de lire la suite du fichier.
La routine ci-après réalise ce travail.

Voici tout d'abord les paramètres utilisés par la routine:

```
SIZE:    .BLKW   1              ; size of input buffer
SAVCHA:  .BLKB   1              ; temporary channel save
FLGEOF:  .BLKB   1              ; file EOF status
```

Seul le paramètre SIZE doit être préalablement initialisé. Au début la longueur utile DE est évidemment égale a SIZE. Par la suite c'est la valeur rendue par la routine elle même. Le paramètre *soft end* BC, est généralement déterminé par le traitement de l'information. Il pointe la première position non significative.

```
;----------\
;  FILBUF  >
;==========/

; This routine fills the buffer pointed by HL of usefull length
; DE with software end. That means shift unused part at beginning
; and complete with maximum possible number of blocs ; from channel A file.
; Return new usefull length. If end of file occurs  A = zero.
; If end of file and buffer empty occur return CS and file is closed.

; BUFFER DEFINITION

;  |--used-part-of-buffer--|--unused-part--|  empty part  |
;   ^                       ^               |              ^
;  HL = begin               BC = soft end   |          physical end
;  <----------DE = usefull length----------->|          of buffer

; in:   HL = begin of buffer pointer
;       DE = usefull length of buffer
;       BC = software buffer end pointer
;       A  = channel
; out:  DE = new usefull length of buffer
;       CS et DE = 0 if end of file (mean buffer empty)
;       A  = disk end of file status (null if EOF occur)
; mod:  AF,BC,DE
```

```
FILBUF:
        LOAD    SAVCHA,A        ; save channel
        LOAD    FLGEOF,A        ; set no EOF flag
        PUSH    HL              ; save twice
        PUSH    HL              ; begin of buffer
        ADD     HL,DE           ; compute buffer end
        EX      HL,DE           ; DE = buffer end
        LOAD    H,B             ; HL = software buffer end
        LOAD    L,C
        .W      ?COMPHLDE       ; test if still valid bytes
        JUMP,HS FILBU4          ; if not, jump
        PUSH    HL              ; else save soft buffer end
        EX      HL,DE           ; HL = end DE = soft end
        OR      A,A             ; clear carry
        SUBC    HL,DE           ; compute offset
        LOAD    B,H             ; BC = offset
        LOAD    C,L
        POP     HL              ; restore soft end as from
        POP     DE              ; restore beg of buffer as to
        PUSH    BC              ; save offset
        LDIR                    ; move offset to beg of buffer
FILBU0:
        LOAD    HL,SIZE         ; HL = full size of the buffer
        POP     BC              ; restore offset
        OR      A,A             ; clear carry
        SUBC    HL,BC           ; compute free space in buffer
        LOAD    L,#0            ; in bloc multiple
        PUSH    BC              ; save offset
        LOAD    B,H             ; BC = free space in bloc
        LOAD    C,L
        LOAD    A,SAVCHA        ; restore channel
        .W      ?RDBLK          ; read blocs
        JUMP.,CC FILBU1         ; if no error skeep
        COMP    A,#EREOF        ; else test if end of file
        JUMP,NE ERROR           ; if not, abort
        XOR     A,A
        LOAD    FLGEOF,A        ; clear no EOF flag
FILBU1:
        POP     HL              ; restore offset
        ADD     HL,BC           ; compute new buffer length
        LOAD    A,H             ; test if buffer
        OR      A,L             ; buffer empty
        JUMP.,EQ FILBU3         ; if yes, jump
FILBU2:
        EX      HL,DE           ; else DE = new length
        POP     HL              ; restore begin of buffer
        LOAD    A,FLGEOF        ; A = EOF status
        RET                     ; and return
FILBU3:
        LOAD    A,SAVCHA        ; restore channel
        .W      ?CLOSE          ; close the file
        JUMP,CS ERROR           ; abort if error
        SETC                    ; set carry
        JUMP    FILBU2
FILBU4:
        POP     DE              ; restore begin of buffer
        LOAD    HL,#0           ; set 0 on stack
        PUSH    HL              ; as null offset
        JUMP    FILBU0
```