COMMENT EXPLOITER TOUTES LES RESSOURCES ET AUGMENTER LES PERFORMANCES DE VOTRE

ATARI®



ÉDITIONS WEKA



Guide de l'utilisateur ou "La quête du savoir", jeu d'aventures sur Atari

Les Atari ST et Méga ST sont des machines performantes dans nombre de domaines comme le graphisme ou la génération de sons complexes.

Leur technologie moderne fait qu'ils sont aujourd'hui les plus puissants des ordinateurs familiaux. Le microprocesseur Motorola 68000 qui les équipe était encore, il y a peu de temps, le plus sophistiqué et l'un des plus rapides. Leur capacité mémoire, de 520 kilo-octets à 4 Méga-octets, leur permet même de revendiquer une place parmi les grands et les applications professionnelles développées pour eux en font des machines capables de rivaliser avec bon nombre d'ordinateurs dits professionnels. Preuve en est qu'ils sont à peu près les seuls à avoir résisté et résistent encore au raz-de-marée des compatibles IBM PC qui déferle sur le monde de la micro depuis quelques années.

Pour exploiter au mieux de telles machines, il fallait créer un ouvrage de référence digne d'elles.

Comment exploiter toutes les ressources et augmenter les performances de votre Atari ST 520, 1040 et Méga est le résultat d'années d'expériences dans la programmation des Atari et de nombreuses heures passées à rédiger, structurer, illustrer les divers chapitres qui le constituent.

Vous pourrez consulter cet ouvrage comme une encyclopédie que l'on feuillette avec plaisir, sans but précis, vous laissant glisser au fil des pages. Vous serez alors souvent surpris des découvertes que vous y ferez. Vos nuits seront alors peut-être un peu courtes...

Vous aurez également, sans doute dans la plupart des cas, à rechercher une information précise. C'est dans ce but que nous avons essayé de structurer au mieux l'ouvrage et que nous vous donnons les quelques conseils qui suivent.

La carte de l'oeuvre

La figure 1 représente en quelque sorte la carte générale de l'ouvrage, chaque bloc figurant une partie.

Ces blocs sont disposés sur un champ représentant les grands types d'application, cours, programme et matériel ainsi que la nature du contenu qui peut être soit une description, soit une réalisation originale.

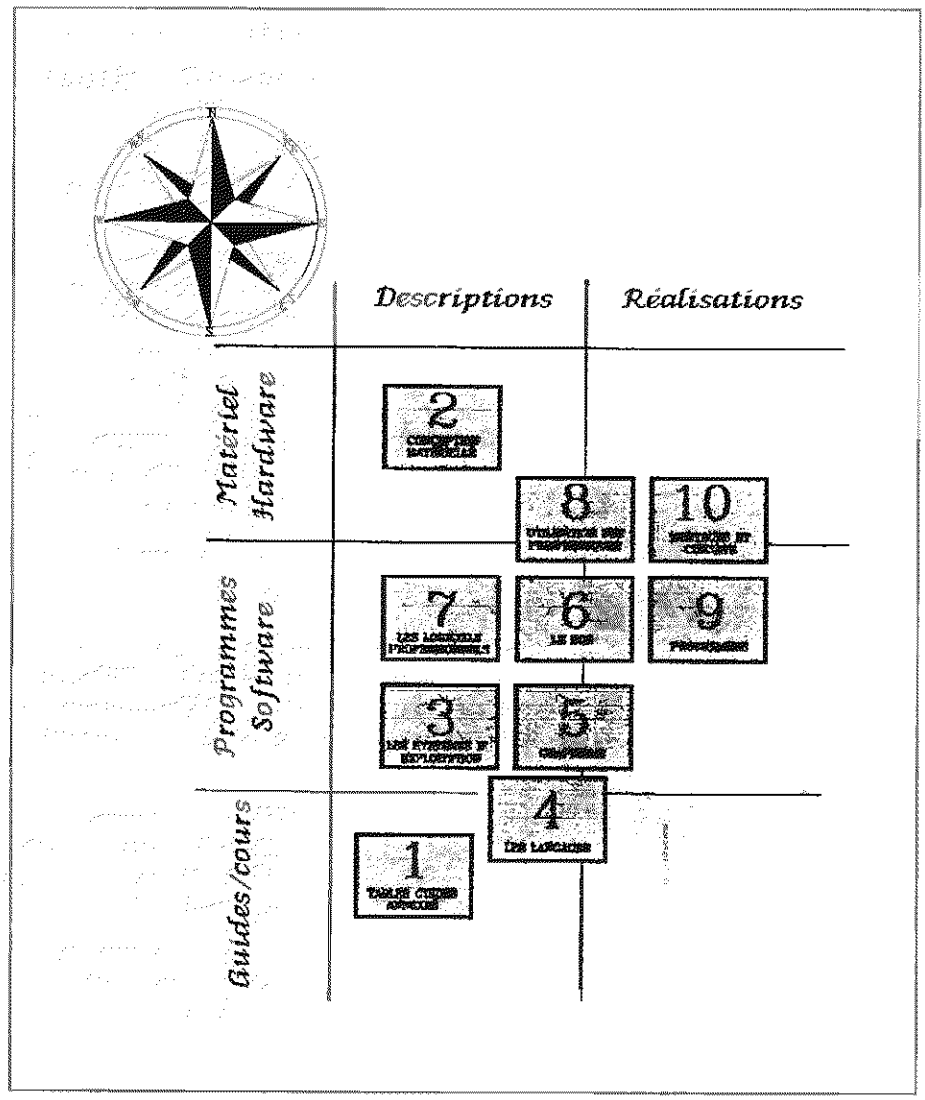


Fig. 1

La position des blocs chevauchant les zones représente sensiblement la répartition du contenu rédactionnel.

Cette *photographie* très synthétique de l'ouvrage devrait vous donner, très rapidement, les indications de base nécessaires à toute recherche.

- C'est ainsi que la Partie 1 sera constituée de la table des matières générale, des index ainsi que des annexes.
- La Partie 2 intitulée *Conception matérielle des ST* est consacrée au *hardware*, la quincaillerie en français. Vous avez deviné que l'on y décrit les cartes mères, les divers composants constituant votre ST, les connecteurs, les moniteurs, etc.
- La Partie 3, quant à elle, est plutôt le lieu de prédilection de la matière grise des ST, à savoir les systèmes d'exploitation. BIOS, XBIOS,

GEMDOS, autant de termes assez hermétiques, derrière lesquels se cachent des trésors que seul l'initié saura utiliser.

- En Partie 4, nous trouvons les langages de programmation. Assembleur, divers Basic, C ou autres, ces langages constituent en fait le seul lien entre vous et votre ST. Sans eux, point de compréhension mutuelle possible, votre ST ne serait qu'une machine inerte. Cette partie vous est assurément indispensable.
- Nous voici dans la Partie 5, un peu plus frivole puisqu'elle traite du graphisme. Graphisme doit être pris au sens large et plutôt compris comme *traitement de l'image*. En effet, nous avons intégré dans cette partie aussi bien les systèmes d'exploitation graphiques GDOS ou VDI que la description critique de logiciels graphiques du commerce, ou encore des programmes originaux dont le but essentiel est le graphisme.
- En Partie 6, domaine du son, un des points forts des ST, vous trouverez, sans que cela fasse double emploi avec les autres parties, des descriptions de matériels comme le générateur sonore des programmes de traitement du son ou de gestion de la très célèbre interface MIDI.
- La Partie 7 est en quelque sorte le salon du logiciel. En effet, nous y décrirons en détail les logiciels à caractère professionnel tournant sur ST. Ces descriptions complètent la documentation souvent réduite, fournie avec le logiciel.

  Des exemples d'utilisation, des astuces, ou bien des critiques pourront venir compléter cette partie.
- La Partie 8, c'est l'ouverture ; mot à la mode qui signifie, pour les ST, le transfert de données vers l'extérieur ou la réception de ces données. Que ce soit avec une imprimante, un serveur télématique, un disque dur externe, une tablette à digitaliser, toute conversation du ST avec l'un de ces périphériques trouvera sa description ici.
- Les programmes originaux, qui peuvent être les vôtres, seront présentés en Partie 9. Divisée en chapitres, utilitaires, jeux, gestion, etc., cette partie s'ornera de programmes décrits en détail aussi bien dans leur utilisation que dans leur programmation.
- Les bricoleurs, amateurs impénitents du fer à souder, seront au paradis dans la Partie 10. Des montages originaux, décrits dans leurs moindres détails, viendront constituer cette partie, véritable caverne d'Ali Baba du micromaniaque.

Le grimoire secret

Toute information visuelle, dans votre *quête du savoir*, est bonne à prendre et ne figure pas sur vos pages sans raison.

Chaque page de votre ouvrage est munie de repères que nous essayons de rendre le plus pratiques possible, que ce soit lors de la consultation ou lors de l'insertion des mises à jour.

C'est ainsi que les pages de chaque partie sont marquées d'un pictogramme schématisant le domaine concerné.

La figure 2 vous présente les pictogrammes affectés à chaque partie. Bientôt, ils vous seront aussi familiers que des panneaux routiers.


Guide de l'utilisateur


Partie 1 Tables des matières, index, annexes


Partie 2 Conception matérielle des ST

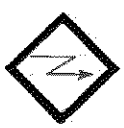

Partie 3 Systèmes d'exploitation


Partie 4 Langages des Atari ST


Partie 5 Graphisme


Partie 6 Son


Partie 7 Logiciels à caractère professionnel


Partie 8 Utilisation des périphériques


Partie 9 Programmes


Partie 10 Fabrication de montages et circuits additionnels

Fig. 2

Par ailleurs, sur chaque page figure également, en haut et à droite sur les rectos et en haut à gauche sur les versos, une « têtière » donnant les bornes de numérotation.

Sur l'exemple de la figure 3, vous pouvez lire « Partie ... Chapitre ... à ... page 1 ».

La numérotation des pages se fera donc de façon continue depuis ... jusqu'à .... Dès le chapitre suivant (...), la numérotation reprendra à partir de la page 1.

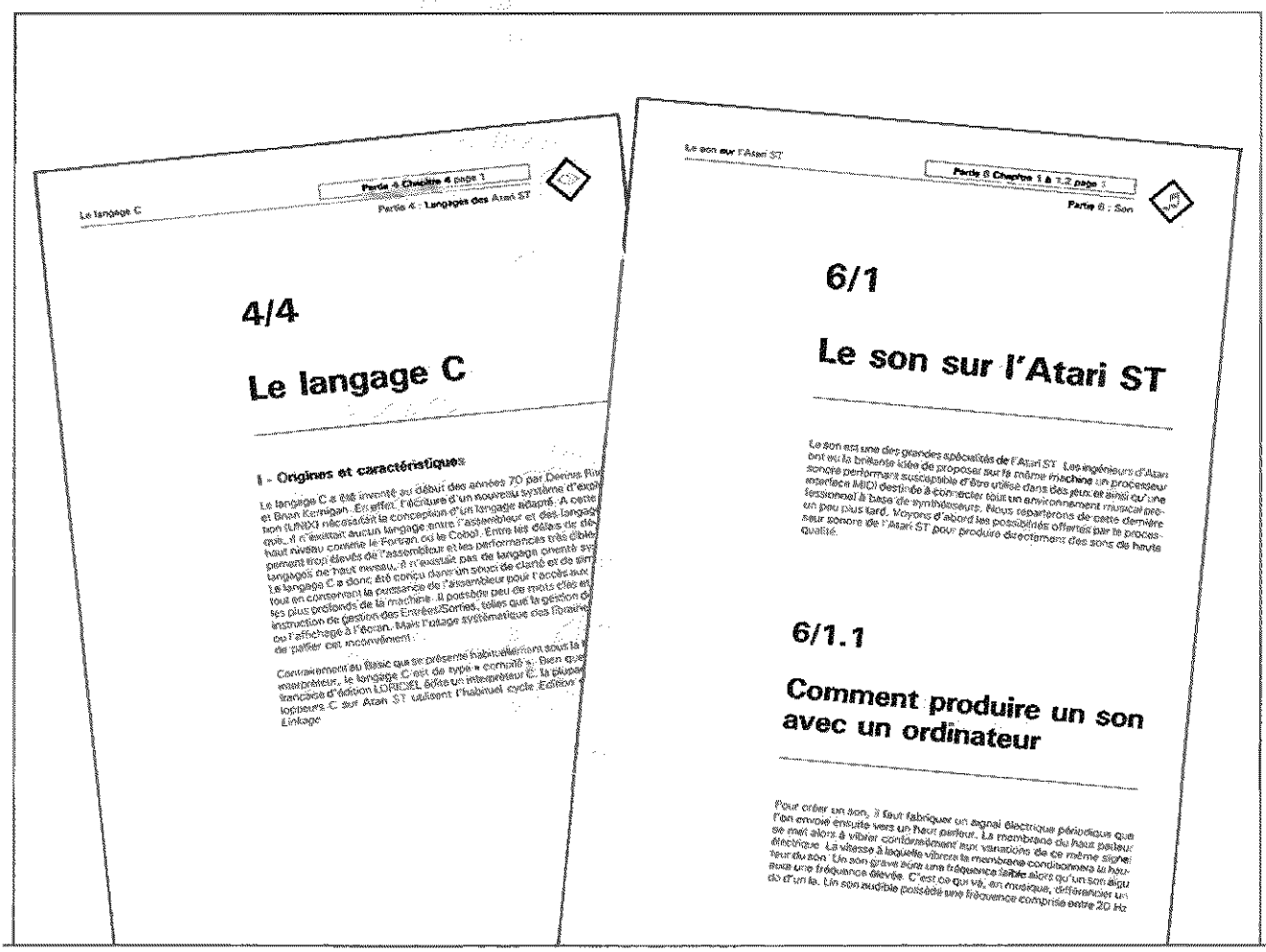


Fig. 3

En tête de l'ouvrage, en Partie 1 précisément, se trouve une table des matières générale. Celle-ci reprend les titres de chaque partie et chapitre. C'est-à-dire tout ce qui sera numéroté avec 2 chiffres.

Un niveau de détail plus fin est donné par le sommaire de chaque partie, figurant bien sûr en tête de chacune.

Les clés du savoir

Les index constituent en quelque sorte les clés d'accès à l'information que vous recherchez. Là encore, nous avons essayé de rendre ces index les plus pratiques possible.

Afin d'assurer efficacement leur consultation et leur remise à jour, ceux-ci sont organisés en une structure arborescente (Fig. 4).

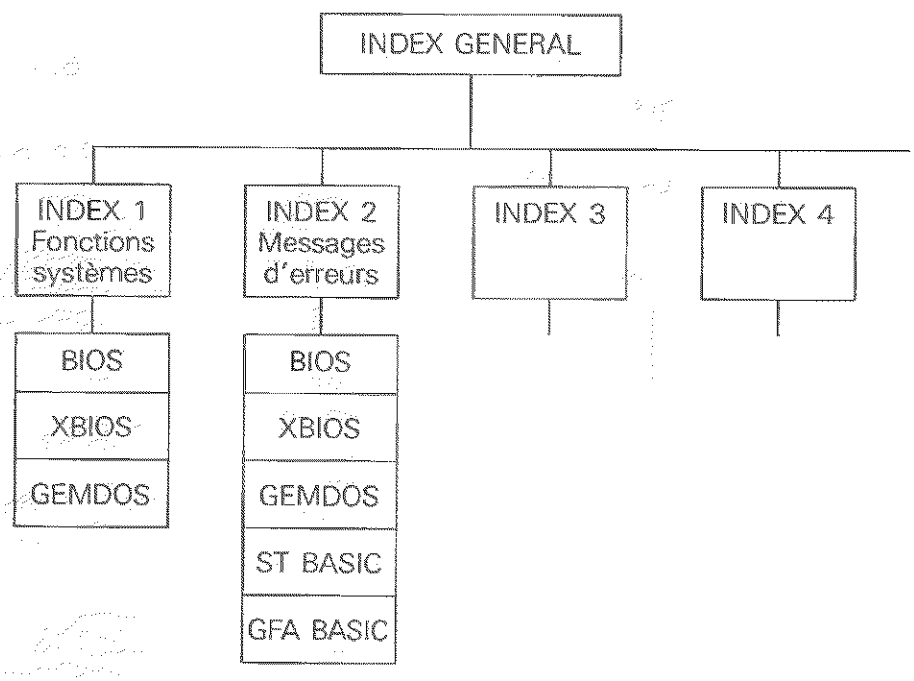


Fig. 4



De nouveaux index pourront être créés si besoin est.

Dans la mesure du possible, nous avons essayé de combiner la fonction INDEX avec la fonction GLOSSAIRE. En effet, il nous a semblé que les deux informations devaient être rassemblées en un seul document puisque leur but est le même, tout connaître d'un thème ou d'un terme.

L'oracle du Sage

Avant de vous jeter à corps perdu dans cette quête du savoir, recevez le conseil qui suit. Conquérant impétueux du savoir, consultez votre ouvrage et ses index sans précipitation et avec méthode. Suivez le petit schéma de la figure 5 et, ainsi, vous ne vous perdrez jamais dans les circonvolutions multiples de la connaissance.

La démarche du méthodique

Vous désirez vous reporter aux chapitres où est décrit un terme précis. Vous consulterez dans ce cas l'index général qui vous donnera la réponse désirée ou, le cas échéant, vous renverra vers un ou plusieurs index spécialisés.

La démarche de l'initié

Vous avez une idée assez précise de la nature de l'information recherchée ; par exemple, vous savez qu'il s'agit d'un message d'erreur, vous pouvez alors consulter directement l'index spécialisé des messages d'erreur (Fig. 5).

En cas de panique, ou si nous avons oublié quelque chose, n'hésitez pas à nous écrire.

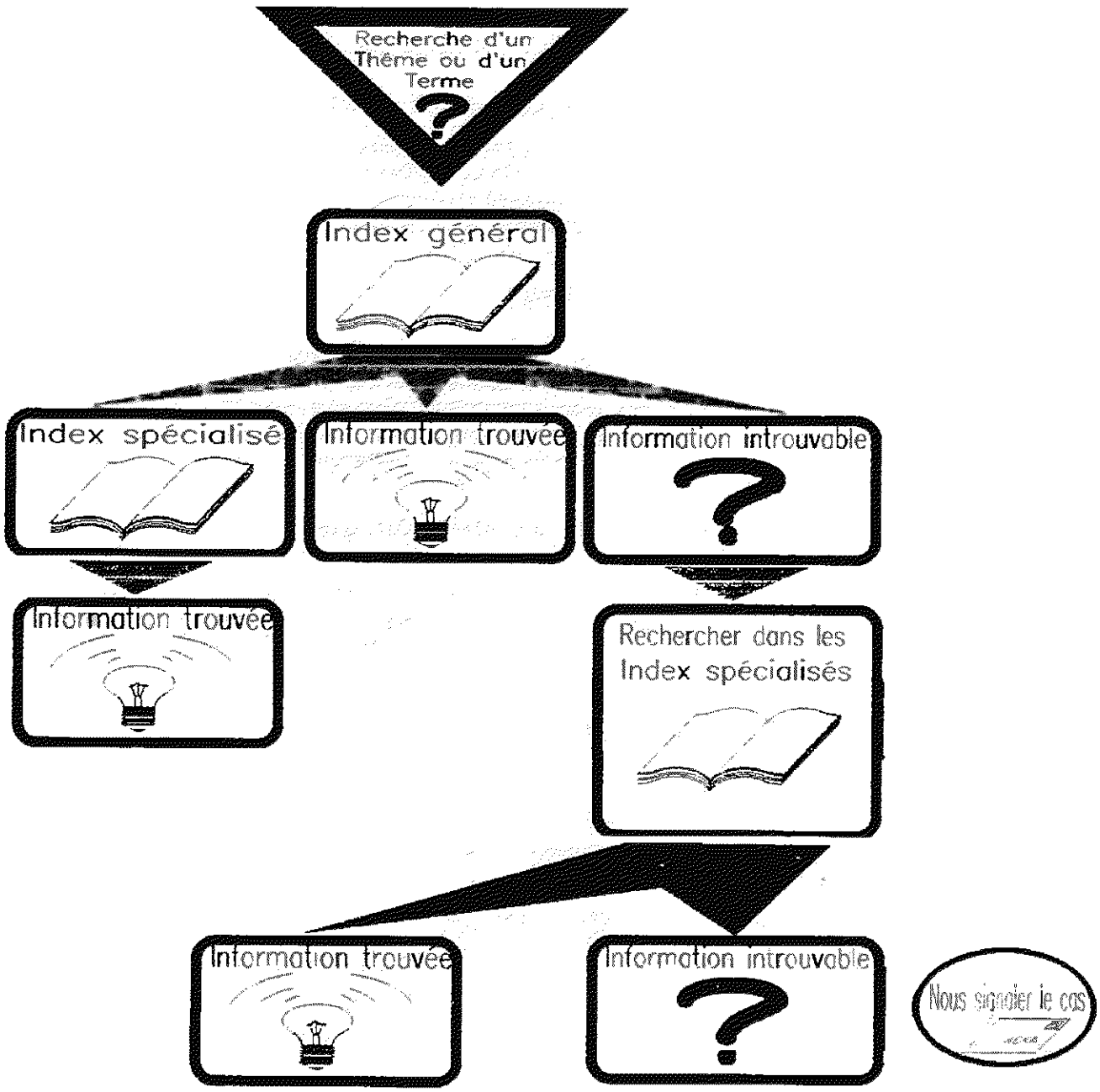


Fig. 5

Armé de ces quelques conseils, tel un chevalier moderne, partez à la *quête du savoir*. Découvrez le trésor et aussi les terribles dangers de votre Atari. Affrontez les horribles Bugs, savourez les délices de la programmation, tremblez avant le fatidique « RUN » mais n'oubliez jamais que votre guide *« Comment exploiter toutes les ressources et augmenter les performances de votre Atari ST 520, 1040 et Méga »* : est là, tout près de vous, pour vous seconder dans cette palpitante aventure.



1/1

Table des matières générale

Tome 1

Guide de l'utilisateur

1 Table des matières - Index - Annexes

1/1 Table des matières générale
1/2 Index général
1/2.1 Index 1 : Les fonctions BIOS, XBIOS et GEMDOS utilisables sur les Atari ST
1/2.2 Les messages d'erreur sur Atari ST
1/3 Annexes
1/3.0 Annexes 0 : Plus loin
1/3.0.1 Le nouvel ATARI STE
1/3.0.2 Vive la différence. Compatibilité jusqu'où ?
1/3.1 Annexe 1 : Les codes ASCII des caractères Atari
1/3.2 Annexe 2 : La table des SCANCODES des claviers Atari
1/3.3 Annexe 3 : Résumé des codes opératoires du 68000
1/3.4 Annexe 4 : Les principales abréviations utilisées dans le monde des Atari
1/3.5 Annexe 5 : Résumé des instructions du ST Basic

2 Conception matérielle des ST

2/0 Plus loin
2/1 Le système ST

2/2 La connectique Atari
2/3 Les périphériques de base

3 Systèmes d'exploitation

3/0 Plus loin
3/1 Définitions et rappels
3/2 Le BIOS
3/3 L'extended BIOS (XBIOS)
3/4 Le GEMDOS
3/5 L'AES
3/6 Le Bureau GEM
3/7 La gestion des périphériques de base
3/8 Les éditeurs de ressources

Tome 2

4 Langages des Atari ST

4/1 La programmation structurée
4/2 Les Basic sur Atari ST
4/3 L'assembleur
4/4 Le langage C
4/5 Le langage Pascal
4/6 Le langage logo

5 Graphisme

5/0 Plus loin
5/1 Le système d'exploitation graphique GDOS
5/2 LE VDI
5/3 Les routines de la LINE-A et leur utilisation
5/4 Les programmes de dessin du marché
5/5 Les programmes de graphisme
5/6 Traitement d'image

Tome 3

6 Son

6/1 Le son sur l'Atari ST
6/2 L'environnement MIDI



6/3 Plus loin
6/4 Les grands logiciels musicaux

7 Logiciels à caractère professionnel

7/1 Les traitements de texte
7/2 Les tableurs et les grapheurs
7/3 Les utilitaires
7/4 Les Systèmes de Gestion de Bases de Données (SGBD)
7/5 La Publication Assistée par Ordinateur

8 Utilisation des périphériques

8/1 La communication avec une imprimante par le port Centronics
8/2 La transmission des données
8/3 Un Modem pour l'Atari
8/4 Serveur télématique

Tome 4

9 Programmes

9/1 ...
9/2 Les utilitaires
9/3 Les jeux
9/4 Les mathématiques
9/5 La gestion familiale et professionnelle

10 Fabrication de montages et circuits additionnels

10/1 La connexion de ST au Minitel
10/2 La commande de circuits intégrés TTL, CMOS et de puissance
10/3 Une rupture des cartes d'extension Atari ST
10/4 Un générateur sonore performant : le SAA 1099

11 Notions scientifiques de base

11/0 Avant-propos
...
11/3 Résolution de problèmes physiques

1/2

Index général

A

ACCES DIRECT EN MEMOIRE	comme son nom l'indique	2/1.1, 2/1.2, 2/1.5
ACIA	*Asynchronous Communication Interface Adaptor*. Circuit spécialisé dans la gestion des signaux lors d'une communication inter-systèmes	2/1.2 p. 1, 2/3.2 p. 1
ADRESSE	valeur numérique correspondant au numéro (octet) d'une case mémoire	2/1.3 p. 4
AES	*Application Environment System*. Programme qui prend en charge l'interface utilisateur	3/5
ALIMENTATION	circuit électronique chargé de fournir l'énergie électrique sous les diverses formes requises par les divers montages ou composants d'un système	2/1.2, 2/2.1 p. 5
ARCHITECTURE	organisation d'un système logiciel ou matériel	2/1.1, 4/3.5
ASCII	code normalisé affecté à tout caractère compréhensible par un ordinateur	1/3.1
ASSEMBLEUR	langage très proche du langage machine ou outil d'assemblage d'un code source	4/2.2 p. 6, 4/3, 9/2.1
AY3-8910	nom du processeur sonore utilisé sur les Atari ST	2/1.1, 2/1.2, 2/1.4 p. 11 et 12, 3/3 p. 19, 20, 22, 6/1.2

B

BASE DE DONNEES	fichier de données organisé de façon particulière en vue d'y faciliter l'accès par des opérations de tri, l'application de masques, les tabulations	7/4, 9/5.1

BASIC	langage de programmation simple mis au point par le Dartmouth College (USA) existe en version interprétée et maintenant en versions compilables	4
BASIC GFA	Basic développé par GFA, ce Basic puissant est compilable, la dernière version existant aujourd'hui est la version 3.03	4/2.2
BAUD	unité de vitesse de transmission (1 bit/s)	10/1.1 p. 3
BIDIRECTIONNEL	traduit l'indifférence du sens de circulation de l'information	2/1.3 p. 3, 2/1.4 p. 2
BINAIRE	qualifie tout ce qui est défini dans un système de numération de base 2	
BINARY	voir BINAIRE	
BIOS	programme constructeur, écrit en mémoire morte et gérant les opérations d'entrée/sortie de base	3/2, 1/2.1, 1/2.2
BIT	unité élémentaire d'information, peut prendre seulement 2 valeurs 0 ou 1	
BRANCHEMENT	connexion à une instruction non adjacente à l'instruction en cours, ce branchement peut être conditionnel ou systématique	4/3 p. 14, 17, 4/4 p. 15
BROCHAGE	affectation des diverses broches d'un circuit ou d'un connecteur	2/1.2, 2/1.3, 2/1.4 p. 2, 2/2, 10/1.1, 10/2 p. 2
BUFFER	voir TAMPON	
BUS DE COMMANDE	ensemble de lignes au travers desquelles circulent les divers signaux de commande	2/1.3 p. 2 et 3
BUS DE DONNEES	ensemble de lignes au travers desquelles circulent les divers signaux de données. Chaque ligne véhicule un bit distinct	2/1.3 p. 2 et 3
BUS D'ADRESSES	ensemble de lignes au travers desquelles circulent les signaux d'adresses, un bit par ligne	2/1.3 p. 2 et 3

C

CARACTERE	élément graphique ou de contrôle généralement défini par un code composé de 8 bits	3/4 p. 4, 8, 4/2.1 p. 8, 18, 5/1 p. 5, 5/2 p. 15, 5/2.2 p. 15, 16, 5/3 p. 5
CARTE MERE	carte-circuit portant le microprocesseur et les organes principaux du micro	2/1
CARTOUCHE	élément additionnel comportant une application (programme) en ROM	2/1.2, 2/2.1 p. 6
CENTRONICS	norme de communication utilisée pour les imprimantes en particulier (parallèle)	2/2.1, 3/2 p. 2, 8/1, 10/2



CHAINE	ensemble de caractères (STRING)	
CIRCUIT MULTIFONCTION	comme son nom l'indique (voir MFP 68901)	
CLAVIER	périphérique d'entrée (saisie des informations) interface avec l'utilisateur	2/1.2, 2/3 p. 1, 3/2 p. 2, 7, 3/3 p. 9, 13, 24, 3/4 p. 2, 3/5.1 p. 8
CLUSTER	ensemble de secteurs d'une piste de disquette ou de disque dur (généralement 4)	3/2 p. 5
CODE CONDITION REGISTER	registre d'un microprocesseur spécialisé dans la sauvegarde d'états et de résultats d'opérations effectuées par le microprocesseur	4/3 p. 8
CODE OPERATOIRE	code symbolisant une opération élémentaire devant être effectuée par le microprocesseur	1/3.3, 4/3 p. 9, 9/2.1.1
COMMANDE DE CIRCUITS	opération consistant à envoyer des signaux de forme adéquate à un circuit de façon à lui ordonner une action déterminée	10/2
COMPARAISON	en général le résultat d'une comparaison est *faux* si les deux objets comparés ne sont pas strictement identiques, *vrai* dans l'autre cas	4/3 p. 18
COMPTEUR	circuit ou programme dont la spécialité est de repérer la position dans une base de temps. Le réveil matin est un compteur	2/1.4, 3/3 p. 21
COMPTEUR DE PROGRAMME	compteur particulier, intégré au microprocesseur dont la spécialité est de repérer le numéro de l'instruction en cours d'exécution	4/3 p. 8
CONNECTEUR	élément mécanique permettant de relier électriquement deux systèmes entre eux	2/2, 8/1 p. 3, 10/2
CONTROLEUR DE DISQUE WD 1772	circuit intégré spécialisé dans la gestion des accès aux disques	2/1.4 p. 12 à 14
CONTROLEUR DE DMA	circuit intégré spécialisé dans le contrôle des accès à la mémoire	2/1.2
COULEUR	sensation visuelle résultant de la synthèse additive des 3 couleurs élémentaires rouge, vert et bleu (dans le cas de l'écran vidéo seulement)	3/3 p. 4, 5, 3/4 p. 2, 4/2.1 p. 22, 5/2.2 p. 12 à 14, 26, 5/3 p. 2
CURSEUR	symbole généralement clignotant repérant la position d'écriture courante sur l'écran vidéo	3/3 p. 16, 1/2.1 p. 2, 3/4 p. 2, 3/5.1 p. 18, 5/2.2 p. 22

D

DATE		3/4 p. 14, 33
DECLARATION	action consistant à informer l'ordinateur de l'existence et de la nature d'une variable ou d'un tableau	4/2.1 p. 23, 4/4 p. 8

DIRECT MEMORY ACCESS	voir ACCES DIRECT EN MEMOIRE	
DISQUE VIRTUEL	unité de disque *fantôme* définie en mémoire centrale dont l'accès est beaucoup plus rapide qu'un disque physique	7/3.3, 9/2.2
DISQUETTE	petit disque de mylar recouvert d'oxyde magnétique sur lequel on peut écrire et lire des informations, voir lecteur de disquettes	

E

ECRAN	organe physique sur lequel les résultats du travail du microprocesseur sont visualisés ou être fugace composé des informations résident un instant	3/3 p. 3, 16, 3/4 p. 2, 6, 8, 5/2.2, 5/3.2, 7/3.4
ERREUR	résultat d'une mauvaise syntaxe ou plus généralement d'un ordre incompréhensible ou d'une opération impossible	1/2.2, 3/3 p. 6, 3/4 p. 3
ETAT	information caractérisant l'état d'un système ou d'un registre	2/1.3 p. 3, 5/2.2 p. 25

F

FAT	*File Allocation Table*, voir Table d'allocation de fichiers	
FENETRE	portion d'écran contenant des informations et qui peut s'ouvrir en surimpression sur un écran d'une autre nature	3/5.1 p. 5, 33 à 38, 4/2.1 p. 2
FICHIER	ensemble d'informations groupées sous un même nom et constituant un programme, un texte ou des données	3/4 p. 21 à 27, 32, 3/5.1 p. 32, 41, 4/2.1 p. 9, 9/5
FLOPPY DISK CONTROLLER	voir CONTROLEUR DE DISQUE	
FLOPPY DISK DRIVER	voir LECTEUR DE DISQUETTES	
FONCTION		3/2, 1/2 1, 3/4 p. 4, 3/5, 4/2.1 p. 26, 5/2.1, 9/4.1
FORMATAGE	opération consistant à définir des pistes et secteurs sur une disquette	3/3 p. 6, 13
FREQUENCE	nombre d'occurrences d'un événement pendant une seconde. Exemple, le courant alternatif EDF s'inverse 50 fois par seconde	3/2 p. 5



G

GDOS	extension graphique optionnelle du système d'exploitation GEM des Atari	5/1
GEMDOS	partie du système d'exploitation gérant les opérations de lecture/écriture sur disque ainsi que l'accès écran et mémoire	2/1.2, 3/4
GLUE	circuit présent sur les cartes mères et gérant les tampons et les décodages d'adresses	2/1.2
GRAPHISME	dans son sens le plus général ici, tout ce qui concerne la génération d'images	3/5.1 p. 23, 4/2.1 p. 19, 9/4.1

H

HARDWARE	voir MATERIEL	
HEURE		3/4 p. 15, 36
HORLOGE	partie d'un circuit intégré ou d'un processeur dont la tâche est d'engendrer un signal périodique de fréquence calibrée, un métronome en somme	1/2.1 p. 2, 2/1.3 p. 3, 2/1.4 p. 2, 3/3 p. 17

I

IMPRIMANTE	périphérique d'impression graphique ou alphanumérique	3/3 p. 16, 23, 8/1
INTERFACE RS 232	voir RS232	
INTERRUPTION	tout ce qui caractérise l'arrêt momentané d'une opération en cours demandée en priorité par un élément extérieur	3/3 p. 9, 18, 25

J

JEU	occupation en général amusante et dont le but essentiel est la détente	9/3
JEU D'ARCADE	type particulier de jeu qui tient son nom des ARCADES anglosaxonnes, lieux où l'on trouve des machines de jeux en général plutôt basés sur les réflexes	9/3.1
JEU D'AVENTURE	type de jeu consistant à simuler des aventures plus ou moins rocambolesques généralement en forme d'énigme ou de jeu de piste	9/3.2
JOYSTICK	voir MANETTE	

L

LANGAGE	ensemble d'instructions appelées par des mots normalisés et constituant un ensemble cohérent permettant à l'utilisateur de dialoguer avec l'ordinateur	4
LANGAGE C	langage moderne, puissant et structuré dont l'utilisation se généralise peu à peu	4/4
LATENCE	vitesse de répétition d'une action déclenchée par l'appui continu sur une touche du clavier	3/3 p. 24
LECTEUR DE DISQUETTE	périphérique d'enregistrement/lecture	2/2.1 p. 3, 3/2 p. 3 à 6, 3/3 p. 5 à 6, 3/4 p. 10, 12, 18, 4/2.1 p. 29, 9/2
LINE A	ensemble de routines graphiques écrites dans les ROM des Atari	5/3
LOGO	langage de programmation inventé par Seymour Papert	4/6
LOW SIGNIFICANT BYTE	voir OCTET MOINS SIGNIFICATIF	

M

MANETTE	périphérique de saisie, en général réservé aux jeux	2/2.1 p. 7
MATERIEL	qualifie tout ce qui existe physiquement : circuits, écrans, lecteurs, etc., par opposition à tout ce qui est immatériel : programmes, mémoires et données	2
MEMOIRE	ensemble de circuits ou d'emplacements dans lesquels sont inscrits les informations, programmes, données, etc., nécessaires au fonctionnement du micro	2/1.5, 3/2 p. 1, 3/4 p. 3, 30 à 32, 4/3 p. 3
MEMOIRE MORTE	ensemble constitué de mémoire accessible uniquement en lecture	2/1.1, 2/1.2, 2/1.5, 2/2.1 p. 6
MEMOIRE VIDEO	partie de la mémoire vive dans laquelle sont inscrites toutes les informations nécessaires pour engendrer un écran	3/3 p. 3 à 4, 5/3.2
MEMOIRE VIVE	ensemble de la mémoire accessible aussi bien en lecture qu'en écriture	2/1.1, 2/1.5
MFP 68901	circuit intégré dont les tâches multiples en font une véritable cheville ouvrière de l'Atari... après le 68000 bien sûr	2/1.4, 3/3 p. 9, 18, 20, 3/4 p. 8
MICROPROCESSEUR	Cœur ou cerveau, à vous de choisir, mais à coup sûr l'élément le plus important de votre Atari	2/1.3, 3/1 p. 3



MIDI	norme réglementant la communication entre un ordinateur et tout instrument de musique numérique comme un synthétiseur par exemple	2/2.1 p. 3, 3/2 p. 2, 3/3 p. 8 à 9, 23, 6/2
MINITEL	petite console télématique, fer de lance de la télématique française et qui plus est... QUASIMENT gratuit	10/1
MMU	circuit de gestion des adresses et chargé du multiplexage	2/1.2
MODE SUPERVISEUR	mode de fonctionnement du microprocesseur 68000 qui autorise l'utilisation de toutes les instructions et de l'ensemble de la mémoire	3/3 p. 25, 3/4 p. 13, 4/3 p. 5
MODE UTILISATEUR	mode de fonctionnement de microprocesseur 68000 autorisant l'utilisation d'un jeu d'instructions réduit et l'accès limité à la mémoire	3/3 p. 25, 3/4 p. 13, 4/3 p. 5
MONITEUR	périphérique de sortie, permet l'affichage des résultats, voir aussi écran	2/1.1, 2/2.1 p. 5, 2/3.1
MOST SIGNIFICANT BYTE	voir OCTET LE PLUS SIGNIFICATIF	

O

OCTET	ensemble de 8 bits, c'est en quelque sorte l'unité de taille d'information, correspond à un caractère	
OCTET MOINS SIGNIFICATIF	octet représentant la partie faible d'un nombre codé sur plus d'un octet (valeur comprise entre 0 et 255)	2/1.3 p. 3
OCTET PLUS SIGNIFICATIF	octet représentant la partie forte d'un nombre codé sur plus d'un octet (partie supérieure à 255)	2/1.3 p. 4

P

PALETTE	ensemble de couleurs constituant le choix possible sur un même écran	3/3 p. 4
PARALLELE	type de transmission, chaque bit d'un octet est transmis par une ligne	2/2.1
PASCAL	langage de haut niveau, structuré et compilable	4/5
PERIPHERIQUE	tout système extérieur à l'ordinateur échangeant des informations avec lui	2/3, 3/1 p. 8, 3/2 p. 2 à 7, 3/3 p. 9, 5/1 p. 2
PERITEL	abréviation de Péritélévision. Environnement normalisé en France pour la connexion de périphériques aux téléviseurs. Magnétoscopes, ordinateurs, etc.	2/2.1 p. 5, 2/3, 3/1 p. 8, 3/2 p. 2 à 6, 5/1 p. 2

PISTE	Couronne concentrique, partie d'une disquette (définie par le formatage). Voir DISQUETTE.	
POINTEUR	registre, variable ou adresse mémoire contenant une valeur qui est en fait une adresse où l'information recherchée se trouve	3/4 p. 32, 4/4 p. 15, 9/2.2
PORT	point de communication par lequel transite toute information échangée avec l'extérieur	2/1.2, 2/2, 3/4 p. 5
PROCESSEUR SONORE	voir AY3-8910	
PROGRAMMATION STRUCTUREE	type de programmation rigoureuse permettant essentiellement par sa présentation d'énoncer clairement les algorithmes	4/1

R

RAM DISK	voir DISQUE VIRTUEL	
RANDOM ACCESS MEMORY	voir MEMOIRE VIVE	
READ ONLY MEMORY	voir MEMOIRE MORTE	
REGISTRE	zone mémoire ou mémoire interne d'un processeur dans lequel sont écrites les informations nécessaires ou résultat d'une opération du dit processeur	2/1.4 p. 4 à 10, 2/1.5, 4/3 p. 6 à 8
REPERTOIRE	sous-ensemble d'un disque dans lequel sont rangés des fichiers. Un répertoire peut également contenir des sous-répertoires	3/4 p. 18, 20
RESET	voir RE-INITIALISATION	
RESOLUTION	qualifie le *piqué* d'un écran vidéo par le nombre de points d'écran définis	3/3 p. 4, 5/3.2
RE-INITIALISATION	opération de remise à zéro du microprocesseur et généralement de l'environnement associé, mémoires, registres, etc.	2/1.2
RS232	Norme de communication en série entre deux systèmes, cablage et nature des signaux sont définis par cette norme	2/2.1 p. 2, 3/2 p. 2, 3/3 p. 11, 10/1.1

S

SCANCODE	code généré pour chaque touche par le clavier	1/3.2
SCREEN	voir ECRAN	



SCROLLING	littéralement défilement. S'applique en général à tout défilement d'écran à la manière d'un panoramique de cinéma. Peut être vertical ou horizontal	9/3.1.1
SECTEUR	partie d'une piste de disquette ou disque dur constituée en général de 256 octets	3/2 p. 3 à 5, 3/3 p. 5
SGBD	Système de Gestion de Bases de Données voir Base de Données	7/4
SIGNAL D'HORLOGE	signal électrique composé d'impulsions à fréquence déterminée engendrée par une horloge (voir ce mot)	2/1.3 p. 3, 3/2 p. 5
SON	bruit, musique... vibration de l'air ambiant	6
SOURIS	périphérique d'entrée (saisie) remplace souvent le clavier	2/2.1 p. 7, 2/3.2 p. 1, 3/3 p. 1, 3/5.1 p. 3, 8, 28, 5/2.2 p. 23, 5/3 p. 5, 6
ST BASIC	langage Basic interprété fourni avec les Atari en France	4/2
SUPERVISEUR	voir MODE SUPERVISEUR	
SUPERVISOR	voir SUPERVISEUR	
SUPERVISOR MODE	voir MODE SUPERVISEUR	
SYNTHETISEUR SONORE	voir AY3-8910	
SYSTEME D'EXPLOITATION	programme écrit par le constructeur et qui gère le fonctionnement général de l'ordinateur, principalement dans les échanges avec les périphériques	3/5

T

TABLE D'ALLOCATION DE FICHIERS	table spéciale engendrée par le système d'exploitation de disque, et contenant la *carte* codée du contenu d'un disque	3/2 p. 5, 9/2.2
TABLEUR	programme permettant les calculs sur une *feuille de calcul électronique*	7/2
TAMPON	zone mémoire dans laquelle sont stockées de manière provisoire des informations en transit d'un ou vers un périphérique	3/3 p. 9, 3/4 p. 11, 15, 3/5.1 p. 4, 9/3.1.1
TEST	essai d'un système ou opération visant à vérifier si un événement déterminé s'est produit ou non	3/2 p. 7, 4/3 p. 15 à 17, 4/4 p. 12
TIMER	voir COMPTEUR	

TOUCHES	micro-contacts constituant un clavier dont la fermeture engendre un signal détecté par le système auquel il est connecté	3/2 p. 7, 3/5.1 p. 8, 28. 5/2.2 p. 24
TRACK	voir piste ou disquette	
TRAITEMENT DE TEXTES	programme permettant la mise en page, la correction et la gestion de textes	7/1
TRAMIEL OPERATING SYSTEM	ensemble des systèmes d'exploitaiton de l'Atari ST	3
TYPE		4/2.1 p. 27

U

UNITES LOGIQUES	unités périphériques vues par l'ordinateur plutôt comme des « canaux » que comme des ensembles physiques	3/2 p. 7
USER MODE	voir MODE UTILISATEUR	
UTILISATEUR	voir MODE UTILISATEUR	

V

VARIABLE		4/2.1 p. 27, 4/4 p. 6
VDI	Virtual Device Interface, interface graphique de GEM	5/2

W

WINDOW	voir FENETRE	

X

XBIOS	extension du BIOS	1/2.1, 1/2.2, 3/3



1/2.1

Index 1 : Les fonctions BIOS, XBIOS et GEMDOS utilisables sur les Atari ST

Fonction OS	Numéro Appartenance	Description de la fonction	Les détails se trouvent :
GETMPB	0 BIOS	Accès aux blocs de gestion Mémoire du système d'exploitation	3/2 p. 1
BCONSTAT	1 BIOS	Périphérique ayant un caractère à transmettre	3/2 p. 2
BCONIN	2 BIOS	Récupération d'un caractère depuis un périphérique	3/2 p. 3
BCONOUT	3 BIOS	Emission d'un caractère vers un périphérique	3/2 p. 3
RWABS	4 BIOS	Lecture/Ecriture de secteurs sur disquette	3/2 p. 4
SETEXEC	5 BIOS	Modification d'un vecteur du 68000	3/2 p. 5
TICKTAL	6 BIOS	Renvoie la fréquence d'horloge du système	3/2 p. 5
GETBPB	7 BIOS	Renvoie l'adresse d'un tableau concernant la disquette spécifiée	3/2 p. 5
BCOSTAT	8 BIOS	Vérification du périphérique prêt à recevoir	3/2 p. 6
MEDIACH	9 BIOS	Test d'introduction d'une nouvelle disquette	3/2 p. 6
DRVMAP	10 BIOS	Renvoie le nombre d'unités logiques connectées	3/2 p. 7
KBSHIFT	11 BIOS	Test sur l'appui des touches spéciales	3/2 p. 7
INITMOUSE	0 XBIOS	Fixe le type de fonctionnement des routines souris	3/3 p. 1
SSBRK	1 XBIOS	Non implantée	3/3 p. 3
PHYBASE	2 XBIOS	Renvoie l'adresse de la mémoire vidéo physique	3/3 p. 3
LOGBASE	3 XBIOS	Renvoie l'adresse de la mémoire vidéo logique	3/3 p. 3
GETREZ	4 XBIOS	Donne la résolution de l'écran	3/3 p. 3
SETSCREEN	5 XBIOS	Modification des adresses physique et logique et de la résolution de l'écran	3/3 p. 4

Fonction OS	Numéro Appartenance	Description de la fonction	Les détails se trouvent :
SETPALETTE	6 XBIOS	Chargement en mémoire d'une palette de couleurs	3/3 p. 4
SETCOLOR	7 XBIOS	Changement d'une seule couleur de la palette	3/3 p. 5
FLOPPYREAD	8 XBIOS	Chargement en mémoire du contenu de secteurs de la disquette	3/3 p. 5
FLOPPYWRITE	9 XBIOS	Permet l'écriture de secteurs sur une disquette	3/3 p. 7
FLOPPYFORMAT	10 XBIOS	Formate une piste de disquette	3/3 p. 7
MIDIWS	12 XBIOS	Envoi d'une chaine à l'interface Midi	3/3 p. 8
MPFINT	13 XBIOS	Vectorisation d'une interruption MFP	3/3 p. 9
IOREC	14 XBIOS	Renvoie l'adresse du tableau d'organisation des périphériques clavier midi et RS232	3/3 p. 9
RSCCONF	15 XBIOS	Reconfiguration totale de l'interface RS232	3/3 p. 11
KEYBLT	16 XBIOS	Reconfiguration complète du clavier	3/3 p. 13
RANDOM	17 XBIOS	Génère un nombre aléatoire de 24 bits	3/3 p. 13
PROTOBT	18 XBIOS	Autorise la création d'un boot sector en mémoire	3/3 p. 14
FLOPVER	19 XBIOS	Vérification de secteurs sur une piste	3/3 p. 15
SCRDMP	20 XBIOS	Permet la copie d'écran sur imprimante	3/3 p. 16
CURSCONF	21 XBIOS	Paramétrage du curseur	3/3 p. 16
SETTIME	22 XBIOS	Réglage de l'horloge interne	3/3 p. 17
GETTIME	23 XBIOS	Renvoie la date et l'heure	3/3 p. 17
BIOSKEYS	24 XBIOS	Remise au standard de la configuration clavier	3/3 p. 18
IKBDWS	25 XBIOS	Envoi de commandes au processeur du clavier	3/3 p. 18
JDSINT	26 XBIOS	Blocage d'une interruption gérée par le MFP 68901	3/3 p. 18
JENABINT	27 XBIOS	Déblocage d'une interruption du MFP 68901	3/3 p. 19
GIACCESS	28 XBIOS	Lecture/écriture dans les registres du processeur sonore	3/3 p. 19 et 6/1.2
OFFGIBIT	29 XBIOS	Effacement des bits du port A du processeur sonore	3/3 p. 20 et 6/1.2
ONGIBIT	30 XBIOS	Mise à un des bits du port A du processeur sonore	3/3 p. 20 et 6/1.2
XBTIMER	31 XBIOS	Programmation d'un des 4 timers du MFP 68901	3/3 p. 21
DOSOUND	32 XBIOS	Programmation simple du processeur sonore	3/3 p. 22 et 6/1.2
SETPRT	33 XBIOS	Configuration de l'imprimante	3/3 p. 23
KBDBASE	34 XBIOS	Renvoie l'adresse d'une table de vecteurs processeurs clavier et midi	3/3 p. 23
KBRATE	35 XBIOS	Réglage latence du clavier	3/3 p. 24
PTRBLK	36 XBIOS	Primitive de SCRDMP	3/3 p. 24



Fonction OS	Numéro Appartenance	Description de la fonction	Les détails se trouvent :
WVBL	37 XBIOS	Attente de la prochaine interruption VBL	3/3 p. 25
SUPEXEC	38 XBIOS	Exécution d'une routine en mode supervisor depuis le mode USER	3/3 p. 25
PUNTAES	39 XBIOS	Non utilisable sur les ST disposant du TOS en ROM	3/3 p. 25
PTERMO	0 GEMDOS	Passe la main au programme appelant	3/4 p. 4
CCONIN	1 GEMDOS	Lecture d'un caractère sur l'entrée standard (clavier)	3/4 p. 4
CCONOUT	2 GEMDOS	Envoi d'un caractère sur la sortie standard (écran)	3/4 p. 5
CAUXIN	3 GEMDOS	Lecture d'un caractère sur l'entrée auxiliaire (port série)	3/4 p. 5
CAUXOUT	4 GEMDOS	Ecriture d'un caractère sur la sortie auxiliaire (port série)	3/4 p. 5
CPRNOUT	5 GEMDOS	Ecriture d'un caractère sur le port parallèle	3/4 p. 6
CRAWIO	6 GEMDOS	Ecriture/lecture sur l'entrée/sortie standard (clavier/écran)	3/4 p. 6
CRAWIN	7 GEMDOS	Lecture sur l'entrée standard (clavier)	3/4 p. 7
CNECIN	8 GEMDOS	Lecture d'un caractère sur l'entrée standard (clavier)	3/4 p. 7
CCONWS	9 GEMDOS	Ecriture d'une chaîne de caractères sur la sortie standard (écran)	3/4 p. 8
CCONRS	10 GEMDOS	Lecture d'une chaîne de caractères sur l'entrée/sortie standard (clavier/écran)	3/4 p. 8
CCONIS	11 GEMDOS	Test de l'état de l'entrée standard	3/4 p. 9
DSETDRV	14 GEMDOS	Sélection du lecteur courant	3/4 p. 10
CCONOS	16 GEMDOS	Test de l'état de la sortie standard	3/4 p. 10
CPRNOS	17 GEMDOS	Test de l'état de la sortie PRN (imprimante en général)	3/4 p. 11
CAUXIS	18 GEMDOS	Test de l'état de l'entrée/sortie AUX (généralement le port série)	3/4 p. 11
CAUXOS	19 GEMDOS	Test de la sortie standard AUX	3/4 p. 12
DGETDRV	25 GEMDOS	Renvoie le numéro du lecteur courant	3/4 p. 12
FSETDTA	26 GEMDOS	Définition d'un tampon utilisé par FSFIRST et FSNEXT	3/4 p. 13
SUPER	32 GEMDOS	Commutation du 68000 en mode supervisor	3/4 p. 13
TGETDATE	42 GEMDOS	Lecture de l'heure stockée dans GEMDOS	3/4 p. 14
TSETDATE	43 GEMDOS	Ecriture de la date dans GEMDOS	3/4 p. 15
TGETTIME	44 GEMDOS	Lecture de l'heure stockée dans GEMDOS	3/4 p. 15
TSETTIME	45 GEMDOS	Ecriture de l'heure dans GEMDOS	3/4 p. 16
FGETDTA	47 GEMDOS	Renvoie l'adresse du tampon utilisé par FSFIRST et FSNEXT (voir FSETDTA #26)	3/4 p. 17

Fonction OS	Numéro Appartenance	Description de la fonction	Les détails se trouvent :
SVERSION	48 GEMDOS	Renvoie le numéro de la version GEMDOS	3/4 p. 17
PTERMRES	49 GEMDOS	Fin d'un programme mise en mode résident	3/4 p. 18
DFREE	54 GEMDOS	Lecture de la taille espace libre sur disquette	3/4 p. 18
DCREATE	57 GEMDOS	Création d'un répertoire	3/4 p. 19
DDELETE	58 GEMDOS	Destruction d'un répertoire	3/4 p. 20
DSETPATH	59 GEMDOS	Définition du répertoire courant	3/4 p. 20
FCREATE	60 GEMDOS	Création d'un fichier	3/4 p. 21
FOPEN	61 GEMDOS	Ouverture d'un fichier existant	3/4 p. 22
FCLOSE	62 GEMDOS	Fermeture d'un fichier	3/4 p. 22
FREAD	63 GEMDOS	Stockage en mémoire d'une partie de fichier	3/4 p. 23
FWRITE	64 GEMDOS	Ecriture d'une partie de fichier préalablement écrite en mémoire	3/4 p. 24
FDELETE	65 GEMDOS	Destruction d'un fichier	3/4 p. 25
FSEEK	66 GEMDOS	Positionnement dans un fichier	3/4 p. 26
FATTRIB	67 GEMDOS	Lecture/écriture des attributs d'un fichier	3/4 p. 27
FDUP	69 GEMDOS	Duplication d'un identificateur standard	3/4 p. 28
FFORCE	70 GEMDOS	Redirection d'un identificateur standard	3/4 p. 29
DGETPATH	71 GEMDOS	Chargement du nom du répertoire d'un lecteur	3/4 p. 29
MALLOC	72 GEMDOS	Réservation d'un bloc mémoire par l'utilisateur	3/4 p. 30
MFREE	73 GEMDOS	Restitution à GEMDOS d'un bloc mémoire (demandé auparavant par Malloc)	3/4 p. 31
MSHRINK	74 GEMDOS	Réduction de la taille d'un bloc mémoire alloué	3/4 p. 31
PEXEC	75 GEMDOS	Opérations diverses de pointage et lancement de fichiers	3/4 p. 32
PTERM	76 GEMDOS	Fin d'un programme depuis un autre programme avec passage d'un diagnostic de fin	3/4 p. 34
FSFIRST	78 GEMDOS	Recherche d'un fichier sur disque (première occurrence)	3/4 p. 34
FSNEXT	79 GEMDOS	Recherche des occurrences suivantes d'un fichier (suite de FSFIRST)	3/4 p. 35
FRENAME	86 GEMDOS	Renomme un fichier	3/4 p. 36
FDATIME	87 GEMDOS	Ecriture/lecture de l'heure/date de création d'un fichier	3/4 p. 36

1/2.2

Index 2 : Les messages d'erreur sur Atari ST

Nom générique	Code erreur	Signification de l'erreur	Appartenance	Concerne
ERROR	-1	Erreur système	XBIOS	3/3
EDVRNR	-2	Lecteur non prêt	XBIOS	3/3
EUNCMD	-3	Commande inconnue	XBIOS	3/3
E_CRC	-4	Erreur CRC	XBIOS	3/3
EBARRQ	-5	Commande incorrecte	XBIOS	3/3
E_SEEK	-6	Piste introuvable	XBIOS	3/3
EMEDIA	-7	Format disquette incorrect	XBIOS	3/3
ESECNF	-8	Secteur introuvable	XBIOS	3/3
EPAPER	-9	Pas de papier imprimante	XBIOS	3/3
EWRITF	-10	Erreur durant l'écriture	XBIOS	3/3
EREADF	-11	Erreur durant la lecture	XBIOS	3/3
	-12	Erreur générale	XBIOS	3/3
EWPRO	-13	Disquette protégée en écriture	XBIOS	3/3
E_CHNG	-14	Disquette changée	XBIOS	3/3
EUNDEV	-15	Périphérique inconnu	BIOS	3/2
EBADSF	-16	Mauvais formatage	BIOS	3/2
EOTHER	-17	Insérer une autre disquette	BIOS	3/2
EINVFN	-32	Numéro de fonction invalidé	GEMDOS	3/4
EFILNF	-33	Fichier introuvable	GEMDOS	3/4
EPTHNF	-34	Chemin introuvable	GEMDOS	3/4
ENHNDL	-35	Plus d'identificateur	GEMDOS	3/4
EACCDN	-36	Accès interdit	GEMDOS	3/4
EIHNDL	-37	Identificateur invalide	GEMDOS	3/4
ENSHEM	-39	Mémoire insuffisante	GEMDOS	3/4
EIMBA	-40	Adresse bloc mémoire invalide	GEMDOS	3/4
EDRIVE	-46	Disque invalide	GEMDOS	3/4
ENMFIL	-49	Plus de fichier	GEMDOS	3/4
ERANGE	-64	Erreur de champ	GEMDOS	3/4
EINTRN	-65	Erreur interne GEMDOS	GEMDOS	3/4
EPLFMT	-66	Fichier non exécutable	GEMDOS	3/4
EGSBF	-67	Erreur accroissement bloc mémoire	GEMDOS	3/4
—	0-1	Erreur non définie	ST BASIC	4/2.1
—	2	Erreur de syntaxe	ST BASIC	
—	3	Return sans GOSUB	ST BASIC	
—	4	Pas assez de data	ST BASIC	
—	5	Appel de fonction illégal	ST BASIC	
—	6	Dépassement de capacité	ST BASIC	

Nom générique	Code erreur	Signification de l'erreur	Appartenance	Concerne
	7	Pas assez de mémoire	ST BASIC	
	8	Numéro de ligne inexistant	ST BASIC	
	9	Valeur d'indice hors limite	ST BASIC	
	10	Définition dupliquée	ST BASIC	
	11	Division par zéro	ST BASIC	
	12	Commande illégale en mode direct	ST BASIC	
	13	Type incorrect	ST BASIC	
	14	Erreur non définie	ST BASIC	
	15	Chaîne trop complexe	ST BASIC	
	16	Expression trop longue	ST BASIC	
	17	Continue seulement après Break	ST BASIC	
	18	Fonction utilisateur non définie	ST BASIC	
	19	Erreur non définie	ST BASIC	
	20	Résumé utilisé sans erreur	ST BASIC	
	21	Erreur non définie	ST BASIC	
	22	Opérande manquant	ST BASIC	
	23	Ligne de programme trop longue	ST BASIC	
	24 à 49	Erreur non définie	ST BASIC	
	50	Dépassement de capacité d'un champ	ST BASIC	
	51	Longueur d'enregistrement invalide	ST BASIC	
	52	Numéro de fichier invalide	ST BASIC	
	53	Fichier non trouvé	ST BASIC	
	54	Mode de fichier non valable	ST BASIC	
	55	Fichier déjà ouvert	ST BASIC	
	56	Erreur non définie	ST BASIC	
	57	Erreur périphérique entrée/sortie	ST BASIC	
	58	Fichier existant	ST BASIC	
	59	Impossibilité de créer un fichier	ST BASIC	
	60	Erreur non définie	ST BASIC	
	61	Disque plein	ST BASIC	
	62	Fin de fichier rencontrée	ST BASIC	
	63	Numéro d'enregistrement non valable	ST BASIC	
	64	Nom de fichier non valable	ST BASIC	
	65	Caractère non valable dans un programme	ST BASIC	
	66	Commande directe dans un fichier	ST BASIC	
	67	Destruction défaillante (kill)	ST BASIC	
	68	Périphérique non disponible	ST BASIC	
	69 à 92	Erreur non définie	ST BASIC	
	93	Segment non défini	ST BASIC	
	94	Fichier protégé	ST BASIC	
	95	Il ne s'agit pas d'un programme Basic	ST BASIC	
	96 à 98	Erreur non définie	ST BASIC	
	99	Arrêt (break)	ST BASIC	
	100	Erreur non définie	ST BASIC	
	101	Programme trop important	ST BASIC	
	102	Erreur non définie	ST BASIC	
	103	Numéro de ligne non valable	ST BASIC	
	104	Numéro de ligne manquant	ST BASIC	
	105	Erreur non définie	ST BASIC	
	106	Déclaration introuvable	ST BASIC	



Nom générique	Code erreur	Signification de l'erreur	Appartenance	Concerne
—	107	Dépassement de capacité pour un entier	ST BASIC	
—	108	Refaire depuis le début	ST BASIC	
—	109	Stop	ST BASIC	
—	110	Imbrications trop nombreuses de GOSUB	ST BASIC	
—	111	Chargement bload de fichier non valable	ST BASIC	
—	112 à 200	Erreur non définie	ST BASIC	
—	201	Option non valable	ST BASIC	
—	202	Commande non autorisée ici	ST BASIC	
—	203	Numéro de ligne requis	ST BASIC	
—	204	For sans next	ST BASIC	
—	206	Virgule manquante	ST BASIC	
—	207	Parenthèse manquante	ST BASIC	
—	208	Option de base doit être 0 ou 1	ST BASIC	
—	209	Erreur non définie	ST BASIC	
—	210	Whiel sans wend	ST BASIC	
—	211	Wend sans while	ST BASIC	
—	212	Erreur non définie	ST BASIC	
—	213	Définition de fonction dupliquée	ST BASIC	
—	214	Saut non valable dans une boucle	ST BASIC	
—	215	Numéro de ligne dupliqué	ST BASIC	
—	216	Etiquette (label) dupliquée	ST BASIC	
—	217 à 220	Erreur non définie	ST BASIC	
—	221	Erreur système	ST BASIC	
—	222	Programme non lancé	ST BASIC	
—	223	Trop de boucles For	ST BASIC	
—	224 à 230	Erreur non définie	ST BASIC	
—	0	Division par zéro	GFA BASIC	4/2.2
—	1	Dépassement de capacité	GFA BASIC	
—	2	Ce nombre n'est pas un entier	GFA BASIC	
—	3	Ce nombre n'est pas codable sur 1 octet	GFA BASIC	
—	4	Ce nombre n'est pas un mot (16 bits)	GFA BASIC	
—	5	Racine carrée d'un nombre négatif	GFA BASIC	
—	6	Logarithme d'un nombre négatif	GFA BASIC	
—	7	Erreur non définie	GFA BASIC	
—	8	Mémoire pleine	GFA BASIC	
—	9	Instruction non valable	GFA BASIC	
—	10	Chaîne trop longue	GFA BASIC	
—	11	Ce programme n'est pas écrit en GFA	GFA BASIC	
—	12	Programme trop important	GFA BASIC	
—	14	Champ dupliqué	GFA BASIC	
—	15	Champ non défini	GFA BASIC	
—	16	Dépassement d'index de champ	GFA BASIC	
—	17	Indice de dimension trop grand	GFA BASIC	
—	18	Nombe d'indices incorrect	GFA BASIC	
—	19	Procédure introuvable	GFA BASIC	
—	20	Marque introuvable	GFA BASIC	
—	21	Utilisation d'options I, O, R ou A (open)	GFA BASIC	

Nom générique	Code erreur	Signification de l'erreur	Appartenance	Concerne
	22	Fichier déjà ouvert	GFA BASIC	
	23	Numéro de fichier incorrect	GFA BASIC	
	24	Fichier fermé	GFA BASIC	
	25	Saisie numérique incorrecte	GFA BASIC	
	26	Fin de fichier rencontrée	GFA BASIC	
	27	Trop de points pour instructions « POLY »	GFA BASIC	
	28	Un champ ne peut avoir qu'une dimension	GFA BASIC	
	29	Nombre de points plus grand que le champ	GFA BASIC	
	30	Fichier non ASCII (merge)	GFA BASIC	
	31	Ligne trop longue (merge)	GFA BASIC	
	32	Erreur de syntaxe	GFA BASIC	
	33	Marque non définie	GFA BASIC	
	34	Données manquantes	GFA BASIC	
	35	Donnée non numérique	GFA BASIC	
	37	Disque plein	GFA BASIC	
	38	Instruction non valable en mode direct	GFA BASIC	
	39	GOSUB impossible	GFA BASIC	
	40	Utilisation de clear interdite (boucle)	GFA BASIC	
	41	Continue non valable	GFA BASIC	
	42	Paramètres manquants	GFA BASIC	
	43	Expression trop complexe	GFA BASIC	
	44	Fonction non définie	GFA BASIC	
	45	Trop de paramètres	GFA BASIC	
	46	Type de paramètre incorrect (numérique)	GFA BASIC	
	47	Type de paramètre incorrect (chaîne)	GFA BASIC	
	48	Enregistrement trop long (accès direct)	GFA BASIC	
	49	Trop de fichiers à accès direct	GFA BASIC	
	50	Pas de fichier à accès direct	GFA BASIC	
	52	Champ trop long	GFA BASIC	
	54	Longueur d'enregistrement incorrecte	GFA BASIC	
	55	Numéro d'enregistrement incorrect	GFA BASIC	
	60	Longueur de chaîne sprite incorrecte	GFA BASIC	
	62	Erreur dans le menu	GFA BASIC	
	63	Erreur réserve	GFA BASIC	
	64	Erreur pointeur	GFA BASIC	
	65	Longueur de champ supérieure à 255	GFA BASIC	
	67	Erreur dans le calcul d'arc SIN/ARC COS	GFA BASIC	
	69	Endfunc sans return	GFA BASIC	
	71	Index trop grand	GFA BASIC	
	90	Erreur local	GFA BASIC	
	91	Erreur For	GFA BASIC	
	92	Résumé impossible	GFA BASIC	
	95	Erreur dans la pile	GFA BASIC	
	102	Erreur sur le bus	GFA BASIC	
	103	Erreur d'adresse (DPEEK/DPOKE)	GFA BASIC	

Nom générique	Code erreur	Signification de l'erreur	Appartenance	Concerne
—	104	Instruction 68000 inadéquate	GFA BASIC	
—	105	Division par zéro en langage 68000	GFA BASIC	
—	106	Interruption 68000 par chk	GFA BASIC	
—	107	Interruption 68000 par trapv	GFA BASIC	
—	108	Interruption 68000 par inst. privilégiée	GFA BASIC	
—	109	Interruption 68000 par trace	GFA BASIC	



1/3.0

Annexe 0
Plus loin

Ce nouveau thème est destiné à alimenter chacune des parties de votre ouvrage. Vous retrouverez le titre chapeau *Plus loin* dans les parties où ce thème est développé. Par exemple, en partie 6, le thème *Plus loin* a été développé avec les capacités sonores du STE (Partie 6, chapitre 3.1).

Aujourd'hui, la grande nouveauté chez Atari étant le STE, *Plus loin* se consacre exclusivement à lui. Les études détaillées des circuits ou encore les programmes spécifiques continueront ainsi à être intégrés dans la structure naturelle de l'ouvrage, mais cette ségrégation permettra aux lecteurs ne disposant pas du STE de ne pas être déçus par la frappe d'un programme spécifique ne fonctionnant pas sur les ST standards, ainsi qu'aux heureux possesseurs de STE d'aller tout droit aux chapitres traitant des spécificités de leur machine.

Notre souci constant de faire de votre ouvrage un guide d'accès rapide et précis se trouve ainsi satisfait.

1/3.0.1

Le nouvel Atari STE

Depuis près de 5 ans, Atari tient le haut du pavé de l'actualité avec sa gamme d'ordinateurs développée autour d'un excellent microprocesseur, le 68000 de Motorola.

La montée en puissance de la gamme s'est faite très tôt avec les Mega ST : c'était à l'époque où l'on mesurait la puissance de l'ordinateur par la taille de sa mémoire. Depuis peu, cette débauche de kilo-octets fait place à d'autres qualités telles que l'affichage haut en couleurs et en résolution ou encore la

génération de bruits, sons et musiques de plus en plus sophistiqués. Cette tendance, imposée essentiellement par le regain d'intérêt du public pour les jeux, tant d'arcade que d'aventure, trouve sa pleine expression dans la conception du dernier né d'Atari, à savoir le maintenant célèbre STE.

A la fois différent et semblable à ses aînés, le STE présente des qualités qui en font une machine tout à fait intéressante.

Votre ouvrage *Comment améliorer toutes les ressources et augmenter les performances de votre Atari ST 520/1040 et Méga* se devait de disséquer ce nouveau venu aussi bien sur le plan de l'anatomie que de l'intellect. Vous trouverez donc dans chacune des parties spécialisées de votre ouvrage, si, bien sûr, les différences du STE avec les autres machines le justifient, un chapitre traitant de ces différences.

Ce thème est ainsi aujourd'hui abordé par :

1/3.0 Annexe 0 Plus loin
1/3.0.1 Le nouvel Atari STE
1/3.0.2 Vive la différence. Compatibilité jusqu'où ?

2/0 Plus loin
2/0.1 L'anatomie de l'Atari STE

3/0 Plus loin
3/0.1 Le nouveau TOS 1.6 de l'Atari STE

5/0 Plus loin
5/0.1 L'affichage graphique de l'Atari STE

6/3 Plus loin
6/3.1 Les possibilités sonores de l'Atari STE
(publié au complément 5)

1/3.0.2

Vive la différence. Compatibilité jusqu'où ?[1)]

Il nous a semblé important de rassembler, sous forme de tableaux, les principales différences ainsi que les incompatibilités et compatibilités connues et vérifiées entre les gammes ST et STE. Consultant ces tableaux, le lecteur pressé ou désireux d'acquérir un STE saura alors très rapidement à quoi il s'expose ou quels seront les bénéfices d'un tel achat.

I - Caractéristiques principales de l'Atari STE

• Unité centrale, configuration de base

— Microprocesseur Motorola MC68000 16 bits

— RAM 512 Ko extensible à 4 Mégaoctets (Barrettes SIMM)

— ROM 256 Ko

— Lecteur de disquettes 3,5 pouces double face 720 Ko

— Intégrateur graphique GEM, système d'exploitation TOS 1.6

— Câble sortie péritel

— Souris Atari

— 10 circuits coprocesseurs : Blitter, Shifter, GLUE/MMU, DMA, MFP68901, FD1772, YM2149, LMC1992, 2 ACIAS

• Clavier

— AZERTY/QWERTY selon versions 85 touches + 10 touches de fonction

— Pavé numérique 18 touches

— Pavé touches de commandes curseur

• Affichage

— Sortie couleur RVB et péritel

— Synchro source extérieure possible

— 16 niveaux pour chaque couleur de base R, V, B

— Palette de 4096 couleurs

1) Les éditions WEKA remercient ATARI MAGAZINE de les avoir autorisées à reprendre certaines informations publiées dans le Dossier STE du numéro 6 du mois de novembre 1989.

— Résolutions : 640×200 4 couleurs
320×200 16 couleurs

• Génération sonore

— 2 processeurs musicaux (YM 2149 et LMC 1992)

— Sorties stéréo

— Sortie son numérique format PCM 8 bits signés

— Générateur de bruits

— Interface MIDI

— 4 fréquences d'échantillonnage (6,25 12,5 25 et 50 kilohertz)

• Interfaces

— Vidéo monochrome haute résolution 640×400 pixels

— Lecteur de disquettes externe

— Série (RS232)

— Parallèle (Centronics)

— 2 interfaces manettes 15 broches

— 2 interfaces manettes 9 broches (Standard ST)

— Port cartouche (Standard ST)

— Port disque dur (vitesse 10 Mégabits/s)

— 2 prises son RCA

II - Les principales différences matérielles et performances

	520ST	520STF	1040STF	MegaST1	520STE
Année de sortie	1985	1988	1988	1988	1989
RAM	516	512	1024	1024	512
RAM extensible	non	non	non	non	oui 4Mo
ROM	64	192	192	192	256
Alimentation	Externe	Interne	Interne	Interne	Interne
Lecteur disquette	Externe	Interne	Intégré	Intégré	Interne
Capacité lecteur	360 Ko	360 Ko	720 Ko	720 Ko	720 Ko
Système d'exploitation	TOS	TOS1.2	TOS1.2	TOS1.3	TOS1.6
Manettes	2	2	2	2	6
Paddles	0	0	0	0	2
Pistolets	0	0	0	0	2
Blitter	non	non	non	oui	oui
Scrolling	soft	soft	soft	soft	hard
Son numérique	non	non	non	non	oui
Son stéréo	non	non	non	non	oui
Mémoire écran	fixe	fixe	fixe	fixe	extens



III - Compatibilité jusqu'où ?

En règle générale, lorsqu'un concepteur d'ordinateurs travaille sur un modèle complétant une gamme existante, il essaie de respecter les règles de compatibilité ascendante. C'est-à-dire qu'à quelques exceptions près, tous les périphériques, extensions ou logiciels développés pour l'ancienne version doivent tourner sans problème sur le nouveau venu.

Ces règles sont souvent transgressées soit du fait du concepteur d'ordinateur qui n'a pas forcément connaissance de tous les logiciels du marché, soit du fait du concepteur de logiciel qui n'a pas toujours respecté les règles propres à lui assurer cette compatibilité en utilisant des astuces de programmation qui rendront son logiciel un peu marginal.

Ces problèmes de compatibilité sont souvent résolus par les fabricants sérieux qui proposent des échanges contre des versions compatibles à des prix relativement intéressants.

Dans tous les cas, lorsque vous rencontrez ce type de problème, nous ne saurions que trop vous encourager à nous les indiquer afin d'éviter ce genre de désagrément au plus grand nombre d'entre vous, et à vous adresser à votre distributeur pour négocier un échange.

LES PÉRIPHÉRIQUES

Les périphériques classiques de marque Atari comme l'imprimante laser ATARI SLM804, le disque dur MEGAFILE et le CD-ROM Atari fonctionnent sans aucun problème sur les STE.

Les échantillonneurs, Master Sound, ST Replay 4 et STOS Maestro ne présentent aucune anomalie tant sur le plan matériel que logiciel.

Les cartes accélératrices, quant à elles, qui se connectent en lieu et place du 68000 d'origine pour le remplacer par une version plus rapide, ne peuvent encore s'implanter sur les STE du fait de la forme carrée du 68000 du STE au lieu de la forme traditionnelle rectangulaire du circuit 40 pattes.

Les émulateurs posent également quelques problèmes. Supercharger, PC Ditto 3.96 fonctionnent parfaitement alors que PC-SPEED rencontre le même handicap de connexion que les cartes accélératrices.

Si SPECTRE 128 et SPECTRE GCR ne posent aucun problème, Aladin, lui, ne fonctionne absolument pas sur STE.

LES LOGICIELS

- **Séquenceurs et éditeurs musicaux**

Les versions de logiciels STEINBERG tournant sous M-ros refusent obstinément de fonctionner. Il faut savoir que M-ros, adaptation du système d'exploitation rendant le ST multitâche, a contraint Steinberg à réécrire une bonne partie de GEM.

PRO 24 version 2.1 fonctionne sans problème sur 520 STE. La version 3.0 réclame l'extension de mémoire possible avec les barettes SIMM.

Aucun autre logiciel musical, séquenceur, éditeur, échantillonneur ou logiciel pédagogique ne semble présenter d'incompatibilité.

Il en est ainsi de Creator, Notator, Track 24, Studio 24 et Pro Score.

Les éditeurs Synthworks de Steinberg, éditeur K1 de PA-decoder, K5 Kawai editor, Pro Sample Editor, Digidesign ou Roland D50 sont entièrement compatibles. Certains, comme Sound designer ou Turbosynth, fonctionnent bien avec une qualité sonore accrue sur le STE.

En ce qui concerne les logiciels pédagogiques comme Orphée, Eurydice, Clé de sol et Dictée musicale, ils ne présentent aucun problème.

- **Les logiciels de graphisme**

Degas Elite, Néochrome, Spectrum 512 n'offrent aucune difficulté d'utilisation sur STE. Ils n'utilisent cependant que la palette de 512 couleurs des ST.

Dali et ZZ Rough existent en version STE et utilisent la palette de 4096 couleurs du STE.

Dans la gamme des logiciels vectoriels, citons Eazy Draw, ZZ draft et ZZ-2D qui fonctionnent parfaitement ainsi que Funface, ZZ-Lazypaint ou ZZ-Scan. Les logiciels 3D CyberSculpt et CyberControl tournent sur STE comme des charmes.

Quelques petits problèmes de compatibilité sont à noter sur GFA-Artist de Micro Application. CyberPaint ne fonctionne que dans la version 1.0 alors que GFA Raytrace, quant à lui, génère 3 bombes immanquablement et vous renvoie au bureau GEM avant même la fin de son chargement.

- **Les utilitaires bureautiques**

Les bons élèves sont LDW Power, Calcomat, Superbase II, Superbase PRO III, Adimens, First Word Plus, Calamus, Timeworks, Emulcom 3. Le Rédacteur, élève dissipé, ne fonctionne que dans sa version 1.98... mais pas toutes, car il y en a plusieurs... Difficile de s'y retrouver !

TurboST, G++ et GDOS fonctionnent parfaitement quant à eux.

- **Les jeux**

Difficile de faire un tour exhaustif de tous les jeux, et c'est sans doute là que les surprises (mauvaises) seront les plus nombreuses. C'est en effet dans l'écriture de programmes de jeux que les concepteurs prennent le plus de liberté avec les spécifications d'Atari comme le non respect des points d'entrée par exemple, ou encore parce que le concepteur a protégé son logiciel au prix du sacrifice de ces spécifications. Le serveur Atari vous donne la liste des logiciels présentant quelques difficultés sur le STE mais rassurez-vous, cette liste reste limitée.



1/3

Annexes

1/3.1

Annexe 1 : Les codes ASCII des caractères Atari

DEC	HEX	ASCII	CAR. CTRL	Origine, fonction ou nom
0	0	NUL	^@	Nul
1	1	SOH	^A	Start of heading
2	2	STX	^B	Start of text
3	3	ETX	^C	End of text
4	4	EOT	^D	End of transmission
5	5	ENQ	^E	Enquiry
6	6	ACK	^F	Acknowledge
7	7	BEL	^G	Bell
8	8	BS	^H	Back space

DEC	HEX	ASCII	CAR. CTRL	Origine, fonction ou nom
9	9	HT	^I	Tab horizontally
10	0A	LF	^J	Line Feed
11	0B	VT	^K	Tab vertically
12	0C	FF	^L	Form feed
13	0D	CR	^M	Carriage return
14	0E	SO	^N	Shift out
15	0F	SI	^O	Shift in
16	10	DLE	^P	Data link escape
17	11	DC1	^Q	Device control 1
18	12	DC2	^R	Device control 2
19	13	DC3	^S	Device control 3
20	14	DC4	^T	Device control 4
21	15	NAK	^U	Negative acknowledge
22	16	SYN	^V	Synchronous idle
23	17	ETB	^W	End of transmitted block
24	18	CAN	^X	Cancel line
25	19	EM	^Y	End of medium
26	1A	SUB	^Z	Substitute
27	1B	ESC	^[	Escape
28	1C	FS	^\	File separator
29	1D	GS	^]	Group separator
30	1E	RS	^^	Record separator
31	1F	US	^_	Unit separator
32	20	SPACE		
33	21	!		
34	22	"		Double quote
35	23	#		Cross-Hatch
36	24	$		Dollar
37	25	%		
38	26	&		Ampersand



DEC	HEX	ASCII	CAR. CTRL	Origine, fonction ou nom
39	27	'		Quote
40	28	(		
41	29	)		
42	2A	*		
43	2B	+		
44	2C	,		
45	2D	-		
46	2E	.		
47	2F	/		Slash
48	30	0		
49	31	1		
50	32	2		
51	33	3		
52	34	4		
53	35	5		
54	36	6		
55	37	7		
56	38	8		
57	39	9		
58	3A	:		
59	3B	;		
60	3C	<		
61	3D	=		
62	3E	>		
63	3F	?		
64	40	@		Arrobas
65	41	A		
66	42	B		
67	43	C		
68	44	D		

DEC	HEX	ASCII	CAR. CTRL	Origine, fonction ou nom
69	45	E		
70	46	F		
71	47	G		
72	48	H		
73	49	I		
74	4A	J		
75	4B	K		
76	4C	L		
77	4D	M		
78	4E	N		
79	4F	O		
80	50	P		
81	51	Q		
82	52	R		
83	53	S		
84	54	T		
85	55	U		
86	56	V		
87	57	W		
88	58	X		
89	59	Y		
90	5A	Z		
91	5B	[		Crochet ouvert
92	5C	\		Anti-slash
93	5D	]		Crochet fermé
94	5E	^		
95	5F	_		
96	60	`		
97	61	a		
98	62	b		



DEC	HEX	ASCII	CAR. CTRL	Origine, fonction ou nom
99	63	c		
100	64	d		
101	65	e		
102	66	f		
103	67	g		
194	68	h		
105	69	i		
106	6A	j		
107	6B	k		
108	6C	l		
109	6D	m		
110	6E	n		
111	6F	o		
112	70	p		
113	71	q		
114	72	r		
115	73	s		
116	74	t		
117	75	u		
118	76	v		
119	77	w		
120	78	x		
121	79	y		
122	7A	z		
123	7B	{		
124	7C	¦		
125	7D	}		
126	7E	~		Tilde
127	7F	Δ		Delta Grec
128	80	Ç		

DEC	HEX	ASCII	CAR. CTRL	Origine, fonction ou nom
129	81	ü		
130	82	é		
131	83	â		
132	84	ä		
133	85	à		
134	86	å		
135	87	ç		
136	88	ê		
137	89	ë		
138	8A	è		
139	8B	ï		
140	8C	î		
141	8D	ì		
142	8E	Ä		
143	8F	Å		
144	90	É		
145	91	æ		
146	92	Æ		
147	93	ô		
148	94	ö		
149	95	ò		
150	96	û		
151	97	ù		
152	98	ÿ		
153	99	Ö		
154	9A	Ü		
155	9B	¢		
156	9C	£		Livre anglaise
157	9D	¥		Yen japonais
158	9E	P_t		



DEC	HEX	ASCII	CAR. CTRL	Origine, fonction ou nom
159	9F	ƒ		
160	A0	á		
161	A1	í		
162	A2	ó		
163	A3	ú		
164	A4	ñ		N tilde minuscule
165	A5	Ñ		N tilde majuscule
166	A6	ª		
167	A7	º		
168	A8	¿		
169	A9	⌐		
170	AA	¬		
171	AB	½		
172	AC	¼		
173	AD	¡		
174	AE	«		
175	AF	»		
176	B0	ã		A tilde
177	B1	õ		O Tilde
178	B2	ø		Grand O barré
179	B3	ϕ		Petit o barré
180	B4	œ		O E dans l'O minuscule
181	B5	Œ		O E dans l'O majuscule
182	B6	À		A accent grave majuscule
183	B7	Ã		A tilde majuscule
184	B8	Õ		O tilde majuscule
185	B9	¨		Tréma
186	BA	´		
187	BB	⊤		
188	BC	¶		

DEC	HEX	ASCII	CAR. CTRL	Origine, fonction ou nom	
189	BD	©		Copyright	
190	BE	®		Registered	
191	BF	™		Trademark	
192	C0	﬚			
193	C1	﬛			
194	C2	א		ALEF	Hébreu son H
195	C3	ב		BET	Hébreu son B
196	C4	ג		GUIMEL	Hébreu son GUE
197	C5	ד		DALETH	Hébreu son D
198	C6	ה		HE	Hébreu son HE
199	C7	ו		VAV	Hébreu son V
200	C8	ז		ZAïN	Hébreu son Z
201	C9	ח		H'ET	Hébreu R GUTTURAL
202	CA	ט		TET	Hébreu son T
203	CB	י		YOUD	Hébreu son I
204	CC	כ		KAF	Hébreu son QUE
205	CD	ל		LAMED	Hébreu son L
206	CE	מ		MEM	Hébreu son M
207	CF	נ		NOUN	Hébreu son N
208	D0	ס		SAMER	Hébreu son S
209	D1	ע		AïN	Hébreu son A
210	D2	פ		PE	Hébreu son P
211	D3	צ		TSADIK	Hébreu son TSE
212	D4	ק		KOUF	Hébreu son K
213	D5	ר		RECH	Hébreu son R
214	D6	ש		CHIN	Hébreu son CHE
215	D7	ת		TAV	Hébreu son T
216	D8	ן		Fin de mot hébreu	
217	D9	ך		Fin de mot hébreu	
218	DA	ם		Fin de mot hébreu	
219	DB	ף		Fin de mot hébreu	



DEC	HEX	ASCII	CAR. CTRL	Origine, fonction ou nom
220	DC	ך		Fin de mot hébreu
221	DD	§		
222	DE	Λ		LAMBDA Grec
223	DF	∞		Symbole de l'infini
224	E0	α		ALPHA Grec
225	E1	β		BETA Grec
226	E2	Γ		GAMMA Grec
227	E3	Π		PI Grec
228	E4	Σ		SIGMA Grec
229	E5	σ		Petit SIGMA
230	E6	μ		MU Grec
231	E7	t		TAU Grec
232	E8	Φ		
233	E9	Θ		THETA Grec
234	EA	Ω		OMEGA Grec
235	EB	δ		DELTA Grec
236	EC	ϕ		
237	ED	ø		PHI Grec
238	EE	ε		EPSILON Grec
239	EF	∩		
240	F0	≡		
241	F1	±		
242	F2	≥		
243	F3	≤		
244	F4	⌠		1/2 signe intégral sup.
245	F5	⌡		1/2 signe intégral inf.
246	F6	÷		
247	F7	≈		Symbole de environ
248	F8	°		
249	F9	•		

DEC	HEX	ASCII	CAR. CTRL	Origine, fonction ou nom
250	FA	·		
251	FB	√		Symbole racine
252	FC	n		Puissance N
253	FD	²		Puissance 2
254	FE	³		Puissance 3
255	FF			

1/3.2

Annexe 2 : La table des SCANCODES des claviers Atari

TOUCHE	F1	F2	F3	F4	F5	F6	F7	F8	F9	F0											
CODE	3B	3C	3D	3E	3F	40	41	42	43	44											
TOUCHE	ESC	1	2	3	4	5	6	7	8	9	0	°	_	£	BS	HELP	UNDO	(	)	/	*	
CODE	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	29	0E	62	61	63	64	65	66	
TOUCHE	TAB	A	Z	E	R	T	Y	U	I	O	P	¨	£		DEL	INS	↑	CLR	7	8	9	-
CODE	0F	10	11	12	13	14	15	16	17	18	19	1A	1B		53	52	48	47	67	68	69	4A
TOUCHE	CTRL	Q	S	D	F	G	H	J	K	L	M	%	RETURN		←	↓	→	+	4	5	6	+
CODE		1E	1F	20	21	22	23	24	25	26	27	28	1C		2B	4B	50	4D	6A	6B	6C	4E
TOUCHE	SHIFT	>	W	X	C	V	B	N	?	.	/	+	SHIFT						1	2	3	ENT
CODE		60	2C	2D	2E	2F	30	31	32	33	34	35							6D	6E	6F	72
TOUCHE		ALT			S	P	A	C	E			CAP							0		.	
CODE							39												70		71	

Le clavier standard azerti.



1/3.3

Annexe 3 : Résumé des codes opératoires du 68000

Dans cette annexe, les codes opératoires sont systématiquement représentés selon le modèle suivant :

Code Op Signification
Syntaxe(s)
Indicateurs modifiés

La ou les syntaxe(s) de chaque code opératoire suit (suivent) les conventions suivantes :

Dx	=	Registre de données source,
Dy	=	Registre de données destination,
Dn	=	Un des registres de données,
Ax	=	Registre d'adresses source,
Ay	=	Registre d'adresses destination,
An	=	Un des registres d'adresses,
<AE>	=	Adresse effective,
# <Donnée>	=	Donnée immédiate,
<Etiquette>	=	Adresse mémoire,
CCR	=	Condition Code Register
SR	=	Registre d'état

Les indicateurs utilisent les conventions de codage suivantes :

*	=	Positionné,
I	=	Indéfini,
-	=	Non affecté,
0	=	Mis à zéro,
1	=	Mis à un.

Code opératoire	Signification
ABCD	Addition décimale avec retenue ABCD Dy,Dx ou ABCD −(Ay), −(Ax) XNZVC = *U*U*
ADD	Addition binaire ADD <AE>,Dn ou ADD Dn,<AE> XNZVC = *****
ADDA	Addition binaire avec une adresse ADDA <AE>,An XNZVC = -----
ADDI	Addition immédiate ADDI #<Donnée>,<AE> XNZVC = *****
ADDQ	Addition rapide ADDQ #<Donnée>,<AE> XNZVC = *****
ADDX	Addition binaire étendue ADDX Dy,Dx ou ADDX −(Ay), −(Ax) XNZVC = *****
AND	ET logique AND <AE>,Dn ou AND Dn,<AE> XNZVC = −**00
ANDI	ET logique immédiat ANDI #<Donnée>, <AE> XNZVC = −**00
ASL	Décalage arithmétique à gauche ASL Dx,Dy ou ASL #<Donnée>,Dy ou ASL <AE> XNZVC = *****
ASR	Décalage arithmétique à droite ASR Dx,Dy ou ASR #<Donnée>,Dy ou ASR <AE> XNZVC = *****
Bcc	Branchement conditionnel Bcc <Etiquette> où cc=CC, CS, EQ, NE, GE, GT, HI, LE, LS, LT, MI, PL, VC, VS, T OU F XNZVC = -----
BCHG	Test d'un bit et complémentation BCHG Dn,<AE> ou BCHG #<Donnée><AE> XNZVC = --*--
BCLR	Test d'un bit et remise à zéro BCLR Dn,<AE> ou BCLR #<Donnée>,<AE> XNZVC = --*--



Code opératoire	Signification
BRA	Branchement inconditionnel BRA <Etiquette> XNZVC = ----
BSET	Test d'un bit et remise à un BSET Dn,<AE> ou BSET #<Donnée>,<AE> XNZVC = --*--
BSR	Branchement à un sous-programme BSR <Etiquette> XNZVC = ----
BTST	Test d'un bit BTST Dn,<AE> ou BTST #<Donnée>,<AE> XNZVC = --*--
CHK	Test du contenu d'un registre CHK <AE>,Dn XNZVC = -*III
CLR	Mise à zéro de l'opérande CLR <AE> XNZVC = -0100
CMP	Comparaison CMP <AE>,Dn XNZVC = -****
CMPA	Comparaison avec un registre d'adresses CMPA <AE>,An XNZVC = -****
CMPI	Comparaison immédiate CMPI #<Donnée>,<AE> XNZVC = -****
CMPM	Comparaison avec le contenu d'une mémoire CMPM (Ay)+,(Ax)+ XNZVC = -****
DBcc	Test d'une condition, décrémentation et branchement en fonction du résultat du test DBcc Dn,<Etiquette> XNZVC = ----
DIVS	Division signée DIVS <AE>,Dn XNZVC = -***0
DIVU	Division non signée DIVU <AE>,Dn XNZVC = -***0

Code opératoire	Signification
EOR	OU exclusif EOR Dn,<AE> XNZVC = −**00
EORI	OU exclusif immédiat EORI #<Donnée>,<AE> XNZVC = -**00
EORI to CCR	OU exclusif immédiat avec CCR EORI #<Donnée>,CCR XNZVC = *****
EORI to SR	OU exclusif immédiat avec SR EORI #<Donnée>,SR XNZVC = *****
EXG	Echange de deux registres EXG Rx,Ry XNZVC = ----
EXT	Extension signée de l'opérande EXT Dn XNZVC = -**00
ILLEGAL	Exception 4 Syntaxe quelconque XNZVC = ----
JMP	Branchement inconditionnel JMP <AE> XNZVC = ----
JSR	Appel d'un sous-programme JSR <AE> XNZVC = ----
LEA	Chargement d'une adresse effective LEA <AE>,An XNZVC = ----
LINK	Registre d'adresse dans la pile LINK An,#<Offset> XNZVC = ----
LSL	Décalage logique vers la gauche LSL, Dx,Dy ou LSL #<Donnée>,Dy ou LSL <AE> XNZVC = ***0*
LSR	Décalage logique vers la droite LSR Dx,Dy ou LSR #<Donnée>,Dy ou LSR <AE> XNZVC = -**00



Code opératoire	**Signification**
MOVE	Opérande source dans l'opérande destination MOVE <AE>,<AE> XNZVC = *****
MOVE to CCR	Opérande source dans le registre des codes condition MOVE <AE>,CCR XNZVC = -----
MOVE to SR	Opérande source dans le registre d'état MOVE <AE>,SR XNZVC = *****
MOVE from SR	Registre d'état dans le registre destination MOVE SR,<AE> XNZVC = -----
MOVE USP	Pointeur de pile dans un registre d'adresses et inversement MOVE USP,A1 ou MOVE A1,USP XNZVC = -----
MOVEA	Opérande source dans un registre d'adresses MOVEA <AE>,An XNZVC = -----
MOVEM	Registres sources transférés à partir de l'adresse effective ou inversement MOVEM <registres>,<AE> ou MOVEM <AE>,<registres> XNZVC = -----
MOVEP	Transfert d'un registre de données dans la mémoire indiquée et inversement MOVEP Dx,d(Ay) ou MOVEP d(Ay),Dx XNZVC = -----
MOVEQ	Transfert rapide d'un registre de données dans la mémoire indiquée MOVEQ # <Donnée>,Dn XNZVC = -**00
MULS	Multiplication signée sur 16 bits MULS <AE>,Dn XNZVC = -**00
MULU	Multiplication non signée sur 16 bits MULU <AE>,Dn XNZVC = -**00
NBCD	Négation décimale étendue NBCD <AE> XNZVC = *-*-*

Code opératoire	Signification
NEG	Négation NEG <AE> XNZVC = *****
NEGX	Négation étendue NEGX <AE> XNZVC = *-*-*
NOP	Pas d'opération NOP XNZVC = -----
NOT	Complémentation logique NOT <AE> XNZVC = -**00
OR	OU logique OR <AE>,Dn ou OR Dn,<AE> XNZVC = -**00
ORI	OU logique immédiat ORI #<Donnée>,<AE> XNZVC = -**00
ORI to CCR	OU logique immédiat avec CCR ORI #<Donnée>,CCR XNZVC = *****
ORI to SR	OU logique immédiat avec SR ORI #<Donnée>,SR XNZVC = *****
PEA	Adresse effective dans la pile PEA <AE> XNZVC = -----
RESET	Remise à zéro du 68000 RESET XNZVC = -----
ROL	Rotation sans extension vers la gauche ROL Dx,Dy ou ROL #<Donnée>,Dy ou ROL <AE> XNZVC = -**0*
ROR	Rotation sans extension vers la droite ROR Dx,Dy ou ROR #<Donnée>,Dy ou ROR <AE> XNZVC = -**0*
ROXL	Rotation avec extension vers la gauche ROXL Dx,Dy ou ROXL #<Donnée>,Dy ou ROXL <AE> XNZVC = ***0*



Code opératoire	Signification
ROXR	Rotation avec extension vers la droite ROXR Dx,Dy ou ROXR #<Donnée>,Dy ou ROXR <AE> XNZVC = ***0*
RTE	Retour d'exception (en mode superviseur), Génération d'un exception (en mode utilisateur) RTE XNZVC = *****
RTR	Retour et restauration des codes condition RTR XNZVC = *****
RTS	Retour d'un sous-programme RTS XNZVC = -----
SBCD	Soustraction décimale étendue SBCD Dy,Dx ou SBCD −(Ay),−(Ax) XNZVC = *-*-*
Scc	Initialisation d'un octet en fonction d'une condition Scc <AE> où cc=CC, CS, EQ, NE, GE, GT, HI, LE, LS, LT, MI, PL, VC, VS, T ou F XNZVC = -----
STOP	Chargement du registre d'état et arrêt (en mode superviseur) STOP #<Donnée> XNZVC = *****
SUB	Soustraction binaire SUB <AE>,Dn ou SUB Dn,<AE> XNZVC = *****
SUBA	Soustraction d'une adresse SUBA <AE>,An XNZVC = -----
SUBI	Soustraction immédiate SUBI #<Donnée>,<AE> XNZVC = *****
SUBQ	Soustraction rapide SUBQ #<Donnée>,<AE> XNZVC = *****
SUBX	Soustraction étendue SUBX Dy,Dx ou SUBX −(Ay),−(Ax) XNZVC = *****

Code opératoire	Signification
SWAP	Echange des mots de poids fort et de poids faible d'un registre SWAP Dn XNZVC = -**00
TAS	Test d'un octet, et mise à un de son bit 7 TAS <AE> XNZVC = -**00
TRAP	Exécution d'une exception TRAP #<Vecteur> XNZVC = -----
TRAPV	Exécution d'une exception si débordement TRAPV XNZVC = -----
TST	Test d'un opérande TST <AE> XNZVC = -**00
UNLK	Transfert d'un registre d'adresses dans le pointeur de pile UNLK An XNZVC = -----



1/3.4

Annexe 4 : Les principales abréviations utilisées dans le monde des Atari

Abréviation	Signification
ABS	Absolute
AC	Analog Channel
ACIA	Asynchronous Communication Interface Adaptor
AES	Application Environment System
ALT	Alternate
ALU	Arithmetic and Logic Unit
ASCII	American Standard for Information Interchange
BC	Bus Controller
BERR	Bus Error
BIOS	Basic Input Output System
BR	Bus Request
CCR	Condition Code Register
CLK	Clock
CLR	Clear
CRC	Cyclic Redundancy Check
CS	Chip Select
CTRL	Control
CTS	Clear To Send
DCR	Data Carrier Detect
DDR	Data Direction Register
DMA	Direct Memory Access
DTACK	Data Transmission Acknoledge
DTR	Data Terminal Ready
EOF	End Of File
EPROM	Erasable Programmable ROM
ESC	Escape
FAT	File Allocation Table
FDC	Floppy Disk Controller
FF	Form Feed
GDOS	Graphic Device Operating System
GEMDOS	Graphic Environment Manager Disk Operating System
GND	Ground

Abréviation	Signification
GPO	General Purpose Output
IACK	Interrupt Acknoledgment
ID	Identification Device
INTR	Interrupt Request
IP	Interruption Priority
I/O	Input/Output
LDS	Lower Data Selection
LF	Line Feed
LSB	Lowest Significant Byte
LSI	Large Scale Integration
MFP	Multi Functions Processor
MIDI	Musical Instrument Digital Interface
MPB	Memory Parameter Block
MSB	Most Significant Byte
NC	Not Connected
PSG	Programmable Sound Generator
RAM	Random Access Memory
RGB	Red Green Blue
ROM	Read Only Memory
RR	Receive Register
RSR	Receiver Status Register
RTSPC	Request To Send
R/W	Read/Write
SCRDMP	Screen Dump
SI	Serial Input
SO	Serial Output
TAI	Timer A Input
TAO	Timer A Output
TBI	Timer B Input
TBO	Timer B Output
TCO	Timer C Output
TDO	Timer D Output
TOS	Tramiel Operating System
TR	Transmission Register
TROFF	Trace OFF
TRON	Trace ON
TSR	Transmitter Status Register
UART	Universal Asynchronous Receiver Transmitter
UDS	Upper Data Selection
USART	Universal Synchronous Asynchronous Receiver Transmitter
VDI	Virtual Device Adapter
VMA	Valid Memory Adress
VPA	Valid Peripheral Adress
WD	Western Digital
WD	Write Data
WPRT	Write Protected
XBIOS	Extended Basic Input Output System

1/3.5

Annexe 5 : Résumé des instructions du ST Basic

Abréviation	Signification
ABS	Valeur absolue
ASC	Code ASCII
ATN	Arc tangente
AUTO	Numérotation automatique
BLOAD	Chargement d'un fichier binaire en mémoire
BREAK	Arrêt du programme
BSAVE	Sauvegarde d'une partie de la mémoire dans un fichier
CALL	Appel à une routine Assembleur
CHAIN	Mixage de deux programmes
CHR$	Conversion d'un code ASCII en un caractère
CINT	Arrondi d'un réel
CIRCLE	Tracé de cercles et arcs de cercles
CLEAR	Effacement des variables
CLEARW	Effacement des fenêtres du ST Basic
CLOSE	Fermeture de fichiers
CLOSEW	Fermeture d'une fenêtre ST Basic
COLOR	Choix des couleurs de l'écran
COMMON	Déclaration de variables globales
CONT	Suite d'un programme suspendu par un BREAK
COS	Cosinus d'un angle
CVD	Conversion d'une chaîne de 8 bits en une valeur numérique
CVI	Conversion d'une chaîne de 2 bits en une valeur numérique
CVS	Conversion d'une chaîne de 4 bits en une valeur numérique
DATA	Définition d'une ou plusieurs constantes
DEF FN	Définition d'une fonction utilisateur
DEF SEG	Définition de l'adresse de départ du segment
DEFDBL	Initiales des variables double précision
DEFINT	Initiales des variables entières
DEFSNG	Initiales des variables simple précision
DEFSTR	Initiales des variables chaînes
DELETE	Effacement de lignes dans le programme

Abréviation	Signification
DIM	Dimensionnement d'un tableau
DIR	Liste des fichiers qui se trouvent sur un disque
EDIT	Edition d'une ligne de programme
ELLIPSE	Tracé d'ellipses et d'arcs d'ellipses
END	Fin du programme
EOF	Indicateur de fin de fichier séquentiel
ERA	Effacement de fichiers sur disque
ERASE	Effacement de tableaux en mémoire
ERL	Ligne où s'est produite la dernière erreur
ERR	Numéro de la dernière erreur
ERROR	Simulation d'une erreur ST Basic
EXP	Exponentielle
FIELD	Définition de la dimension des champs d'un fichier à accès direct
FILL	Remplissage de contours fermés
FIX	Partie entière d'un réel
FLOAT	Conversion d'un entier en un réel
FOLLOW	Mémorisation de la valeur des variables
FOR	Boucle dont le nombre de parcours est fixé
FRE	Taille de la mémoire libre
FULLW	Taille maximale des fenêtres du ST Basic
GEMSYS	Accès à l'interface AES
GET	Lecture d'un enregistrement dans un fichier à accès direct
GOSUB	Exécution d'un sous-programme
GOTO	Débranchement au numéro de ligne spécifié
GOTOXY	Positionnement du curseur sur l'écran
HEX$	Conversion hexadécimale d'un nombre
IF	Test logique
INP	Lecture d'une valeur sur un port
INPUT	Lecture de données au clavier
INPUT #	Lecture de données dans un fichier séquentiel
INPUT$	Lecture de n caractères au clavier
INSTR	Recherche de la position d'une chaîne dans une autre chaîne
INT	Valeur entière d'une expression
KILL	Effacement d'un fichier
LEFT$	Extraction des n caractères les plus à gauche dans une chaîne
LEN	Longueur d'une chaîne
LET	Affectation d'une valeur à une variable ou un tableau
LINE INPUT	Lecture de données au clavier
LINE INPUT #	Lecture de données dans un fichier séquentiel
LINEF	Tracé d'une ligne
LIST	Affichage du programme dans la fenêtre LIST
LLIST	Impression du programme
LOAD	Chargement d'un programme
LOC	Numéro d'enregistrement ou nombre d'octets échangés avec un fichier



Abréviation	Signification
LOF	Taille d'un fichier
LOG	Logarithme népérien
LOG10	Logarithme décimal
LPOS	Position de la tête d'impression
LPRINT	Impression de variables et de constantes
LSET	Initialisation d'un champ d'un fichier à accès direct et justification à gauche
MERGE	Mixage de deux programmes ST Basic
MID$	Extraction d'une sous-chaîne dans une chaîne
MKD$	Conversion d'une valeur réelle double précision (8 octets) en une chaîne
MKI$	Conversion d'une valeur entière (2 octets) en une chaîne
MKS$	Conversion d'une valeur réelle simple précision (4 octets) en une chaîne
NAME	Modification du nom d'un fichier
NEW	Effacement du fichier en mémoire
NEXT	Marque de fin de boucle FOR
OCT$	Conversion octale
OLD	Lecture d'un programme
ON	Débranchement en fonction de la valeur d'une variable
ON ERROR	Débranchement en cas d'erreur
OPEN	Ouverture d'un fichier ou début de dialogue avec un périphérique
OPENW	Ouverture d'une fenêtre de ST Basic
OPTION BASE	Premier indice des tableaux par défaut
OUT	Envoi d'un octet sur un port de sortie
PCIRCLE	Tracé de cercles et arcs de cercle pleins
PEEK	Lecture du contenu d'une adresse en mémoire
PELLIPSE	Tracé d'ellipses et arcs d'ellipse pleines
POKE	Stockage d'un octet à une adresse mémoire
POS	Position du curseur sur l'écran ou de la tête sur l'imprimante
PRINT	Impression de données dans la fenêtre OUTPUT
PRINT #	Impression de données dans un fichier séquentiel
PRINT USING	Affichage de données formatées dans la fenêtre OUTPUT
PUT	Sauvegarde d'un enregistrement dans un fichier à accès direct
QUIT	Sortie de ST Basic, et retour sous GEM
RANDOMIZE	Initialisation du générateur pseudo-aléatoire
READ	Lecture de données dans des instructions DATA
REM	Insertion de remarques dans un programme
RENUM	Renumérotation d'un programme
REPLACE	Sauvegarde du programme en mémoire
RESET	Fenêtre OUTPUT dans le buffer graphique
RESTORE	Relecture des données dans des instructions DATA
RESUME	Poursuite de l'exécution du programme après une erreur

Abréviation	Signification
RETURN	Fin de sous-programme
RIGHT$	Extraction de n caractères à droite dans une chaîne
RND	Extraction d'un nombre pseudo-aléatoire
RSET	Initialisation d'un champ d'un fichier à accès direct et justification à droite
RUN	Exécution du programme
SAVE	Sauvegarde du programme
SGN	Signe d'une expression
SIN	Sinus d'un angle
SOUND	Emission d'un son
SPACE$	Chaîne d'espaces
SPC	Affichage d'espaces dans une instruction PRINT
SQR	Racine carrée
STEP	Pas de progression dans l'exécution d'un programme
STOP	Arrêt du programme courant
STR$	Conversion d'une expression numérique en chaîne
STRING$	Répétition multiple d'une chaîne
SWAP	Echange de deux variables
SYSTAB	Début de la table des paramètres système
SYSTEM	Retour sous GEM
TAB	Tabulation
TAN	Tangente d'un angle
TRACE	Suivi de l'exécution d'un programme
TROFF	Mode TRACE OFF
TRON	Mode TRACE ON
UNBREAK	Annulation d'une commande BREAK
UNFOLLOW	Annulation de la commande FOLLOW
UNTRACE	Annulation de la commande TRACE
VAL	Conversion d'une chaîne en nombre
VARPTR	Adresse d'une variable
VDISYS	Accès à l'interface VDI
WAIT	Attente d'une donnée sur un port
WAVE	Forme d'onde de l'instruction SOUND
WEND	Fin d'une boucle WHILE
WHILE	Boucle en fonction de la valeur d'une variable
WIDTH	Dimensionnement d'une ligne
WRITE	Affichage de données sur l'écran
WRITE#	Envoi de données dans un fichier séquentiel

2

Sommaire

2/0 Plus loin
2/0.1 L'anatomie de l'Atari STE
I. Apparence extérieure et connecteurs
II. Les circuits spécialisés de l'Atari STE

2/1 Le système ST
2/1.1 L'architecture interne
2/1.2 Les schémas des cartes mères ST 520 et ST 1040, MEGA ST2 et MEGA ST4
2/1.3 Le microprocesseur 68000
2/1.4 Les circuits intégrés spécialisés
I. Le multi-fonction MFP 68901
II. Le générateur sonore AY3 8910
III. Le contrôleur de disque WD 1772
IV. ACIA 6850
V. GLUE
VI. MMU
VII. SHIFTER
VIII. DMA
2/1.5 La configuration mémoire

2/2 La connectique Atari
2/2.1 La description matérielle des connecteurs de base
I. Le connecteur Centronics
II. Le connecteur RS 232
III. Le connecteur musical MIDI
IV. Le connecteur lecteurs de disquettes
V. Le connecteur disque dur
VI. Le connecteur d'alimentation
VII. Le connecteur de moniteur
VIII. Le connecteur de cartouche
IX. Le connecteur de souris et manette

2/3 Les périphériques de base
2/3.1 Les moniteurs
2/3.2 Les périphériques de saisie

Partie 2 : Conception matérielle des ST

2/0

Plus loin

Vous retrouverez dans chacune des parties où cela est nécessaire, un chapitre au titre évocateur *Plus loin*. Comme vous le savez, ce nouveau chapitre incluera toutes les nouveautés importantes de la gamme Atari. Ces nouveautés, à l'instant de la publication de *Plus loin*, sont connues dans leurs grandes lignes. Ce sont ces descriptions générales qui seront inclues dans *Plus loin*. Les descriptions et études détaillées dont vous êtes familiers seront ensuite intégrées dans l'ouvrage aux places qui leur sont dues lorsque la machine en question aura été mise en notre possession aux fins d'examen détaillé.

2/0.1

L'anatomie de l'Atari STE[1)]

I - Apparence extérieure et connecteurs

Son « look » est identique à celui de ses grands frères, et seuls quelques petits détails discrets vous le feront reconnaître.

Tous les secrets du STE sont en fait placés à l'arrière de l'appareil ou à l'intérieur.

Si l'on retrouve les traditionnelles prises des ST, leurs familiers distingueront au premier coup d'œil quelques nouvelles venues.

1) Les éditions WEKA remercient ATARI MAGAZINE de les avoir autorisées à reprendre certaines informations publiées dans le Dossier STE du numéro 6 du mois de novembre 1989.

• Deux prises Cinch permettant de connecter votre STE à une chaîne Hi-Fi afin d'assurer une reproduction sonore stéréophonique de grande qualité.

• Deux connecteurs broches permettant l'adjonction de périphériques de type ludique.

Ces quelques particularités externes du STE étant citées, passons aux différences internes plus nombreuses et surtout plus importantes.

II - Les circuits spécialisés de l'Atari STE

• Le SHIFTER du STE et le câblage associé ont été totalement modifiés afin de permettre les *scrolling* d'écran câblés. L'avantage par rapport à un *scrolling* par logiciel est une rapidité accrue ainsi que des capacités quasiment illimitées. Il est en effet possible de créer des écrans virtuels de très grande taille, l'écran réel agissant alors comme une fenêtre que l'on promène sur cette image gigantesque.

• Le BLITTER, jusqu'ici réservé aux grands frères Méga ST, honore le STE de sa présence et de ses sensationnelles facultés.

Ce Blitter a pour mission une tâche très simple : il transfère des blocs mémoire bit à bit d'une adresse à une autre. Trivial, direz-vous... Oui, mais toute opération réalisée par un microprocesseur est triviale, seule la vitesse effarante à laquelle il travaille est exceptionnelle.

Ce Blitter, sorte de coprocesseur graphique, est un champion de transfert de mémoire, opération réalisée lors de tout affichage graphique. Outre le fait de décharger le processeur principal de ces tâches d'affichage et de les faire mieux que lui, contenant le programme de transfert en lui, ce dernier n'existe donc plus en mémoire morte, économisant ainsi plusieurs kilo-octets de mémoire.

• Le DAC (convertisseur Numérique-Analogique)

Comme son nom l'indique, ce circuit convertit des signaux numériques composés des bits à l'état haut ou bas en signaux analogiques de tensions continuellement variables.

Sa fonction est de traduire les signaux numériques en provenance du générateur sonore Yamaha YM2149, toujours fidèle au poste, en signaux analogiques propres à exciter les bobines des membranes des deux haut-parleurs externes. Tout ceci se fait en accès direct, c'est-à-dire sans aucune intervention du 68000 qui trouve alors évidemment le temps de faire autre chose.

Un petit circuit spécialisé, le LMC 1992 de National Semiconductors, permet le réglage de volume et de tonalité à la manière d'une véritable chaîne Hi-Fi.

Le STE dispose donc maintenant de 5 voies de restitution sonore, deux numériques dont nous venons de parler en plus des 3 voies combinées du générateur Yamaha.

• La mémoire vive SIMM (*Single In-line Memory Module*)

De technologie nouvelle, ces circuits de mémoire vive permettent d'éten-



dre la capacité du STE à 4 Méga-octets en enfichant très simplement ces petites barettes de mémoire SIMM sur les 4 supports de la carte mère. Sur le marché, sont proposées des barettes de mémoire SIMM de 256 Ko, 512 Ko ou 1 Mo.

• Le nouveau GLUE/MMU

Sur les anciens ST, ces deux circuits étaient bien séparés et ne se parlaient qu'à travers quelques-unes de leurs 68 broches (voir Partie 2, chapitre 1.4). La STE dispose maintenant d'un circuit unique combinant l'ensemble des fonctions du MMU et du GLUE. La partie GLUE de ce nouveau circuit assure comme auparavant le décodage des zones d'adressage de la mémoire ainsi que des périphériques, il génère les signaux d'interruption et contrôle les accès mémoire.

Ce GLUE est constamment nourri d'un signal d'horloge à 8 Mégahertz. Muni de divers diviseurs de fréquence, il délivre donc des fréquences de pilotage pour les divers périphériques comme le 2 Mégahertz pour le générateur de son ou le 500 kilohertz pour les divers ACIA du clavier ou l'interface MIDI. Les signaux de synchronisation du moniteur vidéo sont aussi à la charge de ce très courageux GLUE. Et c'est là qu'une nouveauté intéressante apparaît sur le STE : il est en effet maintenant possible de désactiver ces signaux vidéo et de synchroniser l'image engendrée par lui avec une source extérieure, une caméra vidéo par exemple. Vous devinez déjà l'immense intérêt de cette possibilité.

La partie MMU, quant à elle, prend en charge le couplage du bus d'adresse mémoire avec le bus 68000. Cette fois, la capacité d'adressage de 4 Méga-octets est pleinement utilisée de façon optionnelle avec les barettes mémoires SIMM. Les deux autres tâches du MMU, à savoir l'assistance au SHIFTER dans la production des signaux vidéo et l'adressage mémoire en accès direct (à travers le DMA), n'ont pas reçu de modifications majeures hormis évidemment le cablâge.

Pourquoi un GLUE/MMU intégré ? Tout simplement parce qu'un circuit un peu plus compliqué revient moins cher que deux circuits déjà compliqués. Mais aussi parce que la réduction des connexions améliore la fiabilité et la rapidité d'exécution. Viendra peut-être le temps où tous les circuits de votre Atari ne seront plus qu'un seul et même circuit...



2/1

Le système ST

2/1.1

L'architecture interne

Les ordinateurs de la gamme Atari ST sont des systèmes mono-carte dont les fonctionnalités peuvent être divisées en quatre sous-systèmes :

— Cœur :

Microprocesseur 16/32 bits 68000 de Motorola,
Mémoire ROM (192 Koctets extensible à 320 Koctets),
Mémoire RAM (512 ou 1024, 2048 ou 4096 Koctets de base selon la version [520 ST ou 1040 ST, Méga ST2, Méga ST4]),
Système d'accès direct à la mémoire DMA.

— Sous-système graphique :

Utilisation d'un écran monochrome haute résolution ou d'un écran couleur (moniteur RGB ou téléviseur équipé d'une prise PERITEL).

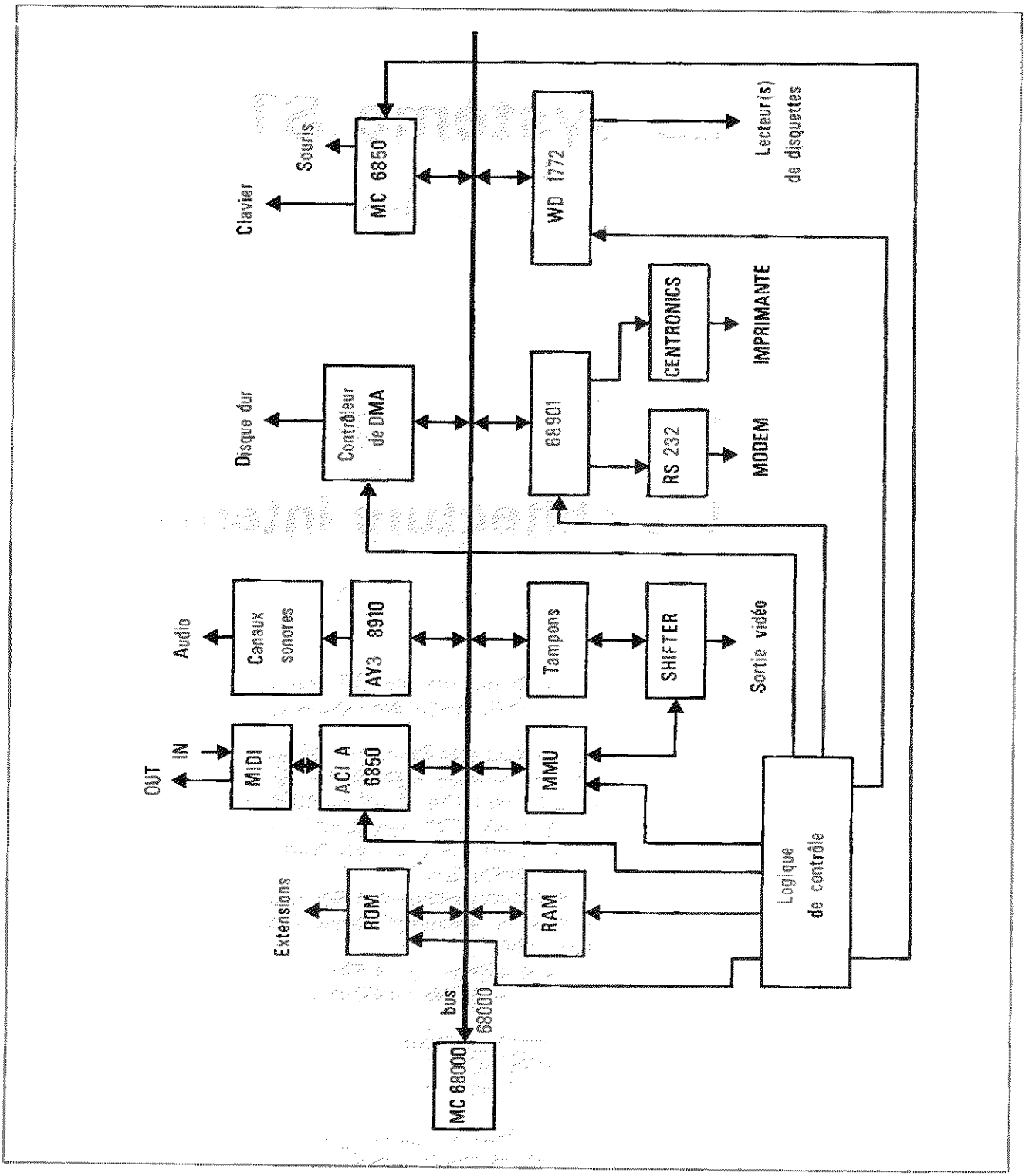
— Sous-système sonore :

Synthétiseur sonore interne : AY3-8910,
Possibilité de commander des instruments de musique qui satisfont les normes MIDI.

— Sous-systèmes périphériques :

Connecteur CENTRONICS,
Connecteur RS 232 (communication avec un modem ou un autre ordinateur),
Contrôleur de disquettes et interface DMA,
Connecteurs Souris et manette de jeu.

Le schéma suivant matérialise ces quatre sous-systèmes, leurs composants et leurs interconnexions :



2/1.2

Les schémas des cartes mères ST 520 et ST 1040, MEGA ST 2 et MEGA ST 4

ST 520 ET ST 1040

Le schéma HARD 1 laisse apparaître :

— l'alimentation ;

— le contrôleur de DMA (Accès direct à la mémoire) ;

— le contrôleur de disque ;

— les deux ACIA 6850 (connexion du clavier et de l'interface MIDI) ;

— le circuit multi-fonctions 68901 (interface série, timers, port parallèle, gestionnaire d'interruptions) ;

— le circuit sonore AY3 et l'amplificateur associé ;

— divers ports d'entrée/sortie :
- parallèle,
- RS 232,
- clavier,
- MIDI IN et MIDI OUT,
- lecteur de disquettes,
- lecteur de disque dur.

Le schéma HARD 2 laisse apparaître :

— le microprocesseur 68000 ;

— le GLUE qui prend en charge la logique de tous les tampons et des décodages d'adresses et qui divise le signal d'horloge destiné aux ACIA et au SHIFTER ;

— le SHIFTER qui prend en charge la gestion de l'écran (zone mémoire de 32 Koctets) et la logique d'écran associée ;

— la MMU qui gère le multiplexage des adresses et aide le DMA et le SHIFTER au niveau de la gestion des adresses ;

— les 6 ROM qui contiennent le système d'exploitation GEMDOS ;

— la logique de gestion de la touche **RESET** ;

— le port d'extension pour cartouches ROM.

Le schéma HARD 3 laisse apparaître :

— la logique d'interconnexion des mémoires RAM ;

— le détail du connecteur FLOPPY DISC.

Le schéma HARD 4 concerne la logique liée au clavier des Atari 520 ST.

Enfin, le schéma HARD 5 concerne la logique liée au clavier des Atari 1040 ST.

MÉGA ST 2 ET 4

Le schéma HARD 6 représente l'interconnexion du processeur et des coprocesseurs principaux :

— le microprocesseur 68000,

— le GLUE qui prend en charge la logique des tampons, les décodages d'adresses, et la division du signal d'horloge,

— le MMU (STCMU) spécialisé dans la gestion et le multiplexage des adresses,

— le blitter (STB) qui permet des déplacements rapides de blocs mémoire graphiques.

Le schéma HARD 7 représente l'interconnexion des circuits spécialisés dans les entrées/sorties :

— les deux ACIA 6850 qui s'occupent de la gestion du clavier et de l'interface MIDI,

— le générateur sonore YM 2149 et son amplificateur,

— le circuit multi-fonctions 68901 qui gère l'interface série, les timers, le port parallèle et le gestionnaire d'interruptions),

— le contrôleur d'accès direct à la mémoire DMAC,

— le contrôleur de disque.

Sur le schéma HARD 8 apparaissent :

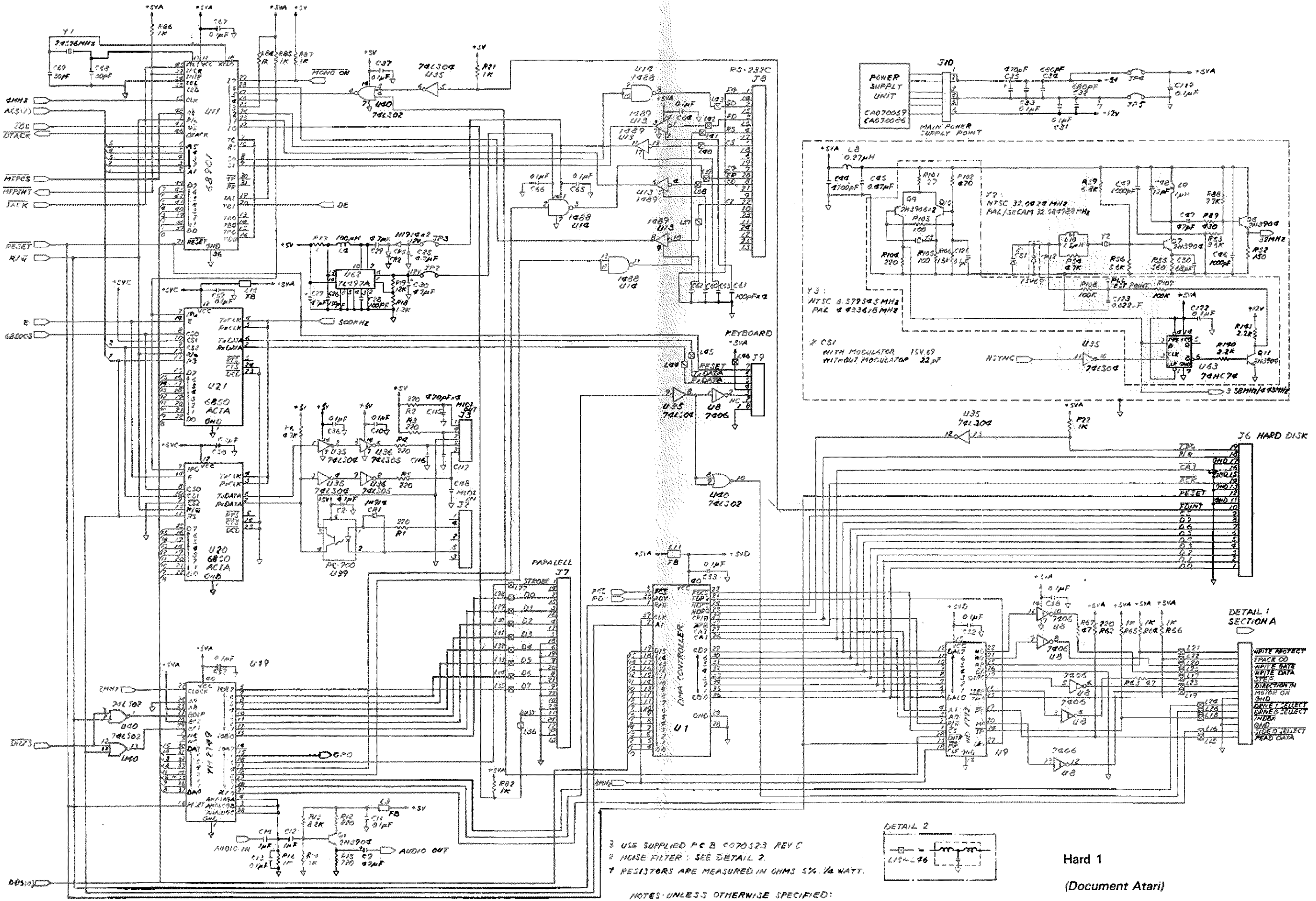
— l'interconnexion du circuit vidéo (SHIFTER) et des composants spécialisés vidéo,

— l'alimentation de la carte mère,

— le circuit qui produit le signal d'horloge.

Le schéma HARD 9 montre l'interconnexion des circuits de mémoire morte avec les bus d'adresses et de données.

Le schémé HARD 10 montre l'interconnexion des circuits de mémoire vive (RAM) avec le bus d'adresses et de données pour un Mega ST 2. Le schéma HARD 11 représente cette même interconnexion, mais pour un Méga ST 4. Seul ce schéma diffère entre les Méga ST 2 et 4.



3 USE SUPPLIED PCB C070523 REV C

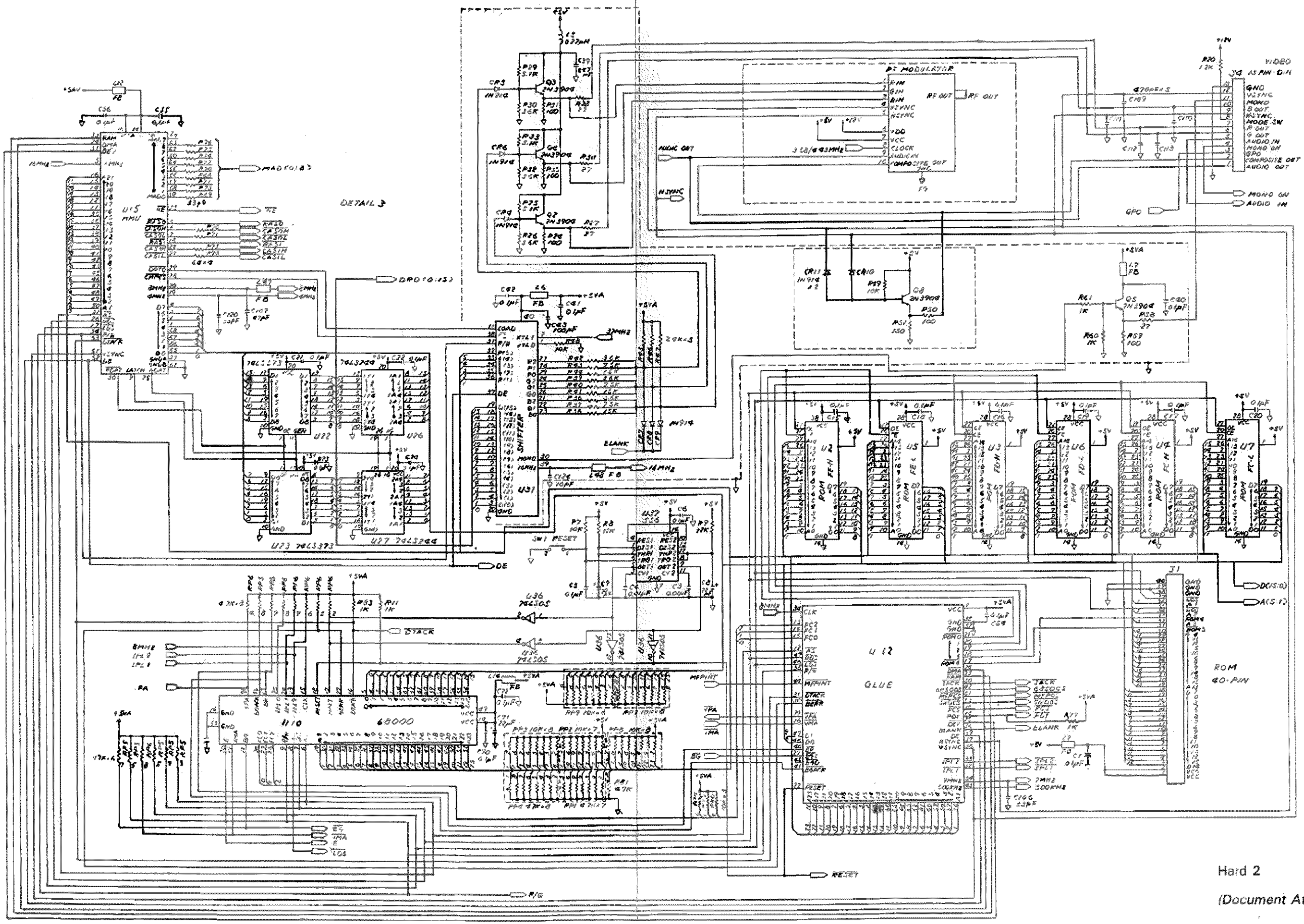
2 NOISE FILTER : SEE DETAIL 2.

1 RESISTORS ARE MEASURED IN OHMS 5% 1/4 WATT.

NOTES: UNLESS OTHERWISE SPECIFIED:

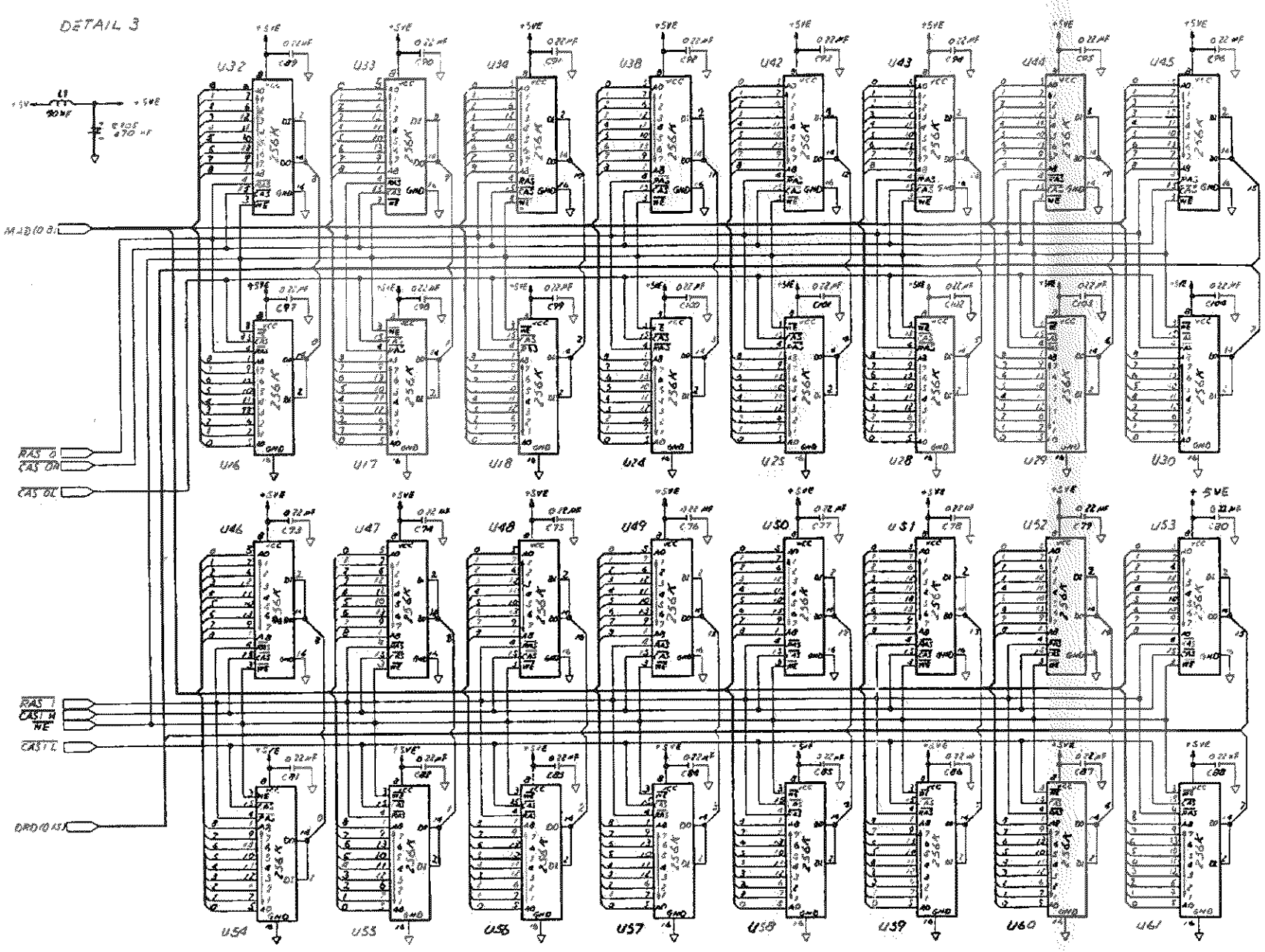
Hard 1

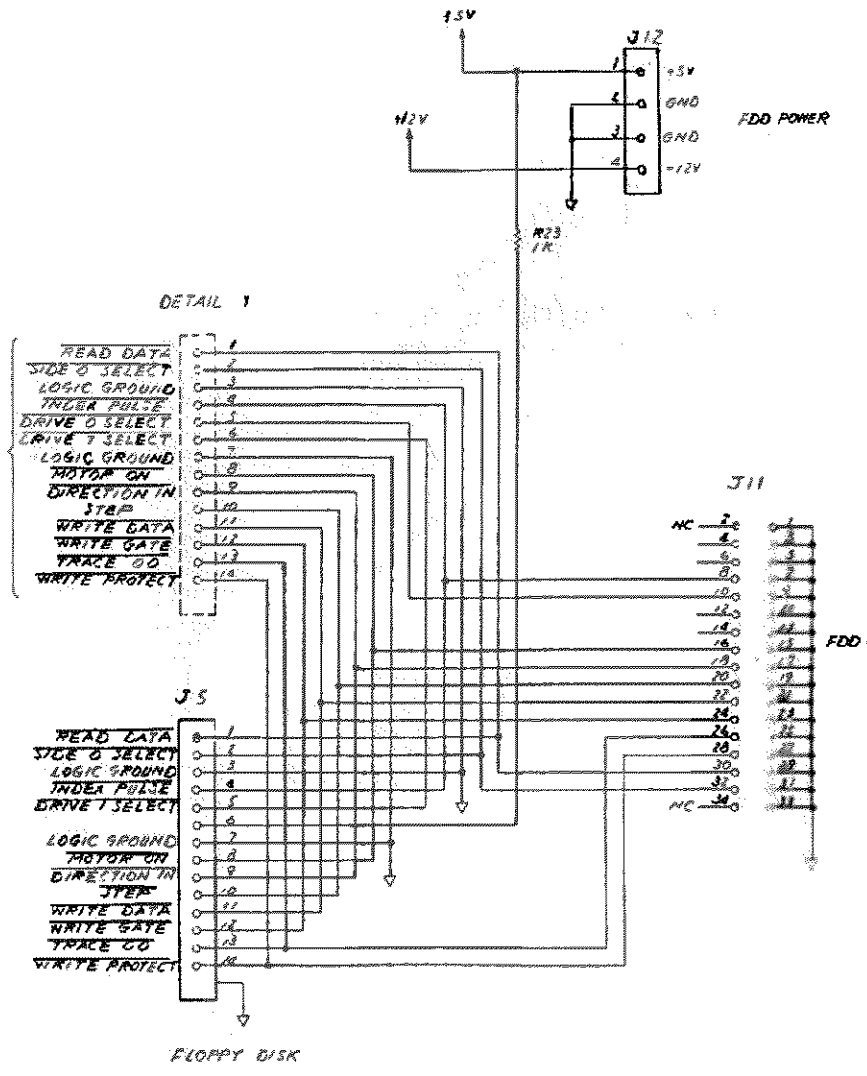
(Document Atari)



Hard 2

(Document Atari)

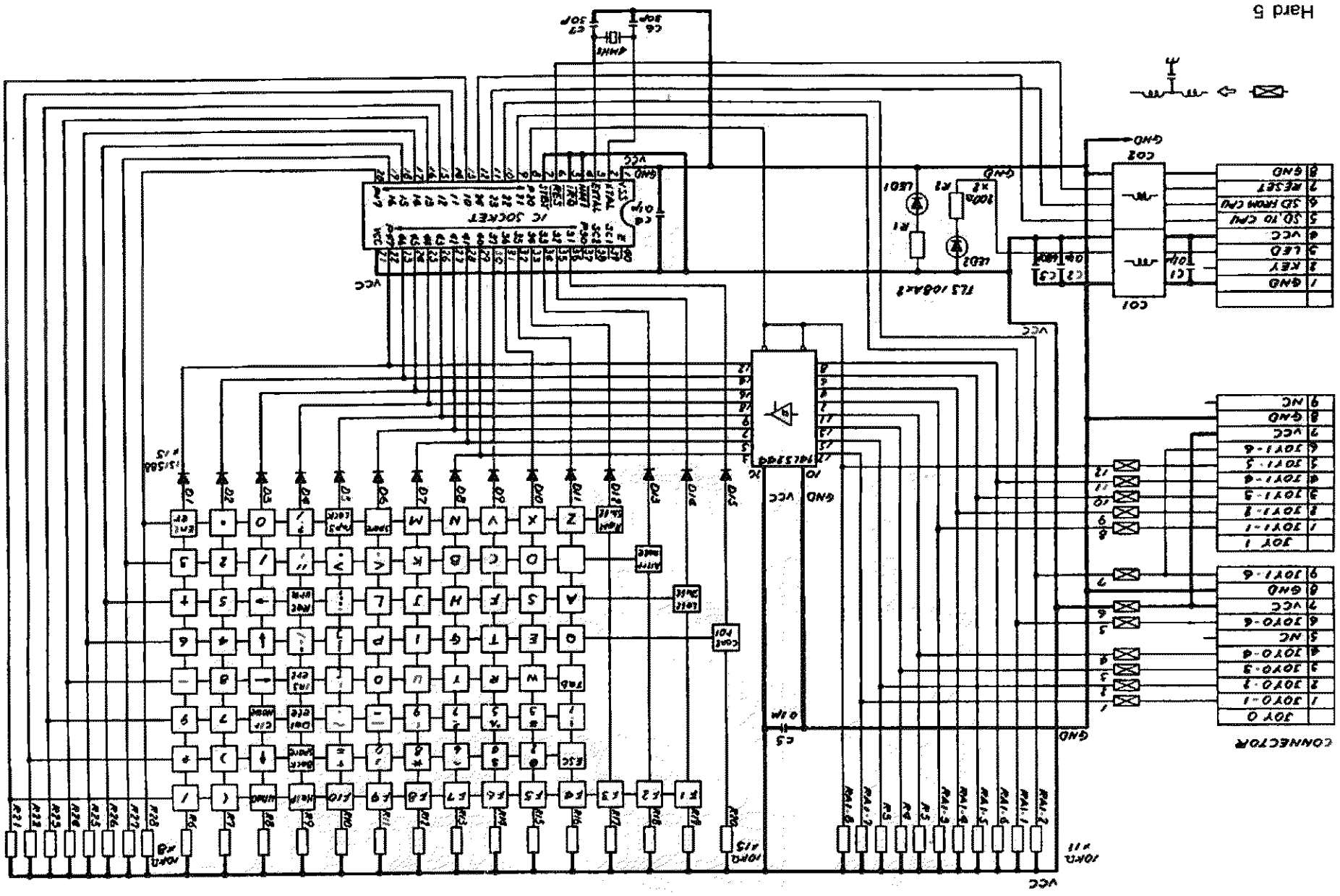




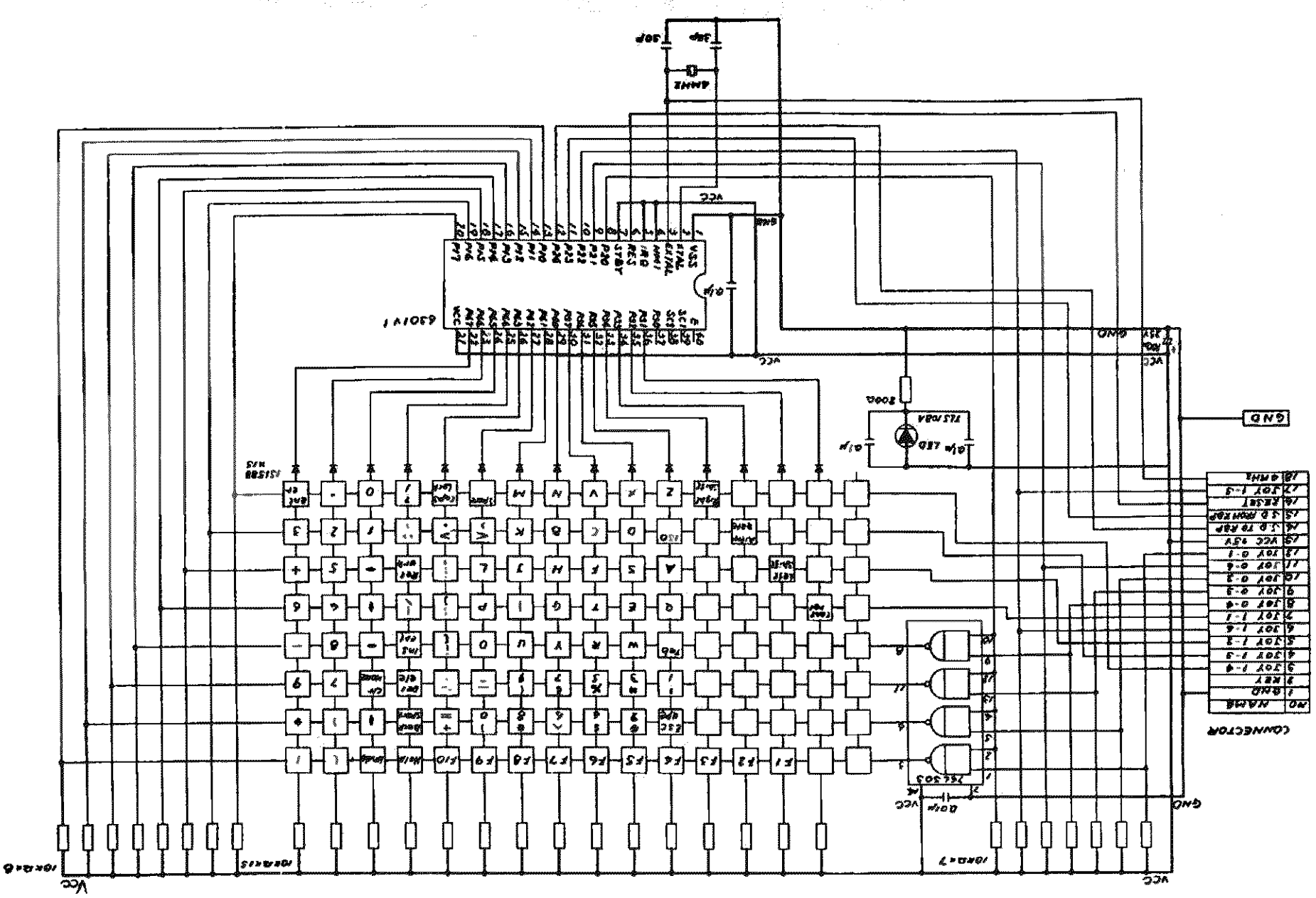
Hard 3

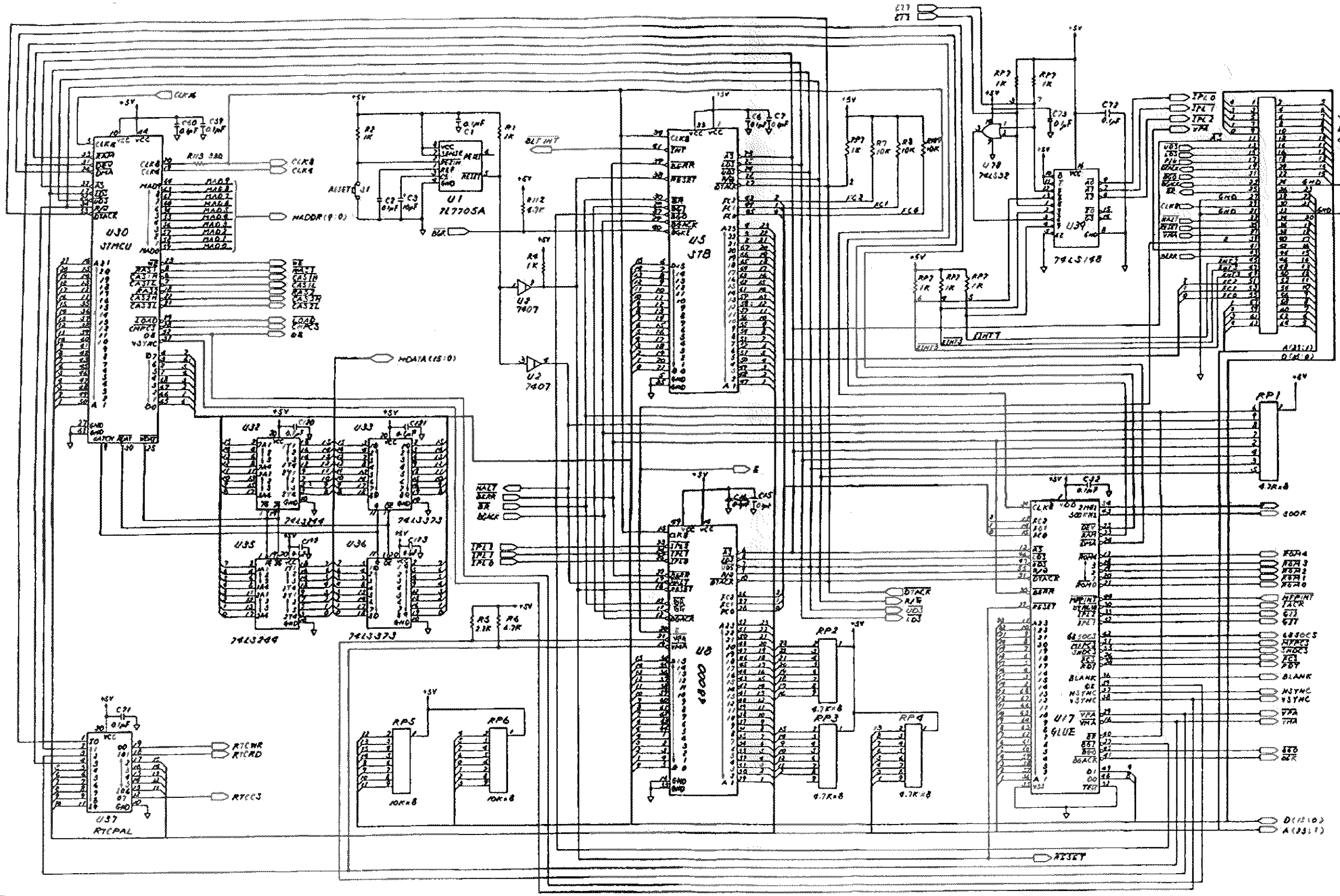
(Document Atari)

Hard 5



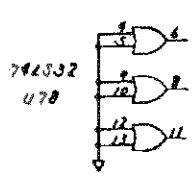
Hard 4





5. ROM TABLE : SEE SHEET 5 OF 6
4. NOT REQUIRED FOR ASSY NO. CA2000SS-XX1.
3. PUT JUMPER WIRE FROM (10) TO (8) (IN DETAIL 'A' L2-4, 6-8, 13-22, 24-30, 33-43) ASSY NO. CA2000SS-XX2.
2. NOISE FILTER : SEE DETAIL 7.
1. RESISTORS ARE MEASURED IN OHMS.

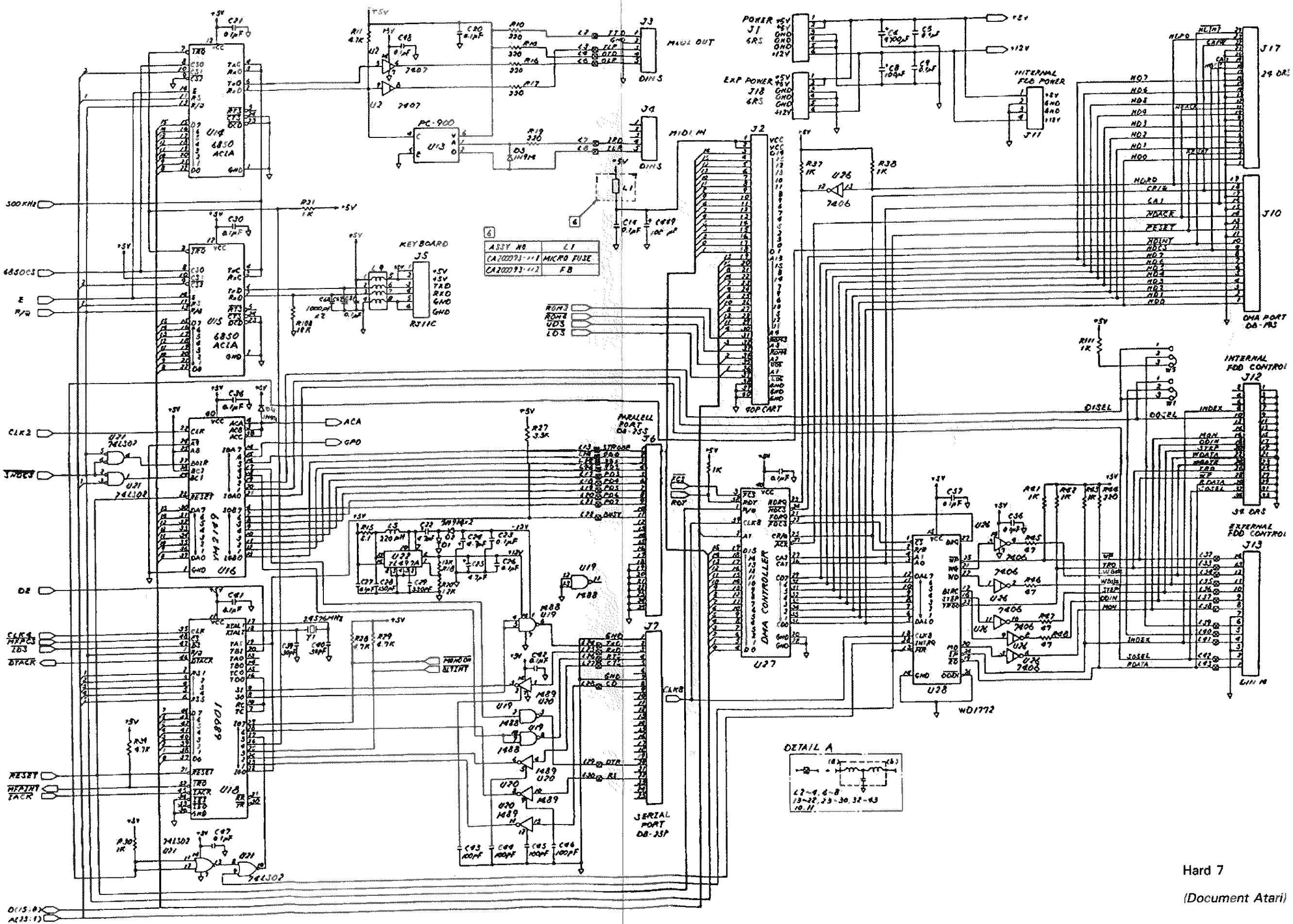
NOTES - UNLESS OTHERWISE SPECIFIED :

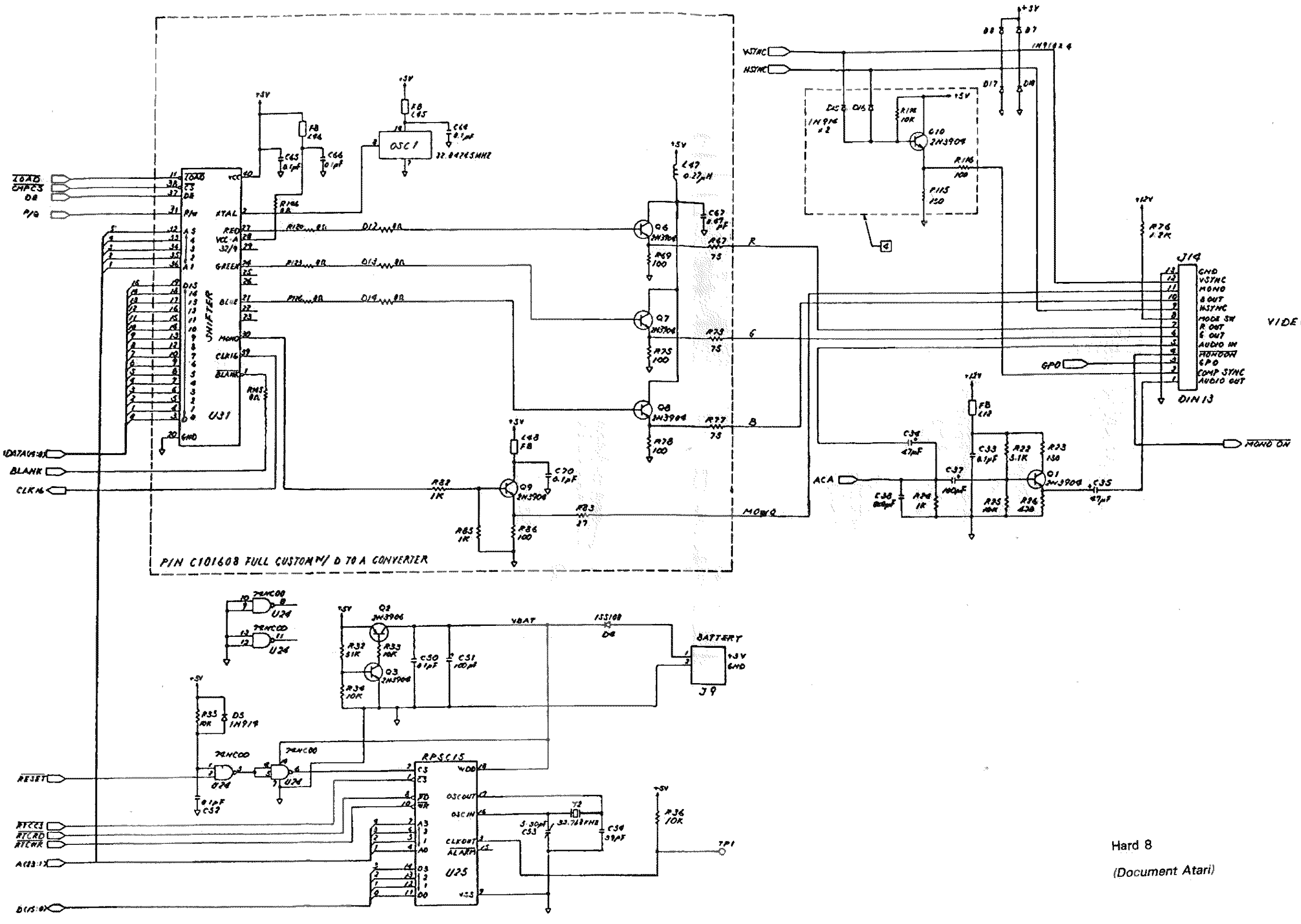


Hard 6

Hard 6

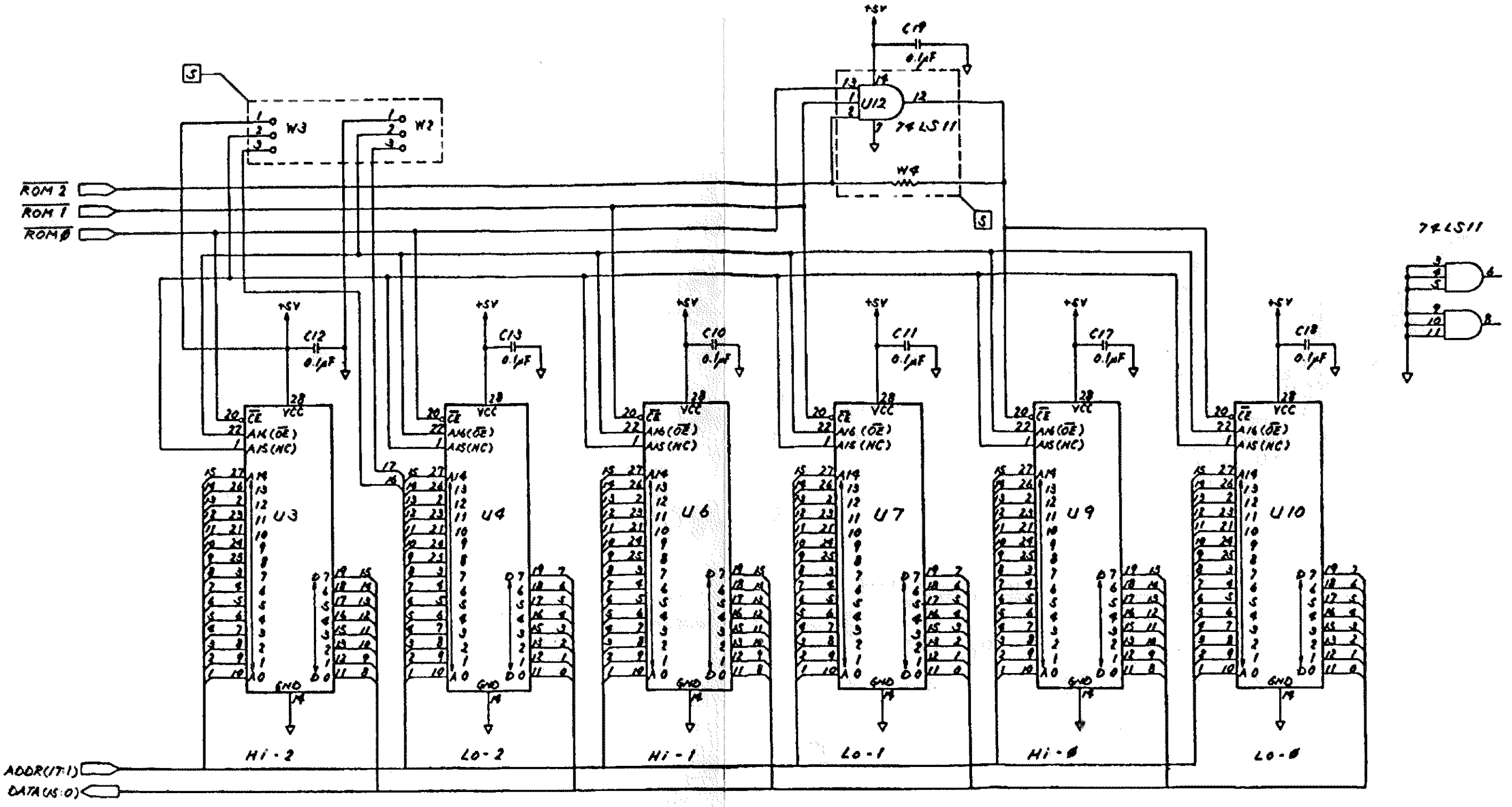
(Document Atari)


U14 6850 ACIA
U15 6850 ACIA
U16 68901
U17 6850 ACIA
U18 68901
U20 74LS02
U21 74LS08
U22 74LS02
U23 74LS74A
U24 74LS02
U25 74LS02
U26 7406
U27 DMA CONTROLLER
U28 WD1772
U29 7407
U30 74LS02
U31 74LS02
U32 7407
U33 74LS74A
U34 7406
U35 7406
U36 7406
U37 7406
U38 7406
U39 7406
U40 7406
U41 7406
U42 7406
U43 7406
U44 7406
U45 7406
U46 7406
U47 7406
U48 7406
U49 7406
U50 7406
U51 7406
U52 7406
U53 7406



Hard 8

(Document Atari)



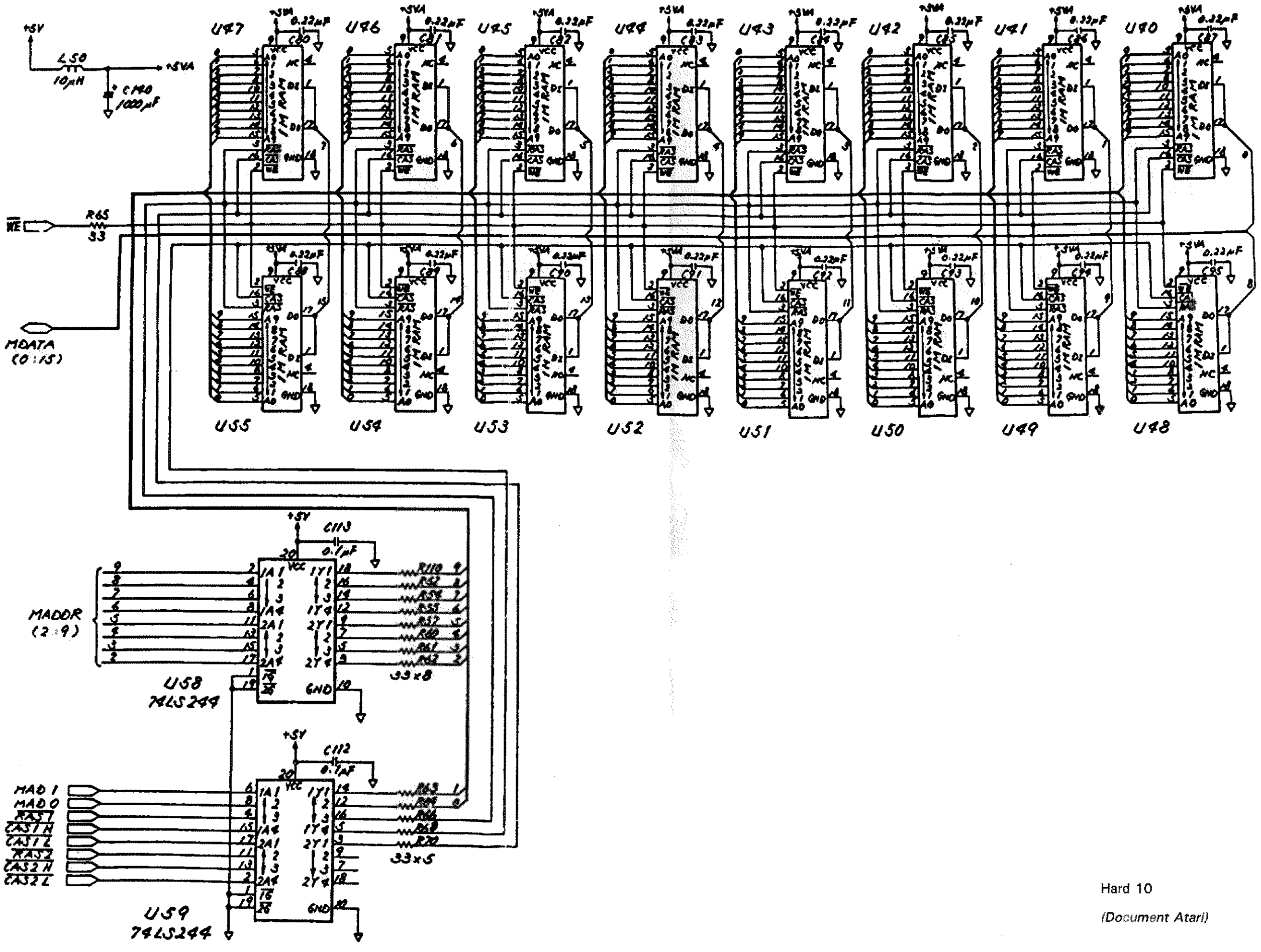
[S] ROM TABLE

IC ; U12 (74LS11)
JUMPER ; W2, W3, W4

ROM	LOCATION			
	W2	W3	W4	U12
256K	1-2	1-2	JUMPER	NO USE
1M x 2	2-3	2-3	NON	USE
1M x 8	2-3	2-3	JUMPER	NO USE

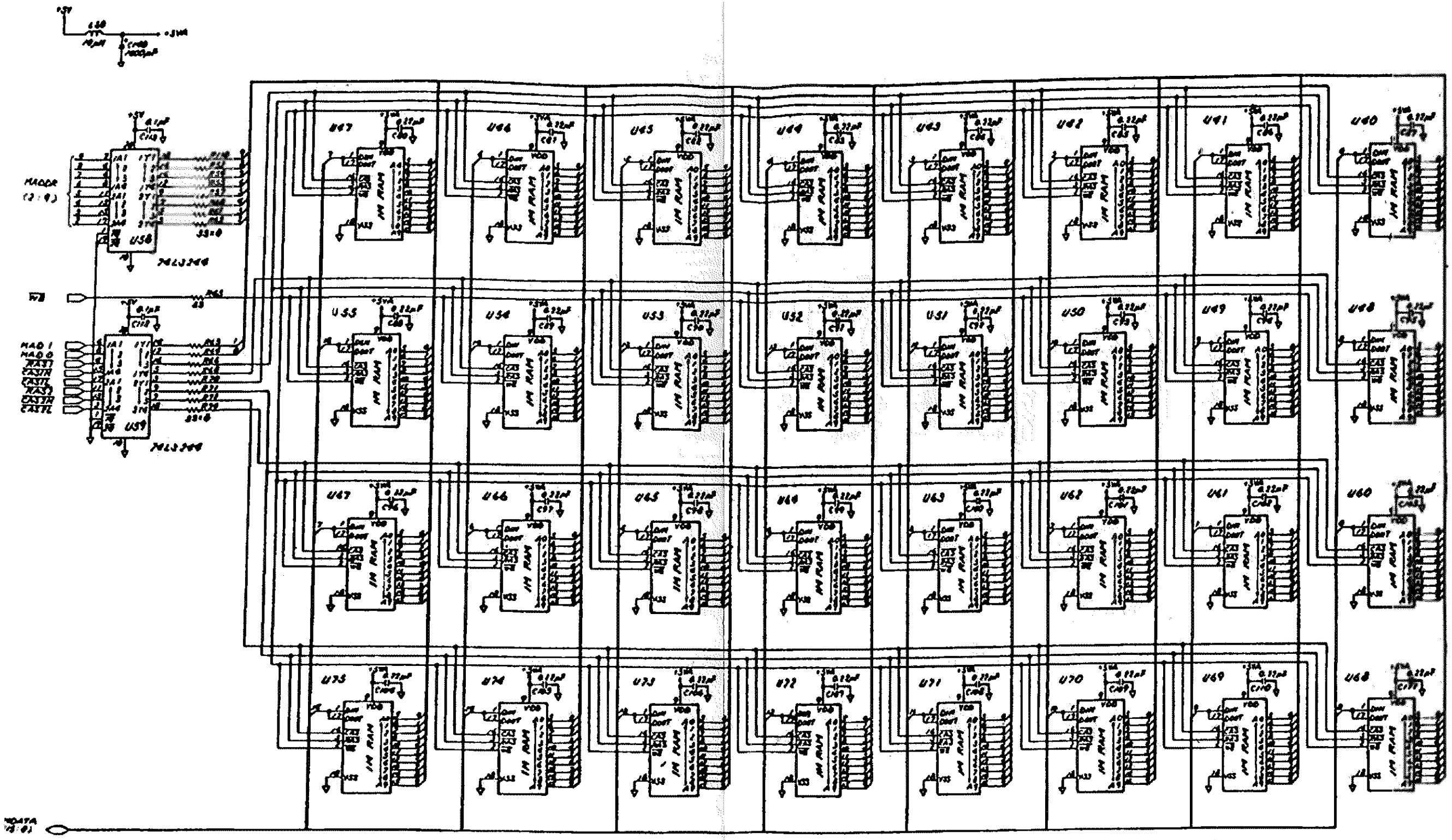
Hard 9

(Document Atari)



Hard 10

(Document Atari)



Hard 11

(Document Atari)



2/1.3

Le microprocesseur 68000

Le 68000 est un microprocesseur fabriqué par Motorola. Il est bâti selon une architecture 16/32 bits (32 bits-interne, 16 bits sur le bus de communication externe). Son bus de données de 24 bits lui permet d'adresser directement 2^24 octets, c'est-à-dire 16 Méga Octets de mémoire (vive ou morte).

Ce microprocesseur se trouve dans un boîtier de 64 connecteurs dont le brochage est le suivant :

D4	1		64	D5
D3	2		63	D6
D2	3		62	D7
D1	4		61	D8
D0	5		60	D9
AS	6		59	D10
UDS	7		58	D11
LDS	8		57	D12
R/W	9		56	D13
DTACK	10		55	D14
BG	11		54	D15
BGACK	12		53	Vss
BR	13		52	A23
VDD	14		51	A22
CLK	15		50	A21
VSS	16	68000	49	Vdd
HALT	17		48	A20
RESET	18		47	A19
VMA	19		46	A18
E	20		45	A17
VPA	21		44	A16
BERR	22		43	A15
IPL2	23		42	A14
IPL1	24		41	A13
IPL0	25		40	A12
FC2	26		39	A11
FC1	27		38	A10
FC0	28		37	A9
A1	29		36	A8
A2	30		35	A7
A3	31		34	A6
A4	32		33	A5

A1-A23 Sortie : Bus d'adresses threestate
Ce bus permet d'adresser directement 8 Moctets de mémoire en dehors des cycles d'interruption. Pendant les cycles d'interruption, les connecteurs A1 à A3 donnent le niveau d'interruption courant, et les connecteurs A4 à A23 se trouvent au niveau logique haut.

AS Sortie : Adresse valide
A l'état bas, cette sortie indique qu'une adresse valide se trouve sur le bus d'adresses.

BG Sortie : Attribution des bus
Un niveau bas sur cette sortie indique aux éventuels circuits maîtres que le 68000 n'aura plus le contrôle des bus à la fin du cycle courant.



BERR — Entrée : Erreur bus
Un niveau bas sur cette entrée signale qu'une erreur disque s'est produite pendant le cycle courant.

BGACK — Entrée : Détection de l'attribution des bus
Cette entrée indique qu'un périphérique a pris la maîtrise des bus.

BR — Entrée : Demande d'accès aux bus
A l'état bas, cette entrée indique qu'un périphérique demande la maîtrise des bus.

CLK — Entrée : Signal d'horloge compatible TTL

D0-D15 — Bidirectionnel : Bus de données three-state
Ce bus permet d'échanger des données avec tous les circuits périphériques. Pendant une séquence d'interruption, il véhicule notamment le vecteur d'interruption.

DTACK — Entrée : Transfert de données effectué
A un niveau bas, cette entrée indique que le transfert de données est terminé.

E — Sortie : Validation d'horloge
Signal d'horloge délivré par le 68000 égal à dix fois la période d'horloge (6 périodes basses, 4 périodes hautes).

FC0-FC2 — Sortie : Signaux d'état du 68000
Etat et type du cycle en cours d'exécution

FC2	FC1	FC0	Type du cycle en cours
0	0	0	Indéfini
0	0	1	Donnée utilisateur
0	1	0	Programme utilisateur
0	1	1	Indéfini
1	0	0	Indéfini
1	0	1	Donnée superviseur
1	1	0	Programme superviseur
1	1	1	Validation d'interruption

HALT — Bidirectionnel : Signal d'arrêt
A l'état bas, ce connecteur provoque l'arrêt du 68000 (demandé par un équipement externe).

IPL0-IPL2 — Entrée : Niveau de priorité des interruptions
Ces connecteurs indiquent le niveau de priorité du périphérique qui demande une interruption.

LDS — Sortie : Sélection du LSB (*Last Significative Byte*)
Cette sortie, couplée à UDS et R/W, contrôle le flux de données sur le bus de données (voir UDS).

RESET — Bidirectionnel : Initialisation du 68000

R/W — Sortie : Read/Write (Lecture/Ecriture)

UDS

Sortie : Sélection du MSB (*Most Significative Byte*)
Cette sortie, couplée à LDS et R/W, contrôle le flux de données sur le bus de données :

UDS	LDS	R/W	D8-D15	D0-D7
0	0	0	bits 8-15 valides	bits 0-7 valides
0	0	1	bits 8-15 valides	bits 0-7 valides
0	1	0	bits 8-15 valides	bits 8-15 valides
0	1	1	bits 8-15 valides	non valides
1	0	0	bits 0-7 valides	bits 0-7 valides
1	0	1	non valide	bits 0-7 valides
1	1	1/0	non valide	non valide

VMA

Sortie : Validation d'adresse mémoire
Cette sortie indique aux périphériques qu'une adresse valide est disponible sur le bus d'adresses (périphériques de la gamme MC 6800).

VPA

Entrée : Validation d'adresse de périphérique
Cette entrée indique qu'un périphérique (de type MC 6800) doit se synchroniser avec le signal E (*Enable*) issu du 68000.



2/1.4

Les circuits intégrés spécialisés

I - Le multi-fonction MFP 68901

Le MFP 68901 est un circuit intégré multi-fonction qui permet de gérer la plupart des périphériques qui se trouvent dans un environnement 68000. Lorsque ce circuit est utilisé, le reste des composants nécessaires pour piloter la carte du micro-ordinateur est réduit au maximum. Les caractéristiques de ce circuit sont les suivantes :

— boîtier 48 broches,

— logique compatible 68000,

— 8 broches d'entrée/sortie parallèle,

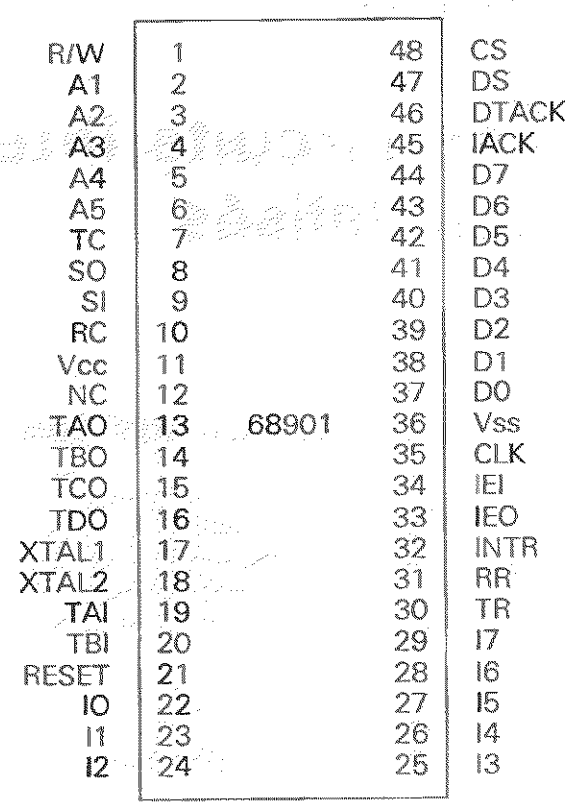
— 2 contrôleurs d'interruptions 8 bits,

— 16 origines d'interruptions possibles,

— 4 timers programmables individuellement,

— 1 canal USART (interface série).

Le brochage du MFP 68901 est le suivant :



A1-A5 Entrée : Choix d'un registre de commande pendant un cycle de lecture/écriture.

CLK Entrée : Horloge (base de temps du MFP)

CS Entrée : Chip Select (Sélection du circuit)
Cette entrée permet d'accéder aux registres internes du MFP.

D0-D7 Bidirectionnel threestate : Bus de données.
Ce bus permet de transférer des données entre le bus et un des registres du MFP et à communiquer un vecteur d'interruption.

DS Entrée : Data Select (Sélection de l'octet de donnée)
Cette broche est utilisée pendant les phases de reconnaissance d'interruption et l'accès au MFP.

DTACK Sortie threestate : Fin d'un cycle (lecture ou écriture)
Elle signale que le bus de données est prêt ou que les données ont été prises en compte par le MFP.

I0-I7 Bidirectionnel : Bus de données qui peuvent être utilisées pour générer des interruptions externes ou comme port d'entrée/sortie parallèle.

IACK Entrée : Autorisation d'interruption
Cette entrée signale au MFP que le 68000 est en phase de reconnaissance d'interruption.



IEI	Entrée : Validation d'interruption Cette entrée signale au MFP qu'aucun périphérique de priorité supérieure ne sollicite une interruption.
IEO	Sortie : Validation d'interruption Cette sortie signale aux périphériques que le MFP ne demande pas d'interruption.
INTR	Sortie : Demande d'interruption par le MFP.
RC	Entrée : Horloge pour la réception de données sur le port série.
RESET	Entrée : Remise à zéro du circuit Dévalidation des canaux d'interruption, effacement des interruptions en attente, registres internes remis à zéro (sauf timer, registre d'état de la transmission et registres de données USART).
RR	Sortie : Etat du registre de réception Cette sortie donne l'état du buffer de réception du port 15 pour les opérations de DMA.
R/W	Entrée : Read/Write (Sens du cycle courant : Lecture ou Ecriture).
SI	Entrée : Entrée série de l'USART
SO	Sortie : Sortie série de l'USART
TAI	Entrée : Horloge compteur A
TAO	Sortie : Compteur A
TBI	Entrée : Horloge compteur B
TBO	Sortie : Compteur B
TC	Entrée : Horloge d'émission de l'USART
TCO	Sortie : Compteur C
TDO	Sortie : Compteur D
TR	Sortie : Etat du registre d'émission Cette sortie donne l'état du buffer de réception du port 16 pour les opérations de DMA.
Vcc	Entrée : Alimentation + 5 V
Vss	Entrée : Masse de l'alimentation
XTAL1, XTAL2	Entrée : Horloge (Quartz)

Le MFP 68901 se programme grâce à 24 registres accessibles au programmeur. En voici le résumé.

Numéro	Adresse	Nom	Utilisation
1	0	GPIP	Registre général d'entrée/sortie
2	1	AER	Sélection du front actif
3	2	DDR	Direction des données
4	3	IERA	Validation d'interruptions A
5	4	IERB	Validation d'interruptions B
6	5	IPRA	Interruptions en attente A
7	6	IPRB	Interruptions en attente B
8	7	ISRA	Interruptions en service A
9	8	ISRB	Interruptions en service B
10	9	IMRA	Masque des interruptions A
11	A	IMRB	Masque des interruptions B
12	B	VR	Vecteur
13	C	TACR	Contrôle du Timer A
14	D	TBCR	Contrôle du Timer B
15	E	TCDCR	Contrôle des timers C et D
16	F	TADR	Registre de données Timer A
17	10	TBDR	Registre de données Timer B
18	11	TCDR	Registre de données Timer C
19	12	TDDR	Registre de données Timer D
20	13	SCR	Caractère de synchronisation
21	14	UCR	Contrôle général de l'USART
22	15	RSR	Etat du récepteur
23	16	TSR	Etat de l'émetteur
24	17	UDR	Données USART

Voici le détail d'utilisation de chacun de ces registres :

GPIP

Registre général d'entrée/sortie
Ce registre donne accès aux bits du port parallèle, en entrée et en sortie.

AER

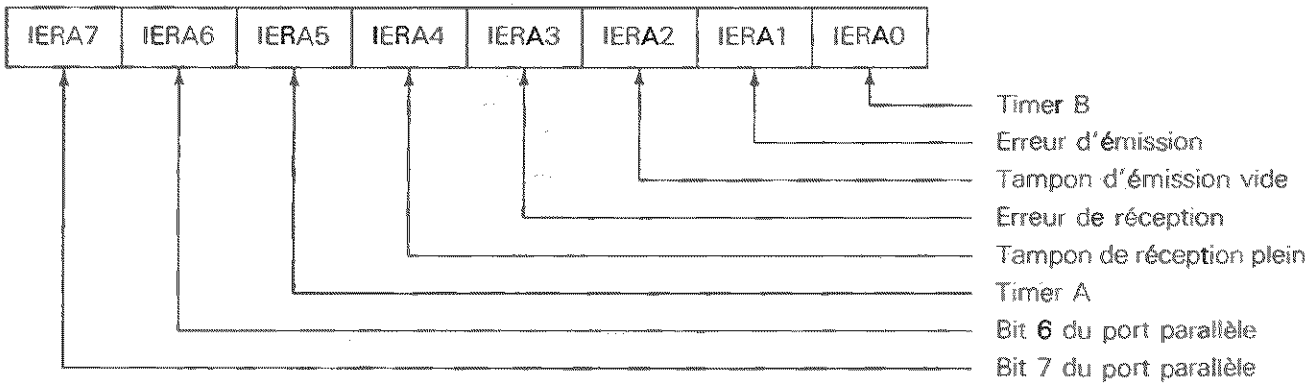
Registre de sélection du front actif.
Les bits du registre GPIP peuvent être utilisés comme entrées d'interruptions. Lorsque c'est le cas, les bits du registre AER permettent de déterminer le front sur lequel doit se produire chaque interruption. Un bit de AER à 1 signifie que l'interruption correspondante dans le registre GPIP est sensible à la transition niveau bas vers le niveau haut. Un bit à zéro dans AER signifie que l'interruption correspondante dans le registre GPIP est sensible à la transition niveau haut vers niveau bas.

DDR

Registre de direction des données.
Le sens d'utilisation de chaque bit du port parallèle est déterminé à l'aide de ce registre. Un bit à 1 dans DDR signifie que la broche correspondante dans le port parallèle est programmée en sortie. Ce même bit à zéro signifie que la broche du port parallèle est programmée en entrée.

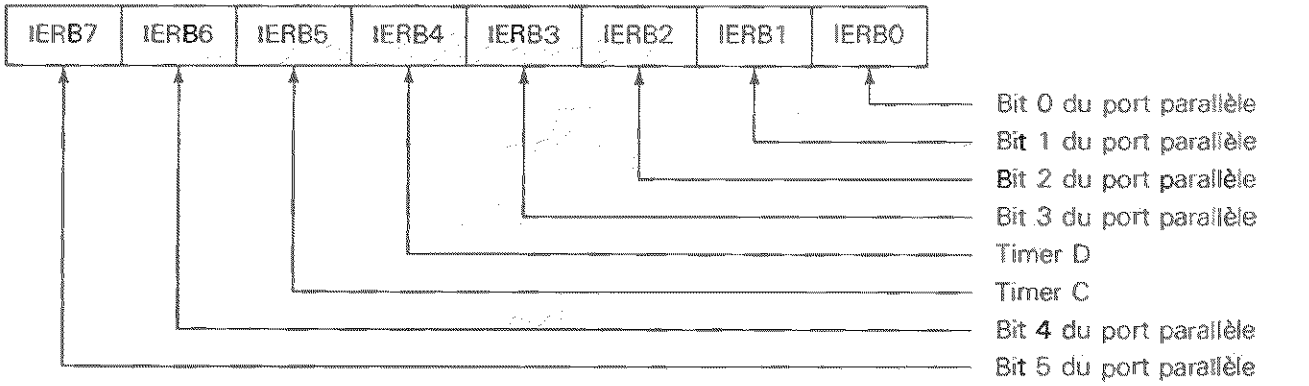
IERA

Registre de validation d'interruptions A
Ce registre valide ou inhibe 8 des 16 sources d'interruption possibles. Un bit à 1 dans ce registre valide l'interruption correspondante. Un bit à zéro dans ce registre l'inhibe. Voici la signification de chaque bit du registre :



IERB

Registre de validation d'interruptions B
Ce registre valide ou inhibe 8 des 16 sources d'interruption possibles. Un bit à 1 dans ce registre valide l'interruption correspondante. Un bit à zéro dans ce registre l'inhibe. Voici la signification de chaque bit du registre :



IPRA

Registre d'interruptions en attente A
Les bits de ce registre signalent l'occurrence d'une interruption, selon le même codage que dans le registre IERA.

IPRB

Registre d'interruptions en attente B
Les bits de ce registre signalent l'occurrence d'une interruption, selon le même codage que dans le registre IERB.

ISRA

Registre d'interruptions en service A
Si le bit 3 du registre 12 (registre vecteur) vaut 0 : le MFP met les bits IPR et ISR à zéro lorsque le 68000 a reçu un vecteur d'interruption.

Si le bit 3 du registre 12 vaut 1, le bit ISR est mis à un lorsqu'un vecteur d'interruption est demandé par le 68000. Toutes les interruptions de priorité inférieure ne sont pas traitées tant que ce bit est à un. Il repasse à zéro par l'écriture d'un octet dont ce bit (et seulement ce bit) est à zéro.

ISRB Registre d'interruptions en service B
Voir ISRA

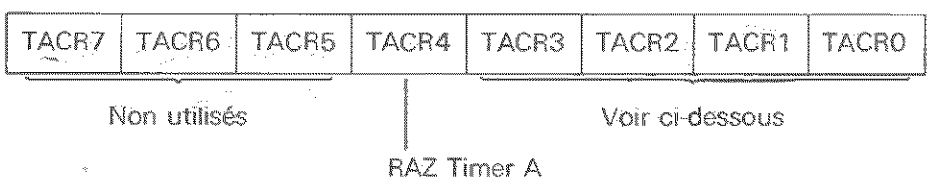
IMRA Registre de masque des interruptions A
Ce registre permet de trouver la source d'une interruption (8 parmi 16) et de modifier le registre IPR en conséquence.

IMRB Registre de masque des interruptions B
Voir IMRA.

VR Registre vecteur
Lorsque les interruptions sont vectorisées, ce registre est utilisé pour générer un vecteur d'interruption propre à chaque interruption.
Les bits 0 à 2 sont inutilisés.
Le bit 3 de ce registre permet de choisir le mode travail : *Automatic End Of Interrupt* (s'il vaut zéro) ou *Software End Of Interrupt* (s'il vaut un).
Les bits 4 à 7 sont à la disposition de l'utilisateur.
Dans le vecteur d'interruption, les bits 0 à 2 identifient le canal d'interruption. Ils sont automatiquement fournis par le MFP.

TACR Registre de contrôle du Timer A
Ce registre permet de spécifier le mode de fonctionnement du timer A : délai, compteur d'événements, génération de signaux.
Dans le mode délai, le compteur est décrémenté à chaque impulsion d'horloge.
Dans le mode compteur d'événements, les impulsions externes qui arrivent sur la broche TAI sont prises en compte (cette broche est reliée à la broche 3 du port parallèle).
Dans le mode génération de signaux, le timer A peut être activé ou inhibé à l'aide de l'entrée TAI.

Les bits du registre TACR ont les fonctions suivantes :



TACR3-TACR0	Fonction
0000	Arrêt du timer
0001	Mode délai, diviseur par 4
0010	Mode délai, diviseur par 10
0011	Mode délai, diviseur par 16
0100	Mode délai, diviseur par 50
0101	Mode délai, diviseur par 64
0110	Mode délai, diviseur par 100
0111	Mode délai, diviseur par 200
1000	Mode compteur d'événements
1001	Mode génération de signaux, diviseur par 4
1010	Mode génération de signaux, diviseur par 10
1011	Mode génération de signaux, diviseur par 16
1100	Mode génération de signaux, diviseur par 50
1101	Mode génération de signaux, diviseur par 64
1110	Mode génération de signaux, diviseur par 100
1111	Mode génération de signaux, diviseur par 200

TBCR

Registre de contrôle du Timer B
Voir TACR.

TCDCR

Registre de contrôle des timers C et D
Le registre TCDCR permet de spécifier le choix du facteur de division des timers C et D. Ces timers fonctionnent obligatoirement en mode délai.
Les bits du registre TCDCR sont les suivants :

TCDCR7	TCDCR6	TCDCR5	TCDCR4	TCDCR3	TCDCR2	TCDCR1	TCDCR0
	Timer D				Timer C		

TCDCR2-TCDCR0 (*) TCDCR6-TCDCR4 (**)	Fonction
000	Arrêt du timer
001	Mode délai, diviseur par 4
010	Mode délai, diviseur par 10
011	Mode délai, diviseur par 16
100	Mode délai, diviseur par 50
101	Mode délai, diviseur par 64
110	Mode délai, diviseur par 100
111	Mode délai, diviseur par 200

TADR

Registre de données Timer A
Ce registre contient la valeur à décompter pour le compteur A.

TBDR

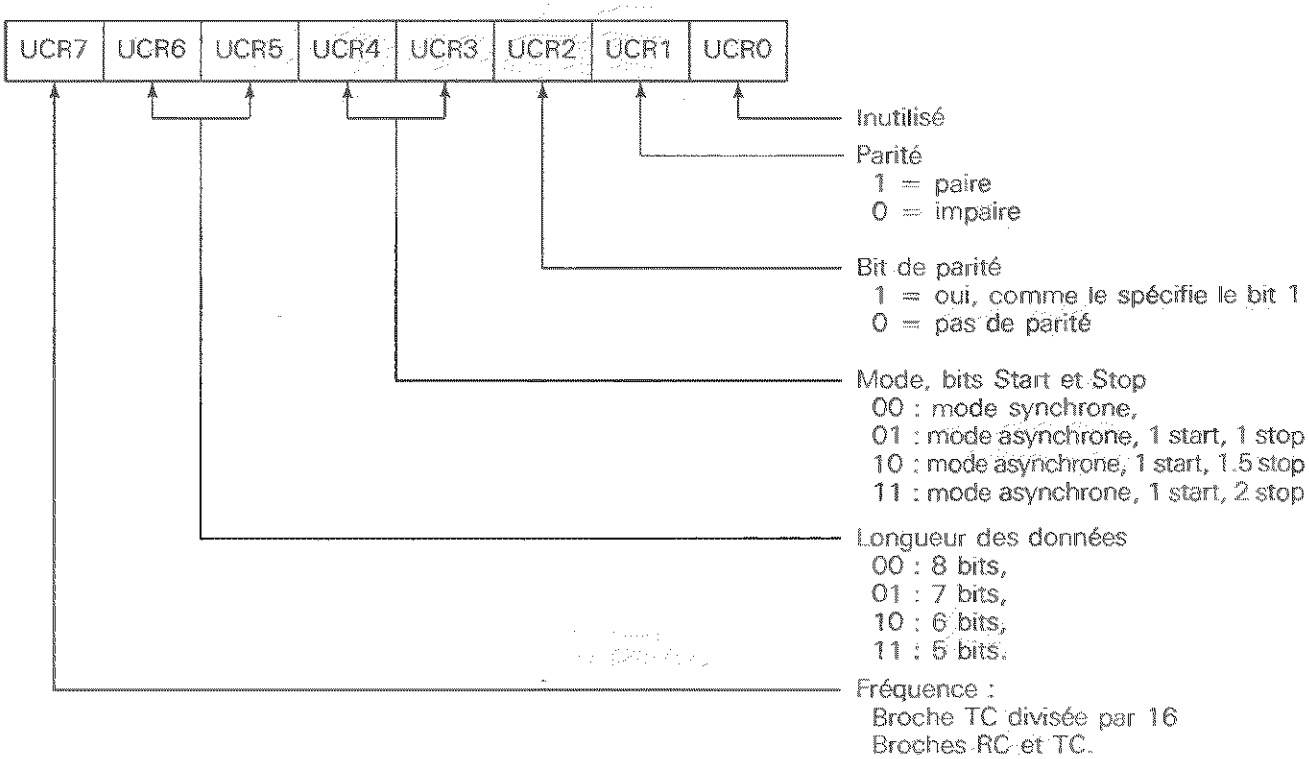
Registre de données Timer B
Ce registre contient la valeur à décompter pour le compteur B.

TCDR Registre de données Timer C
Ce registre contient la valeur à décompter pour le compteur C.

TDDR Registre de données Timer D
Ce registre contient la valeur à décompter pour le compteur D.

SCR Registre caractère de synchronisation.
Dans une réception asynchrone, ce registre doit contenir le caractère de synchronisation. Le MFP rajoute automatiquement un bit de parité.

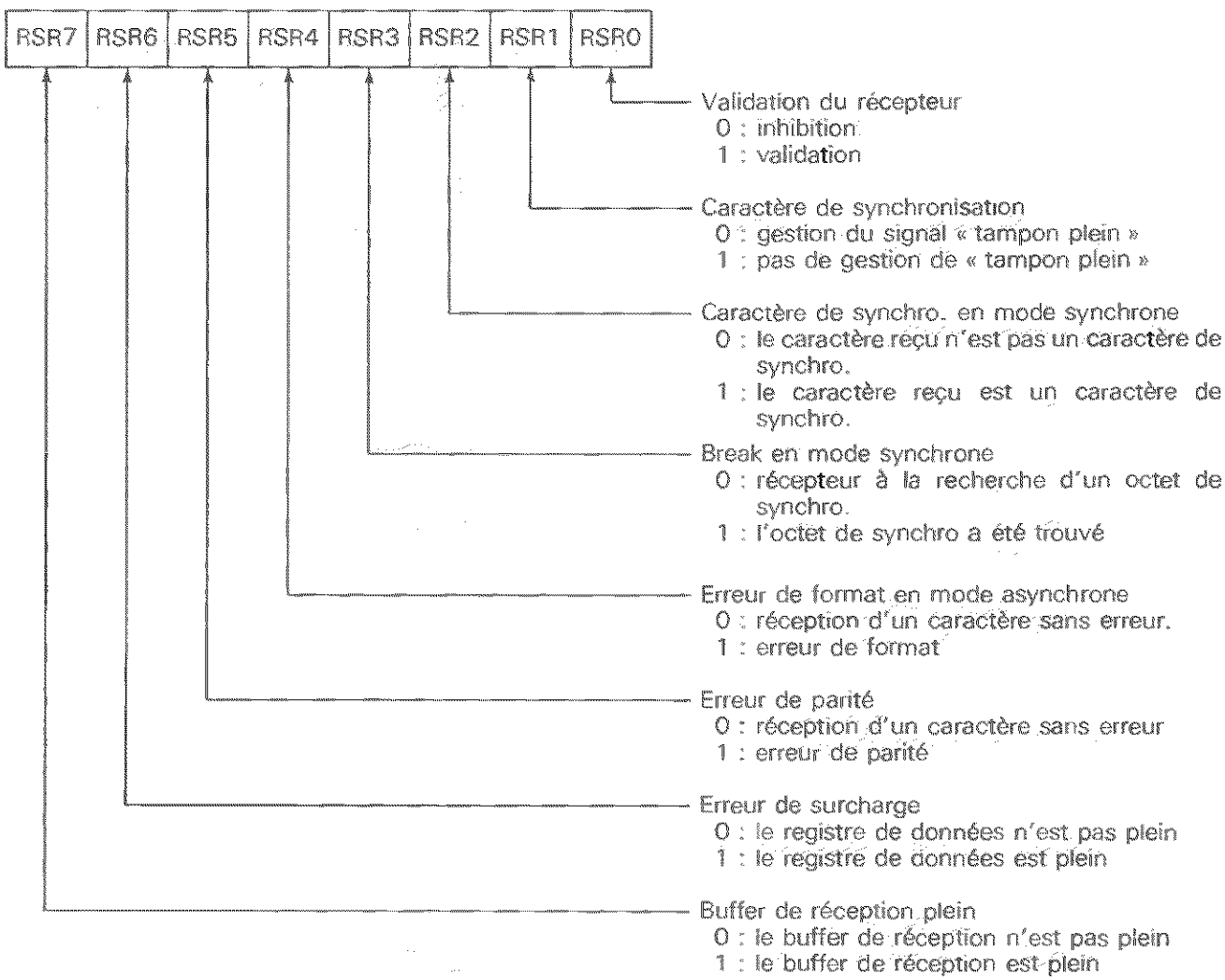
UCR Registre de contrôle général de l'USART.
Ce registre détermine les paramètres des transmissions asynchrones. Il contient les informations suivantes :





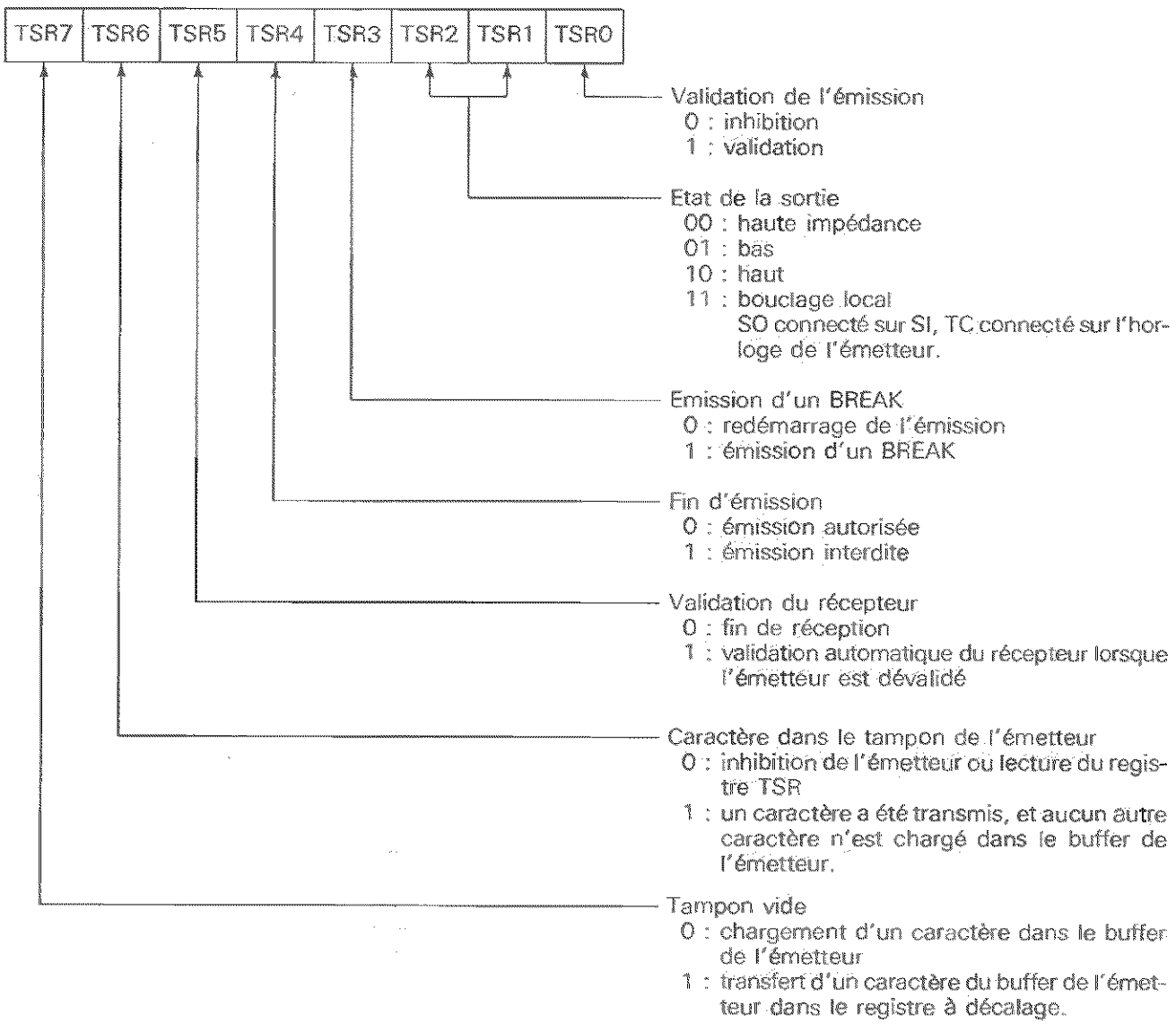
RSR

Registre d'état du récepteur
Ce registre donne des informations sur l'état de la réception lors d'une transmission asynchrone. Les informations qu'il contient sont les suivantes :



TSR Registre d'état de l'émetteur

Ce registre donne des informations sur l'état de l'émission lors d'une transmission asynchrone. Les informations qu'il contient sont les suivantes :



UDR Registre de données de USART

Par ce registre transitent les données à émettre et à recevoir en mode asynchrone.



II - Le générateur sonore AY3 8910

Ce circuit LSI (*Large Scale Integration*, c'est-à-dire intégration à grande échelle), de chez General Instruments, permet de générer des sons sur trois voies et du bruit blanc sur une voie, mixable avec les trois autres. Il est prévu pour s'interfacer facilement avec toutes sortes de microprocesseurs 8 ou 16 bits, ce qui explique sa popularité sur les micro-ordinateurs personnels.

Il est disponible en boîtier 40 broches sous la référence AY3-8910 et en boîtier 28 broches sous la référence AY3-8912. La version 40 broches a été choisie par ATARI pour ses ordinateurs de la gamme ST. Voici l'implantation et la désignation des pattes sur le boîtier :

Vss	1		40	Vdd
ND	2		39	TESTI
ACB	3		38	ACC
ACA	4		37	DA0
NC	5		36	DA1
IOB7	6		35	DA2
IOB6	7		34	DA3
IOB5	8		33	DA4
IOB4	9	AY3	32	DA5
IOB3	10	8910	31	DA6
IOB2	11		30	DA7
IOB1	12		29	BC1
IOB0	13		28	BC2
IOA7	14		27	BDIR
IOA6	15		26	SEL
IOA5	16		25	A8
IOA4	17		24	A9
IOA3	18		23	RESET
IOA2	19		22	CLOCK
IOA1	20		21	IOA0

A8 Entrée : toujours à 1

A9 Entrée : toujours à 0

AC Sortie : Canal analogique.
Ces sorties sont directement issues des convertisseurs digital/analogique. Elles produisent des signaux crête-crête d'un volt maximum.

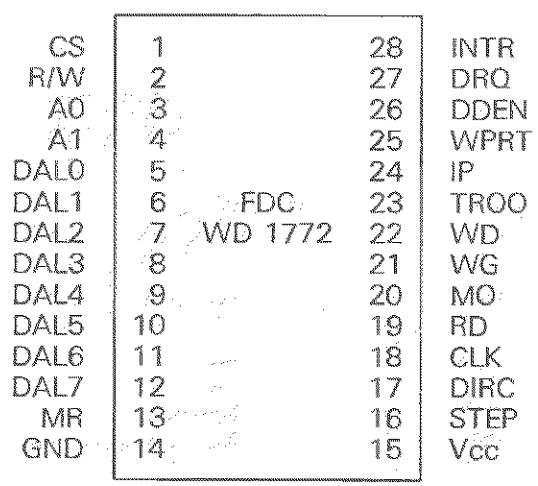
BC1, BC2 Entrée : Contrôle de bus 1 et 2
BDIR et ces entrées permettent de définir la fonction du circuit sonore comme indiqué dans le tableau suivant :

BDIR	BC2	BC1	Fonction
0	1	0	Inactif
0	1	1	Lecture
1	1	0	Ecriture
1	1	1	Adresse à mémoriser par le AY3

BDIR	Entrée : Contrôle de la logique interne
CLOCK	Entrée : Signal d'horloge Cette broche permet de cadencer les générateurs de ton, de bruit et d'enveloppe.
DA0-DA7	Bidirectionnel : Bus de données Ces 8 broches véhiculent : — les données et adresses envoyées par le microprocesseur au circuit sonore ; — les données envoyées par le circuit sonore et à destination du microprocesseur. Quand le mode adresse est actif, les bits DA0 à DA3 sélectionnent un des registres internes.
IOA0-IOA7	Bidirectionnel : Port de sortie A
IOB0-IOB7	Bidirectionnel : Port de sortie B
NC	Non connecté
RESET	Entrée : Remise à zéro des registres internes du circuit
SEL	Entrée : Sélection de la fréquence d'horloge
TESTI	Entrée : Broche de test
VDD	Entrée : Alimentation + 5 V
VSS	Entrée : Masse de l'alimentation

III - Le contrôleur de disque WD 1772

Le contrôleur de disque WD 1772 est un circuit LSI (*Large Scale Integration*, c'est-à-dire intégration à grande échelle) de la société américaine Western Digital. Il permet de gérer un ou deux lecteurs de disquettes aux formats IBM 3740 (simple densité) et IBM 34 (double densité). Le brochage de ce circuit est le suivant :



A0-A1 Entrée : Sélection des registres de commande
Ces deux connecteurs permettent de sélectionner un des registres internes du contrôleur. Le tableau suivant donne le nom du registre en fonction de la valeur de ces connecteurs et du connecteur de lecture/écriture (R/W) :

A1	A0	Lecture (R/W = 1)	Ecriture (R/W = 0)
0	0	Registre d'état	Registre de commande
0	1	Registre de piste	Registre de piste
1	0	Registre de secteur	Registre de secteur
1	1	Registre de données	Registre de données

CLK Entrée : Horloge
Base de temps interne (8 MHz).

CS Entrée : Chip Select (Sélection du circuit)
A l'état bas, le circuit WD est sélectionné et des données peuvent être échangées entre lui et le processeur maître.

DAL0-DAL7 Bidirectionnel : Bus de données
Ce bus est utilisé pour tous les transferts de données. Il est validé par les broches CS et R/W.

DDEN Entrée : Densité (simple ou double)
Un état bas sur cette broche définit le mode double densité. Un état haut définit le mode simple densité.

DIRC Sortie : Direction du déplacement de la tête
Un état haut sur cette broche indique que le bras du lecteur doit se déplacer vers le centre du disque. Un état bas indique que le bras doit se déplacer vers l'extérieur du disque.

DRQ Sortie : Etat du registre de données
Cette sortie est dans un état haut lorsque le registre de données est plein en lecture, ou vide en écriture.

INTR Sortie : Demande d'interruption
Cette sortie est positionnée (état haut) à la fin de l'exécution de chaque commande. Elle est mise à zéro par la lecture du registre d'état.

IP Entrée : Détection du trou d'index
Cette entrée est mise à un à chaque tour complet de la disquette.

GND Entrée : Masse de l'alimentation

MO Sortie : Mise sous tension du moteur
Lorsque cette sortie vaut 1, le moteur du lecteur de disquettes est mis sous tension.

MR Entrée : Remise à zéro du circuit
Une impulsion (état haut vers état bas) sur cette entrée initialise le circuit et ses registres internes.

RD Entrée : Direction des données
Cette entrée reçoit des impulsions du lecteur de disquettes qui renseignent le WD sur la fréquence et la lecture des données.

R/W Entrée : Read/Write (Lecture/Ecriture)
Lorsque cette entrée vaut zéro, une donnée qui se trouve dans un des registres internes du WD est envoyée au 68000. Lorsque cette entrée vaut un, le 68000 initialise un des registres internes du WD.

STEP Sortie : Pas du moteur
Cette sortie est reliée au lecteur de disquettes. Elle permet de déplacer la tête de lecture/écriture d'un pas variable.

TR00 Entrée : Piste 0
Lorsqu'elle est dans un état bas, cette entrée signale au WD que la tête de lecture/écriture est positionnée au-dessus de la piste zéro.

Vcc Entrée : Alimentation + 5 V

WD Sortie : Emission de données
Sur cette sortie sont envoyées toutes les impulsions (destinées au lecteur de disquettes) de fréquences et de données.

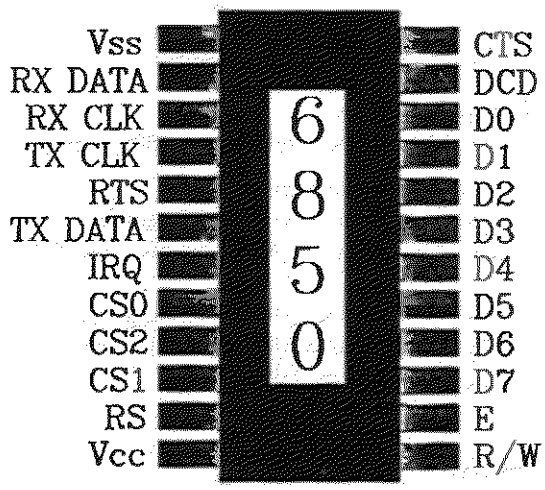
WG Sortie : Verrouillage de la logique d'écriture
Cette sortie est dans un état haut avant chaque opération d'écriture. Mis à part ce cas précis, elle est toujours dans un état bas.

WPRT Entrée : Protection contre l'écriture
Cette entrée est dans un état bas lorsque la disquette est protégée en lecture et dans un état haut si la disquette n'est pas protégée en écriture. Elle est testée avant toutes les opérations d'écriture sur disque.

IV - ACIA 6850

Le MC 6850 est un circuit d'émission/réception série de données (ACIA = *Asynchrone Communication Interface Adapter*) fabriqué par Motorola. Il est utilisé pour les communications asynchrones entre le microprocesseur et un organe série périphérique.

Son brochage est le suivant :



Vss Entrée : Masse de l'alimentation.

Vcc Entrée : + 5 Volts.

RX DATA Entrée : Receive DATA
Réception série des bits de donnée.

RX CLK Entrée : Receive Clock
Horloge des données en réception.

TX CLK Entrée : Transmitter Clock
Horloge des données en émission.

RTS Sortie : Request To Send
Lorsque cette broche est au niveau bas, l'ordinateur désire émettre des données.

TX DATA Sortie : Transmitter Data
Cette broche est utilisée pour la transmission série des données de l'ordinateur vers un périphérique.

IRQ Sortie : Interrupt Request
Lorsque cette broche est au niveau bas, un service (émission ou réception) est demandé au microprocesseur.

CS0 à CS2 Entrée : Chip Select
Ces broches permettent de mettre en ou hors fonction l'ACIA.

RS Entrée : Register Status
Cette broche permet d'accéder à un des registres internes de l'ACIA.

CTS Entrée : Clear To Send
Le microprocesseur autorise l'émission de données suite à un signal RTS actif.

DCD Entrée : Data Carrier Detect (détection de porteuse)
Inhibition ou validation de la réception de données série.

D0 à D7 Bidirectionnel : Data Bus
Bus de données bidirectionnel dans l'état haute impédance jusqu'à ce que l'ACIA entre en fonction.

E Entrée : Enable
Cette broche permet de synchroniser l'entrée ou la sortie de données.

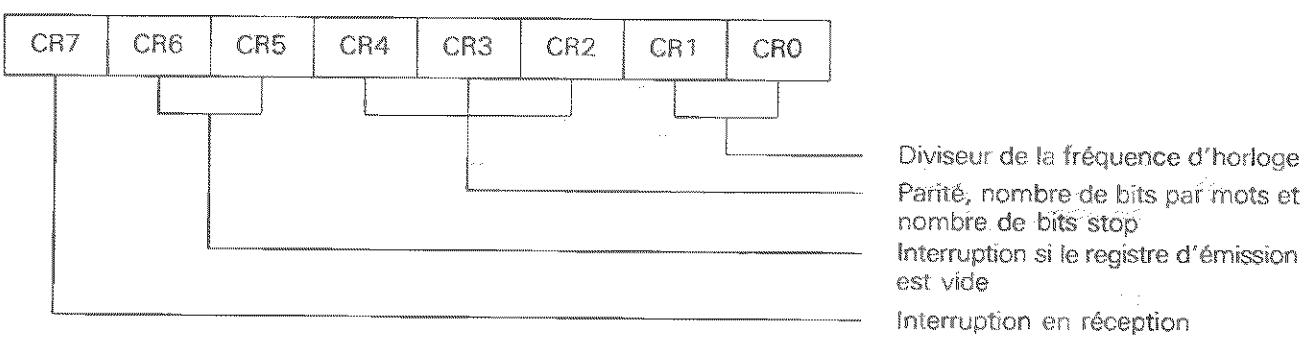
R/W Entrée : Read/Write
Cette broche, associée à RS, permet d'accéder en lecture ou en écriture à un des registres internes de l'ACIA.

L'ACIA MC 6850 comporte quatre registres internes accessibles en lecture et en écriture par l'intermédiaire des connecteurs RS (*Register Status*) et R/W (*Read/Write*).

Registre de contrôle

Ce registre est accessible en écriture lorsque CS = 0 et R/W = 0.

Il contient diverses informations codées comme suit :



Ces bits de données doivent être utilisés de la manière suivante :

CR1	CR0	Fonction
0	0	Pas de division
0	1	Division par 16
1	0	Division par 64
1	1	Reset de l'ACIA

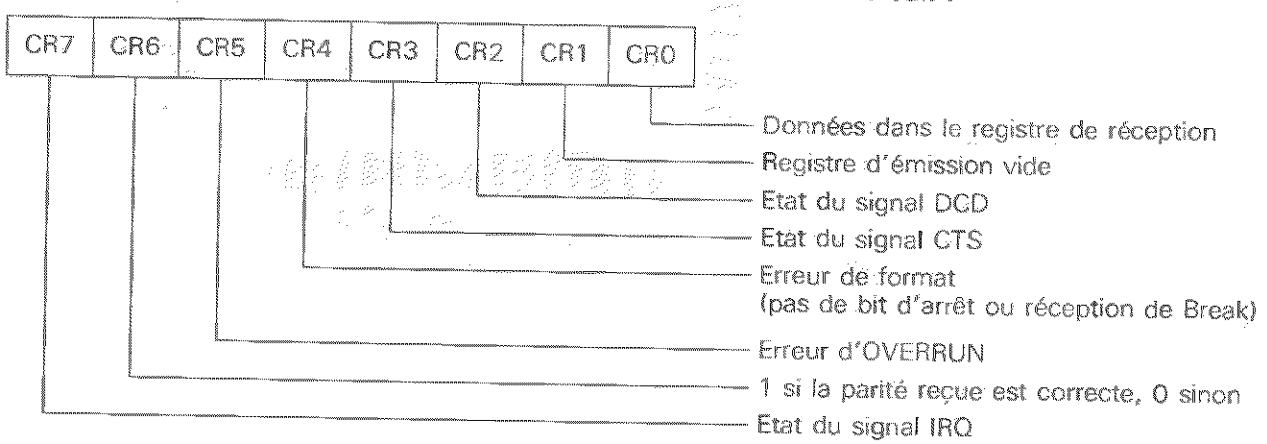
CR4	CR3	CR2	Fonction Bits de données, bits stop, parité
0	0	0	7, 2, paire
0	0	1	7, 2, impaire
0	1	0	7, 1, paire
0	1	1	7, 1, impaire
1	0	0	8, 2, sans
1	0	1	8, 1, sans
1	1	0	8, 1, paire
1	1	1	8, 1, impaire

CR6	CR5	Fonction
0	0	IT TX interdite si RTS = 0
0	1	IT TX autorisée si RTS = 0
1	0	IT TX interdite si RTS = 1
1	1	IT TX interdite si RTS = 0, et envoi d'un BREAK

CR7	Fonction
0	IT RX interdites
1	IT RX autorisées (registre RX plein, modif du signal DCD, donnée écrasée)

Registre d'état

Ce registre est accessible en lecture lorsque CS = 0 et R/W = 1. Il contient diverses informations codées comme suit :



Registre d'émission

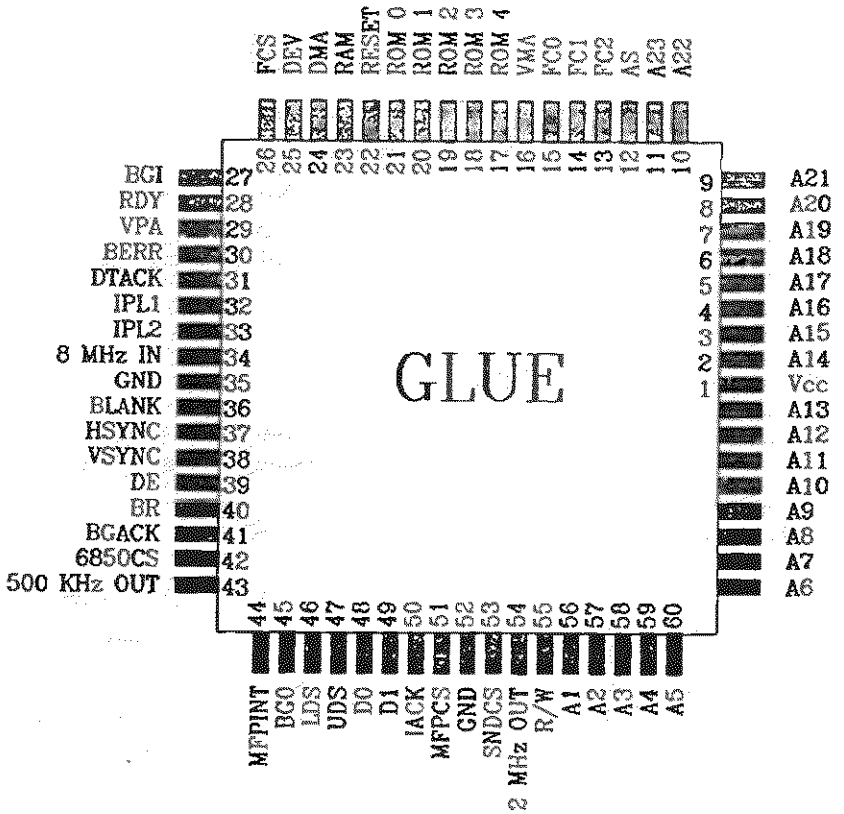
Ce registre est accessible en écriture lorsque CS = 1 et R/W = 0. Il est chargé par le bus de données et contient les caractères à émettre sur le port série.

Registre de réception

Ce registre est accessible en lecture lorsque CS = 1 et R/W = 1. Il est chargé par le bus de données et contient les données reçues sur le port série.

V - GLUE

Ce circuit développé par Atari réalise de nombreuses fonctions dont, entre autres, la gestion des tampons, le décodage des adresses système, la division de la fréquence d'horloge, etc. Il est totalement inaccessible au programmeur. Son brochage est le suivant :

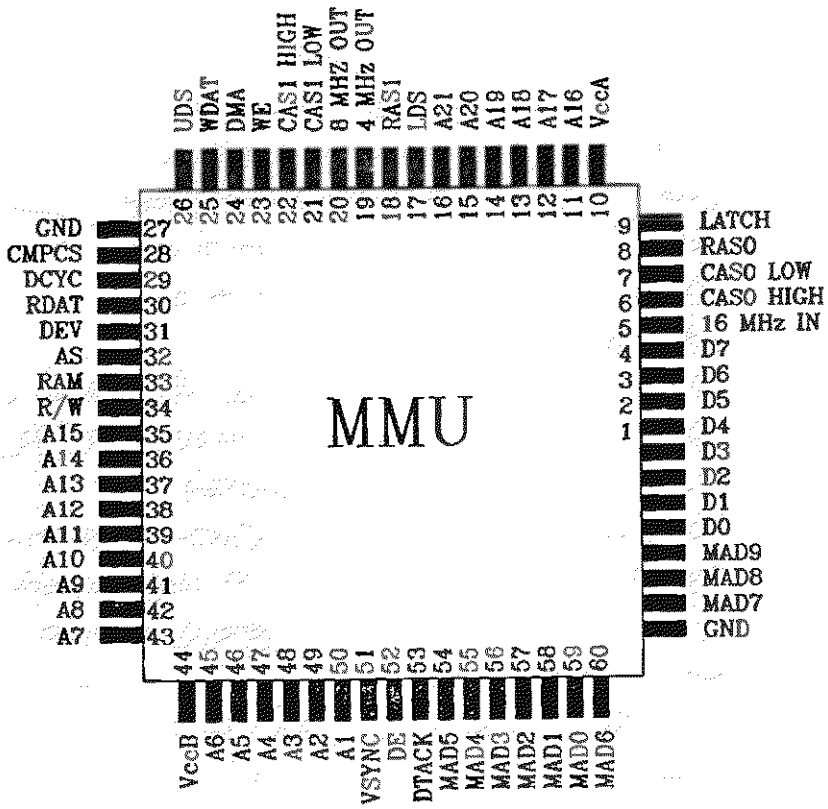


Vcc	Alimentation +5 Volts.
Gnd	Masse de l'alimentation.
A1-A23	Entrée : Bus d'adresses.
AS	Entrée : Adresse valide.
FC0-FC2	Entrée : Etat du processeur.
D0-D1	Bidirectionnel : Bus de données.
VMA	Entrée : Validation d'adresse mémoire.
RESET	Entrée : Initialisation du circuit.
BG1	Entrée : Attribution de bus
8 MHz IN	Entrée : Horloge.
LDS	Entrée : Sélection du LSB (octet de poids faible).
UDS	Entrée : Sélection du MSB (octet de poids fort).
R/W	Entrée : Lecture/Ecriture.
ROM0-ROM4	Sortie : Adresses d'activation des ROM.
SNDCS	Sortie : Activation du générateur sonore.
2 MHz OUT	Sortie : ACIA et générateur sonore.
500 KHz OUT	Sortie : ACIA.
6850 CS	Sortie : Chip Select de l'ACIA.
BLANK	Sortie : Gestion d'écran.
HSYNC-VSYNC	Sortie : Synchronisation horizontale et verticale du spot lumineux.
BR	Sortie : Demande d'accès au bus.
BGACK	Sortie : Attribution du bus.
DEV	Sortie : Validation du MMU.
FCS, IACK, MFPCS, RDY	Sortie : Interfaçage avec le 68901.
VPA	Sortie : Validation de l'adresse d'un périphérique.
BERR	Sortie : Erreur bus.
DTACK	Sortie : Validation de transfert de données.
IPL1-IPL2	Sortie : Priorité des interruptions.

VI - MMU

Ce circuit développé par Atari est utilisé comme complément des circuits SHIFTER et DMA. Il est totalement inaccessible au programmeur.

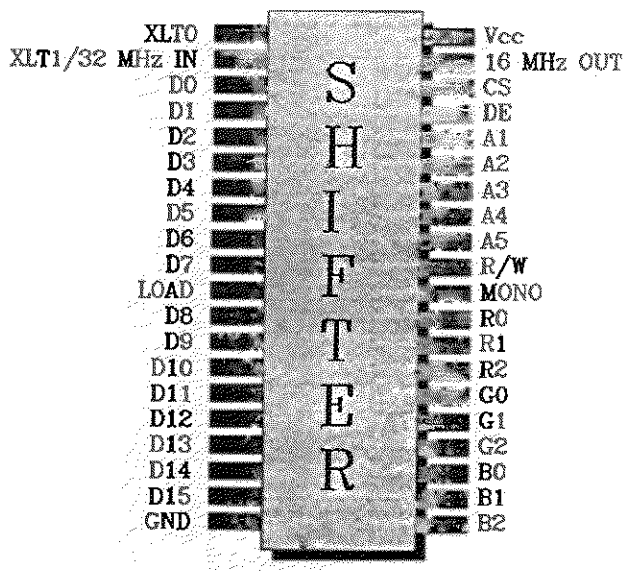
Son brochage est le suivant :



Vcc	Alimentation + 5 volts.
Gnd	Masse de l'alimentation.
16 MHz IN	Entrée : Horloge.
A1 à A21	Entrée : Bus d'adresses.
AS	Entrée : Adresse valide.
R/W	Entrée : Lecture/Ecriture.
D0 à D7	Bidirectionnel : Bus de données.
CAS, RAS	Sortie : Activation de la mémoire RAM.
RDAT, WDAT	Sortie : Lecture et écriture de données.
CMPCS	Sortie : Chip Select du Shifter.
MAD0 à MAD9	Sortie : Activation des adresses mémoire.
DIACK	Sortie : Validation de transfert de données.

VII — SHIFTER

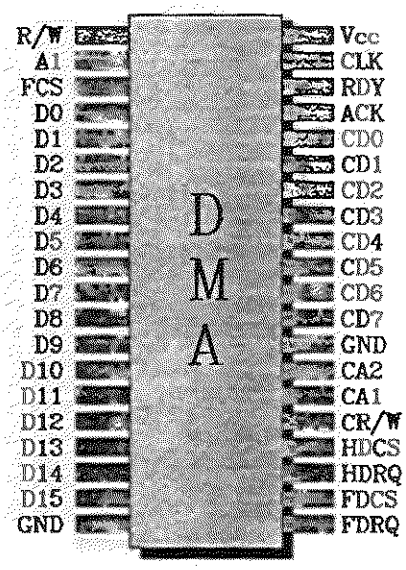
Ce circuit développé par Atari délivre des impulsions destinées au moniteur, à partir des données qui se trouvent dans la RAM vidéo. Il est donc responsable de la partie affichage, quels que soient la résolution ou le type du moniteur. Son brochage est le suivant :



Vcc	Alimentation + 5 volts.
Gnd	Masse de l'alimentation.
XLT	Entrée : Horloge.
LOAD	Entrée : Chargement MMU.
CS	Entrée : Sélection du Shifter.
A1 à A5	Entrée : Bus d'adresses.
R/W	Entrée : Lecture/Ecriture.
D0 à D15	Bidirectionnel : Bus de données.
MONO	Sortie : Vidéo monochrome.
16 MHz OUT	Sortie : Horloge/2.
R0 à R2	Sortie : Composante rouge du signal vidéo.
G0 à G2	Sortie : Composante verte du signal vidéo.
B0 à B2	Sortie : Composante bleue du signal vidéo.

VIII — DMA

Ce circuit développé par Atari permet d'effectuer des transferts de données rapides sans passer par le microprocesseur. Il fonctionne essentiellement avec les lecteurs de disquettes et de disques durs. Son brochage est le suivant :



Vcc	Alimentation + 5 volts.
Gnd	Masse de l'alimentation.
R/W	Entrée : Lecture/Ecriture.
A1	Entrée : Choix d'un registre adresse de base.
FCS	Entrée : Sélection du circuit DMA.
CLK	Entrée : Horloge.
RDY	Entrée : Prêt à l'émission/réception de données.
CR/W	Entrée : Transfert de données en lecture ou en écriture.
D0 à D15	Bidirectionnel : Bus de données.
ACK	Sortie : Autorisation de DMA.
CA1 et CA2	Sortie : Choix des registres de sélection.
HDCS et FDCS	Sortie : Choix du circuit de transfert.
HDRQ et FDRQ	Sortie : Réception des demandes de transfert.



2/1.5

La configuration mémoire

Le domaine adressable des Atari ST est divisé en trois grands groupes :
— RAM : vecteurs d'exception, écran et mémoire de travail ;
— ROM : système d'exploitation et ROM d'extension ;
— Entrées/Sorties : zones de communication avec les circuits intégrés spécialisés.
Le schéma suivant fait apparaître les zones adressables occupées par chacun de ces groupes :

FFFE00	I/O : Registres (ACIA)
FFFC00	I/O : Registres (MPF)
FFFA00	I/O : Registres (Circuit sonore)
FF8800	I/O : Registres (DMA et FDC)
FF8600	RESERVE
FF8400	I/O : Registres (SHIFTER)
FF8200	I/O : Registres (configuration)
FF8000	ROM : Fin de la ROM
FEFFFF	ROM : Boot
FC0000	ROM : Connecteur d'extension
FA0000	RAM : Fin de la RAM pour les Méga ST 2
200000	RAM : Fin de la RAM pour les 1040 ST
100000	RAM : Fin de la RAM pour les 520 ST
080000	RAM
020000	RAM : Table des vecteurs d'exception
000008	ROM : Initialisation du PC
000004	ROM : Initialisation de la pile superviseur
000000	

2/2

La connectique Atari

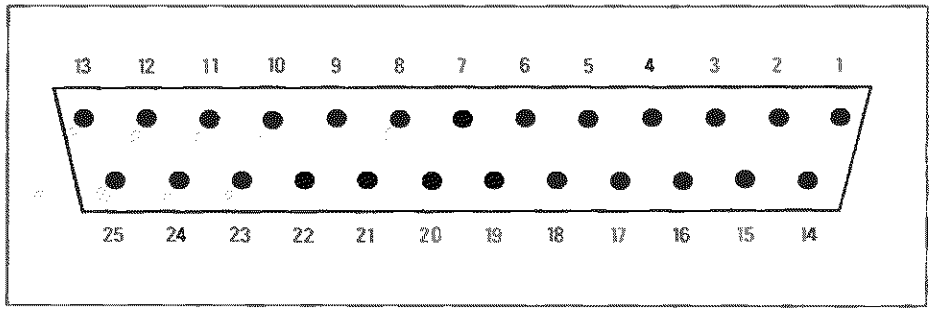
2/2.1

La description matérielle des connecteurs de base

I - Le connecteur Centronics

Ce connecteur permet de relier l'Atari à une imprimante parallèle compatible CENTRONICS.

Son brochage est le suivant :



Le brochage du connecteur Centronics.

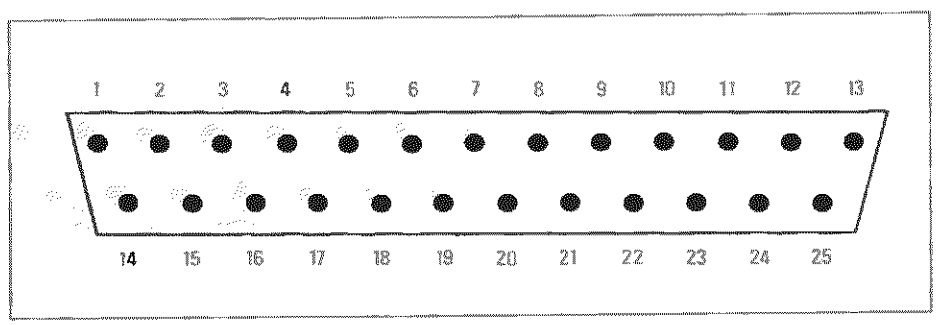
Broche	Fonction
1	STROBE
2	Data 0
3	Data 1
4	Data 2
5	Data 3
6	Data 4
7	Data 5
8	Data 6
9	Data 7
10	Non connecté
11	BUSY
12 à 17	Non connecté
18 à 25	Masse

II - Le connecteur RS 232

Ce connecteur permet de relier l'Atari à un modem ou à un autre ordinateur, via un programme de (télé)communications.

Son brochage est le suivant :

Broche	Fonction
1	Masse châssis
2	Données transmises
3	Données reçues
4	RTS (*Request To Send*)
5	CTS (*Clear To Send*)
6	Non connecté
7	Masse
8	DCR (*Data Carrier Detect*)
9 à 19	Non connecté
20	DTR (*Data Terminal Ready*)
21	Non connecté
22	Ring
23-25	Non connectés



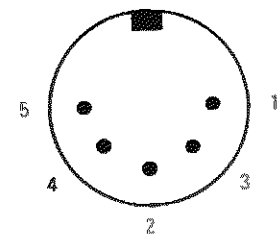
Le brochage du connecteur RS 232.

III - Le connecteur musical MIDI

Le connecteur MIDI IN permet de relier l'Atari à l'entrée d'instruments qui respectent la norme MIDI (*Musical Instrument Digital Interface*). Ces instruments sont par exemple des séquenceurs, synthétiseurs, boîtes à rythmes, etc.

Son brochage est le suivant :

Broche	Fonction
1	THRU (Emission des données)
2	Masse
3	Masse THRU
4	IN (Réception des données)
5	Masse IN

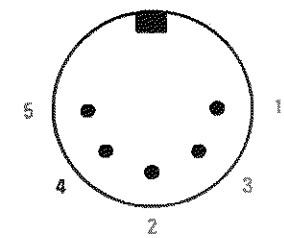


Connecteur MIDI « IN ».

Le connecteur MIDI OUT permet de relier l'Atari à la sortie d'instruments qui respectent la norme MIDI.

Son brochage est le suivant :

Broche	Fonction
1	THRU (Emission des données)
2	Masse
3	Masse THRU
4	OUT (Emission des données)
5	Masse OUT



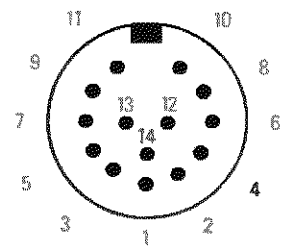
Connecteur MIDI « OUT ».

IV - Le connecteur lecteurs de disquettes

Le connecteur lecteurs de disquettes permet de relier l'Atari à un ou deux lecteurs de disquettes 80 pistes (simple ou double face, 3' 1/2 ou 5'1/4).

Son brochage est le suivant :

Broche	Fonction
1	Lecture des données
2	Face 0
3	Masse
4	Impulsion Index
5	Lecteur 0
6	Lecteur 1
7	Masse
8	Déclenchement du moteur
9	Direction
10	Pas de déplacement
11	Ecriture des données
12	Porte d'écriture
13	Piste 0
14	Protection en écriture



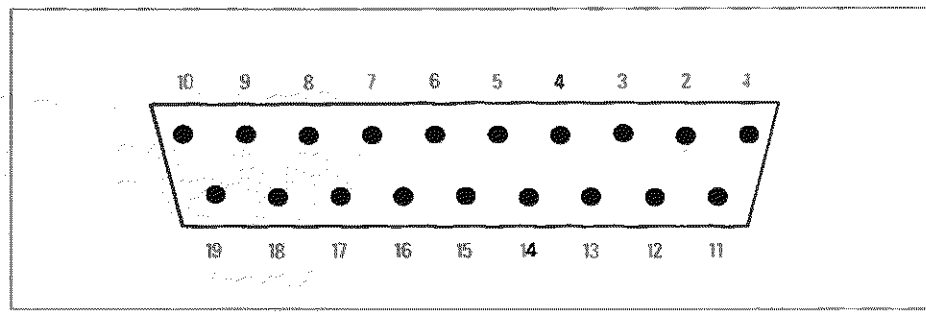
Connecteur du lecteur de disquettes.

V - Le connecteur disque dur

Le connecteur disque dur permet de relier l'Atari à un disque dur (à travers l'interface AHDI qui se trouve sur l'unité de disque).

Son brochage est le suivant :

Broche	Fonction
1	Data 0
2	Data 1
3	Data 2
4	Data 3
5	Data 4
6	Data 5
7	Data 6
8	Data 7
9	CS (*Chip Select*)
10	Demande d'interruption
11	Masse
12	Reset
13	Masse
14	ACK (accusé de réception)
15	Masse
16	AI
17	Masse
18	Lecture/Ecriture
19	DRQ (Demande de données)



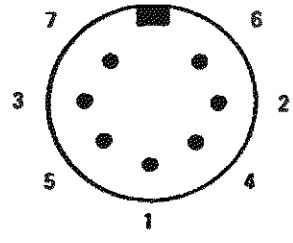
Connecteur de disque dur.

VI - Le connecteur d'alimentation

Le connecteur d'alimentation permet de relier l'Atari au secteur. Il contient les diverses tensions stabilisées nécessaires à l'alimentation des composants de la carte.

Son brochage est le suivant :

Broche	Fonction
1	+ 5 Volts
2	Non connecté
3	Masse
4	+ 12 Volts
5	− 12 Volts
6	+ 5 Volts
7	Masse



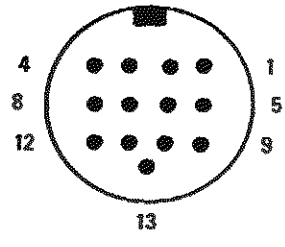
Connecteur d'alimentation.

VII - Le connecteur de moniteur

Le connecteur de moniteur permet de réaliser l'Atari à un moniteur monochrome ou couleur, ou à un téléviseur muni d'un connecteur PERITEL.

Son brochage est le suivant :

Broche	Fonction
1	Sortie audio
2	Réservé (Sortie)
3	GPO (*General Purpose Output*) : Sortie à usage général
4	Détection monochrome (niveau bas)
5	Entrée audio
6	Vert
7	Rouge
8	Masse
9	Synchronisation horizontale
10	Bleu
11	Luminosité en monochrome
12	Synchronisation verticale
13	Masse



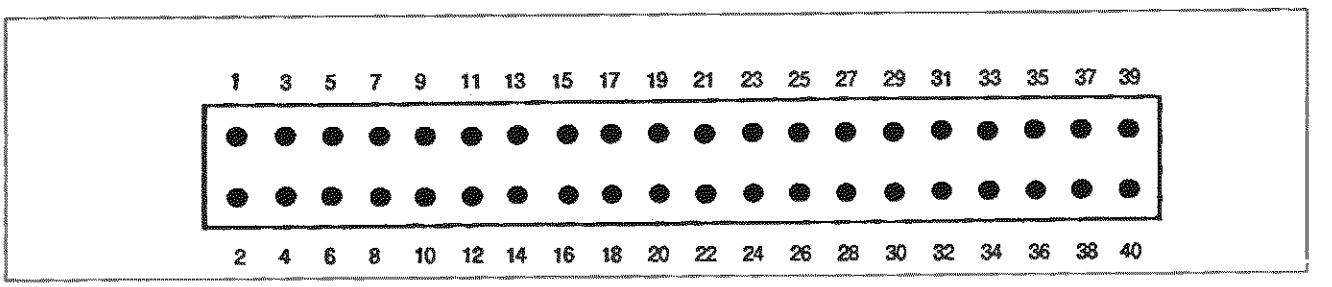
Connecteur de moniteur.

VIII - Le connecteur de cartouche

Le connecteur de cartouche permet de relier l'Atari à une cartouche d'extension de ROM de 128 Koctets (applications ou diagnostics).

Son brochage est le suivant :

Broche	Fonction
1 et 2	+ 5 V
3	Data 14
4	Data 15
5	Data 12
6	Data 13
7	Data 10
8	Data 11
9	Data 8
10	Data 9
11	Data 6
12	Data 7
13	Data 4
14	Data 5
15	Data 2
16	Data 3
17	Data 0
18	Data 1
19	Adresse 13
20	Adresse 15
21	Adresse 8
22	Adresse 14
23	Adresse 7
24	Adresse 9
25	Adresse 6
26	Adresse 10
27	Adresse 5
28	Adresse 12
29	Adresse 11
30	Adresse 4
31	ROM Select 3
32	Adresse 3
33	ROM Select 4
34	Adresse 2
35	UDS (Validation donnée supérieure)
36	Adresse 1
37	LDS (Validation donnée inférieure)
38-40	Masse



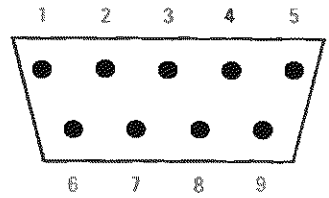
Connecteur de cartouche (ROM).

IX - Le connecteur de souris et manette

Le connecteur de Souris (port 0) permet de relier l'Atari à une souris Atari.

Son brochage est le suivant :

Broche	Fonction
1	XB/UP
2	XA/DOWN
3	YA/LEFT
4	YB/RIGHT
5	Non connecté
6	LEFT BUTTON/FIRE
7	+ 5 V
8	GND
9	RIGHT BUTTON

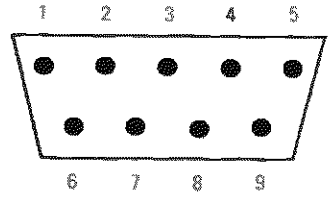


Connecteur souris et manette.

Le connecteur manette (port 1) permet de relier l'Atari à une manette de jeu.

Son brochage est le suivant :

Broche	Fonction
1	UP
2	DOWN
3	LEFT
4	RIGHT
5	Réservé
6	FIRE
7	+ 5 V
8	GND
9	Non connecté



Connecteur manette.

2/3

Les périphériques de base

2/3.1

Les moniteurs

Le faible prix des Atari ST est lié au nombre important de fonctions réalisées par les circuits spéciaux (MMU, GLUE et SHIFTER).

Le SHIFTER délivre des impulsions destinées au moniteur vidéo en fonction des informations qui se trouvent dans la mémoire RAM vidéo. Lorsque le moniteur est de type monochrome, l'information est délivrée sur la broche MONO. Dans le cas d'un moniteur couleur, l'information est envoyée au moniteur grâce aux connecteurs R (Red), G (Green) et B (Blue).

La mémoire RAM vidéo est constituée d'un ou de plusieurs plans logiques. Elle occupe 32 Koctets et peut être configurée comme suit :

— un plan logique :
640 × 400 pixels, monochrome,

— deux plans logiques :
640 × 200 pixels, 4 couleurs,

— quatre plans logiques :
320 × 200 pixels, 16 couleurs.

2/3.2

Les périphériques de saisie

LE CLAVIER

Le clavier possède son propre processeur (6301). Ce dernier dialogue avec l'unité centrale à travers un des circuits ACIA. Il lui retourne un code sur 8 bits qui correspond à la touche (ou à la combinaison de touches) enfoncée(s). Grâce à cet artifice, l'unité centrale n'est ''dérangée'' que lorsqu'un code est disponible dans l'ACIA, d'où un gain de temps non négligeable.

Comme le montrent les schémas HARD 4 et HARD 5 (voir Partie 2, chapitre 1.2), la touche actionnée est repérée grâce à la fermeture d'un circuit électrique dans une matrice.

Remarque :

Le processeur clavier s'occupe également de la gestion d'une souris et d'une manette.

LA SOURIS

La souris transmet des informations à l'unité centrale à travers un des ACIA.

Ces informations concernent :

— la direction des déplacements dans les deux axes (X et Y),

— la valeur de ces déplacements,

— l'éventuel appui sur un des boutons de la souris.

Les informations concernant le déplacement de la souris sont acheminées grâce à quatre connecteurs :

— XA (2) pour les déplacements vers le bas,

— XB (1) pour les déplacements vers le haut,

— YA (3) pour les déplacements vers la gauche,

— YB (4) pour les déplacements vers la droite.

Ces informations sont transmises sous la forme d'impulsions électriques produites par quatre portes optiques et mises en forme par un amplificateur opérationnel.

L'appui sur le bouton de gauche est signalé sur le connecteur LEFT BUTTON (6). L'appui sur le bouton de droite est signalé sur le connecteur RIGHT BUTTON (9).



3

Sommaire

3/0 Plus loin
3/0.1 Le nouveau TOS/GEM 1.6 de l'Atari STE
I. Performances accrues
II. Principales différences

3/1 Définitions et rappels
3/1.1 Utilisation des toutines du TOS

3/2 Le BIOS

3/3 L'extended BIOS (XBIOS)
3/3.1 Programmation avec XBIOS
3/3.1.1 Le port A du circuit son
3/3.1.2 Le swap d'écran

3/4 Le GEMDOS
I. La présentation du GEMDOS
II. La gestion de fichiers
III. La gestion de l'écran et du clavier
IV. La gestion de la mémoire
V. Les messages d'erreur
VI. Les fonctions GEMDOS
3/4.1 Utilisation des fonctions GEMDOS
3/4.1.1 Le GEMDOS, les piles et quelques ruses

3/5 L'AES
3/5.1 La disposition des fonctions de l'AES
I. Les fonctions de contrôle des applications
II. Les fonctions de contrôle des événements
III. Les fonctions de gestion des menus
IV. Les fonctions de gestion des objets
V. Les fonctions de gestion des formulaires
VI. Les fonctions graphiques
VII. Les fonctions de gestion du presse-papier

VIII. Les fonctions de sélection d'un fichier
IX. Les fonctions de gestion des fenêtres
X. Les fonctions de gestion des ressources
XI. Les fonctions de gestion des commandes

3/6 Le Bureau GEM

3/6.1 L'émulateur de terminal VT 52

3/7 La gestion des périphériques de base

3/7.1 La gestion du clavier

3/8 Les éditeurs de ressources

3/0

Plus loin

Les nouveautés en matière de systèmes d'exploitation sont plutôt rares ces derniers temps. La sortie d'un nouveau modèle de la gamme Atari est le prétexte à une nouvelle version du TOS, système d'exploitation de la Gamme Atari ST.

Ce chapitre est donc inauguré par la description générale du TOS 1.6. Nous insisterons ici sur les différences avec les versions précédentes. Le contenu des chapitres de la partie 3 de votre ouvrage reste applicable, aux remarques mentionnées ci-dessous près, pour sa plus grande part au STE.

3/0.1

Le nouveau TOS/GEM 1.6 de l'Atari STE[1)]

La plus notable évolution de ce TOS/GEM est qu'il réside maintenant entièrement en mémoire morte (amenée à 256 Ko pour l'occasion) : plus n'est donc besoin de charger ce TOS depuis une disquette comme par le passé.

L'intégrateur graphique GEM est donc accessible dès la mise sous tension de l'Atari STE.

1) Les éditions WEKA remercient ATARI MAGAZINE de les avoir autorisées à reprendre certaines informations publiées dans le Dossier STE du numéro 6 du mois de novembre 1989.

Une autre amélioration notable est la compatibilité totale et bidirectionnelle des disquettes Atari et IBM PC/PS 3,5 pouces. Cette compatibilité n'était que partielle jusqu'à présent car si un Atari ST savait lire les disquettes IBM, l'inverse était impossible.

Attention, il n'est pas pour autant question de faire tourner tel ou tel logiciel IBM sur un Atari, la compatibilité s'arrêtant en effet au format et à la structure des disquettes. Un fichier ASCII pourra donc être lu indifféremment par les deux machines quelle qu'en soit l'origine. Un fichier exécutable sur l'une ne le sera pas sur l'autre.

Cette compatibilité limitée n'en est pas moins intéressante car elle permet d'échanger très facilement les données d'un standard à l'autre, et même des programmes non compilés, pourvu que le langage utilisé existe sur les deux machines.

La présentation graphique des différentes boîtes de GEM est sensiblement identique à celle maintenant familière des ST.

I - Performances accrues

Ce nouveau TOS a été grandement amélioré en ce qui concerne la gestion des fichiers et disques. Les temps de réalisation de certaines opérations comme la copie de fichier, la création de répertoire ou la compilation de programmes Basic GFA ont été divisés par 3 et quelquefois par 4.

Malgré ces améliorations, le TOS 1.6 n'occupe pas la totalité des 256 Ko de mémoire morte. Il semble bien que des extensions ou versions plus puissantes de ce TOS soient prévues dans un avenir que seul Atari est en mesure de préciser.

II - Principales différences

RÉINITIALISATION

Autrefois, le seul moyen de réinitialiser la machine à chaud était purement *hardware* et consistait à presser la touche RESET située à l'arrière de l'appareil. Avec TOS 1.6, il est possible, à l'instar des IBM PC et PS, de réinitialiser en pressant simultanément les touches CTRL *(control)*, ALT *(alternate)* et DEL *(delete)*. La réinitialisation à froid, quant à elle, c'est-à-dire complète avec perte totale de la mémoire dynamique, est possible maintenant en pressant les touches CTRL, ALT et SHIFT droite. Cette opération est nettement moins préjudiciable à l'électronique que la brutale mise hors puis sous tension.

GESTION DES FICHIERS

- Duplication

Cette opération, ô combien fastidieuse !, nécessitait une sorte de danse très désagréable consistant à passer de la disquette source à la disquette destination un peu comme si votre Atari ne savait pas vraiment ce qu'il voulait.



Ce cauchemar est terminé car la mémoire disponible est entièrement utilisée comme un tampon et les opérations mentionnées seront réduites au strict minimum voire inexistantes si vous disposez de 4 Mégaoctets.

- Déplacement de fichiers

Impossible sur les versions ST, cette opération très fréquente chez les possesseurs de compatibles IBM (*prune and graft*) consiste à recopier un fichier dans un répertoire et à l'effacer simultanément dans le répertoire où il était. Maintenant, avec TOS1.6, c'est possible... Il suffit de maintenir la touche CTRL enfoncée pendant tout le déplacement du fichier ou plutôt de son nom à l'aide de la souris. Lâchez la touche et le fichier tombe immédiatement dans le répertoire où vous vous trouvez.

- Indications de la boîte de dialogue de copie, destruction ou déplacement de fichiers

Dans les versions précédentes, seul le nombre de fichiers restant à traiter était indiqué. Le STE vous indique en plus le nom du fichier en cours de traitement.

- Interruptions de la copie, destruction ou déplacement de fichiers

Avant, c'était tout simplement impossible. Maintenant, la touche UNDO permet cela, une confirmation vous étant tout de même demandée.

- Gestion des conflits de noms de fichiers

Deux possibilités vous étaient offertes : écrasement pur et simple du fichier portant le même nom ou annulation de la commande de sauvegarde. Le TOS 1.6 offre une troisième option qui consiste à sauter au fichier suivant sans pour autant annuler la commande pour les fichiers restant à traiter. A noter que cette option peut être activée ou non dans la fenêtre **définir des préférences** du bureau GEM.

- Date et heure de création de fichier

Elles étaient réactualisées lors des opérations de copie (nous dirons sans vouloir vexer personne que c'était un BUG). Cette fois, elles restent identiques à celles d'origine.

- Sélection multiple de fichiers dans la fenêtre active

Il fallait maintenir la touche SHIFT enfoncée pendant les opérations de click sur les noms de fichiers successifs. Le pointage click à côté d'un nom réinitialisait à zéro la liste (encore ce que l'on pourrait appeler un BUG). Sur le STE, cette caractéristique gênante a disparu, et même un mauvais tireur atteint de strabisme et de la danse de Saint-Guy saura sélectionner des fichiers en grand nombre.

- Boîtes d'informations fichiers et répertoires

Les ST affichaient des boîtes différentes dans ces deux cas. La cohérence a été rétablie sur le STE : elles fournissent maintenant le nombre de fichiers et répertoires contenus par le répertoire sélectionné.

- **Nom des fichiers et répertoires**

Certains caractères illégaux (* et ?) étaient autorisés dans les noms de fichiers et répertoires, caractéristique pouvant conduire à bien des soucis. Le STE rétablit la rigueur en interdisant ces jokers dans les noms de fichiers et de répertoires.

- **Changement de nom des répertoires**

Impossible avec les TOS précédents, cette opération est tout à fait possible maintenant. Les remords quant aux noms de répertoires sont alors rapidement oubliés.

- **Visualisation des textes**

La visualisation se faisait par un défilement vertical ultrarapide, donc peu pratique. Le STE autorise le défilement des textes par demi-pages en appuyant sur les touches D ou CTRL D.

- **Impression des fichiers textes**

Impossible de stopper l'impression d'un texte autrement qu'en mettant l'imprimante hors tension. Sur le STE, on peut stopper cette impression par l'appui sur Q et UNDO ou sur CTRL C car le clavier est scruté tous les 16 caractères.

- **Sélection des fichiers**

Munie d'un superbe BUG sur l'option **abandonner**, cette sélection a été revue. De même, le choix du lecteur, inexistant sur l'ancienne version, est possible aujourd'hui. Le nombre de fichiers n'est plus limité comme précédemment et les noms de chemins (*path*) peuvent être beaucoup plus longs.

- **Fichiers cachés**

Ces fichiers, quoique théoriquement cachés, apparaissaient sur le bureau et dans le sélecteur de fichiers ... BUG, oserons-nous le dire ... corrigé cette fois.

GESTION DE DISQUES

- **Formattage et duplicata**

Les deux boîtes de dialogue de ces deux opérations ont été rendues semblables.

- **Boîte formattage**

L'option par défaut était le formattage (très dangereux). L'option par défaut est maintenant l'annulation ... ouf ... merci !

- **Boot sector**

Ce secteur si précieux était incompatible avec le MS/DOS (système d'exploitation des compatibles IBM). Son format a été rendu compatible.

- **Accès disques**

Les temps d'accès ont été divisés par 2 et 3 en ce qui concerne les créations, suppressions et écritures de fichiers.



L'INTEGRATEUR GRAPHIQUE GEM

- Lancement des programmes GEM

Le lancement automatique des programmes GEM, jusqu'alors impossible, peut se faire maintenant en installant l'application (option du bureau GEM) et en sélectionnant l'option **AUTO**. On sauve le tout dans le bureau GEM et c'est tout.

- Affichage d'une fenêtre lors de son ouverture

Nous dirons qu'il se fait un peu au petit bonheur la chance, c'est-à-dire n'importe où, dans l'écran tout de même. Le nouveau GEM ouvre toute nouvelle fenêtre là où la précédente a été fermée... C'est une logique comme une autre, mais au moins, c'est logique.

- Demande du catalogue d'une disquette non formattée

Une fenêtre vide était affichée, très peu différente de celle d'une disquette formattée mais vide. La nouvelle version de GEM n'ouvre plus de fenêtre en présence d'une disquette non formattée.

- Les fenêtres catalogue

Ces fenêtres ne pouvaient afficher que 400 fichiers au maximum. La seule limite, maintenant, est la mémoire vive disponible. Les fenêtres inactives n'étaient pas remises à jour après une manipulation de fichier ou de disque : elles le sont cette fois.

D'autre part, le système ne relisait que les catalogues des disques A et B alors qu'aujourd'hui, il relit tous les catalogues de tous les disques après chaque manipulation de fichier ayant pu altérer leur contenu.

- Relecture de catalogue

L'appui de la touche ESC (*escape*) forçait le système à relire le catalogue du lecteur de la fenêtre active sauf si la disquette contenue dans celui-ci n'avait pas été changée. Dans la nouvelle version, on relit le catalogue dans tous les cas.

- Lancement d'application

On ne pouvait lancer une application qu'à partir de la fenêtre active. On peut maintenant lancer à travers toute fenêtre.

LE BUREAU GEM

- Sauvegarde du bureau

La sauvegarde du bureau GEM était immédiate alors qu'un message d'alerte apparaît s'il ne reste pas assez de place sur le disque. Un BUG existe cependant sur les premières versions de TOS 1.6 qui empêche l'initialisation en moyenne résolution. (BUG sur DESKTOP.INF).

- Boîtes de dialogue du bureau

Dans les champs numériques, les vides étaient signalés par des points alors que des blancs sont maintenant indiqués. De plus, les commentaires sont beaucoup plus concis.

- Boîte d'initialisation de lecteur

L'option par défaut **annuler** fait place à l'option **installer**.

3/1

Définitions et rappels

Le système d'exploitation des Atari ST porte le nom de GEMDOS (ou de TOS : *Tramiel Operating System*, Jack Tramiel étant le nom du directeur d'Atari). Pour mieux comprendre son utilité, rappelons sommairement ce qu'est un ordinateur.

Pour ce faire, nous allons utiliser la technique des boîtes noires. Considérons qu'un ordinateur n'est ni plus ni moins qu'une boîte noire, dans laquelle entrent et sortent des informations :

entrées → [ordinateur] → sorties

Quelle est la nature des informations à fournir à l'ordinateur ou délivrées par l'ordinateur ?

Entrées : données entrées au clavier ou reçues par des interfaces de communications (MODEM par exemple),

Sorties : données affichées sur l'écran, imprimées, stockées dans des mémoires de masse (disquettes, disque dur, cartes magnétiques, etc.), envoyées à des périphériques par l'intermédiaire d'interfaces de communication.

Un ordinateur est donc un système :

— capable de traiter des informations issues d'organes d'entrée, comme des ordres utilisateurs provenant du clavier, des ordres ou données arrivant par une interface de communication, etc.

— capable d'envoyer le résultat du traitement sur différents périphériques, de les stocker sur des supports magnétiques et de les émettre vers des interfaces de communication.

Pour réaliser ces opérations, un ordinateur est composé d'une carte mère regroupant des composants qui peuvent être répartis en trois grandes familles :

1) le ou les microprocesseur(s),

2) les mémoires (RAM, ROM, EPROM, etc.),

3) les circuits spécialisés dans la gestion des périphériques (écran, clavier, modem, etc.).

Les composants des types 2 et 3 suivent les directives du ou des microprocesseurs exprimées dans un langage de communication de bas niveau : le binaire.

Pour faciliter l'utilisation des ressources de l'ordinateur, des langages semi-évolués ou évolués ont été créés : Assembleur, Basic, Pascal, C, etc.

Le volume des données à traiter n'a cessé d'augmenter. Alors est apparu le besoin de stocker ces données sur des unités magnétiques (cassettes, disquettes, puis disques durs).

Devant les grandes possibilités de stockage des lecteurs de disquettes ou de disques durs, les constructeurs ont réagi en créant des systèmes d'exploitation.

Ces logiciels facilitent :

— la gestion de l'espace disque,

— la gestion des entrées/sorties.

Les systèmes d'exploitation sont des programmes constitués d'un ensemble de fonctions représentées par des mots clés.

FACILITÉ ET RAPIDITÉ D'ACCÈS AUX FICHIERS SUR DISQUE

Pour accéder à un fichier sur disque, il suffit de donner son nom au TOS. C'est le TOS qui se charge de convertir ce nom en une adresse physique et d'extraire les données demandées du support magnétique.

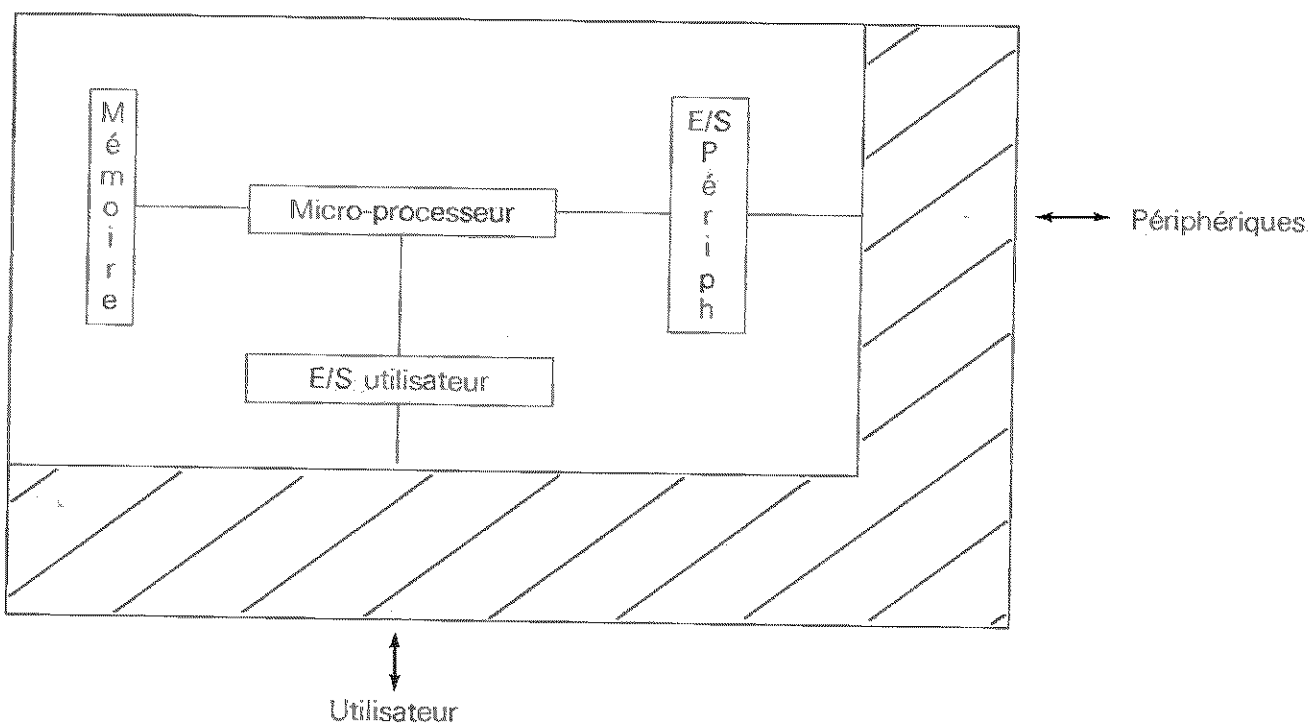
De même, le TOS est capable de stocker des données sur disque, rapidement, et là où aucune autre donnée n'est déjà stockée. Les données à stocker seront éventuellement morcelées pour occuper les trous libres sur le disque si celui-ci est relativement plein.

Le TOS permet également de visualiser la liste et l'espace occupé par les fichiers stockés sur une unité de disque. Ces fichiers peuvent également être effacés et renommés par son intermédiaire.

FACILITÉ ET RAPIDITÉ D'ACCÈS AUX PÉRIPHÉRIQUES

Les organes périphériques sont essentiellement le clavier, l'écran, l'imprimante et le lecteur de disquettes. Ils sont reliés à l'ordinateur par des connexions standard, et échangent avec lui des messages selon un protocole de communication bien défini. Les différences existant entre deux protocoles sont très grandes. Grâce au système d'exploitation, les entrées/sorties sont banalisées, et l'utilisateur peut s'adresser aux périphériques sans tenir compte de ces différences.

Pour schématiser, le système d'exploitation TOS est une couche logicielle d'interfaçage entre la carte mère et les périphériques. Ces périphériques communiquent avec la carte mère grâce à un protocole de communication qui leur est propre.



Le TOS comporte plusieurs parties dédiées à la gestion de l'écran, des périphériques, de la mémoire, etc. Dans la suite de ce chapitre, nous allons donner en détail la manière optimale d'utiliser les fonctions liées à ces diverses parties.

3/1.1

Utilisation des routines du TOS

LES FONCTIONS GEMDOS, BIOS ET XBIOS

Généralités

Le système d'exploitation de l'Atari (TOS) est un ensemble de routines ou sous-programmes qui permettent à l'utilisateur de dialoguer avec les périphériques. Le TOS est constitué de trois composantes : le GEMDOS *(GEM Disk Operating System)*, le BIOS *(Basic Input Output System)* et le XBIOS *(eXtended BIOS)*. Le GEMDOS est le niveau supérieur du TOS et gère les traitements orientés « problème » tandis que le BIOS et le XBIOS sont les deux parties qui traitent du hardware. Le BIOS s'occupe des opérations classiques que l'on retrouve sur tous les ordinateurs alors que le XBIOS traite des particularités de l'Atari. Le GEMDOS n'a aucun contact avec le hardware. Les actions sont traduites en plusieurs fonctions de base système qui sont transmises aux BIOS et XBIOS pour être exécutées. On peut bien sûr aussi s'adresser directement au BIOS et au XBIOS. On y gagnera en rapidité mais par le GEMDOS les routines d'entrées sorties seront spécifiques à chaque périphérique.

Appel des fonctions

Suivant le langage de programmation que vous utilisez, l'appel des fonctions diffère dans la syntaxe, mais les paramètres d'appel et le fonctionnement général restent identiques.

Assembleur

L'appel du TOS se fait au moyen de trois instructions TRAP.
TRAP est une instruction qui introduit une exception, donc exécute un saut comme lors d'un appel à un sous-programme ; les paramètres communiqués à la fonction le sont par l'intermédiaire de la pile, (mais dans l'ordre inverse par rapport au C et aux Basic). Le numéro du TRAP est lui aussi placé sur la pile. Les trois valeurs de TRAP utilisées sont :

— GEMDOS : **TRAP #1**

— BIOS : **TRAP #13**

— XBIOS : **TRAP #14**

Ces fonctions reviennent dans le programme appelant juste derrière le TRAP et doivent être traitées comme un sous-programme. Le pointeur

de pile doit être repositionné à sa valeur d'avant l'appel. Il suffit pour cela de lui ajouter le nombre d'octets utilisés pour le transfert des paramètres et du numéro du TRAP. La valeur de retour est communiquée dans le registre D0. Dans le cas du GEMDOS, le registre A0 est aussi détruit. Dans les cas du BIOS et du XBIOS ce sont les registres D1, D2, A0, A1, A2.

Exemple d'appel :

```
move.l      para1, – (sp)          ;un mot long sur la pile
move.l      para2, – (sp)          ;un second mot long
move.w      #numfonction, – (sp)   ;le numéro de fonction
                                   ;est un mot
trap        #13                    ;appel du BIOS
add.l       #10,sp                 ;correction de la pile
```

Si vous avez quelques problèmes, revoyez l'article traitant de la gestion des piles (Partie 4 chapitre 3.2.2).

Le C

Le fichier OSBIND.H *(Operating System Bindings)* permet d'appeler toutes les fonctions du TOS par leur nom. Il suffit de le charger en début de programme par la ligne :

```
#include <osbind.h>
```

Les fonctions **gemdos( )**, **bios( )** et **xbios( )** existent aussi en C.

Pour les utiliser directement, il faut les déclarer comme des fonctions longues :

```
extern long gemdos( );
extern long bios( );
extern long xbios( );
```

Le Basic OMIKRON

Les fonctions GEMDOS BIOS et XBIOS sont accessibles en OMIKRON. Leur nom doit être suivi de l'expression suivante entre parenthèses :

(variable de retour,numéro de fonction,para1,para2,...)

Si vous ne désirez pas utiliser la valeur tournée, vous pouvez omettre la variable mais devez garder les virgules correspondantes. Tous les paramètres étant des mots en OMIKRON, si la fonction attend des mots longs en paramètres (comme dans l'exemple en assembleur), vous devrez transmettre successivement le poids fort (high-word) puis le poids faible (low-word).

Exemple :

```
GEMDOS(,9,HIGH(Nompara%L),LOW(Nompara%L))
```



Le Basic GFA

Les fonctions GEMDOS, BIOS et XBIOS sont connues en Basic GFA. Elles retournent toujours une valeur. Il faut donc affecter cette valeur à une variable ou l'utiliser dans une expression.

Si vous ne désirez pas utiliser la valeur retournée, écrire VOID ou tilde(~) avant l'appel de la fonction. Les paramètres indiqués sont des mots par défaut, mais vous pouvez utiliser des mots longs en faisant précéder le nom de la variable ou la valeur par L suivi de deux points.

Exemples :

```
VOID BIOS(1,3)

IF XBIOS(1,3)< >0
resultat%=GEMDOS(71,L:adr%,d%)
```

Affichage de tous les caractères disponibles sur Atari par les fonctions du BIOS

Le petit programme qui suit se propose d'afficher à l'écran tous les caractères disponibles sur Atari au moyen des fonctions BIOS. La fonction Bconout(dev,caractère) envoie un caractère sur le périphérique repéré par son numéro dev. C'est la fonction numéro 3 du BIOS. La fonction Bconin(dev) lit un caractère sur le périphérique repéré par son numéro dev. C'est la fonction numéro 2 du BIOS. Le numéro de périphérique (*device* en Anglais) peut prendre les valeurs suivantes :

0 Interface Centronics
1 Interface RS 232
2 Clavier et écran
3 Interface Midi
4 Processeur clavier
5 Ecran (canal ASCII)

Le périphérique numéro 2 reconnaît et exécute les caractères de contrôle dont les codes sont inférieurs à 32 alors que le périphérique 5 n'en reconnaît que les codes ASCII et peut donc afficher tous les caractères.

Pour bien comprendre la différence entre les deux périphériques 5 et 2, les affichages se font avec ces deux valeurs de façon consécutives, après attente d'une touche clavier.

La pause n'est là que pour vous aider à repérer visuellement le ping, le retour chariot, le backspace et le line feed.

L'initialisation du curseur en haut à gauche et l'effacement de l'écran se fait au moyen des fonctions escape et des séquences VT 52.

Le code de la touche escape est 27 (ou 33 en octal). Il doit être suivi par la sortie du « E ».

Remarquez que le programme distingue les majuscules des minuscules. Les codes 13 (retour chariot) et 10 (saut de ligne) ne sont pas précédés d'escape et permettent de positionner l'affichage du texte au début de la ligne suivante. La valeur de il MOD 13 (i% 13 en C) ou du reste de la division du numéro de caractère par 13 permet de revenir en début

de ligne suivante dès qu'on a écrit 13 caractères à l'écran. Cela permet de disposer tous les caractères et leur code ASCII à l'écran.

Le code 32 est celui de l'espace qui est utilisé entre l'affichage de deux caractères consécutifs. Rappelons que le code du caractère à afficher est une variable qui varie entre 0 et 255.

VERSION ASSEMBLEUR

```
        .TEXT
; Avec périphérique N°5

; fonction escape
            move.w    #27,-(sp)
            move.w    #2,-(sp)
            move.w    #3,-(sp)
            trap      #13
            addq.l    #6,sp

;Effacer l'écran et curseur ligne suivante
            move.w    #'E',-(sp)
            move.w    #2,-(sp)
            move.w    #3,-(sp)
            trap      #13
            addq.l    #6,sp

; saut de ligne
            move.w    #10,-(sp)
            move.w    #2,-(sp)
            move.w    #3,-(sp)
            trap      #13
            addq.l    #6,sp

;Initialisation de d3 à 0 et de d4 à 255
            clr.w     d3
            move.w    #$ff,d4

boucle5:    move.l    d3,d5
            divu.w    #13,d5
            andi.l    #$ffff,d5          ;modulo 13
            cmpi.l    #0,d5
            bne.s     pasdemodulo5

            move.w    #13,-(sp)
            move.w    #2,-(sp)
            move.w    #3,-(sp)
            trap      #13
            addq.l    #6,sp
```

(suite)

```
                move.w    #10,-(sp)
                move.w    #2,-(sp)
                move.w    #3,-(sp)
                trap      #13
                addq.l    #6,sp

        pasdemodulo5:
                move.w    d3,-(sp)  ;caractère à afficher
                move.w    #5,-(sp)
                move.w    #3,-(sp)
                trap      #13
                addq.l    #6,sp
        ;espace entre deux caractères
                move.w    #' ',-(sp)
                move.w    #5,-(sp)
                move.w    #3,-(sp)
                trap      #13
                addq.l    #6,sp

        ;caractère suivant
                addq.l    #1,d3

                move.w    #500,d5
        attente5:
                dbra      d5,attente5

                dbra      d4,boucle5

                move.w    #2,-(sp)
                move.w    #2,-(sp)
                trap      #13
                addq.l    #4,sp

        ; Version avec périphérique N° 2

                move.w    #27,-(sp)
                move.w    #2,-(sp)
                move.w    #3,-(sp)
                trap      #13
                addq.l    #6,sp

                move.w    #'E',-(sp)
                move.w    #2,-(sp)
                move.w    #3,-(sp)
                trap      #13
                addq.l    #6,sp
```

(suite)

```
                move.w    #10,-(sp)
                move.w    #2,-(sp)
                move.w    #3,-(sp)
                trap      #13
                addq.l    #6,sp

                clr.w     d3
                move.w    #$ff,d4

boucle2:        move.l    d3,d5
                divu.w    #13,d5          ;modulo 13
                andi.l    #$ffff,d5
                cmpi.l    #0,d5
                bne.s     pasdemodulo2

                move.w    #13,-(sp)
                move.w    #2,-(sp)
                move.w    #3,-(sp)
                trap      #13
                addq.l    #6,sp
                move.w    #10,-(sp)
                move.w    #2,-(sp)
                move.w    #3,-(sp)
                trap      #13
                addq.l    #6,sp

pasdemodulo2:

                move.w    d3,-(sp)
                move.w    #2,-(sp)
                move.w    #3,-(sp)
                trap      #13
                addq.l    #6,sp

                move.w    #' ',-(sp)
                move.w    #2,-(sp)
                move.w    #3,-(sp)
                trap      #13
                addq.l    #6,sp

                addq.l    #1,d3

                move.w    #500,d5
attente2:
                dbra      d5,attente2

                dbra      d4,boucle2
```



(suite)

```
                move.w    #2,-(sp)
                move.w    #2,-(sp)
                trap      #13
                addq.l    #4,sp


                clr.w     -(sp)
                trap      #1
                .END
```

VERSION EN C

```
#include <osbind.h>

int i,j;

main()

{

Bconout (2,27);
Bconout (2,'E');
Bconout (2,10);

printf ("\n\n\nles caractères de l'ATARI N° de périph=5\n";

    for (i=0; i<256; i++)
    {
        if (i % 13 ==0)
        {
            Bconout (2, 13);
            Bconout (2, 10);
        }
        Bconout (5, i);
        Bconout (5, ' ');
        for (j=0; j<500; j++)
        ;
    }
Bconin (2);
Bconout (2,27);
Bconout (2,'E');
Bconout (2,10);

printf ("\n\n\nles caractères de l'ATARI N° de périph=2\n";
```

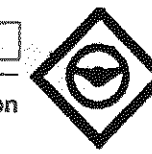
(suite)

```
        for (i=0; i<256; i++)
        {
            if (i % 13 ==0)
            {
                Bconout (2, 13);
                Bconout (2, 10);
            }
            Bconout (2, i);
            Bconout (2, ' ');
            for (j=0; j<500; j++)
            ;
        }
    Bconin (2);

    }
```

```
VERSION BASIC OMIKRON

BIOS(,3,2,27)
BIOS(,3,2,ASC("E"))
BIOS(,3,2,10)
PRINT @(20,2);"les caractères de l'ATARI N° de périph=5"
FOR i%=0 TO 255
  IF i% MOD 13=0
    THEN BIOS(,3,2,13)
         BIOS(,3,2,10)
  ENDIF
  BIOS(,3,5,i|)
  BIOS(,3,5,32)
  PRINT i%;
  WAIT 5
NEXT i%
BIOS(,2,2)
BIOS(,3,2,27)
BIOS(,3,2,ASC("E"))
BIOS(,3,2,10)
PRINT @(20,2);"les caractères de l'ATARI N° de périph=2"
FOR i%=0 TO 255
  IF i% MOD 13=0
    THEN BIOS(,3,2,13)
         BIOS(,3,2,10)
  ENDIF
  BIOS(,3,2,i|)
  BIOS(,3,2,32)
  PRINT i%;
  WAIT 5
NEXT i%
BIOS(,2,2)
END
```



```
VERSION BASIC GFA

~BIOS(3,2,27)
~BIOS(3,2,ASC("E"))
~BIOS(3,2,10)
PRINT AT(20,2);"les caractères de l'ATARI N° de périph=5"
FOR il=0 TO 255
  IF il MOD 13=0
    ~BIOS(3,2,13)
    ~BIOS(3,2,10)
  ENDIF
  ~BIOS(3,5,il)
  ~BIOS(3,5,32)
  PRINT il'
  PAUSE 5
NEXT il
~BIOS(2,2)
~BIOS(3,2,27)
~BIOS(3,2,ASC("E"))
~BIOS(3,2,10)
PRINT AT(20,2);"les caractères de l'ATARI N° de périph=2"
FOR il=0 TO 255
  IF il MOD 13=0
    ~BIOS(3,2,13)
    ~BIOS(3,2,10)
  ENDIF
  ~BIOS(3,2,il)
  ~BIOS(3,2,32)
  PRINT il'
  PAUSE 5
NEXT il
~BIOS(2,2)
END
```

les caractères de l'ATARI N° de périph=2

```
 0  1  2  3  4  5  6  7 8       9
                                  10
                                     11
                                        12
 13  14  15  16  17  18  19  20  21  22  23  24  25
 26 27  28  29  30  31    32 ! 33 " 34 # 35 $ 36 % 37 & 38
' 39 ( 40 ) 41 * 42 + 43 , 44 - 45 . 46 / 47 0 48 1 49 2 50 3 51
4 52 5 53 6 54 7 55 8 56 9 57 : 58 ; 59 < 60 = 61 > 62 ? 63 @ 64
A 65 B 66 C 67 D 68 E 69 F 70 G 71 H 72 I 73 J 74 K 75 L 76 M 77
N 78 O 79 P 80 Q 81 R 82 S 83 T 84 U 85 V 86 W 87 X 88 Y 89 Z 90
[ 91 \ 92 ] 93 ^ 94 _ 95 ` 96 a 97 b 98 c 99 d 100 e 101 f 102 g 103
h 104 i 105 j 106 k 107 l 108 m 109 n 110 o 111 p 112 q 113 r 114 s 115 t 116
u 117 v 118 w 119 x 120 y 121 z 122 { 123 | 124 } 125 ~ 126 △ 127 Ç 128 ü 129
é 130 â 131 ä 132 à 133 ã 134 ç 135 ê 136 ë 137 è 138 ï 139 î 140 ì 141 Ä 142
Å 143 É 144 æ 145 Æ 146 ô 147 ö 148 ò 149 û 150 ù 151 ÿ 152 Ö 153 Ü 154 ¢ 155
£ 156 ¥ 157 ß 158 ƒ 159 á 160 í 161 ó 162 ú 163 ñ 164 Ñ 165 ª 166 º 167 ¿ 168
⌐ 169 ¬ 170 ½ 171 ¼ 172 ¡ 173 « 174 » 175 ã 176 õ 177 Ø 178 ø 179 œ 180 Œ 181
À 182 Ã 183 Õ 184 ¨ 185 ´ 186 † 187 ¶ 188 © 189 ® 190 ™ 191 Ĳ 192 IJ 193 ʞ 194
ɹ 195 ɐ 196 ⊥ 197 ∩ 198 ɯ 199 ɾ 200 ᴨ 201 ∪ 202 ' 203 ɔ 204 ɔ 205 ɑ 206 ɹ 207
ᗡ 208 ᗺ 209 ᘔ 210 ᔭ 211 ᔨ 212 ᔦ 213 ᔮ 214 ᔪ 215 ᔦ 216 ᔦ 217 ᔮ 218 ᔨ 219 ᔨ 220
§ 221 ∧ 222 ∞ 223 α 224 β 225 Γ 226 π 227 Σ 228 σ 229 µ 230 τ 231 Φ 232 Θ 233
Ω 234 δ 235 ϕ 236 ϕ 237 ∈ 238 ∩ 239 ≡ 240 ± 241 ≥ 242 ≤ 243 ∫ 244 ⌡ 245 ÷ 246
≈ 247 ° 248 • 249 · 250 √ 251 ⁿ 252 ² 253 ³ 254   255
```



les caractères de l'ATARI N° de périph=5

	0	⇧	1	⇩	2	◇	3	◇	4	⊠	5	▨	6	⊠	7	✓	8	⊙	9	♣	10	♪	11	⁷⁄₈	12								
ᶜ⁄ₖ	13	⅄	14	⅄	15	⊔	16	⁝	17	⅁	18	⅃	19	⅄	20	⅄	21	⅂	22	⅂	23	⅃	24	⅄	25								
ə	26	ᵉ⁄ₛ	27	ᵉ	28	ᵓ	29	ᵡ	30	ᵝ	31		32	!	33	"	34	#	35	$	36	%	37	&	38								
'	39	(	40	)	41	*	42	+	43	,	44	-	45	.	46	/	47	0	48	1	49	2	50	3	51								
4	52	5	53	6	54	7	55	8	56	9	57	:	58	;	59	<	60	=	61	>	62	?	63	@	64								
A	65	B	66	C	67	D	68	E	69	F	70	G	71	H	72	I	73	J	74	K	75	L	76	M	77								
N	78	O	79	P	80	Q	81	R	82	S	83	T	84	U	85	V	86	W	87	X	88	Y	89	Z	90								
[	91	\	92	]	93	^	94	_	95	`	96	a	97	b	98	c	99	d	100	e	101	f	102	g	103								
h	104	i	105	j	106	k	107	l	108	m	109	n	110	o	111	p	112	q	113	r	114	s	115	t	116								
u	117	v	118	w	119	x	120	y	121	z	122	{	123	|	124	}	125	~	126	△	127	Ç	128	ü	129								
é	130	â	131	ä	132	à	133	ä	134	ç	135	ê	136	ë	137	è	138	ï	139	î	140	ì	141	Ä	142								
Å	143	É	144	æ	145	Æ	146	ô	147	ö	148	ò	149	û	150	ù	151	ÿ	152	Ö	153	Ü	154	¢	155								
£	156	¥	157	ß	158	ƒ	159	á	160	í	161	ó	162	ú	163	ñ	164	Ñ	165	ª	166	º	167	¿	168								
⌐	169	¬	170	½	171	¼	172	¡	173	«	174	»	175	ã	176	õ	177	Ø	178	ø	179	œ	180	Œ	181								
À	182	Ã	183	Õ	184	¨	185	´	186	†	187	¶	188	©	189	®	190	™	191	ÿ	192	Ÿ	193	¤	194								
⅃	195	⅄	196	⊤	197	⅂	198	⅂	199	⅂	200	⅃	201	⅃	202	⁻	203	⅃	204	⅄	205	⅄	206	⅃	207								
⅃	208	⅃	209	⅄	210	⅄	211	⅃	212	⅂	213	⅃	214	⅃	215	⅂	216	⅂	217	⅃	218	⅂	219	⅂	220								
§	221	∧	222	∞	223	α	224	β	225	Γ	226	π	227	Σ	228	σ	229	µ	230	τ	231	Φ	232	θ	233								
Ω	234	δ	235	∅	236	Φ	237	Є	238	Π	239	≡	240	±	241	≥	242	≤	243	⌠	244	⌡	245	÷	246								
≈	247	°	248	•	249	∙	250	√	251	ⁿ	252	²	253	³	254	¯	255																

3/2

Le BIOS

Cet ensemble de routines se charge, avec le XBIOS, de servir de liaison entre le GEMDOS et le hardware du ST. On y trouve donc la plupart des routines de base du GEMDOS (lecture ou écriture sur disquette, lecture de l'interface clavier, etc.). Après un appel BIOS, les registres D0 à D2 et A0 à A2 peuvent être modifiés. Il conviendra, si vous programmez en assembleur, de ne pas laisser sur ces registres de valeur importante avant un appel en ROM. Tout appel à une routine BIOS s'effectue par l'intermédiaire du **TRAP #13**. En C, les exemples sont donnés pour le compilateur Megamax. Pour tous les appels BIOS et XBIOS, la librairie 'osbind.h' devra être chargée par l'instruction :

```
#include <osbind.h>
```

LES FONCTIONS BIOS

Les fonctions BIOS ont pour nom :

GETMPB, BCONSTAT, BCONIN, BCONOUT, RWABS, SETEXEC, TICKTAL, GETBPB, BCOSTAT, MEDIACH, DRVMAP, et KBSHIFT.

Voyons maintenant en détail ces fonctions BIOS :

#0:
— GETMPB

Grâce à cette fonction, l'utilisateur peut avoir accès aux blocs de gestion mémoire du système d'exploitation.

Le seul paramètre nécessaire est l'adresse d'un tampon de 12 octets.

Exemple d'utilisation en assembleur :

```
    move.l    #pointeur,-(sp)     ; Adresse des pointeurs
    clr.w     -(sp)               ; Code GETMPB
    trap      #13                 ; Appel en ROM
    addq.w    #6,sp               ; Et restauration de la pile
    ...
pointeur:
    ds.l 1
```

Après l'appel, le pointeur contient l'adresse de trois autres pointeurs à des tableaux de 16 octets chacun :

— 1er pointeur (4 octets) : adresse du Memory Free List
— 2e pointeur (4 octets) : adresse du Memory Allocated List
— 3e pointeur (4 octets) : adresse du Roving Pointer

Memory Free list :

liste des blocs de mémoire libres
liste des blocs alloués
liste des blocs de mémoire libres

Memory Allocated List :

adresse du memory allocated list suivant (ou 0 si dernier)
adresse du bloc
taille du bloc en octets
adresse de la basepage du programme propriétaire de ce bloc de mémoire

Roving Pointer :

structure identique au Memory Allocated List

Exemple d'utilisation en C :

```
void getmpb(tampon);
/* Définition des variables */
long tampon
```

#1 :
— BCONSTAT

Cette fonction prévient l'utilisateur si un périphérique d'entrée a un caractère à transmettre.

Les périphériques disponibles sont :

0 : Interface Centronics
1 : Interface RS232
2 : Clavier et écran
3 : Interface MIDI
4 : Port clavier

Le seul paramètre utilisé est le type de périphérique dont on demande l'état. Après l'appel, le registre D0 contient −1 si un caractère est disponible (0 dans le cas contraire).

Exemple d'utilisation en assembleur :

```
move.w    #2,-(sp)    ; Appui sur une touche du clavier
move.w    #1,-(sp)    ; Code BCONSTAT
trap      #13         ; Appel en ROM
addq.w    #4,sp       ; Et restauration de la pile
...                   ; Si D0 = -1 alors un caractère est disponible
```



Exemple d'utilisation en C :

```
int Bconstat(dev)
/* Définition des variables */
int dev;
```

#2 :
— BCONIN

Cette fonction récupère un caractère dans le périphérique d'entrée spécifié. Pour le choix du périphérique, voir BCONSTAT. Après l'appel, le registre D0 contient, pour le clavier (périphérique numéro 2), le scancode et le code ASCII de la touche pressée.

Exemple d'utilisation en assembleur :

```
move.w    #2,-(sp)      ; Lire un caractère du clavier
move.w    #2,-(sp)      ; Code BCONIN
trap      #13           ; Appel en ROM
addq.w    #4,sp         ; Et restauration de la pile
```

Exemple d'utilisation en C :

```
long Bconin(dev)
/* Définition des variables */
int dev;
```

#3 :
— BCONOUT

Au contraire de BCONIN, BCONOUT envoie un caractère au périphérique choisi.

Exemple d'utilisation en assembleur :

```
move.w    #'1',-(sp)    ; Envoi du caractère 1
move.w    #2,-(sp)      ; A l'écran
move.w    #3,-(sp)      ; Code BCONOUT
trap      #13           ; Appel en ROM
addq.w    #6,sp         ; Et restauration de la pile
```

Exemple d'utilisation en C :

```
Bconout(dev, c)
/* Définition des variables */
int dev, c;
```

#4 :
— RWABS

Permet la lecture ou l'écriture d'un ou de plusieurs secteurs sur une disquette. Les paramètres suivants sont nécessaires :

DEV : unité de disquette utilisée [0 = Disque A, 1 = Disque B, 2 ou plus = autre périphérique (ex. : disque dur)],

RECNR : numéro du premier secteur à lire ou écrire,

NUMBER : nombre de secteurs à lire ou écrire,

BUFFER : tampon contenant ou les données à écrire, ou les données lues après l'appel,

RWFLAG : drapeau spécifiant le type d'accès :
0 : lecture de secteurs,
1 : écriture de secteurs,
2 : lecture de secteurs, sans s'occuper d'un éventuel changement de disquette,
3 : écriture de secteurs, sans s'occuper d'un éventuel changement de disquette.

Après l'appel, D0 contient 0 si tout s'est bien déroulé. Un nombre négatif est synonyme d'une erreur (voir à ce sujet la fonction **#8** du chapitre sur le XBIOS).

Exemple d'utilisation en assembleur :

```
    clr.w       -(sp)              ; Disque A
    move.w      #0,-(sp)           ; Secteur numéro 1
    move.w      #1,-(sp)           ; 1 secteur concerné
    move.l      #tampon,-(sp)      ; Tampon de travail
    move.w      #3,-(sp)           ; Ecriture
    move.w      #4,-(sp)           ; Code RWABS
    trap        #13                ; Appel en ROM
    add.w       #$e,sp             ; Et restauration de la pile
    ...
    ...
tampon:
    ds.b 512                       ; Tampon pour un secteur
                                   (512 octets)
```

L'exemple ci-dessus formate le secteur de boot d'une disquette. Cela est bien entendu fortement déconseillé !

Exemple d'utilisation en C :

```
long Rwabs(rwflag, buf, count, recno, dev)

/* Définition des variables */

int rwflag;
char *buf;
int count, recno, dev;
```



#5 :
— SETEXEC

Grâce à SETEXEC, on peut modifier un des vecteurs du 68000. Deux paramètres sont nécessaires : le numéro du vecteur à modifier ainsi que la nouvelle adresse à placer sur le vecteur. En retour, la routine place dans le registre D0 l'ancienne adresse contenue. Si l'on veut seulement connaître le contenu d'un vecteur sans pour autant le modifier, il faut donner à la routine la valeur −1.

Exemple d'utilisation en assembleur :

```
    move.l      #routine,-(sp)      ; Adresse de la routine
    move.w      #2,-(sp)            ; En vecteur numéro 2 ; (bus
                                      error)
    move.w      #5,-(sp)            ; Code SETEXEC
    trap        #13                 ; Appel en ROM
    addq.w      #8,sp               ; Et restauration de la pile
    ...
    ...
routine:
    ...
    rte
```

#6 :
— TICKTAL

Donne la fréquence de l'horloge du système. Le résultat est normalement de 20 ms (stocké dans D0).

Exemple d'utilisation en assembleur :

```
    move.w      #6,-(sp)            ; Code TICKTAL
    trap        #13                 ; Appel en ROM
    addq.w      #2,sp               ; Et restauration de la pile
```

Exemple d'utilisation en C :

```
long Ticktal( )
```

#7 :
— GETBPB

Cette fonction rend l'adresse d'un tableau concernant la disquette insérée dans l'un des deux lecteurs. Le seul paramètre demandé est l'unité de disquette choisie [0 : lecteur A, 1 : lecteur B, 3 ou plus : autre périphérique (ex. : disque dur)].

Exemple d'utilisation en assembleur :

```
    clr.w       -(sp)               ; Disque A
    move.w      #7,-(sp)            ; Code GETBPB
    trap        #13                 ; Appel en ROM
    addq.w      #4,sp               ; Et restauration de la pile
```

Après l'appel, le registre D0 contient l'adresse du tableau BPB.

Tableau BPB :

2 octets :	Taille des secteurs en octets (ex. : 512)
2 octets :	Taille des clusters en secteurs (ex. : 2)
2 octets :	Taille des clusters en octets (ex. : 1024)
2 octets :	Taille du directory en secteurs (ex. : 7)
2 octets :	Taille de la FAT en secteurs (ex. : 5)
2 octets :	Numéro de secteur du début de la deuxième FAT (ex. : 6)
2 octets :	Numéro du premier secteur réservé à l'utilisateur (ex. : 18)
2 octets :	Nombre de clusters sur la disquette (ex. : 351)
2 octets :	Réservé

Exemple d'utilisation en C :

```
long Getbpb(dev)

/* Définition des variables */

int dev;
```

#8 :
— BCOSTAT

Cette fonction vérifie qu'un périphérique est prêt à recevoir des données. Au retour de l'appel, D0 contient 0 si l'utilisateur doit encore attendre, ou — 1 si le périphérique est prêt. Le choix du périphérique est identique à la fonction **#1 (BCONSTAT)**.

Exemple d'utilisation en assembleur :

```
move.w    #3,-(sp)     ; Port MIDI prêt à recevoir des données ?
move.w    #8,-(sp)     ; Code BCOSTAT
trap      #13          ; Appel en ROM
addq.w    #4,sp        ; Et restauration de la pile
...                    ; D0 contient l'état du périphérique
```

Exemple d'utilisation en C :

```
long Bcostat(dev)

/* Définition des variables */

int dev;
```

#9 :
— MEDIACH

Grâce à cette fonction, on peut savoir si une nouvelle disquette a été introduite dans le lecteur depuis la dernière opération de lecture ou d'écriture. Le résultat est un nombre stocké sur le registre D0 :

0 : La disquette n'a pas été changée.

1 : La disquette a peut-être été changée.

2 : La disquette a été changée.



Le paramètre suivant est nécessaire :

DEV : unité de disquette (voir les fonctions **#4** et **#7** ci-dessus).

Exemple d'utilisation en assembleur :

```
clr.w       -(sp)          ; Lecteur A
move.w      #9,-(sp)       ; Code MEDIACH
trap        #13            ; Appel en ROM
addq.w      #4,sp          ; Et restauration de la pile
...                        ; Résultat dans D0
```

Exemple d'utilisation en C :

```
long Mediach(dev)
/* Définition des variables */
int dev;
```

#10 : — DRVMAP

Cette fonction renvoie à l'utilisateur un nombre sur 4 octets, reflétant le nombre d'unités logiques connectées au ST codé de la façon suivante :

bit 0 : unité A (1 si présent, 0 sinon)

bit 1 : unité B (idem...)

bit 2 : unité C ...

bit 4 : unité D ...

etc.

Sur un ST sans disque dur, la routine renvoie normalement le chiffre '3' (bits 0 et 1 allumés pour les unités logiques A et B).

Exemple d'utilisation en assembleur :

```
move.w      #10,-(sp)      ; Code DRVMAP
trap        #13            ; Appel en ROM
addq.w      #2,sp          ; Et restauration de la pile
```

Exemple d'utilisation en C :

```
long Drvmap ( )
```

#11 : — KBSHIFT

Cette fonction offre la possibilité de tester ou de modifier très facilement l'appui ou non sur les touches *spéciales* du clavier du ST. Pour cela, l'utilisateur reçoit après l'appel en ROM un nombre stocké sur le registre D0. A chacun des huit bits de poids faible de ce registre, correspond l'état d'une des touches suivantes :

bit 0 : Touche **Shift** de droite

bit 1 : Touche **Shift** de gauche

bit 2 : Touche **Control**

bit 3 : Touche **Alt**

bit 4 : Touche **Caps Lock** en action (sélection des majuscules)

bit 5 : Touche de droite de la souris **(CLR/HOME)**

bit 6 : Touche de gauche de la souris **(INSERT)**

bit 7 : Réservé

Un seul paramètre est demandé :

MODUS : Si celui-ci est négatif, la routine renvoie l'état des touches. S'il est positif, ce nombre sera pris comme nouvel état des touches.

Exemple d'utilisation en assembleur :

```
move.w    #%00010000,-(sp)    ; Nombre positif (on enclenche
                              Caps Lock)
move.w    #11,-(sp)           ; Code KBSHIFT
trap      #13                 ; Appel en ROM
addq.w    #4,sp               ; Et restauration de la pile
...                           ; Maintenant, le clavier est en
                              majuscules...
```

Exemple d'utilisation en C :

```
long Kbshift(mode)

/* Définition des variables */

int mode;
```

3/3

L'extended BIOS (XBIOS)

Sous ce nom barbare, se cachent en fait des routines implantées en mémoire morte qui permettent, comme pour le GEMDOS ou le BIOS, de faciliter la vie du programmeur. Ces routines sont au nombre de 39. Elles peuvent être utilisées indifféremment en mode USER ou SUPER et sont appelées par l'intermédiaire du **TRAP #14**. Certains appels renvoient au programme des valeurs qui sont alors stockées sur les registres D0 à D2 ou A0 à A2. Le contenu de ces registres peut donc être modifié après l'appel en ROM.

LES FONCTIONS XBIOS

Ces routines sont :

INITMOUSE, SSBRK, PHYSBASE, LOGBASE, GETREZ, SETSCREEN, SETPALETTE, SETCOLOR, FLOPRD, FLOPWR, FLOPFMT, MIDIWS, MFPINT, IOREC, RSCONF, KEYTBL, RANDOM, PROTOBT, FLOPVER, SCRDMP, CURSCONF, SETTIME, GETTIME, BIOSKEYS, IKBDWS, JDISINT, JENABINT, GIACCESS, OFFGIBIT, ONGIBIT, XBTIMER, DOSOUND, SETPRT, KBDVBASE, KBRATE, PRTBLK, WVBL, SUPEXEC et PUNTAES.

Voyons maintenant en détail le rôle de toutes les fonctions disponibles.

Remarque :

Les exemples d'utilisation en C sont prévus pour le Mégamax C mais peuvent être utilisés sans grande transformation avec d'autres compilateurs de marque différente.

#0 :
— INITMOUSE

Cette fonction initialise les routines souris et fixe leur type de fonctionnement. Trois paramètres sont demandés : l'adresse de la nouvelle routine souris, son mode de fonctionnement ainsi que l'adresse d'un petit tampon de travail.

Quatre possibilités sont offertes pour le mode de fonctionnement : connexion en mode absolu (2), relatif (1), clavier (4) ou déconnexion (0).

Le tampon de travail doit avoir une taille de 12 octets :

TOPMODE : 1 octet : 0 si l'origine des Y en bas de l'écran, 1 si en haut

BUTTONS : 1 octet : les deux bits de poids inférieur donnent l'état des boutons de la souris : 1 ou 0 suivant que le bouton est appuyé ou non.

XREL : 1 octet : déplacement relatif en X

YREL : 1 octet : déplacement relatif en Y

Les huit octets suivants ne sont utilisés qu'en mode absolu.

XMAX : 2 octets : coordonnée maximale en X

YMAX : 2 octets : coordonnée maximale en Y

XSTART : 2 octets : coordonnée d'origine en X

YSTART : 2 octets : coordonnée d'origine en Y

La nouvelle routine souris doit être terminée par un RTS.

Exemple d'utilisation en assembleur :

```
    move.l    #nouvelle_routine,-(sp)   ; Adresse de la nouvelle routine
    move.l    #parametres,-(sp)         ; Adresse du nouveau tampon de travail
    move.w    #2,-(sp)                  ; Souris en mode absolu
    clr.w     -(sp)                     ; Code INITMOUSE
    trap      #14                       ; Appel en ROM
    add.w     #12,sp                    ; Et restauration de la pile
    ...
nouvelle_routine:
    ...
parametres :
    ds.b 12
```

Rappel :

Pour tous les appels GEMDOS, BIOS et XBIOS, l'ordre d'envoi des paramètres dans la pile doit être à tout prix respecté. Pour connaître cet ordre, il suffit de regarder l'exemple assembleur accompagnant toute description de fonction.

Exemple d'utilisation en C :

```
Initmous(type, paramp, vec)

/* Définition des variables */

int type ;
struct param *paramp ;
int vec (*vec) ( ) ;
```



```
struct param {
     char topmode ;
     char buttons ;
char xparam ;
     char yparam ;
     int xmax, ymax ;
     int xinital, yinital ;
     }
```

#1 :
— SSBRK

Cette fonction qui sert théoriquement à réserver de la place en mémoire n'est pas implantée en ROM. Elle n'est donc pas, malgré ce que peuvent écrire certaines documentations, utilisable.

#2 :
— PHYSBASE

Cette fonction délivre l'adresse de la mémoire vidéo physique.

Exemple d'utilisation en assembleur :

```
move.w    #2,-(sp)      ; Code PHYSBASE
trap      #14           ; Appel en ROM
addq.w    #2,sp         ; Et restauration de la pile
```

Après l'appel, le registre D0 contient l'adresse recherchée.

Exemple d'utilisation en C :

```
long Physbase ( )
```

#3 :
— LOGBASE

Identique à la fonction PHYSBASE, à la seule différence près que c'est l'adresse de la mémoire vidéo logique qui est fournie.

#4 :
— GETREZ

Donne la résolution de l'écran.

Le résultat de l'appel est situé dans le registre D0.

#0 pour la basse résolution couleur

#1 pour la moyenne résolution couleur

#2 pour la haute résolution monochrome

Exemple d'utilisation en assembleur :

```
move.w    #4,-(sp)      ; Code GETREZ
trap      #14           ; Appel en ROM
```

```
addq.w      #2,sp          ; Et restauration de la pile
...                        ; D0 contient la résolution
```

Exemple d'utilisation en C :

```
int Getrez ( )
```

#5 :
— SETSCREEN

Cette fonction permet la modification d'un ou de plusieurs des trois paramètres écran que sont l'adresse physique, l'adresse logique et la résolution. Si un paramètre ne doit pas être changé, il suffit d'envoyer à la fonction un nombre négatif.

Exemple d'utilisation en assembleur :

```
clr.w       -(sp)          ; Passe en basse résolution
move.l      #$a0000,-(sp)  ; Ram vidéo physique en $a0000
move.l      #-1,-(sp)      ; Ram vidéo logique inchangée
move.w      #5,-(sp)       ; Code SETSCREEN
trap        #14            ; Appel en ROM
add.w       #12,sp         ; Et restauration de la pile
```

Exemple d'utilisation en C :

```
Setscreen(log_loc, phys_loc, rez)

/* Définition des variables */

char*log_loc, *phys_loc;
int rez;
```

#6 :
— SETPALETTE

Procède au chargement en mémoire d'une nouvelle palette de couleurs. Si l'ordinateur est en moyenne résolution, seules les quatre premières seront prises en compte. En haute résolution les deux seules couleurs possibles sont bien entendu le noir et le blanc.

Exemple d'utilisation en assembleur :

```
    move.l      #nouvelle_palette,-(sp)  ; Adresse de la nouvelle
                                         palette
    move.w      #6,-(sp)                 ; Code SETPALETTE
    trap        #14                      ; Appel en ROM
    addq.w      #6,sp                    ; Et restauration de la pile
    ...
nouvelle_palette:
    dc.w $000,$777,$666,$555,$444,$333,$222,$111
    dc.w $123,$234,$345,$456,$567,$700,$070,$007
```



Exemple d'utilisation en C :

```
Setpalette(newpalette)

/* Définition des variables */

int newpalette[16];
```

Note :

En haute résolution, l'ordinateur fixe automatiquement la seconde couleur par rapport à la première. Si la première est blanche, la seconde sera noire et vice-versa. Il est ainsi impossible d'avoir un écran tout noir ou tout blanc sans effacer l'écran.

#7 :
— SETCOLOR

Permet de ne changer, contrairement à l'appel précédent, qu'une seule couleur de la palette. Les paramètres demandés sont la nouvelle couleur ainsi que son numéro (de 0 à 15).

Exemple d'utilisation en assembleur :

```
move.w    #$333,-(sp)     ; Nouvelle couleur (gris)
move.w    #10,-(sp)       ; Comme dixième couleur
move.w    #7,-(sp)        ; Code SETCOLOR
trap      #14             ; Appel en ROM
addq.w    #6,sp           ; Et restauration de la pile
```

Exemple d'utilisation en C :

```
int Setcolor(colornum, color)

/* Définition des variables */

int colornum, color;
```

#8 :
— FLOPPYREAD

Cette fonction procède au chargement en mémoire d'un ou de plusieurs secteurs d'une disquette. Sept paramètres sont demandés :

COMPTEUR : Nombre de secteurs à lire sur la piste (1 à 9).

FACE : Précise la face de lecture (0 pour les lecteurs SF. 0 ou 1 pour les lecteurs DF).

TRACK : Numéro de la piste à lire (0 à 79 pour les disquettes formatées par le bureau).

SECTEUR : Désigne le premier secteur à lire (0 à 9).

DEV : Indique l'unité de disquette utilisée (0 ou 1 pour les lecteurs A ou B).

Quatre octets doivent être mis à zéro sur la pile, correspondant sans doute à une extension des possibilités de la routine non encore réalisée.

TAMPON : Adresse du tampon de stockage des données. Chaque secteur ayant normalement une taille de 512 octets, la taille du tampon doit être de 512 fois le nombre de secteurs lus.

Exemple d'utilisation en assembleur :

```
move.w    #5,-(sp)          ; lire les secteurs
move.w    #0,-(sp)          ; 2, 3, 4, 5, 6
move.w    #20,-(sp)         ; de la 20e
move.w    #2,-(sp)          ; piste
clr.w     -(sp)             ; sur la face A
clr.l     -(sp)             ; de la disquette
move.l    #tampon,-(sp)     ; et les stocker dans le tampon
move.w    #8,-(sp)          ; appel FLOPPYREAD
trap      #14
add.w     #20,sp
...
tampon:
ds.b 512*5
```

Après l'appel **TRAP**, le registre D0 contient la valeur 0 si la lecture s'est bien déroulée. Une valeur négative signifie une erreur.

Liste des erreurs (XBIOS) :

— 1 Erreur système.
— 2 Lecteur de disquettes non prêt.
— 3 Commande inconnue.
— 4 Erreur CRC *(Cyclic Redundancy Check)*.
— 5 Commande incorrecte.
— 6 Piste introuvable.
— 7 Format de disquette incorrect.
— 8 Secteur introuvable.
— 9 Pas de papier (imprimante).
— 10 Erreur durant l'écriture.
— 11 Erreur durant la lecture.
— 12 Erreur générale.
— 13 Disquette protégée en écriture.
— 14 Disquette changée.
— 15 Périphérique inconnu.
— 16 Secteur endommagé dans la disquette.
— 17 Insérer une autre disquette.

Note :

Les numéros d'erreur compris entre — 32 et — 67 correspondent à une erreur GEMDOS.

Exemple d'utilisation en C :

```
int Floprd(buf, filler, devno, sectno, trackno, sideno, count)
/* Définition des variables */
int *buf;
long filler;
int devno, sectno, trackno, sideno, count;
```

#9 : — FLOPPYWRITE

Cette fonction est le pendant de la fonction précédente et permet donc l'écriture d'un ou de plusieurs secteurs sur disquette. Les paramètres utilisés sont identiques. Seul l'appel est naturellement différent : il faut stocker non pas la valeur **#8** mais **#9** sur la pile avant l'instruction **TRAP**.

#10 : — FLOPPYFORMAT

Cette routine formate une piste de disquette. Liste des paramètres demandés :

VIRGIN : Nombre inscrit dans tous les octets de la piste formatée.

MAGIC : Constante ayant pour valeur $87654321.

INTERLEAVE : Constante déterminant l'ordre d'écriture des secteurs dans la piste. Normalement 1.

FACE : Sélectionne une des deux faces de la disquette (0 ou 1 pour les faces A et B). La face B n'est accessible qu'avec un lecteur double face.

TRACK : Sélectionne la piste à formater.

SECPTRK : Nombre de secteurs dans la piste.

DEV : Choix de l'unité de disquettes (0 ou 1 pour le lecteur A ou B). Comme pour les deux fonctions précédentes, quatre octets doivent être mis à zéro sur la pile.

TAMPON : Adresse d'un tampon sur lequel l'ordinateur stocke l'ensemble des données d'une piste. Pour un formatage standard (9 secteurs par piste), la taille de ce tampon doit être d'au minimum 8 Ko. Après l'appel, le registre D0 contient 0 si tout s'est bien passé. Une valeur négative indique une erreur (voir FLOPPYREAD).

Exemple d'utilisation en assembleur :

```
move.w   #$e5e5,-(sp)        ; Remplissage standard des DSK
move.l   #$87654321,-(sp)    ; Constante MAGIC
move.w   #1,-(sp)            ; Interleave normal
move.w   #1,-(sp)            ; Deuxième face
move.w   #10,-(sp)           ; Dixième piste
```

```
    move.w    #9,-(sp)              ; 9 secteurs dans la piste
    clr.w     -(sp)                 ; Lecteur A
    clr.l     -(sp)
    move.l    #tampon,-(sp)         ; Adresse du tampon de travail
    move.w    #10,-(sp)             ; Code FLOPPYFORMAT
    trap      #14                   ; Appel en ROM
    add.w     #26,sp                ; Et restauration de la pile
    tst.w d0                        ; Erreur ?
    bmi il_y_a_une_erreur
    ...                             ; Pas d'erreur
    ...
tampon:                             ; Tampon de travail
    ds.b 8192
```

Exemple d'utilisation en C :

```
int Flopfmt(buf, filler, devno, spt, trackno, sideno, interlv, magic,
virgin)

/* Définition des variables */

char *buf;
long filler;
int devno, spt, trackno, sideno, interlv, virgin;
long magic;
```

#11 :

Non implantée.

#12 :
— MIDIWS

Permet l'envoi d'une chaîne au travers de l'interface MIDI. Les deux paramètres demandés sont :

POINTER : Pointeur sur la chaîne de caractères à envoyer

COUNTER : Nombre de caractères — 1 dans la chaîne

Exemple d'utilisation en assembleur :

```
    move.l    #chaîne,-(sp)         ; Adresse de la chaîne
    move.w    #6,-(sp)              ; 7 caractères — 1 = 6
    move.w    #12,-(sp)             ; code MIDIWS
    trap      #14                   ; Et appel en ROM
    addq.w    #8, sp                ; Restauration de la pile
    ...
chaîne:
    dc.b      "bonjour"             ; 7 caractères
```

Exemple d'utilisation en C :

```
midiws(cnt, ptr)
```



```
/* Définition des variables */
int cnt;
```

#13 :
— MFPINT

Par cette fonction, on revectorise une des interruptions du MFP. Attention, le simple fait de mettre l'adresse d'une de vos routines dans un vecteur MFP n'est pas toujours suffisant. Si, par exemple, vous revectorisez un TIMER, il vous faudra auparavant régler sa fréquence. Une routine en interruption doit toujours se terminer, non pas par un RTS *(Return from Subroutine)*, mais par un RTE *(Return from Exception)*.

Les deux paramètres à donner sont :

VECTOR : Adresse de votre routine,

NUMBER: Numéro de l'interruption MFP qui effectuera votre routine (0-15).

Exemple d'utilisation en assembleur :

```
    move.l    #ma_routine,-(sp)   ; Adresse de la nouvelle routine
    move.w    #6,-(sp)            ; Sur l'interruption CLAVIER/MIDI
    move.w    #13,-(sp)           ; Code MFPINT
    trap      #14                 ; Appel en ROM
    addq.w    #8,sp               ; Et restauration de la pile
    ...
ma_routine:
    ...
    rte
```

Exemple d'utilisation en C :

```
Mfpint(interno, vector)
/* Définition des variables */
int interno;
int (*vector) ( )
```

#14 :
— IOREC

Cette fonction retourne l'adresse du tableau d'organisation d'un des périphériques suivants :

0 : Interface RS232

1 : Interface Clavier

2 : Interface MIDI

Ce tableau a une longueur de 14 octets :

4 octets : adresse du tampon de données (Ibuf)

2 octets : taille du tampon de données (Ibufsize)

2 octets : pointeur en lecture sur le tampon (Tail index)
2 octets : pointeur en écriture sur le tampon (Head Index)
2 octets : High Water Mark
2 octets : Low Water Mark

Prenons par exemple le tampon clavier.

Son adresse (qui varie selon les ROMs) est donnée par Ibuf, et sa taille par Ibufsize (normalement 256 octets). Les données écrites ont une longueur de 4 octets (2 octets pour le scancode de la touche pressée, et 2 pour son code ASCII). A chaque fois que le processeur clavier décèle l'appui sur une touche, il code cet appui en 4 octets à l'adresse pointée par le Head Index. Quand l'utilisateur vérifie si une touche a été pressée, il compare les valeurs du Head et du Tail Index. Si celles-ci sont identiques, c'est qu'il ne s'est rien passé. Dans le cas contraire, il y a eu pression sur une touche. On peut alors lire sur le tampon son code ASCII et son SCANCODE.

Exemple d'utilisation en assembleur :

```
Initialisation_de_inkey :
    move.w    #1,-(sp)                  ; On récupère l'adresse
    move.w    #14,-(sp)                 ; du tableau de description
    trap      #14                       ; des données du clavier
    addq.w    #4,sp
    move.l    d0,a0                     ; Adresse dans A0
    move.l    (a0),ibuf                 ; Adresse du tampon clavier
    addq.w    #6,a0
    move.l    a0,tail_index_address     ; Adresse du Tail Index
    addq.w    #2,a0
    move.l    a0,head_index_address     ; Adresse du Head Index
    ...
inkey:                                  ; Routine inkey
    moveq     #0,d0                     ; Par défaut sur 'rien à rece-

    move.l    ibuf,a0
    move.l    tail_index_address,a2     ; Tail et Head
    move.l    head_index_address,a1     ; à la même valeur ?
    move.w    (a2),d2
    cmp.w     (a1),d2
    beq       pas_de_touche_pressée     ; Si oui, on sort de la routine
    addq.w    #4,d2
    addq.w    #4,(a2)                   ; On incrémente le Tail
    move.l    0(a0,d2.w),d0             ; et on récupère le code de
                                        la touche dans D0
pas_de_touche_pressée:
    rts                                 ; Retour
ibuf:
    ds.l 1
tail_index_address:
    ds.l 1
head_index_address:
    ds.l 1
```



La routine Inkey remplit le même rôle qu'un Inkey$ du basic, à savoir qu'elle regarde si une touche a été pressée. On l'appelle par un 'bsr inkey'. Au retour, si D0 contient 0, c'est qu'il n'y avait rien à récupérer dans le tampon clavier. Sinon, D0 contient le scancode et le code ASCII de la touche.

Une étude attentive de l'exemple ci-dessus vous aura sans doute fait remarquer que le Head pointe sur le dernier code à avoir été écrit. De même, le Tail pointe sur le dernier code à avoir été lu. C'est pour cette raison qu'il faut incrémenter le Tail avant de lire les données du tampon.

Un point particulier n'a pas été expliqué : la signification et l'utilité des LOW WATER MARK et HIGH WATER MARK. Ceux-ci pointent respectivement sur le premier et le troisième quart du tampon et sont utilisés par l'ordinateur pour la réception des données provenant de l'interface RS232. Si les données écrites dépassent le HIGH WATER MARK, la transmission s'arrête et ce jusqu'à ce que l'utilisateur en ait lu assez pour faire redescendre le Head au LOW WATER MARK. Cela évite que le tampon soit dépassé par un trop grand nombre de données lui arrivant.

Exemple d'utilisation en C :

```
long iorec(devno)

/* Définition des variables */

int devno;
```

 # 15 :
— RSCCONF

Permet une reconfiguration totale de l'interface RS232.

Six paramètres sont demandés :

SCR : valeur du registre MFP des caractères synchrones (registre 20).

TSR : valeur du registre MFP de l'état du transmetteur (registre 23).

RSR : valeur du registre MFP de l'état du récepteur (registre 22).

UCR : valeur du registre MFP du contrôle de l'USART (registre 21).

CTRL : mode de transmission. Sont disponibles les quatre choix suivants :

0 Pas de Handshake
1 Mode XON/XOFF
2 Mode RTS/CTS
3 Mode XON/XOFF et RTS/CTS

BAUD : définit la vitesse de transmission des données. 16 choix vous sont offerts :

Valeur du paramètre	Vitesse de transmission
0	19200 Bauds
1	9600
2	4800
3	3600
4	2400
5	2000
6	1800
7	1200
8	600
9	300
10	200
11	150
12	134
13	110
14	75
15	50

Exemple d'utilisation en assembleur :

```
    move.w      scr,-(sp)
    move.w      tsr,-(sp)
    move.w      rsr,-(sp)
    move.w      usr,-(sp)
    move.w      ctrl,-(sp)
    move.w      baud,-(sp)
    move.w      #14,-(sp)
    trap        #14
    add.w       #14,sp
    ...
scr:
    ds.w 1
tsr:
    ds.w 1
rsr:
    ds.w 1
usr:
    ds.w 1
ctrl:
    ds.w 1
baud:
    ds.w 1
```

Remarque :

Si un des paramètres ne doit pas être modifié, il suffit de mettre pour valeur −1 sur la pile (voir fonction SETSCREEN).

Exemple d'utilisation en C :

```
Rsconf(speed, flowctl, ucr, rsr, tsr, scr)
/* Définition des variables */
int speed, flowctl, ucr, rsr, tsr, scr;
```



#16 :
— KEYBLT

Cette fonction permet une redéfinition totale du clavier (y compris le pavé numérique et les flèches). Il faut pour cela construire des tables contenant les codes ASCII à retourner pour chaque appui sur une touche, y compris avec **SHIFT** ou **CONTROL**.

Trois tables sont nécessaires.

Si une des trois tables du GEM ne doit pas être changée, la valeur −1 doit être donnée à la place.

Exemple d'utilisation en assembleur :

```
    move.l    #ctrl,-(sp)       ; Nouvelle table CTRL
    move.l    #-1,-(sp)         ; Table SHIFT inchangée
    move.l    #-1,-(sp)         : Table normale inchangée
    move.w    #16,-(sp)         ; Code KEYTBL
    trap      #14               ; Appel en ROM
    add.w     #14,sp            ; Et restauration de la pile
    ...
ctrl :
    ds.b 128                    ; 128 octets nuls
```

L'exemple ci-dessus conserve deux tables standard, mais renvoie pour toute combinaison de touches avec **CTRL** le code 0. La touche **CTRL** est ainsi inhibée.

Note :

Chacune des tables doit avoir une longueur de 128 octets. Pour connaître l'ordre des touches dans les tables, voir la table des SCANCODES en annexe (Partie 1, chapitre 3, annexe 2).

Exemple d'utilisation en C :

```
long Keybit(unshift, shift, capslock)
/* Définition des variables */
char
```

#17 :
— RANDOM

Cette routine renvoie une valeur aléatoire sur les 24 bits de poids faible du registre D0 (nombre donc compris entre 0 et 16777216).

Exemple d'utilisation en assembleur :

```
move.w    #17,-(sp)         ; Code RANDOM sur la pile
trap      #1                ; Appel en ROM
addq.w    #2,sp             ; Et restauration de la pile
...                         ; D0 contient un nombre aléatoire
```

Exemple d'utilisation en C :

```
long Random( )
```

#18 :
— PROTOBT

Permet la création d'un BOOT SECTOR en mémoire. Après cet appel, il n'a plus qu'à être copié sur le premier secteur de la première piste d'une disquette. Les paramètres demandés sont :

EXECFLAG : Exécutabilité du boot secteur ou non (normalement, un boot secteur n'est pas exécutable, c'est-à-dire qu'il contient uniquement des renseignements sur la disquette, mais pas de programme).

0 pour non exécutable
1 pour exécutable
2 pour inchangé par rapport au dernier bootsecteur chargé.

DISKTYP : Type de formatage.

0 pour 40 pistes simple face (180 Ko)
1 pour 40 pistes double face (360 Ko)
2 pour 80 pistes simple face (360 Ko)
3 pour 80 pistes double face (720 Ko)
4 pour inchangé.

Les DISKTYP 0 et 1 ne sont utiles que pour conserver une certaine compatibilité avec le format MS-DOS. On peut, en écrivant une petite routine, formater des disquettes lisibles sur compatible PC.

Le format ST n'est, quant à lui, pas lisible sur PC mais une disquette PC peut être lue sur ST.

SERIALNR : Numéro d'identification de la disquette (sur 24 bits). Si l'on met −1, un numéro aléatoire sera généré. Un nombre supérieur à 24 bits garde le numéro inchangé.

TAMPON : Adresse d'un tampon de 512 octets dans lequel sera inscrit le boot secteur.

Exemple d'utilisation en assembleur :

```
clr.w       -(sp)              ; Non exécutable
move.w      #2,-(sp)           ; 80 pistes simple face
move.l      #-1,-(sp)          ; Numéro de série inchangé
move.l      #tampon,-(sp)      ; Adresse du tampon de travail
move.w      #18,-(sp)          ; Code de PROTOBT
trap        #14                ; Appel en ROM
add.w       #14,sp             ; Et restauration de la pile
...                            ; Le tampon contient maintenant
                               ; un secteur de boot qu'il suffit
                               ; de copier sur une disquette

tampon:
   ds.b 512
```

Note :

Les routines d'accès au lecteur de disquette fonctionnent indifféremment sur des unités 3.5 pouces ou 5.25 pouces.



Exemple d'utilisation en C :

```
Protobt(buf, serialno, disktype, execflag)
/* Définition des variables */
char buf[512];
long serialno;
int disktype, execflag;
```

#19 :
— FLOPVER

Cette fonction permet la vérification d'un ou de plusieurs secteurs sur une piste. Cinq paramètres sont demandés :

COUNT : Nombre de secteurs à vérifier (1 à 9 pour un format standard).

FACE : 0 ou 1 pour les faces A ou B.

TRACK : piste dans laquelle s'effectue la vérification.

SECTOR : numéro du premier secteur à vérifier.

DEV : unité de disquette (0 ou 1 pour A ou B).

FILLER : quatre octets toujours à 0.

BUFFER : adresse du tampon de travail des ROMs. Doit être égal à 512 fois le nombre de secteurs à vérifier.

Exemple d'utilisation en assembleur :

```
    move.w    #2,-(sp)          ; Vérifier un secteur
    clr.w     -(sp)             ; de la première face
    clr.w     -(sp)             ; de la première piste
    move.w    #1,-(sp)          ; A partir du secteur numéro 1
    clr.w     -(sp)             ; Sur l'unité de disquette A
    clr.l     -(sp)
    move.l    #tampon,-(sp)     ; Adresse du tampon de travail
    move.w    #19,-(sp)         ; Code FLOPVER
    trap      #14,sp            ; Appel en ROM
    add.w     #12,sp            ; Restauration de la pile
    ...
tampon :
    ds.b 512*2                  ; Tampon pour deux secteurs
```

Exemple d'utilisation en C :

```
int Flopver(buf, filler, devno, spt, trackno, sideno, count)
/* Définition des variables */
char *buf;
long filler;
int devno, spt, trackno, sideno, count;
```

#20 :
— SCRDMP

Cette fonction permet d'effectuer une copie de l'écran sur imprimante. C'est elle qui est appelée lors de l'appel des touches **ALT-HELP**.

Exemple d'utilisation en assembleur :

```
move.w    #20,-(sp)
trap      #14
addq.w    #2,sp
```

Note :

Lors d'un appel Screen-Dump (SCRDMP), l'ordinateur utilise la configuration standard d'imprimante. Si votre modèle n'est pas connu par le ST, il vous faudra auparavant utiliser la fonction RSCCONF.

Exemple d'utilisation en C :

```
Scrdmp( )
```

#21 :
— CURSCONF

Cette fonction permet un paramétrage complet du curseur. Plusieurs paramètres sont demandés :

FREQ : Fréquence de flash du curseur. Cette fréquence dépend du moniteur utilisé. En haute résolution, elle est comptée en multiples de 1/70 de seconde. En basse ou moyenne, cela dépend du type de moniteur utilisé (50 ou 60 Hz).

FONC : Fonction curseur demandée :

- 0 : Suppression du curseur.
- 1 : Activation du curseur.
- 2 : Clignotement du curseur autorisé.
- 3 : Clignotement du curseur interdit.
- 4 : Mise en place d'une nouvelle fréquence de clignotement.
- 5 : Demande de la fréquence actuelle de clignotement.

Exemple d'utilisation en assembleur :

```
clr.w     -(sp)
move.w    #5,-(sp)        ; Demande de la
move.w    #21,-(sp)       ; fréquence de
trap      #14             ; clignotement
addq.w    #6,sp           ; Résultat dans D0
lsr.w     #1,d0           ; On divise D0 par 2
move.w    d0,-(sp)        ; et on le place
move.w    #4,-(sp)        ; comme nouvelle
move.w    #21,-(sp)       ; fréquence
trap      #1
addq.w    #6,sp
```

L'exemple ci-dessus multiplie par 2 la vitesse de clignotement.

Exemple d'utilisation en C :

```
int Cursconf(function, operand)
/* Définition des variables */
int function, operand;
```

#22 :
— SETTIME

Cette fonction permet de régler l'horloge interne du ST. Attention, après chaque extinction de la machine, celle-ci devra être remise à la bonne heure (sauf pour les mégas ST qui disposent d'une horloge alimentée par piles).

Paramètre nécessaire :

TIME codé sur 32 bits (4 octets) :

bits 0 à 4 : seconde (les secondes sont comptées 2 par 2 → 0/29)
bits 5 à 10 : minute (0 →59)
bits 11 à 15 : heure (0→ 24)
bits 16 à 20 : jour (1 → 31)
bits 21 à 24 : mois (1 → 12)
bits 25 à 31 : année (0 → 127) à partir de 1980.

Exemple d'utilisation en assembleur :

```
clr.l       -(sp)          ; 1er janvier 1980 à 0 heure, 0 minute,
                             0 seconde
move.w      #22,-(sp)
trap        #14
addq.w      #6,sp
```

Exemple d'utilisation en C :

```
Settime(datetime)
/* Définition des variables */
long datetime;
```

#23 :
— GETTIME

Retourne la date et l'heure au format décrit dans la fonction précédente.

Exemple d'utilisation en assembleur :

```
move.w      #23,-(sp)
trap        #14
addq.w      #2,sp
```

Exemple d'utilisation en C :

```
long Gettime( )
```

#24 :
— BIOSKEYS

Voici le pendant de la fonction XBIOS #16. En effet, si une nouvelle configuration clavier a été choisie, BIOSKEYS permet de le remettre dans sa configuration standard.

Exemple d'utilisation en assembleur :

```
move.w    #24,-(sp)      ; Code BIOSKEYS
trap      #14            ; Appel en ROM
addq.l    #2,sp          ; Et restauration de la pile
```

Exemple d'utilisation en C :

```
Bioskeys ( )
```

#25 :
— IKBDWS

Le processeur clavier 6850 étant en partie contrôlable par l'utilisateur, cette fonction permet l'envoi de commandes. Deux paramètres sont utilisés : l'adresse de la chaîne à envoyer, ainsi que sa longueur −1.

Exemple d'utilisation en assembleur :

```
    move.l    #chaine,-(sp)     ; Adresse de la chaîne
    clr.w     -(sp)             ; 1 octet à envoyer -1=0
    move.w    #25,-(sp)         ; Code IKBDWS
    trap      #14               ; Appel en ROM
    addq.w    #8,sp             ; Et restauration de la pile
    ...
    ...
chaine:
    dc.b $f                     ; Inversion du sens vertical de
                                ; déplacement de la souris
```

Exemple d'utilisation en C :

```
Ikbdws(cnt, prt)

/* Définition des variables */

int cnt;
char *ptr;
```

#26 :
— JDSINT

Un appel à cette fonction permet le blocage d'une des interruptions gérées par le MFP 68901. Cette fonction est bien entendu à manier avec la plus grande précaution car plusieurs de ces interruptions sont d'une importance capitale (gestion clavier et souris, disquettes, etc.).

Le seul paramètre est le numéro de l'interruption à bloquer (0 à 15).



Exemple d'utilisation en assembleur

```
move.w    #6,-(sp)        ; Blocage de la gestion
move.w    #26,-(sp)       ; Clavier et midi
trap      #14             ; Appel en ROM
addq.w    #4,sp           ; Et restauration de la pile
```

Exemple d'utilisation en C :

```
Jdisint(interno)

/*Définition des variables*/

int interno;
```

#27 :
— JENABINT

Cette fonction, contrairement à la précédente, débloque une interruption 68901. Le paramètre demandé est bien entendu le même.

Exemple d'utilisation en assembleur :

```
move.w    #6,-(sp)        ; Déblocage de la gestion
move.w    #27,-(sp)       ; Clavier et midi
trap      #14             ; Appel en ROM
addq.w    #4,sp           ; Et restauration de la pile
```

Exemple d'utilisation en C :

```
Jenabint(interno)

/* Définition des variables */

int interno;
```

#28 :
— GIACCESS

Par cette fonction, vous pouvez écrire ou lire des données dans l'un des 16 registres du processeur sonore. Vous devez, pour cela, fournir à la routine le numéro du registre concerné et, le cas échéant, la valeur à écrire.

Après l'appel **TRAP**, le registre D0 contient, en cas de lecture, la valeur demandée. C'est le bit 7 du numéro de registre qui précise si c'est une lecture ou une écriture qui est voulue. Par exemple :

$0 : lecture du registre 0

$1 : lecture du registre 1

$80 : écriture du registre 0

$81 : écriture du registre 1

Exemples d'utilisation en assembleur

```
move.w    #88,-(sp)        ; Ecriture de la valeur
move.w    #5,-(sp)         ; 5 sur le registre 8
move.w    #28,-(sp)        ; Code GIACCESS
trap      #14              ; Appel en ROM
addq.w    #6,sp            ; Et restauration de la pile
...
...
move.w    #5,-(sp)         ; Lecture du registre 5
clr.w     -(sp)            ; Mettre ici n'importe quelle valeur
move.w    #28,-(sp)        ; Code GIACCESS
trap      #14              ; Appel en ROM
addq.w    #6,sp            ; Et restauration de la pile
...                        ; D0 contient la valeur lue
```

Exemple d'utilisation en C :

```
char Giaccess (data, regno)

/* Définition des variables */

char regno;
int regno;
```

#29 :
— OFFGIBIT

Cette fonction permet d'effacer un ou plusieurs des bits du port A du circuit sonore. Le seul paramètre demandé est un octet où sont mis à 0 tous les bits devant être effacés.

Exemple d'utilisation en assembleur :

```
move.w    #%00001011,-(sp)  ; Effacement des bits 7, 6, 5, 4
                            ; 2 (conservation des bits 3,
                            ; 1, 0)
move.w    #29,-(sp)         ; Code OFFGIBIT
trap      #14               ; Appel en ROM
addq.w    #4,sp             ; Et restauration de la pile
```

Exemple d'utilisation en C :

```
Offgibit(bitno)

/* Définition des variables */

int bitno;
```



#30 :
— ONGIBIT

Voici le pendant de la fonction précédente. ONGIBIT permet, contrairement à OFFGIBIT, de mettre à 1, un ou plusieurs des bits du port A.

Exemple d'utilisation en assembleur :

```
move.w    #%00000011,-(sp)    ; Mise à 1 des bits 0 et 1
                              ; (conservation des bits 7,
                              ; 6, 5, 4, 3, 2)
move.w    #30,-(sp)           ; Code ONGIBIT
trap      #14                 ; Appel en ROM
addq.w    #4,sp               ; Et restauration de la pile.
```

Exemple d'utilisation en C :

```
Ongibit(bitno)

/* Définition des variables */

int bitno;
```

#31 :
— XBTIMER

Cette fonction reprogramme un des 4 timers du MFP 68901.

TIMER : Numéro du timer à programmer

0 pour le timer A
1 pour le timer B
2 pour le timer C
3 pour le timer D

DATA et CONTROL correspondent aux registres de données et de contrôle du timer choisi. Vecteur est l'adresse de la routine que vous voulez voir effectuée par l'interruption.

Note :

La routine en interruption doit être terminée par un RTE et non pas par un RTS.

Exemple d'utilisation en assembleur :

```
    move.l    #routine,-(sp)
    move.w    #7,-(sp)          ; CONTROL (prédiviseur/200)
    move.w    #$ff,-(sp)        ; DATA (décomptage de 256)
    move.w    #2,-(sp)
    move.w    #31,-(sp)
    trap      #14
    add.w     #12,sp
    ...
    ...
routine:
    ...
    rte
```

Exemple d'utilisation en C :

```
Xbtimer(timer, control, data, vec)

/* Définition des variables */

int timer, control, data;
int (*vec) ( );
```

#32 :
— DOSOUND

Cette fonction autorise une certaine programmation du processeur sonore du ST. Le seul paramètre demandé est l'adresse d'un tableau de données à transmettre. Le premier octet de ces données fixe le type de transfert choisi.

Valeur du premier octet :

0 à 15 ($0 à $f)

Numéro du registre du processeur sonore. L'octet suivant correspond à l'octet à transmettre dans le registre choisi.

128 ($80)
Place l'octet suivant dans le registre temporaire du processeur sonore.

129 ($81)
Cette commande nécessite 3 octets comme argument. Le premier est le numéro du registre à utiliser, le second une valeur à ajouter au registre temporaire, et le troisième la valeur de fin. L'instruction sera exécutée jusqu'à ce que la valeur du registre temporaire soit égale à celle de la valeur finale.

130 à 255 ($82 à $ff)
Nécessite un paramètre. Si ce dernier est zéro, le son s'arrête. Sinon, il représente le nombre d'interruptions *timer système* avant le prochain changement du son.

Exemple d'utilisation en assembleur :

```
    move.l    #octets,-(sp)
    move.w    #32,-(sp)
    trap      #14
    addq.w    #6,sp
    ...
octets:
    dc.b      $a,$50
```

Exemple d'utilisation en C :

```
void Dosound(octets)

/* Définition des variables */

long octets;
```

#33 : — SETPRT

Cette fonction permet de changer la configuration d'imprimante du ST. Elle ne nécessite qu'un paramètre appelé CONFIG. Structure de configuration :

bit	0	1
0	imprimante à marguerite	imprimante à ruban
1	imprimante couleur	imprimante monochrome
2	imprimante Atari	imprimante compatible Epson
3	mode listing	mode courrier
4	port parallèle	port série
5	papier type listing	feuille à feuille
6-14	réservé	réservé
7-15	toujours 0	toujours 0

Exemple d'utilisation en assembleur :

```
move.w    #%0000000000000100,-(sp)    ; Imprimante
                                      ; compatible
                                      Epson
move.w    #33,-(sp)                   ; Code SETPRT
trap      #14                         ; Appel en ROM
addq.w    #4,sp                       ; Et restauration
                                      ; de la pile
```

Exemple d'utilisation en C :

```
Setprt(config)
/* Définition des variables */
int config;
```

#34 : — KBDBASE

Cette fonction renvoie à l'utilisateur l'adresse d'une table de vecteurs relatifs aux processeurs clavier et MIDI.

Exemple d'utilisation en assembleur :

```
move.w    #34,-(sp)      ; Code KBDBASE
trap      #14            ; Appel en ROM
addq.w    #2,sp          ; Et restauration de la pile
...                      D0 contient l'adresse en question
```

Structure de la table :

4 octets:	midivec	routine MIDI
4 octets:	vkbderr	erreur clavier (non utilisée)
4 octets:	vmiderr	erreur midi (non utilisée)

4 octets:	statvec	état du clavier (non utilisée)
4 octets:	mousevec	routine souris
4 octets:	clockvec	routine horloge
4 octets:	joyvec	routine joystick (non utilisée)
4 octets:	sysmidi	routine système MIDI
4 octets:	syskbd	routine système clavier + souris

Exemple d'utilisation en C :

```
long Kbdbase ( )

/* Pour la structure du tableau dont l'adresse est donnée par l'appel,
voir l'exemple en assembleur. */
```

#35 :
— KBRATE

Grâce à cette fonction, on peut régler le délai de répétition des touches du clavier (la latence). Deux paramètres sont nécessaires.

REPET : Temps d'attente entre deux écritures de la touche compté en multiples de 20 ms (50 Hz)

RETARD : Temps d'attente entre la première et la deuxième écriture du caractère.

Exemple d'utilisation en assembleur :

```
move.w    #50,-(sp)      ; 1 seconde de délai de répétition
move.w    #200,-(sp)     ; 4 secondes
                         ; avant la première répétition
move.w    #35,-(sp)      ; Code KBRATE
trap      #14            ; Appel en ROM
addq.w    #6,sp          ; Et restauration de la pile
...
(le clavier est très ralenti)
```

Exemple d'utilisation en C :

```
int Kbrate(initial, repeat)

/* Définition des variables */

unsigned char initial, repeat;
```

#36 :
— PTRBLK

Primitive de la fonction SCRDMP de hardcopy à l'écran. Il est, dans la plupart des cas, plus intéressant de faire directement appel à la fonction #20.



#37
— WVBL

Cette routine a pour fonction d'attendre la prochaine interruption VBL puis de rendre la main à l'utilisateur. Elle peut être, par exemple, utile pour synchroniser une animation avec la fréquence d'apparition des images sur le moniteur.

Exemple d'utilisation en assembleur :

```
move.w    #37,-(sp)      ; Code WVBL
trap      #14            ; Appel en ROM
addq.w    #2,sp          ; Et restauration de la pile
```

Exemple d'utilisation en C :

```
/* Attention ! en C, cette fonction a pour nom Vsync */
Vsync ( )
```

#38 :
— SUPEXEC

Si vous êtes en mode user et qu'une de vos routines doit être effectuée en mode super, alors voici une fonction faite pour vous ! C'est en effet le rôle de SUPEXEC. Le seul paramètre demandé est l'adresse de la routine (qui devra finir par un RTS).

Exemple d'utilisation en assembleur :

```
    move.l    #routine,-(sp)
    move.w    #38,-(sp)
    trap      #14
    addq.w    #6,sp
    ...
    ...
routine:
    ...
    ...
    rts
```

Exemple d'utilisation en C :

```
long Supexec(f)
/* Définition des variables */
int (*f) ( );
```

#39 :
— PUNTAES

Fonction non utilisable avec tout ST possédant le TOS en ROM. Elle était initialement prévue pour libérer l'espace mémoire pris par l'AES.



3/3.1

Programmation avec XBIOS

3/3.1.1

Le port A du circuit son

En programmant directement les registres du générateur sonore de votre Atari vous avez peut-être eu le malheur ou l'angoisse de constater que la led de sélection du lecteur de disquette s'allumait et que même parfois ce même Floppy driver refusait par la suite d'obtempérer à vos commandes. La cause et le remède vous paraîtront sans doute plus clairs après l'utilisation des fonctions **XBIOS $1D** (Offdigit) et **XBIOS $1E** (Ondigit).

La première fonction **(XBIOS (29))** permet d'annuler de façon sélective un des bits du port A du circuit son. Le bit 1 représente le bit de sélection de l'unité de disquette A. Pour annuler ce bit sans toucher aux autres (le principe de fonctionnement logique est le **AND**) il faut envoyer le code 11111101 en binaire. Attention à ne pas confondre les 0 et les 1 comme c'est le cas dans de trop nombreux ouvrages !

En mettant ce bit à 0, on sélectionne le lecteur de disquette A alors que la logique humaine pousse à penser qu'on sélectionne en mettant à un. Après l'appel de **XBIOS(29)** avec ce masque, vous verrez donc le lecteur sélectionné, mais pas encore bloqué car le son ne marche pas en même temps et ce n'est pas le but du programme.

La seconde fonction **(XBIOS(30))** réalise l'opération inverse. Les attributions de bits sont inchangées mais le principe de fonctionnement logique est différent (le **OR**). Pour désactiver le bit 1 sans toucher aux autres, il faut le mettre à un, donc faire un **OU** logique avec le masque 00000010 qui est traduit en 10 en binaire par l'interpréteur.

Le programme effectue un va et vient itératif entre sélection et désélection de l'unité de disquette A.

En utilisant les masques différemment, vous pourriez aussi sélectionner et désélectionner plusieurs bits du port A en même temps sans problème.

VERSION BASIC GFA DU PROGRAMME PORT A

```
FOR i=1 TO 10
  ~XBIOS(29,&X11111101) ! bit 1 disk A activé
  PRINT AT(10,10);"weka"
  PAUSE 50
  ~XBIOS(30,&X10)        ! bit 1 désactivé
  PRINT AT(10,10);"      "
  PAUSE 50
NEXT i
~INP(2)
```

VERSION ASSEMBLEUR DU PROGRAMME PORT A

```
 .TEXT
;BOUCLE DE 10 REPETITIONS
            move.w    #9,d5
BOUCLE0move.w    #$fd,-(sp)
            move.w    #29,-(sp)
            trap      #14
            addq.l    #4,sp
;BOUCLE D'ATTENTE
            move.l    #20000,d0
BOUCLE1:
            dbra      d0,BOUCLE1

            move.w    #$2,-(sp)
            move.w    #30,-(sp)
            trap      #14
            addq.l    #4,sp
;BOUCLE D'ATTENTE
            move.l    #20000,d0
BOUCLE2:
            dbra      d0,BOUCLE2
            dbra      d5,BOUCLE0
            move.w    #7,-(sp)
            trap      #1
            addq.l    #2,sp
            clr.w     -(sp)
            trap      #1
            .END
```

VERSION BASIC OMIKRON DU PROGRAMME PORT A

```
FOR i=1 TO 10
  XBIOS(,29,&b11111101) ! bit 1 disk A activé
  TEXT 50,50,"weka"
  WAIT 200
  XBIOS(,30,&b10)        ! bit 1 désactivé
  TEXT 50,50 "      "
  WAIT 200
NEXT i
  GEMDOS(,7)
```



3/3.1.2

Le swap d'écran

La mémoire écran détient toutes les informations nécessaires au shifter pour réaliser l'affichage. L'adresse du bloc de 32000 octets est donnée par le CPU par l'intermédiaire de deux registres. Cette adresse peut donc aussi être spécifiée par programme, la seule règle étant de commencer au début d'une page.

Toute la mémoire d'un ordinateur est divisée en pages de 256 octets ou $100 en hexadécimal. C'est ce que réalisent les programmes GFA et OMIKRON en début de réservation. En général, les deux mémoires utilisées par l'Atari (le bloc du shifter et celui du CPU) sont identiques. Vous retrouvez respectivement ici les notions d'écran physique et d'écran logique. Aucune règle n'impose l'identité entre ces deux mémoires si ce n'est le désir de voir des affichages se réaliser. Le gros intérêt est de pouvoir séparer ces deux mémoires lorsqu'on veut effectuer des animations graphiques. Il suffit de posséder deux écrans distincts, A et B par exemple. Au début A est physique et B est logique. Nous écrivons le texte « WEKA » sur B, rien n'apparaît à l'écran. Nous nommons B physique et A logique : le texte apparaît. Effaçons A et écrivons le texte un peu plus loin. Au moment ou nous inverserons à nouveau les écrans A et B, le texte apparaîtra décalé et ainsi de suite, sans aucun scintillement de l'image car à aucun moment nous ne verrons l'écran avec un vide à la place du texte comme on pourrait le faire avec un TEXT x,y," "directement sur l'écran physique. Si vous deviez faire apparaître une image au lieu d'un texte, le principe serait exactement le même. Le programme ne réserve qu'une mémoire logique et inverse écran physique d'origine (donné par XBIOS(2)) ou Phybase) et nouvelle mémoire, avant de regénérer la même adresse pour les écrans physique et logique en fin de programme. L'adresse écran logique est donnée par XBIOS(3) ou Logbase.

La fonction que nous utilisons ici est la routine XBIOS(5) Setscreen qui attend trois paramètres : deux mots longs qui notent les adresses de début des écrans physique et logique (mettre −1 si on veut conserver la valeur en cours), et un mot spécifiant la résolution de l'écran que l'on ne peut d'ailleurs pas changer impunément. Notez bien la façon de trouver une adresse multiple de 256 par le AND &hffffff00 sur une zone de plus de 32256 octets.

VERSION BASIC GFA DU SWAP D'ECRANS

```
CLS
DEFMOUSE 0
PAUSE 100
ON BREAK GOSUB final
ecran_logique$=STRING$(32700,0)
ecran_physique$=STRING$(32700,0)
orig%=XBIOS(2)
deblog%=(VARPTR(ecran_logique$)+256) AND &HFFFFFF00
debphy%=(VARPTR(ecran_physique$)+256) AND &HFFFFFF00
~XBIOS(5,L:deblog%,L:-1,-1)
xdevl=3
ydevl=2
FOR fl=0 TO 200
  TEXT x1&-xdevl,y1&-ydevl," "
  ADD x1&,xdevl
  ADD y1&,ydevl
  TEXT x1&,y1&,"weka"
  IF XBIOS(2)=deblog%
    ~XBIOS(5,L:deblog%,L:debphy%,-1)
  ELSE
    ~XBIOS(5,L:debphy%,L:deblog%,-1)
  ENDIF
  VSYNC
NEXT fl
final
PROCEDURE final
  ~XBIOS(5,L:-1,L:orig%,-1)
  EDIT
  END
RETURN
```



VERSION BASIC OMIKRON DU SWAP D'ECRAN

```
' on réserve la mémoire au gemdos
CLEAR 33000
DIM Scr%L(1)
XBIOS(Scr%L(0),3)
'l'écran doit commencer au bord de la page
Scr%L(1)=(MEMORY(32256)+256)AND $FFFF00
' boucle du swap d'écrans
FOR I=0 TO 200
XBIOS(,5,-1,-1,HIGH(Scr%L(I MOD 2)),LOW(Scrf%L(I MOD 2)),-1)
XBIOS(,5,HIGH(Scr%L(1-I MOD 2)),LOW(Scrf%L(1-I MOD 2)),-1,-1,-1)
CLS
TEXT I,I,"WEKA"
' attente du retour d'image VSYNC
XBIOS(,37)
NEXT I
' attente d'une touche clavier
GEMDOS(,7)
'on restitue les adresses écrans comme à l'origine
XBIOS(,5,-1,-1,HIGH(Scr%L(0)),LOW(Scrf%L(0)),-1)
XBIOS(,5,HIGH(Scr%L(0)),LOW(Scrf%L(0)),-1,-1,-1)
END
```

3/4

Le GEMDOS

I - La présentation de GEMDOS

GEMDOS (*GEM Disk Operating System*) représente la partie du système d'exploitation qui gère l'accès aux fichiers ainsi que l'écran et la mémoire. Il représente, avec BIOS et XBIOS, le noyau proprement dit de l'Atari ST.

II - La gestion de fichiers

L'utilisation d'un support magnétique comme la disquette impose à l'ordinateur de savoir exactement où se trouve une information. Pour cela, GEMDOS manipule des fichiers qui sont des informations de taille variable. Sans entrer tout de suite dans les détails, sachons que GEMDOS se charge de découper un fichier en petites unités qu'il rangera à sa façon dans la disquette. Cette méthode lui permet de gaspiller le moins de place possible. Un autre niveau d'organisation de la disquette constitue le dossier ou la pochette. Cette notion permet de ranger dans un même tiroir les fichiers ayant trait au même type d'application. Par exemple, il peut être pratique, dans une base de données, de ne pas mélanger la base et ses fichiers d'index avec le fichier programme de gestion de la base de données.

Nous venons de voir que GEMDOS organisait ses données sous forme de fichiers. Voyons maintenant comment il est possible d'utiliser GEMDOS pour travailler nous-même avec ces fichiers. La technique générale consiste à ouvrir un fichier, connaissant son nom, à lire ou à écrire dans ce fichier et le fermer quand c'est terminé. GEMDOS dispose, pour identifier ses fichiers, de Handles ou Identificateurs.

Lors de l'ouverture, la fonction GEMDOS d'ouverture ou de création renvoie au programme appelant (c'est-à-dire le vôtre !) une valeur que celui-ci devra indiquer à chaque fois qu'il fera une quelconque action sur ce fichier. C'est plus pratique que d'indiquer le nom du fichier et c'est bien plus sûr.

III - La gestion de l'écran et du clavier

GEMDOS permet également d'accéder à l'écran. Quelques fonctions affichent du texte à l'écran selon les commandes VT52 et il est également possible de saisir une chaîne au clavier avec écho sur l'écran.

Voyons les commandes VT-52 :

ESC A : Déplace le curseur vers le haut.

ESC B : Déplace le curseur vers le bas.

ESC C : Déplace le curseur vers la droite.

ESC D : Déplace le curseur vers la gauche.

ESC E : Efface l'écran et positionne le curseur en haut à gauche (position HOME).

ESC H : Positionne le curseur en position HOME.

ESC I : Déplace le curseur sur la ligne précédente. Mais dans ce cas, s'il se trouve sur la 1re ligne, un scrolling est généré.

ESC J : Efface l'écran depuis la position du curseur.

ESC K : Efface la ligne depuis la position du curseur.

ESC L : Insère une ligne.

ESC M : Supprime une ligne.

ESC Y : Positionne le curseur à la ligne et la colonne.

Celles-ci sont indiquées dans les deux caractères qui suivent en ajoutant 32 à la valeur désirée. Par exemple, la séquence ESC Y 1 B positionnera le curseur en ligne 17 et en colonne 34.

ESC b : Fixe la couleur d'affichage caractère aux 4 bits de poids faible du caractère qui suit. Par exemple ESC b 5 est équivalent à ESC F (la couleur 5).

ESC c : Fixe la couleur de fond.

ESC d : Efface l'écran depuis son début jusqu'à la position du curseur incluse.

ESC e : Active le curseur.

ESC f : Désactive le curseur.

ESC j : Sauve la position courante du curseur.

ESC k : Positionne le curseur à la dernière position sauvée par ESC j.

ESC l : Efface la ligne entière et positionne le curseur au début.

ESC o : Efface la ligne depuis son début jusqu'à la position du curseur incluse.

ESC p : Passe en vidéo inverse.

ESC q : Sort du mode vidéo inverse.

ESC v : Passe en mode coupure de ligne. Dans ce mode, les caractères qui sont affichés après la dernière colonne seront renvoyés sur la ligne suivante. Si besoin, un scrolling sera réalisé.

ESC w : Désactive le mode coupure de ligne.



IV - La gestion de la mémoire

GEMDOS dispose de trois commandes pour allouer, libérer et réduire une zone mémoire. En effet, il est hors de question d'écrire à une position de la mémoire sans avoir demandé au préalable à GEMDOS si cette position n'était pas déjà utilisée par un autre programme.

V - Les messages d'erreurs

Les fonctions GEMDOS que nous allons étudier renvoient généralement une valeur qui peut être soit le résultat de la fonction soit un code d'erreur.

Par exemple, la fonction Fdup renverra zéro si l'opération s'est bien déroulée mais renverra EIHNDL si l'identificateur passé en paramètre n'est pas standard.

Les numéros des codes d'erreurs sont des longs négatifs.

BIOS utilise les numéros compris entre −1 et −31 et GEMDOS utilise ceux compris entre −32 et −127.

E_OK	0	0×00000000	OK (pas d'erreur)
ERROR	−1	0×FFFFFFFF	Erreur générale
EDVRNR	−2	0×FFFFFFFE	Lecteur non disponible
EUNCMD	−3	0×FFFFFFFD	Commande inconnue
E_CRC	−4	0×FFFFFFFC	Erreur CRC
EBADRQ	−5	0×FFFFFFFB	Mauvaise demande
E_SEEK	−6	0×FFFFFFFA	Erreur de positionnement
EMEDIA	−7	0×FFFFFFF9	Media inconnu
ESECNF	−8	0×FFFFFFF8	Secteur introuvable
EPAPER	−9	0×FFFFFFF7	Manque de papier
EWRITF	−10	0×FFFFFFF6	Erreur d'écriture
EREADF	−11	0×FFFFFFF5	Erreur de lecture
EWRPRO	−13	0×FFFFFFF3	Ecriture sur disquette protégée
E_CHNG	−14	0×FFFFFFF2	Changement de disquette détecté
EUNDEV	−15	0×FFFFFFF1	Périphérique inconnu
EBADSF	−16	0×FFFFFFF0	Mauvais formatage
EOTHER	−17	0×FFFFFFEF	Insérez une autre disquette
EINVFN	−32	0×FFFFFFE0	Numéro de fonction invalide
EFILNF	−33	0×FFFFFFDF	Fichier introuvable
EPTHNF	−34	0×FFFFFFDE	Chemin introuvable
ENHNDL	−35	0×FFFFFFDD	Plus d'indentificateur
EACCDN	−36	0×FFFFFFDC	Accès interdit
EIHNDL	−37	0×FFFFFFDB	Identificateur invalide
ENSMEM	−39	0×FFFFFFD9	Mémoire insuffisante
EIMBA	−40	0×FFFFFFD8	Adresse bloc mémoire invalide
EDRIVE	−46	0×FFFFFFD2	Disque invalide
ENMFIL	−49	0×FFFFFFCF	Plus de fichier
ERANGE	−64	0×FFFFFFC0	Erreur de champ
EINTRN	−65	0×FFFFFFBF	Erreur interne à GEMDOS
EPLFMT	−66	0×FFFFFFBE	Fichier non exécutable
EGSBF	−67	0×FFFFFFBC	Erreur accroissement bloc mémoire

VI - Les fonctions GEMDOS

#0 :
void Pterm0 ( )

Cette fonction permet de sortir du programme en cours et de rendre la main au programme appelant. Tous les fichiers encore ouverts sont fermés et tous les blocs mémoire que cette application a alloués sont libérés, ainsi que la place mémoire utilisée par ce programme. La valeur 0 est renvoyée au programme appelant.

Exemple d'utilisation en assembleur :

```
clr.w     -(SP)
trap      #1
```

Exemple d'utilisation en C :

```
main ( )
{
   Pterm ( );
}
```

#1 :
unsigned long
Cconin( )

Cette fonction lit un caractère sur l'entrée standard (généralement la console). En retour, la fonction renvoie un long indiquant quelle touche a été frappée. Ce long contient le code ASCII de la touche ainsi que le scan code indiquant la position de la touche. Ce scan code est utile dans le cas des touches « spéciales » telles que **HELP, INSERT, TOUCHES DE FONCTIONS, ...** qui ont un code ASCII nul et ne sont indentifiables que par le scan code.

bits 31...24	bits 23...16	bits 15...8	bits 7...0
0 ou bits de shift	scan code	0	code ASCII

Exemple d'utilisation en assembleur :

```
   mov.w     #1, -(SP)        ; Code de la fonction Cconin
   trap      #1               ; Appel de GEMDOS
   addq.l    #2, SP
   move.l    D0,Car           ; D0 contient le code du caractère
   ...
Car:
   dc.l      0
```

Exemple d'utilisation en C :

```
main ( )
{  unsigned long c;
   c=Cconin ( );
}
```



#2 :
void Cconout(c)
unsigned int c;

Cette fonction, à l'inverse de la précédente, permet d'envoyer un caractère sur la sortie standard (généralement l'écran). Seuls les 8 bits de poids faibles sont pris en compte.

Exemple d'utilisation en assembleur :

```
move.w    #'A',-(SP)      ; Lettre à afficher
move.w    #2,-(SP)        ; Code de la fonction Cconout
trap      #1              ; Appel de GEMDOS
addq.l    #4,SP
```

Exemple d'utilisation en C :

```
main ( )
{
   Cconout ('A');
}
```

#3 :
unsigned int Cauxin ( )

Cette fonction permet de lire un caractère sur l'entrée auxiliaire (généralement le port série). Le caractère lu est codé sur 8 bits.

Exemple d'utilisation en assembleur :

```
move.w    #3,-(SP)        ; Code de la fonction Cauxin
trap      #1              ; Appel de GEMDOS
addq.l    #2,SP
```

Exemple d'utilisation en C :

```
main ( )
{  unsigned int c;
   c=Cauxin ( );
}
```

#4 :
void Cauxout (c)
unsigned int c;

Cette fonction permet d'envoyer un caractère sur la sortie auxiliaire (généralement l'interface série).

Exemple d'utilisation en assembleur :

```
move.w    #'A',-(SP)      ; Lettre à envoyer
move.w    #4,-(SP)        ; Code de la fonction Cauxout
trap      #1              ; Appel de GEMDOS
addq.l    #4,SP
```

Exemple d'utilisation en C :

```
main ( )
{
   Cauxout ('A');
}
```

#5 :
void Cprnout (c)
unsigned int c;

Cette fonction permet d'envoyer un caractère sur la sortie imprimante (généralement l'interface parallèle).

Exemple d'utilisation en assembleur :

```
move.w    #'A',-(SP)      ; Lettre à envoyer
move.w    #5,-(SP)        ; Code de la fonction Cprnout
trap      #1              ; Appel de GEMDOS
addq.l    #4,SP
```

Exemple d'utilisation en C :

```
main ( )
{
   Cprnout ('A');
}
```

#6 :
unsigned long Crawio(c)
unsigned int c;

Cette fonction permet de lire ou d'écrire sur l'entrée/sortie standard (généralement la console/écran). Si le paramètre est différent de 0 × 00FF, la fonction le considérera comme le code ASCII du caractère à afficher. Sinon, la fonction renverra le résultat d'une lecture du clavier. Si un caractère est présent, son code sera renvoyé. Dans le cas contraire, la valeur zéro sera renvoyée.

Exemple d'utilisation en assembleur :

Vide Clavier :

```
move.w    #$FF,-(SP)      ; Demande de lecture clavier
move.w    #6,-(SP)        ; Code de la fonction Crawio
trap      #1              ; Appel de GEMDOS
addq.l    #4,SP
tst.l     D0              ; Caractère présent ?
bne.s     VideClavier     ; Oui, on recommence
move.w    #'*',-(SP)      ; Caractère à afficher
move.w    #6,-(SP)        ; Code de la fonction Crawio
trap      #1              ; Appel de GEMDOS
addq.l    #4,SP
```

Exemple d'utilisation en C :

```
main ( )
{
   /* Vide le buffer du clavier et affiche '*' */
```



```
    while (Crawio (0×FF)) ;
    Crawio ('*') ;
}
```

#7 :
long Crawin ( )

Cette fonction permet de lire l'entrée standard (généralement la console). Renvoie la valeur zéro si aucun caractère n'est présent ou le code du caractère s'il est présent. Aucun contrôle de caractère n'est fait.

Exemple d'utilisation en assembleur :

Vide Clavier :

```
move.w    #7,-(SP)      ; Code de la fonction Crawin
trap      #1            ; Appel de GEMDOS
addq.l    #2,SP
tst.l     D0
bne.s                   Vide Clavier
```

Exemple d'utilisation en C :

```
main ( )
{
   /* Vide le buffer clavier */
   while (Crawin ( ) );
}
```

#8 :
long Cnecin ( )

Cette fonction permet de lire un caractère sur l'entrée standard (généralement le clavier). Aucun écho n'est effectué et les caractères de contrôle sont interprétés.

Exemple d'utilisation en assembleur :

```
move.w    #8,-(SP)      ; Code de la fonction Cnecin
trap      #1            ; Appel de GEMDOS
addq.l    #2,SP
move.l    D0, Car
...
Car:dc.l  0
```

Exemple d'utilisation en C :

```
main ( )
{  unsigned long c;
   c=Cnecin ( );
}
```

#9 :
void Cconws (texte)
char *texte;

Cette fonction permet d'envoyer une chaîne de caractères vers la sortie standard (généralement l'écran). Cette fonction prend en compte les séquences VT52. La chaîne à afficher doit être aux normes C et donc se terminer par un zéro de fin de chaîne.

Exemple d'utilisation en assembleur :

```
pea       Texte          ; Texte à afficher
move.w    #9,-(SP)       ; Code de la fonction Cconws
trap      #1             ; Appel de GEMDOS
addq.l    #6,SP
...
```

Texte :

```
dc.b      $1B, 'E'       ; Efface l'écran
dc.b      "Bonjour", 0   ; Texte à afficher
```

Exemple d'utilisation en C :

```
main ( )
{
   /* Efface l'écran et affiche Bonjour */
   Cconws ("\33EBonjour");
}
```

#10 (0×0A) :
Cconrs

Cette fonction permet de saisir une chaîne de caractères sur l'entrée/sortie standard (généralement le clavier/écran). Il faut lui passer l'adresse d'un buffer. Le premier octet de ce buffer contiendra le nombre maximal de caractères à saisir (taille du buffer – 2). En sortie le deuxième octet contiendra le nombre de caractères effectivement lus. La chaîne se trouvera à partir du troisième octet. D'une manière générale, la chaîne ne se termine pas forcément par zéro. L'avantage de cette fonction réside dans le fait qu'elle gère les caractères de contrôle habituels :

RETURN ou CONTROL-J	:	Fin de ligne
BACKSPACE ou CONTROL-H	:	Efface le dernier caractère
CONTROL-U ou CONTROL-X	:	Efface la ligne
CONTROL-R	:	Recopie la ligne
CONTROL-C	:	Sort du programme

Exemple d'utilisation en assembleur :

```
pea       Prompt         ; Texte à afficher
move.w    #9,-(SP)       ; Code de Cconws
trap      #1             ; Appel de GEMDOS
addq.l    #6,SP
move.b    #78, Ligne     ; 78 caractères maxi
clr.b     Ligne+1        ; La ligne est vide
```



```
    pea       Ligne          ; Adresse de la ligne
    move.w    #$A,-(SP)      ; Code de Cconrs
    trap      #1             ; Appel de GEMDOS
    addq.l    #6,SP
    ...                      ; Traitement de la ligne
    .data
Prompt:
    dc.b      ">", 0
    .bss
Ligne:ds.b    80
    .end
```

Exemple d'utilisation en C :

```
main ( )
{
   static char *Prompt =">";      /* A afficher avant */
   char Ligne[80];                /* Ligne de saisie */
   Cconws (Prompt);               /* Affiche le prompt */
   Ligne [0]=78;                  /* 78 caractères maxi */
   Ligne [1]=0;                   /* La ligne est vide */
   Cconrs(Ligne);                 /* Saisie de la ligne */
   /* ... traitement de la ligne saisie */
}
```

#11 (0×0B) :
unsigned int Cconis ( )

Cette fonction teste l'état de l'entrée standard et renvoie 0×FFFF si un caractère est disponible et 0×000 dans le cas contraire.

Exemple d'utilisation en assembleur :

```
wait :

move.w    #$B,-(SP)      ; Code de Cconis
trap      #1             ; Appel de GEMDOS
addq.l    #2,SP
tst.w     D0             ; Caractère présent ?
beq.s     wait           ; Non, on boucle
```

Exemple d'utilisation en C :

```
main ( )
{

   /* Attend qu'un caractère soit disponible */

   while (! Cconis( ) );
}
```

 # 14 (0 × 0E) :
unsigned long Dsetdrv(drv);
int drv;

Cette fonction permet de sélectionner drv comme étant le lecteur actif. En retour, la fonction renvoie la carte des lecteurs présents sur l'Atari (bit 0 = lecteur A, bit 1 = lecteur B, ...).

Exemple d'utilisation en assembleur :

```
move.w    #1,-(SP)        ; Lecteur B
move.w    #$E,-(SP)       ; Code de Dsetdrv
trap      #1              ; Appel de GEMDOS
addq.l    #4,SP
move.l    D0,Carte        ; Sauvegarde de la carte
```

Exemple d'utilisation en C :

```
main ( )
{  unsigned long carte;
   carte=Dsetdrv (1); /* Sélection du lecteur B */
}
```

 # 16 (0 × 10) :
unsigned int Cconos ( )

Cette fonction teste l'état de la sortie standard qui est généralement l'écran. Si la console est prête à recevoir un caractère, la fonction renvoie 0 × FFFF sinon elle renvoie 0 × 000. D'une manière générale sur l'Atari, l'écran est toujours disponible et la fonction renvoie toujours 0 × FFFF.

Exemple d'utilisation en assembleur :

```
wait :

move.w    #$10,-(SP)      ; Code de Cconos
trap      #1              ; Appel de GEMDOS
addq.l    #2,SP
tst.w     D0              ; écran près ?
beq.s     wait            ; Non, on boucle
```

Exemple d'utilisation en C :

```
main ( )
{

   /* Attend que l'écran soit prêt */

   while (! Cconos( ) );
}
```



#17 (0×11) : unsigned int Cprnos ( )

Cette fonction est identique à Cconos mais teste l'état de la sortie standard PRN: qui est habituellement l'imprimante.

Exemple d'utilisation en assembleur :

```
wait :
move.w      #$11,-(SP)      ; Code de Cprnos
trap        #1              ; Appel de GEMDOS
addq.l      #2,SP
tst.w       D0              ; Imprimante prête ?
beq.s       wait            ; Non, on boucle
```

Exemple d'utilisation en C :

```
main ( )
{
   /* Attend que l'imprimante soit prête */
   while (! Cprnos( ) );
}
```

#18 (0×12) : unsigned int Cauxis ( )

Cette fonction est identique à Cconos mais teste l'état de l'entrée standard AUX: qui est habituellement l'interface série.

Exemple d'utilisation en assembleur :

```
wait :
move.w      #$12,-(SP)      ; Code de Cauxis
trap        #1              ; Appel de GEMDOS
addq.l      #2,SP
tst.w       D0              ; Série prête ?
beq.s       wait            ; Non, on boucle
```

Exemple d'utilisation en C :

```
main ( )
{
   /* Attend que l'interface série soit prête */
   while (! Cauxis( ) );
}
```

#19 (0×13) : Cauxos

Cette fonction est identique à Cconos mais teste l'état de la sortie standard AUX: qui est habituellement l'interface série.

Exemple d'utilisation en assembleur :

```
wait :

move.w    #$13,-(SP)      ; Code de Cauxos
trap      #1              ; Appel de GEMDOS
addq.l    #2,SP
tst.w     D0              ; Série prête ?
beq.s     wait            ; Non, on boucle
```

Exemple d'utilisation en C :

```
main ( )
{
   /*Attend que l'interface série soit prête */

   while (! Cauxos( ) );
}
```

#25 (0×19) : unsigned int Dgetdrv ( );

Cette fonction, à l'inverse de Dsetdrv, renvoie le numéro du lecteur courant.

Exemple d'utilisation en assembleur :

```
   move.w    #$19,-(SP)      ; Code de Dgetdrv
   trap      #1              ; Appel de GEMDOS
   addq.l    #2,SP
   move.w    D0, Drv
   ...
   .bss
Drv:
   ds.w      1
```

Exemple d'utilisation en C :

```
main ( )
{
   unsigned int Drv;
   Drv = Dgetdrv( );
}
```



#26 (0×1A) :
void Fsetdta(adr)
char *adr;

Cette fonction permet de définir l'adresse d'un tampon qui sera utilisé pour les fonctions Fsfirst et Fsnext. Ce tampon possède une taille de 44 octets. Nous en parlerons plus en détail dans les fonctions Fsfirst et Fsnext.

Exemple d'utilisation en assembleur :

```
   pea       Tampon       ; Adresse du tampon
   move.w    #$1A,-(SP)   ; Code de Fsetdta
   trap      #1           ; Appel de GEMDOS
   addq.l    #6,SP
   ...
   .bss
Tampon:
   ds.b      44
```

Exemple d'utilisation en C :

```
char Tampon [44] ;

main ( )
{
   Fsetdta(Tampon);
}
```

#32 (0×20) :
unsigned long Super (pile):
long pile;

Cette fonction permet de commuter le microprocesseur 68000 en mode utilisateur ou en mode superviseur. Rappelons que le mode superviseur permet d'accéder aux adresses système de l'Atari.

Le paramètre peut être -1L et, dans ce cas, la fonction renverra 0L si le microprocesseur est en mode utilisateur et 1L s'il est en mode superviseur. Si le paramètre est initialisé à une autre valeur, le microprocesseur changera de mode et renverra l'adresse de la pile superviseur avant l'appel. L'adresse de la pile superviseur est initialisée au paramètre pile si celui-ci est différent de zéro. S'il est nul, l'adresse de la pile superviseur est initialisée à l'adresse de la pile utilisateur. D'une manière générale, quand vous êtes en mode utilisateur et que vous désirez passer momentanément en mode superviseur, la meilleure solution est d'appeler la fonction Super, avec le paramètre à zéro, stocker le résultat dans une variable temporaire, réaliser ce que vous avez à faire et enfin appeler la fonction Super avec, comme paramètre, le contenu de la variable temporaire.

Exemple d'utilisation en assembleur :

```
pea       0.w          ; Pile super = Pile util.
```

```
move.w      #$20,-(SP)      ; Code de Super
trap        #1              ; Appel de GEMDOS
addq.l      #6,SP
move.l      D0,-(SP)        ; Sauve la pile superviseur dans la pile
                            ; puisqu'il faudra de toutes façons l'y
                            ; mettre ! Ici les instructions qui
                            ; nécessitent le mode superviseur
move.w      #$20,-(SP)      ; Code de Super
trap        #1              ; Appel de GEMDOS
addq.l      #6,SP
```

Exemple d'utilisation en C :

```
main ( )
{
   long AncienSSP;
   AncienSSP=Super (0L);
   /* Le code qui nécessite le code superviseur */
   Super(Ancien SSP);
}
```

#42 (0×2A) :
unsigned int Tgetdate ( )

Cette fonction permet de connaître la date stockée dans GEMDOS. Le format est le suivant :

bits 0 à 4 : jour entre 1 et 31
bits 5 à 8 : mois entre 1 et 12
bits 9 à 15 : année depuis 1980 entre 0 et 119

Exemple d'utilisation en assembleur :

```
; L'assembleur ne permettant pas de manipuler aussi facilement que
; le C, des variables telles que la date, nous ne verrons pas l'extrac-
; tion des champs.

   move.w      #$2A,-(SP)      ; Code de Dgetdate
   trap        #1              ; Appel de GEMDOS
   addq.l      #2,SP
   move.w      D0, date        ; Sauve le résultat
   ...
   .bss
date:
   ds.w        1
```

Exemple d'utilisation en C :

```
/* Cet exemple utilise la notion de champs de bits qui est très impor-
tante dans ce type de fonction
```



```
*/
typedef struct {
unsigned int jour:5; /* 5 bits pour le jour */
unsigned int mois:4; /* 4 bits pour le mois */
unsigned int annee:7; /* 7 bits pour l'année*/
} DOSTATE;
main ( )
{  DOSDATE date;
   date=Tgetdate ( );
   printf ("%d %d%d\n", date.jour, date.mois, date.annee+1980);
}
```

#43 (0×2B) :
int Tsetdate (date);
unsigned int date;

Cette fonction, à l'inverse de la précédente, permet de définir la date. Le format est identique. Renvoie zéro si la date est valide.

Exemple d'utilisation en assembleur :

```
   move.w     date,-(SP)        ; Date à affecter
   move.w     #$2B,-(SP)        ; Code de Dsetdate
   trap       #1                ; Appel de GEMDOS
   addq.l     #2,SP
   ...
   .bss
date:
   ds.w       1
```

Exemple d'utilisation en C :

```
typedef struct {
unsigned int jour:5;       /* 5 bits pour le jour */
unsigned int mois:4;       /* 4 bits pour le mois */
unsigned int annee:7;      /* 7 bits pour l'année */
} DOSTATE;
main ( )
{
   DOSDATE date;
   /* Définit la date au 20 février 1989 */
date.jour=20 ; date.mois=2 ; date.annee=1989-1980;
Tsetdate(date);
}
```

#44 (0×2C) :
unsigned int Tgettime ( );

Cette fonction permet de connaître l'heure stockée dans GEMDOS. Le format utilisé est le suivant :

bits 0 à 4 : secondes divisées par 2 entre 0 et 28

bits 5 à 10 : minutes entre 0 et 59
bits 11 à 15 : heure entre 0 et 23

Exemple d'utilisation en assembleur :

```
    move.w      #$2B,-(SP)          ; Code de Tgettime
    trap        #1                  ; Appel de GEMDOS
    addq.l      #2,SP
    move.w      D0,time             ; Stockage du résultat
    ...
    .bss
time:
    ds.w        1
```

Exemple d'utilisation en C :

```
type def struct {
unsigned int sec:5;
unsigned int min:6;
unsigned int heu:5;
} DOSTIME,
main ( )
{
    DOSTIME time;
    time=Tgettime ( );
    printf ("%d %d %d\n", time.heu, time.min, time.sec*2);
}
```

#45 (0×2D) :
int Tsettime (time)
unsigned int time;

Cette fonction permet de définir l'heure. Le format est le même que celui utilisé dans Tgettime. Renvoie 0 si l'heure est valide.

Exemple d'utilisation en assembleur :

```
    move.w      time,-(SP)          ; Heure à affecter
    move.w      #$2C,-(SP)          ; Code de Tgettime
    trap        #1                  ; Appel de GEMDOS
    addq.l      #2,SP
    ...
    .bss
time:
    ds.w        1
```

Exemple d'utilisation en C :

```
typedef struct {
unsigned int sec:5;
unsigned int min:6;
unsigned int heu:5;
} DOSTIME;
```



```
main( )
{
 DOSTIME time;

/* Définit l'heure à 17 h 12 mn 26 sec */

time.heu=17; time.min=12; time.sec=26/2;
Tsettime(time);
}
```

#47 (0×2F) :
char *Fgetdta( )

Cette fonction renvoie l'adresse du buffer de 44 octets utilisé par les fonctions Fsfirst et Fsnext. (Voir FSETDTA fonction #26)

Exemple d'utilisation en assembleur :

```
    move.w      #$2F,-(SP)      ; Code de Fgetdta
    trap        #1              ; Appel de GEMDOS
    addq.l      #2,SP
    move.l      D0,Tampon       ; Stockage du résultat
    .bss
Tampon:
    ds.l        1
```

Exemple d'utilisation en C :

```
main( )
{
    char *Tampon;
    Tampon=Fgetdta( );
}
```

#48 (0×30) :
unsigned int Sversion( );

Cette fonction renvoie le numéro de version de GEMDOS.

Exemple d'utilisation en assembleur :

```
    move.w      #$30,-(SP)      ; Code de Sversion
    trap        #1              ; Appel de GEMDOS
    addq.l      #2,SP
    move.w      D0,Ver          ; Stockage du résultat

    .bss
Ver:
    ds.w        1
```

Exemple d'utilisation en C :

```
main( )
{
   unsigned int Ver;
   Ver = Sversion( );
}
```

#49 (0×31) :
void Ptermres(L,code)
long L;
int code;

Cette fonction permet de terminer un programme en le laissant résidant. Le premier paramètre donne la taille de la zone qui, commençant à la base de la page, devra être conservée par le système. Les buffers alloués par l'application ne seront pas libérés. Le second paramètre représente la valeur qui devra être renvoyée au programme appelant.

Exemple d'utilisation en assembleur :

```
move.w    #-1,-(SP)        ; Code retour
pea       2000             ; Conserve 2000 octets
move.w    #$31,-(SP)       ; Code de Ptermres
trap      #1
```

Exemple d'utilisation en C :

```
main( )
{
   /* Reserve 2000 octets depuis la base page et renvoie -1 au pro-
      gramme appelant
   */
   Ptermres(2000L, -1);
}
```

#54 (0×36) :
void Dfree(buf,lect)
long *buf;
int lect;

Cette fonction permet de connaître la place libre sur un lecteur donné. Le premier paramètre représente l'adresse d'un buffer où GEMDOS placera les informations. Le second paramètre représente le numéro du lecteur à tester.



La structure du buffer est comme suit :

buffer : nombre de blocs disponibles

buffer+4 : nombre de blocs total sur le lecteur

buffer+8 : taille d'un secteur en octets

buffer+12 : taille d'un bloc en secteurs

Exemple d'utilisation en assembleur :

```
    clr.w       -(SP)               ; Lecteur A
    pea         buffer              ; Buffer de stockage
    move.w      #$36,-(SP)          ; Code de Dfree
    trap        #1                  ; Appel de GEMDOS
    addq.l      #8,SP
    lea         Buffer,A0           ; Adresse du buffer
    move.l      (A0),D0             ; Blocs libres
    mulu        8(A0),D0            ; Fois taille d'un secteur
    mulu        12(A0),D0           ; Fois taille d'un bloc
    move.l      D0,taille_libre

    .bss
buffer:
    ds.l        4
taille_libre:
    ds.l        1
```

Exemple d'utilisation en C :

```
typedef struct {
long total;
long libre;
long oct_p_sec;
lonc sec_p_blc;
} DFBLOC;
main( )
{
    DFBLOC buf;
    Dfree(&buf,0);      /* Teste le lecteur A */
    printf("Taille libre = %ld \n",
          buf.libre*buf.oct_p_sec*buf.sec_p_blc);
}
```

#57 (0×39) :
int Dcreate(chemin)
char *chemin;

Cette fonction permet de créer un répertoire. Le paramètre consiste en une chaîne de caractères décrivant ce même répertoire.

Renvoie 0 si la création a été effectuée correctement ou une erreur dans le cas contraire.

Exemple d'utilisation en assembleur :

```
   pea      Rep              ; Répertoire à créer
   move.w   #$39,-(SP)       ; Code de la fonction Ccreate
   trap     #1               ; Appel de GEMDOS
   addq.l   #6,SP
   ...
   .data
Rep:
   dc.b "BIN",0
```

Exemple d'utilisation en C :

```
main( )
{
   /* Création du répertoire BIN
   */
   Dcreate("BIN");
}
```

#58 (0×3A) :
int Ddelete(chemin)
char *chemin;

Cette fonction permet de détruire un répertoire.

Exemple d'utilisation en assembleur :

```
pea      Rep              ; Répertoire à supprimer
move.w   #$3A,-(SP)       ; Code de la fonction Ddelete
trap     #1               ; Appel de GEMDOS
addq.l   #6,SP
...
.data
Rep:dc.b "BIN",0
```

Exemple d'utilisation en C :

```
main( )
{
   /* Détruit le répertoire BIN
   */
   Delete("BIN");
}
```

#59 (0×3B) :
int Dsetpath(chemin)
char *chemin;

Fixe le répertoire courant. Si la chaîne commence par un nom de lecteur, ce chemin sera fixé pour ce même lecteur. Un répertoire courant



est spécifique à chaque lecteur. La fonction renvoie zéro si tout s'est bien passé.

Exemple d'utilisation en assembleur :

```
    pea       Rep              ; Nom du répertoire
    move.w    #$3B,-(SP)       ; Code de la fonction Dsetpath
    trap      #1               ; Appel de GEMDOS
    addq.l    #6,SP
    ...
    .data
Rep:
    dc.b "BIN",0
```

Exemple d'utilisation en C :

```
main( )
{
   /* Fixe le répertoire courant à BIN
   */
   Dsetpath("BIN");
}
```

#60 (0×3C) :
int Fcreate(Nom,att)
char *Nom;
int att;

Cette fonction permet de créer un fichier. Le premier paramètre indique le nom du fichier à créer. Le second paramètre représente les attributs qui devront être associés à ce fichier. Attention, si un fichier du même nom existe déjà, une erreur sera générée.

Les attributs sont codés sur chacun des bits du second paramètre :

bit 0 : fichier en lecture seulement

bit 1 : fichier caché

bit 2 : fichier système

bit 3 : nom de répertoire

La fonction renvoie l'identificateur du fichier si la création a été bien effectuée ou un code d'erreur dans le cas contraire.

Exemple d'utilisation en assembleur :

```
    clr.w     -(SP)            ; Fichier normal
    pea       Nom              ; Nom du fichier à créer
    move.w    #$3C,-(SP)       ; Code de la fonction Fcreate
    trap      #1               ; Appel de GEMDOS
    addq.l    #8,SP
    move.w    D0,Handle        ; Stockage du résultat
    ...
    .data
Nom:
```

```
   dc.b"ADRESSE.DAT",0
   .bss
Handle:
   ds.w 1
```

Exemple d'utilisation en C :

```
main( )
{
   int Hdl;      /* Handle du fichier */
   Hdl=Fcreate("ADRESSE.DAT",0);
}
```

#61 (0×3D) :
int Fopen(nom,mode)
char *nom;
int mode;

Cette fonction permet d'ouvrir un fichier existant. Il est possible d'ouvrir un fichier, en fonction du second paramètre, en lecture seulement (mode=0), en écriture seulement (mode=1) ou en lecture/écriture (mode=2).

La fonction renvoie un identificateur pour ce fichier si l'ouverture s'est effectuée correctement ou une erreur dans le cas contraire.

Exemple d'utilisation en assembleur :

```
   move.w    #2,-(SP)        ; Ouverture en lecture/écriture
   pea       Nom             ; Nom du fichier à ouvrir
   move.w    #$3D,-(SP)      ; Code de la fonction Fopen
   trap      #1              ; Appel de GEMDOS
   addq.l    #8,SP
   move.w    D0,Handle
   ...
   .bss
Handle:
   ds.w      1
```

Exemple d'utilisation en C :

```
main( )
{
   /* Ouvre le fichier ADRESSE.DAT en lecture/écriture
   */
   int Handle;
   Handle=Fopen("ADRESSE.DAT",2);
}
```

#62 (0×3E) :
int Fclose(handle)
int handle;

Cette fonction permet de fermer un fichier ouvert précédemment.

La fonction renvoie zéro si la fermeture s'est bien effectuée ou une erreur dans le cas contraire.



Exemple d'utilisation en assembleur :

```
    clr.w       -(SP)              ; Fichier normal
    pea         Nom                ; Nom du fichier
    move.w      #$3C,-(SP)         ; Code de Fcreate
    trap        #1                 ; Appel de GEMDOS
    addq.l      #8,SP
    move.w      D0,-(SP)           ; Handle du fichier
    move.w      #$3E,-(SP)         ; Code de Fclose
    trap        #1                 ; Code Fclose
    addq.l      #4,SP
    ...
    .data
Nom:
    dc.b "TOTO",0
```

Exemple d'utilisation en C :

```
main( )
{
    int handle;
    handle=Fcreate("TOTO",0);      /* Crée un fichier */
    Fclose(handle);                /* et le ferme     */
}
```

#63 (0×3F) :
long Fread(handle,longueur,buffer)
int handle;
long longueur;
char *buffer;

Cette fonction permet de lire une partie d'un fichier et de la mettre à un endroit de la mémoire. Le fichier doit avoir été ouvert en lecture ou en lecture/écriture.

La fonction renvoie le nombre d'octets lus ou zéro si plus rien n'est à lire dans le fichier. Il faut noter que GEMDOS conserve en mémoire la position d'un index, ce qui permet de lire ou d'écrire un fichier en plusieurs passages.

Exemple d'utilisation en assembleur :

```
    clr.w       -(SP)              ; Mode lecture
    pea         Nom                ; Nom du fichier à ouvrir
    move.w      #$3D,-(SP)         ; Code de la fonction Fopen
    trap        #1                 ; Appel de GEMDOS
    move.w      D0,D7              ; Sauve l'identificateur
    pea         buffer             ; Adresse de chargement
    pea         64.w               ; Longueur à lire
```

```
    move.w      D0,-(SP)        ; Handle du fichier
    move.w      #$3F,-(SP)      ; Code de la fonction Fread
    trap        #1              ; Appel de GEMDOS
    adda.w      #12,SP
    move.w      D7,-(SP)        ; Handle du fichier
    move.w      #$3E,-(SP)      ; Code de Fclose
    trap        #1
    addq.l      #4,SP
    ...
    .data
Nom:
    dc.b"TOTO",0
    .bss
buffer:
    ds.b        64
```

Exemple d'utilisation en C :

```
char Buffer[32];
main( )
{
    int Handle;
    handle=Fopen("TOTO",0);
    Fread(handle,64,Buffer);
    Fclose(handle);
}
```

#64 (0×40) :

```
long Fwrite(handle,longueur,buffer)
int handle;
long longueur;
char *buffer;
```

Cette fonction permet d'écrire une partie d'un fichier depuis un endroit de la mémoire. Le fichier doit avoir été ouvert en écriture ou en lecture/écriture.

La fonction renvoie le nombre d'octets écrits ou une erreur.

Exemple d'utilisation en assembleur :

```
    move.w      #1,-(SP)        ; Mode écriture
    pea         Nom             ; Nom du fichier à ouvrir
    move.w      #$3D,-(SP)      ; Code de la fonction Fopen
    trap        #1              ; Appel de GEMDOS
    move.w      D0,D7           ; Sauve l'identificateur
    pea         buffer          ; Adresse de chargement
    pea         64.w            ; Longueur à lire
    move.w      D0,-(SP)        ; Handle du fichier
```



```
    move.w      #$40,-(SP)      ; Code de la fonction Fwrite
    trap        #1              ; Appel de GEMDOS
    adda.w      #12,SP
    move.w      D7,-(SP)        ; Handle du fichier
    move.w      #$3E,-(SP)      ; Code de Fclose
    trap        #1
    addq.l      #4,SP
    ...
    .data
Nom:
    dc.b''TOTO'',0
    .bss
buffer:
    ds.b        64
```

Exemple d'utilisation en C :

```
char Buffer[32];
main( )
{
    int Handle;
    handle=Fopen(''TOTO'',0);
    Fwrite(handle,64,Buffer);
    Fclose(handle);
}
```

#65 (0×41) :
int Fdelete(Nom)
char *Nom;

Cette fonction efface le fichier dont le nom est donné en paramètre. La fonction retourne zéro en cas de succès et une erreur si le fichier n'existe pas ou s'il est verrouillé.

Exemple d'utilisation en assembleur :

```
pea         Nom             ; Nom du fichier à effacer
move.w      #$41,-(SP)      ; Code de Fdelete
trap        #1
addq.l      #6,SP
```

Exemple d'utilisation en C :

```
main( )
{
    Fdelete(''ADRESSE.DAT'');
}
```

#66 (0×42) :
```
long Fseek(pos,handle,mode)
long pos;
int handle;
int mode;
```

Cette fonction permet de se positionner dans un fichier. Pour cela, il est possible d'utiliser plusieurs modes différents. Le paramètre **handle** représente l'identificateur renvoyé lors de l'ouverture ou de la création du fichier. Le paramètre **mode** peut posséder plusieurs valeurs lui permettant différentes possibilités de positionnement.

mode = 0 : Le paramètre **pos** indique une position par rapport au début du fichier.

mode = 1 : Le paramètre **pos** correspond à un déplacement par rapport à la position courante dans le fichier.

mode = 2 : Le paramètre **pos** est relatif à la fin du fichier. La fonction renvoie zéro en cas de succès de l'opération et une erreur dans le cas contraire.

Exemple d'utilisation en assembleur :

```
    clr.w       -(SP)              ; Fichier en lecture
    pea         Nom                ; Nom du fichier à ouvrir
    move.w      #$3D,-(SP)         ; Code de Fopen
    trap        #1
    addq.l      #8,SP
    move.w      D0,handle
    move.w      #2,-(SP)           ; Rapport à la fin du fichier
    move.w      D0,-(SP)
    pea         100.w              ; 100 octets avant la fin
    move.w      #$42,-(SP)         ; Code de Fseek
    trap        #1
    addq.l      #6,SP
    pea         buffer             ; Buffer de chargement
    pea         100.w              ; 100 octets à charger
    move.w      handle,-(SP)       ; Identificateur du fichier
    move.w      #$3F,-(SP)         ; Code de Fread
    trap        #1
    adda.w      #12,SP
    move.w      handle,-(SP)
    move.w      #$3E,-(SP)         ; Code de Fclose
    trap        #1
    addq.l      #4,SP
    ...
    .data
Nom:
    dc.b''ADRESSE.DAT'',0
    .bss
handle:
```



```
    ds.w        1
buffer:
    ds.w        1
```

Exemple d'utilisation en C :

```
/* Ouvre un fichier ADRESSE.DAT et lit les 100 derniers octets dans un
   buffer
*/

main( )
{
    int handle,buffer[50];
    handle = Fopen("ADRESSE.DAT",0);
    Fseek(100L,handle,2);
    Fread(handle,100L,buffer);
    Fclose(handle);
}
```

#67 (0×43) :

```
int Fattrib(Nom,flag,attrib)
char *Nom;
int flag;
int attrib;
```

Cette fonction permet de modifier ou de lire les attributs d'un fichier dont le nom est donné par le paramètre **Nom**. Le paramètre indique le sens de la fonction. Si sa valeur est 1, la fonction modifiera les attributs tandis que si sa valeur est 0, la fonction lira les attributs. Les attributs d'un fichier sont répartis dans **attrib** comme suit :

bit 0 : fichier en lecture seulement

bit 1 : fichier caché

bit 2 : fichier système

bit 3 : fichier indiquant le nom du volume

bit 4 : fichier indiquant un sous-répertoire

bit 5 : fichier écrit puis refermé

La fonction renvoie, dans le cas de flag = 0, la valeur des attributs du fichier demandé.

Exemple d'utilisation en assembleur :

```
move.w      #$01,-(SP)      ; Lecture seule
move.w      #1,-(SP)        ; Modification attributs
pea         Nom             ; Nom du fichier
move.w      #$43,-(SP)      ; Code Fattrib
```

```
    trap        #1
    adda.w      #10,SP
    ...
Nom:
    dc.b "ADRESSE.DAT",0
```

Exemple d'utilisation en C :

```
main( )
{
    /* Configure le fichier ADRESSE.DAT en lecture seule */
    Fattrib("ADRESSE.DAT",1,0x01);
}
```

#69 (0×45) :
int Fdup(handle)
int handle;

Cette fonction permet de dupliquer un identificateur standard (compris entre 0 et 5). Elle retourne un identificateur non standard (supérieur ou égal à 6) qui peut être utilisé avec les fonctions de manipulation de fichiers (Fread, Fwrite,...). La fonction peut renvoyer une erreur si l'identificateur donné en paramètre n'est pas standard ou que le système ne dispose plus d'identificateurs non standards.

Exemple d'utilisation en assembleur :

```
    clr.w       -(SP)               ; Ecran
    move.w      #$45,-(SP)          ; Code de Fdup
    trap        #1
    addq.l      #4,SP
    pea         Texte               ; Texte à afficher
    move.w      #7,-(SP)            ; 7 caractères de longueur
    move.w      D0,-(SP)            ; Identificateur non standard
    move.w      #$40,-(SP)          ; Code de Fwrite
    trap        #1
    adda.w      #10,SP
    ...
    .data
Texte:
    dc.b        "BONJOUR",0
```

Exemple d'utilisation en C :

```
main( )
{
    int Ecran;
    Ecran=Fdup(0);                  /* Duplique l'écran */
    Fwrite(Ecran,7,"BONJOUR");      /* Affiche bonjour */
}
```



#70 (0×46) :
void Fforce(std,nonstd)
int std;
int nonstd;

Cette fonction permet de rediriger un identificateur standard, tel que l'écran ou l'imprimante, vers un identificateur non standard tel qu'un fichier.

Exemple d'utilisation en assembleur :

```
    move.w      #2,-(SP)            ; Standard de l'imprimante
    move.w      #$45,-(SP)          ; Code de Fdup
    trap        #1
    addq.l      #4,SP
    move.w      D0,-(SP)            ; Non-standard de l'imprimante
    clr.w       -(SP)               ; Standard de l'écran
    move.w      #$46,-(SP)          ; Code de Fforce
    trap        #1
    addq.l      #6,SP
    pea         Texte               ; Texte à « afficher »
    move.w      #9,-(SP)            ; Code de Cconws
    trap        #1
    addq.l      #6,SP
    ...
    .data
Texte:
    dc.b        "BONJOUR",13,10,0
```

Exemple d'utilisation en C :

```
main( )
{
    int Im;
    Imp=Fdup(2);
    Fforce(0,Imp);
    Cconws("BONJOUR \n");
}
```

#71 (0×47) :
void Dgetpath(buffer,lect)
char *buffer;
int lect;

Cette fonction permet de charger à l'adresse **buffer** le nom du répertoire courant pour le lecteur **lect**. Si **lect** est nul, le lecteur courant est utilisé.

Exemple d'utilisation en assembleur :

```
   move.w .    #1,-(SP)          ; Lecteur A
   pea         buffer            ; Adresse de lecture
   move.w      #$47,-(SP)        ; Code de la fonction Dgetpath
   trap        #1
   addq.l      #8,SP
   ...
   .bss
buffer:
   ds.b        64
```

Exemple d'utilisation en C :

```
main( )
{
   char buffer[64];
   Dgetpath(buffer,1);                /* Lecteur A */
}
```

#72 (0×48) :
long Malloc(taille)
long taille;

Cette fonction permet de demander un bloc de mémoire au GEMDOS. Il faut bien noter que c'est la seule façon correcte de s'allouer un bloc de la mémoire de l'Atari. Si le paramètre **taille** est égal à −1L, la fonction retourne la taille du plus grand bloc libre dont elle dispose. Toute autre valeur sera considérée comme la taille d'un bloc à allouer. Dans ce cas, l'adresse du premier bloc correspondant à cette taille sera retournée. Si la mémoire disponible dans le système est insuffisante, la fonction renverra la valeur 0L. Attention, un programme ne peut pas allouer plus de 20 blocs en même temps. Si vous devez allouer un grand nombre de zones, il est préférable d'allouer un grand bloc et de le découper par vous-même. Mais résistez à la tentation d'allouer toute la mémoire disponible et de vous considérer en terre conquise. Il peut arriver que plusieurs programmes coexistent en mémoire et se partagent l'espace libre (accessoires, switcher, etc.).

Exemple d'utilisation en assembleur :

```
   pea         -1.w              ; Demande la taille disponible
   move.w      #$48,-(SP)        ; Code de Malloc
   trap        #1
   addq.l      #6,SP
   move.l      D0,libre          ; Mémoire disponible
   move.w      #10000,D1         ; Taille du bloc à allouer
   cmp.l       D1,D0             ; Y-a assez ?
   ble.s       nomem             ; Non, on sort
   move.l      D1,-(SP)          ; Oui, on en prend un bout
   move.w      #$48,-(SP)        ; Code de Malloc
   trap        #1
   addq.l      #6,SP
   move.l      D0,bloc
```



```
    ...
libre:
    ds.l        1
bloc:
    ds.l        1
```

Exemple d'utilisation en C :

```
main( )
{
    long libre;
    char *bloc;
    libre=Malloc(-1L);
    if (libre>10000L) bloc=Malloc(10000L);
    else Cconws("Mémoire insuffisante \n");
}
```

#73 (0×49) :
int Mfree(bloc)
char *bloc;

Cette fonction permet de rendre au GEMDOS un bloc qui a été alloué auparavant par la fonction Malloc. La fonction renvoie zéro si la libération s'est bien effectuée et une erreur dans le cas contraire.

Exemple d'utilisation en assembleur :

```
pea         10000.w             ; Taille à allouer
move.w      #$48,-(SP)          ; Code de Malloc
trap        #1
addq.l      #6,SP
move.l      D0,-(SP)            ; Adresse du bloc à libérer
move.w      #$49,-(SP)          ; Code de Mfree
trap        #1
addq.l      #6,SP
```

Exemple d'utilisation en C :

```
main( )
{
    char *bloc;
    bloc=Malloc(10000L);
    Mfree(bloc);
}
```

#74 (0×4A) :
int Mshrink(0,bloc,taille)
char *bloc;
long taille;

Cette fonction, malheureusement assez rarement utilisée dans un programme, permet de réduire la taille d'un bloc déjà alloué. En fait, cette fonction est utilisée au moins une fois par tous les programmes dans l'entête pour indiquer au système la taille qu'il occupe en mémoire. En réalité, cette fonction est extrêmement puissante et peut être considérée comme un complément indispensable des fonctions Malloc et Mfree. Dans la fonction Malloc, nous avons vu qu'il fallait éviter d'allouer toute la mémoire disponible. En fait, avec l'aide de l'instruction **Mshrink**, on peut allouer toute la mémoire, travailler à l'intérieur pour déterminer de combien de place on a besoin, puis réduire la mémoire allouée à l'aide de **Mshrink**. Attention **Mshrink** peut réduire un bloc, mais ne peut pas l'agrandir. La fonction renvoie zéro si l'opération a réussi et une erreur dans le cas contraire.

Exemple d'utilisation en assembleur :

```
pea       20000.w          ; 20000 octets
move.w    #$48,-(SP)       ; Code de Malloc
trap      #1
addq.l    #6,SP
pea       10000.w          ; Réduit à 10000 octets
move.w    #$4A,-(SP)       ; Code de Mshrink
trap      #1
addq.l    #6,SP
```

Exemple d'utilisation en C :

```
main( )
{
   char *Bloc;
   Bloc=Malloc(20000L);              /* Alloue 2000 octets */
   Mshrink(0,Bloc,10000L);           /* Réduit à 10000 octets */
}
```

#75 (0×4B) :
void Pexec(mode,ptr1,ptr2,ptr3)
int mode;
char *ptr1;
char *ptr2;
char *ptr3;

Cette fonction permet différentes choses en fonction de la valeur de la variable mode :

- mode = 0 charge et lance un fichier exécutable



ptr1 pointe sur le nom d'un programme exécutable qu'il chargera et lancera.

ptr2 pointe sur une chaîne de caractères représentant les paramètres.

ptr3 pointe sur la chaîne d'environnement.

- mode = 3 charge un fichier exécutable sans le lancer

ptr1 pointe sur le nom d'un programme exécutable qu'il chargera et lancera.

ptr2 pointe sur une chaîne de caractères représentant les paramètres.

ptr3 pointe sur la chaîne d'environnement.

- mode = 4 exécute un programme déjà en mémoire

ptr1 pointe sur la **basepage** qui décrit l'emplacement du programme en mémoire. Cette basepage représente, pour GEMDOS, le seul moyen d'identifier un programme d'un autre. De plus c'est la base page qui permet à un programme de savoir qui l'a lancé.

ptr2 et ptr3 ne sont pas utilisés.

- mode = 5 crée une base page vierge

ptr1 n'est pas utilisé

ptr2 pointe sur une chaîne de caractères représentant les paramètres.

ptr3 pointe sur la chaîne d'environnement.

Exemple d'utilisation en assembleur :

```
    pea       0.w              ; Pas d'environnement
    pea       0.w              ; Pas de paramètres
    pea       Nom              ; Nom du programme
    move.w    #$4B,-(SP)       ; Code de Pexec
    trap      #1
    addq.l    #6,SP
    ...
Nom:
    dc.b"TOTO.PRG",0
```

Exemple d'utilisation en C :

```
main( )
{
    /* lance le programme TOTO.PRG sans paramètres et sans chaîne
       d'environnement
    */
    Pexec(0,"TOTO.PRG", " ", " ");
}
```

#76 (0×4C) :
void Pterm(code)
int code;

Cette fonction permet, exactement comme la fonction 0 (Pterm0), de terminer un programme. De plus, elle permet de renvoyer au programme appelant une valeur indiquant éventuellement comment s'est terminé le programme.

Exemple d'utilisation en assembleur :

```
move.w    #1,-(SP)        ; Code de retour
move.w    #$4C,-(SP)      ; Code de Pterm
trap      #1
```

Exemple d'utilisation en C :

```
main( )
{
   /* Sort du programme et renvoie la valeur 1 à l'appelant
   */
Pterm(1) ;
}
```

#78 (0×4E) :
int Fsfirst(spec,attrib)
char *spec;
int attrib;

Cette fonction permet de trouver, sur le disque, la première occurrence du fichier **spec**. Cette chaîne de spécification peut contenir les caractères joker **'*'** et **'?'** dans la partie nom de fichier mais non dans le chemin. Le paramètre **attrib** permet de choisir un type de fichier particulier. Pour les significations exactes de **attrib** voir la fonction Fattrib( ). La fonction renvoie zéro si une occurrence a été trouvée. Dans ce cas, il faut utiliser le buffer renvoyé par Fgetdta( ) pour avoir plus d'informations sur ce fichier. Ce buffer a une taille de 44 octets et est organisé ainsi :

octets 0 à 20 : réservés
octet 21 : bits d'attributs du fichier
octets 22 à 23 : heure de création du fichier
octets 24 à 25 : date de création du fichier
octets 26 à 29 : taille du fichier sur 32 bits
octets 30 à 43 : nom exact du fichier

Exemple d'utilisation en C :

L'exemple est commun à Fsnext.



#79 (0×4F) :
int Fsnext( )

Cette fonction complète la fonction Fsfirst( ) puisqu'elle renvoie les occurrences suivantes.

Exemple d'utilisation en assembleur :

```
    move.w      #$2F,-(SP)          ; Code de Fgetdta
    trap        #1
    addq.l      #6,SP
    move.l      D0,A5               ; A5 pointera sur le DTA
    clr.w       -(SP)               ; Tous les fichiers normaux
    pea         spec                ; Spécificité
    move.w      #$4E,-(SP)          ; Code de Fsfirst
    trap        #1
    addq.l      #8,SP
wt :            D0                  ; Encore des fichiers ?     tst.w
    bne         Fin                 ; Non, on sort
    pea         30(A5)              ; Adresse du nom dans DTA
    move.w      #9,-(SP)            ; Code de Cconws
    trap        #1
    addq.l      #6,SP
    pea         CRLF                ; On affiche un retour chariot
    move.w      #9,-(SP)            ; Code de Conws
    trap        #1
    addq.l      #6,SP
    move.w      #$4F,-(SP)          ; Code de Fsnext
    trap        #1
    addq.l      #2,SP
    bra         wt
Fin :
    ...
    .data
spec:
    dc.b        "*.PRG",0
CRLF:
    dc.b        13,10,0
```

Exemple d'utilisation en C :

Cet exemple balaye le répertoire courant et affiche le nom de chaque fichier portant l'extension PRG.

```
main( )
{
    char *buffer;
    int ret;
    buffer=Fgetdta( );
    ret=Fsfirst("*.PRG",0×00);
    while (ret=0)
    { printf("%s \n",buffer+30);
      ret=Fsnext( );
    }
}
```

#86 (0×56) :
int Frename(0,ancien,nouveau)
char *ancien;
char *nouveau;

Cette fonction permet de renommer un fichier.

Exemple d'utilisation en assembleur :

```
    pea         nouveau          ; Nouveau nom
    pea         ancien           ; Ancien nom
    clr.w       -(SP)
    move.w      #$56,-(SP)       ; Code de Frename
    trap        #1
    adda.w      #12,SP
    ...
nouveau:
    dc.b        "TITI.PRG",0
ancien:
    dc.b        "TOTO.PRG",0
```

Exemple d'utilisation en C :

```
main( )
{
    Frename(0,"TOTO.PRG","TITI.PRG");
}
```

#87 (0×57) :
void Fdatime(handle,heudat,flag)
int handle;
unsigned long heudat;
int flag;

Cette fonction permet de modifier (si flag est à 1) ou de lire (si flag est à 0), l'heure et la date de création du fichier.

Exemple d'utilisation en assembleur :

```
clr.w       -(SP)            ; Ouverture en lecture
pea         Nom              ; Nom du fichier
move.w      #$3D,-(SP)       ; Code de Fopen
trap        #1
addq.l      8,SP
move.w      D0, D7           ; Sauve l'ident. dans D7
clr.w       -(SP)            ; C'est une lecture
pea         heudate          ; Adresse de la variable
move.w      D7,-(SP)         ; Identificateur du fichier
```



```
move.w      #$57,-(SP)      ; Code de Fdatime
trap        #1
adda.w      #10,SP
move.w      #D7,-(SP)
move.w      #$3E,-(SP)      ; Code de Fclose
trap        #1
addq.l      #2,SP
```

Exemple d'utilisation en C :

```
main( )
{
   unsigned long heudat;
   int handle;
   Handle=Fopen(''TOTO.PRG'',0);
   Fdatime(handle,&heudat,0);
   Fclose(handle);
}
```



3/4.1

Utilisation des fonctions GEMDOS

3/4.1.1

Le GEMDOS, les piles et quelques ruses

But du programme

Ce programme en assembleur s'adresse tout aussi bien aux débutants qui ont bien assimilé les articles de remise à niveau en assembleur qu'aux programmeurs plus confirmés qui découvriront peut-être quelques tournures auxquelles ils n'avaient pas pensé ; sans compter tous ceux qui désirent approfondir un peu plus l'éventail des fonctions GEMDOS.
Le propos du programme est surtout un prétexte de programmation mais son utilisation est plus qu'évidente : sélectionner un fichier sur un chemin quelconque, le transformer et le sauvegarder sous sa nouvelle forme en effectuant tous les tests de conformité dans les opérations d'entrées sorties, choses qui devraient être faites systématiquement mais qu'on a tendance à oublier.

Analyse du programme

Le début teste la mémoire disponible et décide de l'occuper au maximum. Si vous utilisez un assembleur qui réserve toute la mémoire pour lui sans que vous la restreigniez, vous verrez apparaître le message **« Erreur d'allocation mémoire »** en lançant le programme directement après l'assemblage. Ce message disparaîtra dès que vous lancerez le programme depuis le bureau. Tous les messages sont affichés à l'étiquette **FIN** par **Cconws** (Gemdos 9) puis attente de touche (Gemdos 7). L'adresse **dta** est sauvegardée et l'écran effacé par ESC E.

Les deux appels qui suivent se font sans régénérer la pile après appel du trap et en plaçant les paramètres 2(sp), ce qui économise de la place et fait gagner du temps, sans oublier l'élégance de la programmation.

Le nom de source sur 80 octets est sélectionné et sauvegardé avant de tester longueur et attribut du fichier dans le buffer d'information.

Ce buffer est rempli automatiquement dès que s'opère la sélection d'un fichier. Si le fichier n'existe pas on affiche le message correspondant et on va à **FIN**.
Le test suivant permet de tester l'attribut du fichier (on ne lira qu'un fichier normal en lecture et/ou écriture). Attention à supprimer le bit d'archivage avant ce test : c'est le rôle de l'instruction **andi.b$1f,d0**.

La longueur est testée ici (arbitrairement) et on décide de ne traiter que les fichiers de longueur inférieure à $400 et supérieure à 0.

Revoyez la définition du buffer d'informations du DTA si besoin est !

Le nom est correct, le fichier a les conditions requises pour le traitement, on peut ensuite ouvrir le fichier source en lecture dès que l'ouverture s'est effectuée sans problème (n'oubliez pas de sauvegarder le handle d'un fichier dès qu'il est ouvert ou créé et que vous désirez effectuer des opérations).

Par précaution, on efface la zone non initialisée de travail **($100 longs = $400 octets)** et les octets lus sont placés et transformés dans la mémoire de travail par le sous-programme LIRE.

Il ne reste plus qu'à sélectionner un nom de fichier non existant pour sauver le résultat du traitement, puisque nous décidons ici de ne pas écraser de fichier résidant. Dans ce cas là, on pourrait aussi créer un back-up. Notez que tous les chemins sont acceptables dans ce programme, pas uniquement des sélections de fichiers simples sur disque A comme **A:\ FICH.AZE**, mais aussi

C:\ ESSAI\ EXOWEKA\ EX__1.ASS.
Dès que l'on a vérifié la non-existence du fichier destination, on peut le créer (sauver son handle) et le sauver sur disquette (ou ailleurs suivant le chemin défini précédemment) par la routine **ECRIRE**.
Il ne reste plus qu'à fermer tous les fichiers ouverts et rendre la main au système.

Routine LIRE

L'adresse de la zone de stockage des données est empilée puis le fichier **source** est lu. Sa longueur est mémorisée. Le traitement s'effectue sur la totalité du fichier, c'est une mise à un systématique des bits 2, 3, 4, 6. Vous pouvez imaginer et créer toutes les transformations que vous voulez à cet endroit du programme **(LIRE1)**.

Routine ECRIRE

La même adresse de stockage et la même longueur sont empilées avant de sauver, mais vous pouvez raccourcir la longueur du fichier **destination** ou ne pas altérer tous les octets du source. On revient au programme principal si l'écriture s'est effectuée sans erreur.



```
                .DATA

err_blk:        .DC.b $1b,$45," Erreur d'allocation mémoire",$0
effecr:         .DC.b $1b,$45,$7,$0
src_msg:        .DC.b  $a,$d,"        Fichier Source : ",$0
dest_msg:       .DC.b  $a,$d,"        Fichier Destination: ",$0
err_src:        .DC.b $a,$d," Ce fichier n'existe pas !",$0
err_dest:       .DC.b $a,$d," Un fichier porte deja ce nom !",$0
end_msg:        .DC.b $a,$d," Travail terminé !",$7,$a,$d,$0
no_src:         .DC.b $a,$d," pas de fichier de ce nom !",$a,$d,$0
bad_src:        .DC.b $a,$d," pas un fichier autorisé!",$a,$d,$0
src_err:        .DC.b $a,$d," Le fichier source non ouvert",$a,$d,$0
bad_len1:       .DC.b $a,$d," Ce fichier a 0 octets",$a,$d,$0
bad_len:        .DC.b $a,$d," fichier de plus de 1024 octets",$a,$d,$0
errlect:        .DC.b $a,$d," Erreur lecture source",$a,$d,$0
errcreat:       .DC.b $a,$d," Erreur création  destination",$a,$d,$0
errecr:
                .DC.b $a,$d," Erreur écriture dest",$a,$d,$0
fin_trait:
                .DC.b $a,$d," Fin de traitement",$a,$d,$0

                .EVEN

old_stk:        .DC.l $00000000
dta:            .DC.l $00000000
BUFADR:         .DC.l $00000000
B_LEN:          .DC.l $00040000
C_LEN:          .DC.l $00020000
S_LEN:          .DC.l $00000000
I_LEN:          .DC.l $00000000
R_LEN:          .DC.l $00000000
D_LEN:          .DC.l $00000000
src_hdl:        .DC.w $0000
dest_hdl:
                .DC.w $0000

                .BSS
Zone:           .DS.l $100
src_nam:        .DS.b $84
dest_nom:       .DS.b $84
                .EVEN
                .TEXT

                .DC.w $a00a
; Pas de curseur souris.
                movea.l     $4(sp),a0
; Recherche de la memoire disponible
                move.l      sp,old_stk
                move.l      sp,d0
; Calcul de la ram totale.
                swap.w      d0
                move.l      B_LEN,d1
```

(suite)

```
                move.l      C_LEN,d2
                lsr.w       #3,d0
                beq         RAM1
        ; 512 K-Octets.
                lsl.l       #1,d1
                lsl.l       #1,d2
                lsr.w       #1,d0
                beq         RAM0
        ; 1 M-Octets.
                lsl.l       #1,d1
                lsl.l       #1,d2
                lsr.w       #1,d0
                beq         RAM0
        ; 2 M-Octets.
                lsl.l       #1,d1
        ; 4 M-Octets.
                lsl.l       #1,d2
        RAM0:   move.l      d1,B_LEN
                move.l      d2,C_LEN
        RAM1:   move.l      $c(a0),d0
                add.l       $14(a0),d0
                add.l       $1c(a0),d0
                addi.l      #$200,d0
                move.l      d0,d1
                add.l       a0,d1
                andi.w      #$fffe,d1
                movea.l     d1,sp
                move.l      d1,BUFADR
                add.l       B_LEN,d0
                add.l       C_LEN,d0
                addq.l      #4,d0
                move.l      d0,-(sp)
                move.l      a0,-(sp)
                move.l      #$4a0000,-(sp)
                trap        #1
                adda.l      #$c,sp
                tst.l       d0
                beq         suite
                move.l      #err_blk,-(sp)
        ; Probleme d'allocation mémoire.
                bra         FIN
        suite:  move.w      #$2f,-(sp)
        ; Sauvegarde du buffer d'info
                trap        #1
                addq.l      #2,sp
                move.l      d0,dta
                move.l      #effecr,-(sp)
        ; Ecran propre !
                move.w      #$9,-(sp)
                trap        #1
                addq.l      #6,sp
```



(suite)

```
;-----------nom de source
                move.l      #src_msg,-(sp)
                move.w      #$9,-(sp)
                trap        #1
;---------------80 octets max
                movea.l     #src_nam,a5
                move.b      #$50,(a5)
                move.l      a5,$2(sp)
                move.w      #$a,(sp)
                trap        #1
                addq.l      #6,sp
;---------------nom terminé par un $00
                move.b      #$0,$2(a5,d0.w)
                addq.l      #2,a5
                clr.w       -(sp)
                move.l      a5,-(sp)
                move.w      #$4e,-(sp)
                trap        #1
                adda.l      #$8,sp
                tst.w       d0
                beq         FND_SRC
                move.l      #no_src,-(sp)
                bra         FIN
; sauvegarde du dta et test de l'attribut
FND_SRC:movea.l     dta,a0
                move.b      $15(a0),d0
                andi.b      #$1f,d0
                cmpi.b      #$2,d0
                blt         OK_SRC
                move.l      #bad_src,-(sp)
; Pas un fichier autorisé on quitte
                bra         FIN

OK_SRC:  move.l     $1a(a0),d0
; Longueur du fichier source.
                cmpi.l      #$400,d0
                ble         OK_LEN1
                move.l      #bad_len,-(sp)
                bra         FIN
OK_LEN1:cmpi.l      #$0,d0
                bpl         OK_LEN
                move.l      #bad_len1,-(sp)
                bra         FIN
OK_LEN:  move.l     d0,S_LEN

; On ouvre le fichier source en lecture
                clr.w       -(sp)
                move.l      #src_nam,d0
                addq.l      #2,d0
                move.l      d0,-(sp)
                move.w      #$003d,-(sp)
```

(suite)

```
                trap        #1
                adda.l      #$00000008,sp
                tst.w       d0
        ; Ouverture correcte ?
                bpl         SRC_OPEN
                move.l      #src_err,-(sp)
        ; Non, erreur on quitte.
                bra         FIN

        SRC_OPEN: move.w    d0,src_hdl
        ; Sauvegarde du handle source

                lea.l       Zone,a0
        ; Effacer la Zone non initialisée.
                move.w      #$100,d0
        INIT:   clr.l       (a0)+
                dbf         d0,INIT

                bsr         LIRE

                move.l      #dest_msg,-(sp)
        ; On demande le nom du fichier destination
                move.w      #$9,-(sp)
                trap        #1
                movea.l     #dest_nom,a5
                move.b      #$50,(a5)
        ; 80 caracteres maximum.
                move.l      a5,$2(sp)
                move.w      #$a,(sp)
                trap        #1
                addq.l      #6,sp
                move.b      #$0,$2(a5,d0.w)
        ; nom terminé par un #$00
                addq.l      #2,a5
                move.w      #$0,-(sp)
                move.l      a5,-(sp)
                move.w      #$4e,-(sp)
                trap        #1
                adda.l      #$8,sp
                tst.w       d0
        ; Existe-t-il ?
                beq         ERRFND
                move.w      #$10,-(sp)
        ; Verification de la non presence du trajet destination
                move.l      a5,-(sp)
                move.w      #$4e,-(sp)
                trap        #1
                adda.l      #$8,sp
                tst.w       d0
        ; Existe-t-il ?
                bne         CRT_dest
```



(suite)

```
ERRFND: move.l      #err_dest,-(sp)
; Oui, message d'erreur.
        bra         FIN
; On quitte.


CRT_dest:           ; Créer le fichier destination


        move.l      #dest_nom,d0
        addq.l      #2,d0
        clr.w       -(sp)
; Création du fichier destination.
        move.l      d0,-(sp)
        move.w      #$003c,-(sp)
        trap        #1
; Execution !
        addq.l      #8,sp
        tst.w       d0
; Créé correctement ?
        bpl         CRT_OK
; Oui.
        move.l      #errcreat,-(sp)
; Non, erreur signalée,
        bra         FIN
; on quitte.
CRT_OK: move.w      d0,dest_hdl

        bsr         ECRIRE

;fermer tous les fichiers
FERME:
        move.w      src_hdl,-(sp)
        move.w      #$3e,-(sp)
        trap        #1
        adda.l      #$4,sp
        move.w      dest_hdl,-(sp)
        move.w      #$3e,-(sp)
        trap        #1
        adda.l      #$4,sp
        move.l      #fin_trait,-(sp)

FIN:
        move.w      #$9,-(sp)
        trap        #1
        move.w      #$7,-(sp)
        trap        #1
        movea.l     old_stk,sp
        move.w      #$4c,-(sp)
        trap        #1
```

(suite)

```
LIRE:   move.l   S_LEN,d1
        pea      Zone
        move.l   d1,-(sp)          ; Nombre d'octets à lire.
        move.w   src_hdl,-(sp)     ; Handle du fichier source.
        move.w   #$003f,-(sp)      ; Commande de lecture.
        trap     #1
        adda.l   #$c,sp            ; Correction de la pile.
        cmp.l    d0,d1             ; Nombre d'octets lus correct ?
        beq      LIRE1             ; Oui.
        move.l   #errlect,-(sp)    ; Non, erreur signalée,
        bra      FIN               ; on quitte.
LIRE1:
        move.w   d1,d2
        movea.l  #Zone,a0
LIRE2:  ori.b    #$5a,$0(a0,d1.w)
; ici vous pouvez transformer
; vos données comme vous le voulez
        dbf      d1,LIRE2
        rts

ECRIRE:
        move.l   S_LEN,d3
        pea      Zone              ; buffer travail
        move.l   d3,-(sp)          ; Nombre d'octets à sauvegarder
        move.w   dest_hdl,-(sp)    ; Handle du fichier tarjet.
        move.w   #$40,-(sp)        ; Commande d'écriture.
        trap     #1                ; Execution !
        adda.l   #$c,sp            ; Correction de la pile.
        cmp.l    d0,d3             ; Sauvegarde correcte ?
        beq      FINECR            ; Oui.
        move.l   #errecr,-(sp)     ; Non, erreur signalée,
        bra      FIN               ; on quitte.
FINECR:
        rts

        .END
```

3/5

L'AES

L'AES *(Application Environnement System)* représente dans GEM le module qui prend en charge l'interface utilisateur. Ainsi, il met à la disposition du programmeur tout un ensemble de fonctions destinées à faciliter la gestion des dialogues entre le programme et l'utilisateur.

3/5.1

La description des fonctions de l'AES

I - Les fonctions de contrôle des applications

#10 (0×0A) :
int appl_init( ) ;

Cette fonction permet à l'application de se faire connaître par l'AES. Cette opération est indispensable pour tout programme utilisant l'AES et doit être appelée dès le début du programme.

Données en entrée :

CONTRL[0] = 10 ;

Données en sortie :

INT_OUT[0] = Contient l'identificateur retourné par la fonction. Dans le cas d'une erreur, la fonction renvoie −1. De plus, l'identificateur est placé dans le tableau GLOBAL, ce qui permet à l'AES d'identifier une application à chaque appel d'une fonction.

```
#11 (0×0B) :
int appl_read(ap_id, ap_long, ap_buf);
int ap_id ;       /* Identificateur de l'application */
int ap_long ;     /* Nombre d'octets à lire          */
int *ap_buf ;     /* Zone de stockage                */
```

Cette fonction permet de lire un message en provenance de l'AES ou d'une autre application. Elle est utilisée dans le cas où un message de taille non standard est reçu. Nous en reparlerons plus loin dans la gestion des événements.

Données en entrée :

CONTRL[0] = 11

INT_IN[0] = ap_id.

INT_IN[1] = Nombre d'octets à lire.

ADDRIN[0] = Buffer où l'AES stockera les octets lus.

Données en sortie :

INT_OUT = 0 si une erreur s'est produite
différent de 0 s'il n'y a pas d'erreur

```
#12 (0×0C) :
int appl_write(ap_id, ap_long, ap_buf);
int ap_id ;       /* Identificateur de l'application */
int ap_long ;     /* Nombre d'octets à lire          */
int *ap_buf ;     /* Zone de stockage                */
```

Cette fonction permet d'écrire un message à destination d'une autre application. Cela permet à plusieurs programmes de communiquer. Nous en reparlerons plus loin dans la gestion des événements.

Données en entrée :

CONTRL[0] = 12



INT_IN[0] = ap_id.

INT_IN[1] = Nombre d'octets à écrire.

ADDRIN[0] = Buffer contenant les octets à envoyer.

Données en sortie :

INT_OUT = 0 si une erreur s'est produite
différent de 0 s'il n'y a pas d'erreurs.

```
#13 (0×0D) :
int appl_find(nom);
char *nom;      /* Nom de l'application à chercher */
```

Cette fonction permet de retrouver un programme utilisant l'AES. C'est la phase essentielle lorsque l'on désire que plusieurs programmes communiquent entre eux. Le nom à passer se compose du nom du programme sans l'extension et complété par des espaces pour arriver à une longueur de 8 caractères. Pour les bricoleurs, notons que AES peut être adressé sous le nom de SCRENMGR. En cherchant un peu, il doit être possible de provoquer certaines réactions intéressantes dans l'AES.

Données en entrée :

CONTRL[0] = 13

ADDRIN[0] = Nom, c'est un pointeur sur la chaîne de caractères définissant le nom de l'application à retrouver.

Données en sortie :

INT_OUT[0] = Identificateur de l'application recherchée. La fonction renvoie −1 si l'application n'existe pas.

```
#14 (0×0E) :
int appl_tplay(buffer, nbre, vitesse);
int *buffer ;     /* Buffer contenant les actions     */
int nbre ;        /* Nombre d'actions à reproduire */
int vitesse ;     /* Vitesse de reproduction          */
```

Cette fonction permet d'exécuter un certain nombre d'actions enregistrées par la fonction appl_trecord( ). Il est possible de programmer la vitesse à l'aide du paramètre vitesse. Une valeur de 100 correspond à la vitesse d'origine. Avec une valeur de 200, la reproduction se ferait à 2 fois la vitesse d'origine.

Une action est composée de deux parties. La première partie se com-

pose d'un mot de 16 bits et représente le type d'action (0 : timer, 1 : bouton, 2 : souris, 3 : clavier). La seconde partie occupe 4 octets qui dépendent du type d'action :

— Action 0 (timer) : Les 4 octets sont enregistrés comme un mot long qui représente le temps à attendre.

— Action 1 (bouton) : Le premier mot court représente l'état des boutons (bit 0 : bouton gauche, bit 1 : bouton droit) sachant que le bit à 1 signifie bouton appuyé.

— Action 2 (souris) : Le premier mot représente la coordonnée X et le second représente la coordonnée Y.

— Action 3 (clavier) : Le premier mot représente le code de la touche (octet haut : scan code, octet bas : état des touches **SHIFT/CTRL/ALT/CAPS**).

Données en entrée :

CONTRL[0] = 14

INT_IN[0] = Nombre d'actions dans le buffer.

INT_IN[1] = Vitesse de reproduction.

ADDRIN[0] = Adresse du buffer contenant les actions.

Données en sortie :

INT_OUT[0] = 1

#15 (0×0F) :

```
int appl_trecord(buffer, nbre);
int *buffer ;     /* Buffer d'enregistrement          */
int nbre ;        /* Nombre d'actions à enregistrer   */
```

Cette fonction, inversement à la fonction appl_tplay( ), permet d'enregistrer une série d'actions. En sortie, le buffer contiendra les actions enregistrées sous la forme décrite dans la fonction appl_tplay( ).

Données en entrée :

CONTRL[0] = 15

INT_IN[0] = Nombre d'actions à enregistrer.

ADDRIN[1] = Adresse du tampon où seront stockées les actions.

Données en sortie :

INT_OUT[0] = Nombre d'actions enregistrées.



#19 (0×13) :
int appl_exit( ) ;

Cette fonction renseigne AES sur la fin de l'application.

Données en entrée :

CONTRL[0] = 19

Données en sortie :

INT_OUT[0] = 0 indique qu'une erreur s'est produite
différent de zéro s'il n'y a pas d'erreur.

II - Les fonctions de contrôle des événements

L'utilisation de l'AES amène la majorité des programmeurs à changer leurs habitudes de programmation. En effet, un programme AES est organisé autour d'un noyau de détection d'un ou plusieurs événements. En fonction de celui-ci, le programme s'orientera vers une partie traitant cet événement. La méthode présente plusieurs avantages : tout d'abord, l'AES offre la possibilité d'attendre plusieurs événements à la fois. Il est possible d'indiquer à l'AES que l'application attend une pression sur une touche, un délai de 10 secondes ou que la souris sorte d'une zone de l'écran. De plus, c'est un moyen d'indiquer à l'AES que l'application est en attente et qu'elle peut passer la main à une autre application. C'est le cas pour les accessoires qui utilisent la possibilité de AES d'être multitâche. Malheureusement, il n'y a pas d'autre possibilité de le programmer.

Il existe 5 types d'événements :

— Les deux premiers sont très simples, car il s'agit de la pression sur une touche du clavier pour le premier, de la pression sur un des boutons de la souris pour le deuxième.

— Le troisième permet d'être prévenu lorsque la souris entre ou sort d'une zone rectangulaire de l'écran.

— Le quatrième est du type délai car il permet d'attendre un délai fixé.

— Le dernier type est bien plus complexe car il concerne la réception de message depuis l'AES ou une autre application. D'une manière générale, ce message est de longueur variable comprise entre 4 et 8 mots :

— mot 0 : type de message,

— mot 1 : identification de l'application (renvoyé par appl_init),

— mot 2 : nombre d'octets supplémentaires (au-delà de 16),

— mot 3 à 7 : contenu du message.

Selon le type du message, le contenu sera différent. Il existe 13 types de messages pouvant être reçus de l'AES :

MN_SELECTED (10)
Sélection d'une entrée dans un menu. L'utilisateur a cliqué sur un des choix du menu :
— mot 3 : numéro du titre dans l'arbre du menu.
— mot 4 : numéro de l'élément sélectionné.

WM_REDRAW (20)
Demande de rafraîchissement d'une fenêtre.
— mot 3 : identificateur de la fenêtre,
— mot 4 : coordonnée X de la partie à réafficher,
— mot 5 : coordonnée Y de la partie à réafficher,
— mot 6 : largeur à réafficher,
— mot 7 : hauteur à réafficher,

WM_TOPPED (21) :
Demande d'activation d'une fenêtre. L'utilisateur a cliqué sur le contenu d'une fenêtre non active.
— mot 3 : identificateur de la fenêtre.

WM_CLOSED (22) :
Demande de fermeture d'une fenêtre. L'utilisateur a cliqué sur coin supérieur gauche d'une fenêtre active.
— mot 3 : identificateur de la fenêtre.

WM_FULLED (23) :
Demande d'agrandissement d'une fenêtre. L'utilisateur a cliqué sur le coin supérieur droit d'une fenêtre active.
— mot 3 : identificateur de la fenêtre.

WM_ARROWED (24) :
Gestion des flèches et ascenseurs. L'utilisateur a cliqué sur l'un d'eux.
— mot 3 : identificateur de la fenêtre.
— mot 4 : action demandée :
0 : page vers le haut
1 : page vers le bas
2 : ligne vers le haut
3 : ligne vers le bas
4 : page à gauche



5 : page à droite

6 : colonne à gauche

7 : colonne à droite

WM_HSLID (25) :

L'utilisateur a déplacé l'ascenseur horizontal d'une fenêtre.

— mot 3 : identificateur de la fenêtre.

— mot 4 : valeur comprise entre 0 et 1000 indiquant la nouvelle position de l'ascenseur. 0 indique la position extrême à gauche et 1000 celle de droite.

WM_VSLID (26) :

L'utilisateur a déplacé l'ascenseur horizontal d'une fenêtre.

— mot 3 : identificateur de la fenêtre.

— mot 4 : valeur comprise entre 0 et 1000 indiquant la nouvelle position de l'ascenseur. 0 indique la position extrême en haut et 1000 celle du bas.

WM_SIZED (27) :

L'utilisateur a modifié la taille d'une fenêtre active.

— mot 3 : identificateur de la fenêtre,

— mot 4 : coordonnée X de la fenêtre,

— mot 5 : coordonnée Y de la fenêtre,

— mot 6 : largeur de la fenêtre,

— mot 7 : hauteur de la fenêtre.

WM_MOVED (28) :

L'utilisateur a déplacé la fenêtre active.

— mot 3 : identificateur de la fenêtre,

— mot 4 : coordonnée X de la fenêtre,

— mot 5 : coordonnée Y de la fenêtre,

— mot 6 : largeur de la fenêtre,

— mot 7 : hauteur de la fenêtre.

WM_NEWTOP (29) :

Indique à l'application qu'une nouvelle fenêtre est active.

— mot 3 : identificateur de la fenêtre.

AC_OPEN (30) :

Indique à un accessoire de bureau qu'il est sélectionné.

— mot 3 : identificateur de l'entrée dans le menu accessoires. C'est la valeur renvoyée par la fonction menu_register( ).

AC_CLOSE (31) :

Indique à un accessoire qu'il doit se fermer. Cela arrive lorsque le bureau lance une application. L'AES envoie alors un message de ce type à tous les accessoires.

— mot 3 : identificateur de l'entrée dans le menu accessoires. C'est la valeur renvoyée par la fonction menu_register ( ).

#20 (0×14) :
int evnt_keybd( ) ;

Cette fonction renvoie une touche saisie au clavier.

Données en entrée :

CONTRL[0] = 20

Données en sortie :

INT_OUT[0] = Code de la touche détectée. Le code ASCII se trouve dans l'octet bas et le scan-code se trouve dans l'octet haut.

#21 (0×15) :

```
int evnt_button(ev_bclic, ev_bmask, ev_bstate, ev_bmx, ev_bmy, ev_bbutton,
ev_bkstate);
int ev_bclic ;              /* Nombre de clics attendus   */
int ev_bmask ;              /* Boutons attendus           */
int ev_bstate ;             /* Action attendue            */
int *ev_bmx, *ev_bmy ;      /* Position de la souris      */
int *ev_bbutton ;           /* Etat des boutons           */
int *ev_bkstate ;           /* Etat des touches spéciales */
```

Cette fonction permet d'attendre une pression sur une des touches de la souris. De plus, la fonction renvoie le nombre de pressions, la touche pressée, la position de la souris et l'état des touches spéciales du clavier.

Données en entrée :

CONTRL[0] = 21



INT_IN[0] = ev_bclic. Nombre de clics attendus.

INT_IN[1] = ev_bmask. Touches sélectionnées. Chaque touche est représentée par un bit. Le bit 0 correspond à la touche de gauche et le bit 1 correspond à la touche de droite.

INT_IN[2] = ev_bstate. Action attendue. 0 correspond à un relâchement et 1 correspond à une pression.

Données en sortie :

INT_OUT[0] = Nombre de pressions détectées.

INT_OUT[1] = Coordonnée X de la souris.

INT_OUT[2] = Coordonnée Y de la souris.

INT_OUT[3] = Etat du bouton lors de l'événement.

INT_OUT[4] = Etat des touches spéciales du clavier
bit 0 : SHIFT droit
bit 1 : SHIFT gauche
bit 2 : CONTROL
bit 3 : ALTERNATE
Le bit correspondant est à 1 si la touche est appuyée.

#22 (0×16) :

```
int  evnt_mouse(ev_moflags,ev_mox,ev_moy,ev_mowidth,ev_moheight,ev_
momx,ev_momy,ev_mobutton,ev_mokstate) ;
int ev_moflags ;              /* Type de détection          */
int ev_mox,ev_moy ;           /* Rectangle limite           */
ev_mowidth,ev_moheight ;
int *ev_momx ;*ev_momy ;      /* Coordonnées de la souris   */
int *ev_mobutton ;            /* Etat des boutons           */
int *ev_mokstate ;            /* Etat des touches spéciales */
```

Cette fonction permet d'attendre que la souris rentre dans une zone rectangulaire de l'écran, ou qu'elle en sorte.

Données en entrée :

CONTRL[0] = 22

INT_IN[0] = Type de la détection. 0 signifie que l'on attend une rentrée dans le rectangle, et 1 correspond à une sortie.

INT_IN[1] = Coordonnée X du rectangle.

INT_IN[2] = Coordonnée Y du rectangle.

INT_IN[3] = Largeur du rectangle.

INT_IN[4] = Hauteur du rectangle.

Données en sortie :

INT_OUT[0] = 1

INT_OUT[1] = Coordonnée X de la souris.

INT_OUT[2] = Coordonnée Y de la souris.

INT_OUT[3] = Etat des boutons.

INT_OUT[4] = Etat des touches spéciales du clavier

bit 0 : SHIFT droit

bit 1 : SHIFT gauche

bit 2 : CONTROL

bit 3 : ALTERNATE

Le bit correspondant est à 1 si la touche est appuyée.

```
#23 (0×17) :
int evnt_mesag(ev_msg);
int*ev_msg ;     /* Buffer de réception du message */
```

Cette fonction permet d'attendre un message de l'AES.

Données en entrée :

CONTRL[0] = 23

ADDRIN[0] = ev_msg. C'est l'adresse du buffer qui sera utilisé pour recevoir le message.

Données en sortie :

INT_OUT[0] = 1

```
#24 (0×18) :
int evnt_timer(ev_tlocount, ev_thicount);
int ev_tlocount ;     /* Mot bas du délai à attendre   */
int ev_thicount ;     /* Mot haut du délai à attendre  */
```

Cette fonction permet d'attendre un certain délai.

Données en entrée :

CONTRL[0] = 24



INT_IN[0] = Partie basse du délai à attendre en millisecondes.

INT_IN[1] = Partie haute du délai à attendre en millisecondes.

Données en sortie :

INT_OUT[0] = 1

```
#25 (0×19) :
int event_multi(ev_mflags, ev_mbclic, ev_mbmask, ev_mbstate, ev_mm1flags,
ev_mm1x, ev_mm1y, ev_mm1width, mm1height, ev_mm2flags, ev_mm2x,
ev_mm2y, ev_mm2width, mm2height, ev_mmsg, ev_mtlocount, ev_mthicount,
ev_mmox, ev_mmoy, ev_mmobutton, ev_mmokstate, ev_mkreturn, ev_mbreturn) ;
int ev_mflags ;     /* Masque des événements attendus */
int ev_mbclic ;
int ev_mbmask ;
int ev_mbstate ;
int ev_mm1flags ;
int ev_mm1x,ev_mm1y,ev_mm1width,mm1height ;
int ev_mm2flags ;
int ev_mm2x,ev_mm2y,ev_mm2width,mm2height ;
int *ev_mmsg ;
int ev_mtlocount, ev_mthicount ;
int *ev_mmox,*ev_mmoy ;
int *ev_mmobutton ;
int *ev_mmokstate ;
int *ev_mkreturn ;
int *ev_mbreturn ;
```

Cette fonction extrêmement impressionnante permet de combiner les différents types d'événements.

Données en entrée :

CONTRL[0] 25

INT_IN[0] = ev_mflags.
Indique le type d'événements attendus :

bit 0 : événement clavier

bit 1 : événement bouton

bit 2 : événement souris 1er rectangle

bit 3 : événement souris 2e rectangle

bit 4 : événement message

bit 5 : événement timer

INT_IN[1] = ev_mbclic. Nombre de clics attendus.

INT_IN[2] = ev_mbmask. Boutons autorisés.

INT_IN[3] = ev_mbstate. Type d'action attendue. 0 correspond à un relâchement et 1 correspond à une pression.

INT_IN[4] = ev_mm1flags. Action correspondant au 1er rectangle. 0 pour une rentrée, 1 pour une sortie.

INT_IN[5] à INT_IN[8] = coordonnées du premier rectangle.

INT_IN[9] = ev_mm2flags. Action correspondant au 2e rectangle. 0 pour une rentrée, 1 pour une sortie.

INT_IN[10] à INT_IN[13] = Coordonnées du second rectangle.

INT_IN[14] = Partie basse du délai à attendre.

INT_IN[15] = Partie haute du délai à attendre.

ADDRIN[0] = Adresse du buffer de réception des messages.

Données en sortie :

INT_OUT[0] = Evénements détectés. Sa structure est identique à ev_mflags passé dans INTIN[0].

INT_OUT[1] = Coordonnée X de la souris.

INT_OUT[2] = Coordonnée Y de la souris.

INT_OUT[3] = Etat des boutons.

INT_OUT[4] = Etat des touches spéciales (voir fonction evnt_button).

INT_OUT[5] = Touche détectée (voir fonction evnt_keybd).

INT_OUT[6] = Nombre de pressions détectées.

#26 (0×1A) :

```
int evnt_dclick(vitesse,flag);
int vitesse ;     /* Vitesse du double clic */
int flag ;        /* Lecture/modification   */
```

Cette fonction permet de lire ou de modifier l'intervalle de temps définissant le double clic. Si flag est égal à 0, la fonction renvoie la vitesse actuelle. Si flag est égal à 1, la vitesse est actualisée à la valeur du paramètre vitesse qui est compris entre 0 et 4. 0 représente la vitesse la plus faible et 4 la vitesse la plus rapide.

Données en entrée :

CONTRL[0] = 26



INT_IN[0] = Vitesse. N'est pris en compte que si INTIN[1] (flag) est à 1. Doit être compris entre 0 et 4.

INT_IN[1] = Flag. 0 indique une lecture, 1 indique une écriture.

Données en sortie :

INT_OUT[0] = Contient la vitesse actuelle.

III - Les fonctions de gestion des menus

GEM, l'environnement graphique de l'Atari ST, permet de gérer des menus déroulants qui procurent à l'utilisateur un fantastique confort dans l'utilisation d'un programme. Ces menus sont constitués d'une ligne de texte située en haut de l'écran. Cette ligne contient un certain nombre d'entrées qui, en se déroulant sous la souris, permettent d'accéder à un nombre de choix supplémentaires. AES permet de programmer ces menus à l'aide de quelques fonctions de base. Du point de vue du programmeur, ces menus peuvent être créés à l'aide d'un éditeur de ressources tel que celui fourni par ATARI dans son système de développement et intégré dans un fichier ressource (.RSC). Il existe d'autres éditeurs de ressources tels que K-Resource de KUMA ou WERCS de HiSoft. L'application prend en charge, avec la fonction rsrc_load, le chargement du fichier ressource contenant la description du menu.

```
#30 (0x1E) :
int menu_bar(me_btree,me_bshow);
int *me_btree ;     /* Adresse de l'arbre*/
int me_bshow ;      /* Montrer/cacher    */
```

Cette fonction permet d'afficher (me_bshow=1) ou de cacher (me_bshow=0) la barre de menu. Le paramètre me_btree contient l'adresse du menu. Cette adresse est obtenue à l'aide de la fonction rsrc_gaddr( ).

Données en entrée :

CONTRL[0] = 30

INT_IN[0] = me_bshow (0 : effacer, 1 : afficher).

ADDRIN[0] = me_btree. Adresse de l'arbre décrivant le menu.

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, toute autre valeur dans le cas contraire.

```
#31 (0×1F) :
int menu_icheck(me_ctree,me_citem,me_ccheck);
int *me_ctree ;     /* Adresse du menu      */
int *me_citem ;     /* Numéro de l'entrée   */
int me_ccheck ;     /* Affiche/efface       */
```

Cette fonction permet de marquer ou démarquer une entrée du menu. Cette marque se place au début de l'entrée et ressemble à un V.

Données en entrée :

CONTRL[0] = 31

INT_IN[0] = me_citem. Numéro de l'entrée à marquer ou à démarquer.

INT_IN[1] = me_ccheck. 0 pour démarquer, 1 pour marquer.

ADDRIN[0] = me_ctree. Adresse de l'arbre décrivant le menu.

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, toute autre valeur dans le cas contraire.

```
#32 (0×20) :
int menu_ienable(me_etree,me_eitem,me_eenable);
int *me_etree ;     /* Adresse du menu      */
int me_eitem ;      /* Numéro de l'entrée   */
int me_eenable ;    /* Activation/désactivation */
```

Cette fonction valide ou invalide une entrée du menu. Une entrée désactivée est composée de caractères clairs et n'est pas sélectionnée lorsque la souris passe par-dessus.

Données en entrée :

CONTRL[0] = 32

INT_IN[0] = me_eitem. Numéro de l'entrée à activer ou désactiver.

INT_IN[1] = me_eenable. 0 pour désactiver, 1 pour activer.

ADDRIN[0] = me_etree. Adresse de l'arbre décrivant le menu.

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, toute autre valeur dans le cas contraire.



```
#33 (0x21) :
int menu_tnormal(me_ntree,me_ntitle,me_nnormal);
int *me_ntree ;     /* Adresse du menu   */
int me_ntitle ;     /* Numéro du titre   */
int me_nnormal ;    /* Allumer/éteindre  */
```

Cette fonction allume ou éteint un titre du menu. Cette fonction est couramment utilisée lorsque l'application reçoit un message de type MN_SELECTED. Il faut éteindre manuellement l'entrée du menu qui a été sélectionnée.

Données en entrée :

CONTRL[0] = 33

INT_IN[0] = me_ntitle. Numéro du titre à allumer ou éteindre.

INT_IN[1] = me_nnormal. 0 pour éteindre, 1 pour allumer.

ADDRIN[0] = me_ntree. Adresse de l'arbre décrivant le menu.

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, toute autre valeur dans le cas contraire.

```
#34 (0x22) :
int menu_text(me_ttree,me_titem,me_ttext);
int *me_ttree ;     /* Adresse du menu   */
int me_titem ;      /* Numéro de l'entrée */
char *me_ttext ;    /* texte à installer  */
```

Cette fonction modifie le texte contenu dans une entrée du menu. Il ne faut pas oublier que le texte contenu dans le menu avant la modification doit être au moins aussi long que le texte que l'on installe. La fonction utilise un déplacement de chaînes plutôt qu'un déplacement de pointeurs.

Données en entrée :

CONTRL[0] = 34

INT_IN[0] = me_titem. Numéro du titre à modifier.

ADDRIN[0] = me_ntree. Adresse de l'arbre décrivant le menu.

ADDRIN[1] = me_ttext. Adresse de la chaîne de caractères à installer.

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, toute autre valeur dans le cas contraire.

```
#35 (0×23) :
int menu_register(me_rapid,me_rpstring);
int me_rapid ;          /* ID de l'application   */
char *me_rpstring ;     /* Nom de l'accessoire*/
```

Cette fonction installe un accessoire dans le menu réservé aux accessoires. La fonction renvoie une valeur indiquant la ligne où se trouvera ce texte. C'est très utile quand un accessoire utilise plusieurs entrées possibles.

Données en entrée :

CONTRL[0] = 35

INT_IN[0] = me_rapid. Identificateur de l'application. C'est la valeur renvoyée par appl_init( ).

ADDRIN[0] = me_rpstring. C'est le nom qui sera affiché dans le menu.

Données en sortie :

INT_OUT[0] = Numéro de l'accessoire dans le menu.

IV - Les fonctions de gestion des objets

L'AES utilise la notion d'objet et d'arborescence pour manipuler des formulaires et des menus. Nous verrons plus loin comment est organisé un formulaire.

```
#40 (0×28) :
int objc_add(ob_atree,ob_aparent,ob_achild);
int *ob_atree;      /* Adresse de l'arbre */
int ob_aparent ;    /* Objet père         */
int ob_achild ;     /* Objet fils         */
```

Cette fonction ajoute un objet à une famille. Il faut indiquer l'adresse de l'arbre à modifier, le numéro de l'objet père et le numéro de l'objet fils. Cette fonction ne fait que modifier les pointeurs d'en-tête du père et du fils.

Données en entrée :

CONTRL[0] = 40

INT_IN[0] = Objet père.

INT_IN[1] = Objet fils.

ADDRIN[0] = Adresse de l'arbre décrivant le formulaire.



Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

```
#41 (0×39) :
int objc_delete(ob_dltree,ob_dlobject);
int *ob_dltree ;      /* Adresse de l'arbre  */
int ob_dlobject ;     /* Object en supprimer*/
```

Cette fonction effectue la suppression d'un objet d'un arbre. En fait de suppression, cette fonction modifie les pointeurs de l'objet en question pour qu'il ne soit lié à aucun père. L'objet n'est pas supprimé et peut être relié à un père par la fonction objc_add( ).

Données en entrée :

CONTRL[0] = 41

INT_IN[0] = ob_dlobject. Objet à supprimer.

ADDRIN[0] = ob_dltree. Adresse de l'arbre contenant l'objet à supprimer.

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

```
#42 (0×3A) :
int  objc_draw(ob_drtree,ob_drstartob,ob_drdepth,ob_drxclip,ob_dryclip,ob_
drwclip,ob_drhclip);
int *ob_drtree ;      /* Adresse de l'arbre  */
int ob_drstartob ;    /* Objet de départ     */
int ob_drdepth ;      /* Profondeur          */
int ob_drxclip,ob_dryclip,ob_drwclip,ob_drhclip ;
```

Cette fonction permet d'afficher tout ou une partie d'un arbre à l'écran. Elle permet de définir à partir de quel objet on doit dessiner et sur combien de générations. Par exemple, une valeur égale à 1 limitera le retracé à l'objet de départ et ses enfants exclusivement. De plus il est possible de limiter le retracé à un rectangle de restriction.

Données en entrée :

CONTRL[0] = 42

INT_IN[0] = ob_drstart. Objet de départ.

INT_IN[1] = ob_drdepth. Profondeur de dessin.

INT_IN[2] = ob_drxclip. Coordonnée X du rectangle de restriction.
INT_IN[3] = ob_dryclip. Coordonnée Y du rectangle de rectriction.
INT_IN[4] = ob_drwclip. Largeur du rectangle de restriction.
INT_IN[5] = ob_drhclip. Hauteur du rectangle de restriction.
ADDRIN[0] = ob_drtree. Adresse de l'arbre contenant les objets à afficher.

Données en sortie :
INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

```
#43 (0×3B) :
int objc_find(ob_ftree,ob_fstartob,ob_fdepth,ob_fmx,ob_fmy);
int *ob_ftree ;          /* Adresse de l'arbre */
int ob_fstartob ;        /* Objet de départ    */
int ob_fdepth ;          /* Profondeur         */
int ob_fmx,ob_fmy ;      /* Position à tester  */
```

Cette fonction permet de connaître l'objet qui se trouve à une position de l'écran. D'une manière générale, on utilise cette fonction après la détection d'un événement de type clic bouton.

Données en entrée :
CONTRL[0] = 43
INT_IN[0] = ob_fstart. Objet de départ.
INT_IN[1] = ob_fdepth. Profondeur de recherche.
INT_IN[2] = ob_fmx. Coordonnée X à tester.
INT_IN[3] = ob_fmy. Coordonnée Y à tester.
ADDRIN[0] = ob_ftree. Adresse de l'arbre dans lequel se fera la recherche.

Données en sortie :
INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

```
#44 (0×3C) :
int objc_offset(ob_oftree,ob_ofobject,ob_ofx,ob_ofy);
int *ob_oftree ;         /* Adresse de l'arbre  */
int ob_ofobject ;        /* Objet à calculer    */
int *ob_ofx,*ob_ofy ;    /* Variables de retour */
```

Cette fonction permet de connaître la position d'un objet par rapport à l'écran.



Données en entrée :

CONTRL[0] = 44

INT_IN[0] = ob_ofobject. Objet de départ.

INT_IN[2] = ob_ofx. Coordonnée X à tester.

INT_IN[3] = ob_ofy. Coordonnée Y à tester.

ADDRIN[0] = ob_oftree. Adresse de l'arbre dans lequel se trouve l'objet.

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

```
#45 (0×2D) :
int objc_order(ob_ortree,ob_orobject,ob_ornewpos);
int *ob_ortree ;      /* Adresse de l'arbre  */
int ob_orobject ;     /* Objet à modifier    */
int ob_ornewpos ;     /* Nouvelle position   */
```

Cette fonction permet de modifier la position d'un objet par rapport à ses frères. Pratiquement, cela correspond à indiquer si un objet doit être affiché avant ou après un autre objet. C'est une fonction très importante lorsque l'on programme un environnement de type bureau. Lorsqu'un pictogramme du bureau est sélectionné, il doit être réaffiché en vidéo inverse mais en dernière position. Cette fonction permet de l'indiquer.

Données en entrée :

CONTRL[0] = 45

INT_IN[0] = ob_orobject. Objet que l'on désire déplacer.

INT_IN[1] = ob_ornewpos. Nouvelle position.
0 = en dernière position
1 = avant-dernière position
...
-1 = première position

ADDRIN[0] = Adresse de l'arbre.

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

```
#46 (0×2E) :
int objc_edit(ob_edtree,ob_edobject,ob_edchar,ob_edidx,ob_edkind,ob_ednewidx);
int *ob_edtree ;        /* Adresse de l'arbre  */
int ob_edobject ;       /* Objet à éditer      */
int ob_edchar ;         /* Caractère à insérer */
int ob_edidx ;          /* Position du curseur */
int ob_edkind ;         /* Type d'action       */
int *ob_ednewidx ;      /* Curseur après ins.  */
```

Cette fonction permet de modifier le texte contenu dans un objet de type G_TEXT ou G_BOXTEXT. Le paramètre ob_edkind permet de moduler l'effet de cette fonction :

0 : Valeur réservée.

1 : Insère le caractère dans la chaîne et affiche le curseur.

2 : Compare avec la chaîne de validation, modifie le texte et affiche.

3 : Efface le curseur.

Données en entrée :

CONTRL[0] = 46

INT_IN[0] = ob_edobject. Objet à modifier.

INT_IN[1] = ob_edchar. Caractère à insérer.

INT_IN[2] = ob_edidx. Position du curseur avant édition.

INT_IN[3] = ob_edkind. Fonction à réaliser (0 à 3).

ADDRIN[0] = ob_edtree. Adresse de l'arbre.

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

INT_OUT[1] = Position du curseur après édition.

```
#47 (0×3F) :
int  objc_change(ob_ctree,ob_cobject, ob_cresvd, ob_cxclip, ob_cyclip, ob_cwclip,
ob_chclip, ob_cnewstate, ob_credraw);
int *ob_drtree ;        /* Adresse de l'arbre  */
int ob_cobject ;        /* Objet à modifier    */
int ob_cresvd ;         /* Doit être à 0       */
int ob_drxclip,ob_dryclip,ob_drwclip,ob_drhclip ;
int ob_cnewstate ;      /* Nouvel état de l'objet */
int ob_credraw ;        /* Drapeau de retracé  */
```

Cette fonction permet de modifier l'état d'un objet. Le paramètre ob_



credraw permet d'indiquer si l'on désire retracer l'objet dans le nouvel état. De plus il est possible de limiter le tracé à un rectangle de restriction.

Données en entrée :

CONTRL[0] =47

INT_IN[0] = ob_cobject. Objet à modifier.

INT_IN[1] = ob_cresvd. Réservé et doit être nul.

INT_IN[2] = ob_cxclip. Coordonnée X du rectangle de restriction.

INT_IN[3] = ob_cyclip. Coordonnée Y du rectangle de restriction.

INT_IN[4] = ob_cwclip. Largeur du rectangle de restriction.

INT_IN[5] = ob_chclip. Hauteur du rectangle de restriction.

INT_IN[6] = ob_cnewstate. Nouvel état de l'objet.

INT_IN[7] = ob_credraw. 1 si on désire un retracé de l'objet, 0 dans le cas contraire.

ADDRIN[0] = ob_drtree. Adresse de l'arbre contenant l'objet à modifier.

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

V - Les fonctions de gestion des formulaires

En plus des menus, l'AES propose au programmeur un autre moyen pour dialoguer avec l'utilisateur. Ce sont les formulaires que nous avons déjà abordés lors de la gestion des objets. Comme un menu, un formulaire peut être conçu avec l'aide d'un éditeur de ressources. Le formulaire est un moyen très pratique de concevoir une interface utilisateur simple sans que la programmation en soit trop compliquée. En général, un formulaire est inclus dans un fichier ressource et doit être chargé avec la fonction rsrc_load( ). L'adresse du formulaire peut être, comme pour un menu, obtenue avec la fonction rsrc_gaddr( ). A partir de ce point, le formulaire est presque prêt à être utilisé. Il faut tout d'abord déterminer sa position à l'écran, de manière à ce qu'il soit centré. Lors de son utilisation, il suffira de l'afficher avec la fonction objc_draw( ) et de le lancer avec la fonction form_do( ). L'AES prend en charge tous les problèmes de gestion d'objets jusqu'à ce que l'utilisateur clique sur un objet de type **EXIT**. Alors l'AES rendra la main à l'utilisateur pour que celui-ci analyse le formulaire et détermine quels boutons ont été modifiés. Il s'agit là du minimum à réaliser pour utiliser un formulaire. En fait, une autre fonction est couramment utilisée. Il s'agit de la fonction form_dial( ).

```
#50 (0×32) :
int form_do(fo_dotree,fo_dostart);
int *fo_dotree ;
int fo_dostart ;
```

Cette fonction permet d'exécuter un formulaire. L'AES contrôlera celui-ci tant que l'utilisateur ne demandera pas à sortir du formulaire en cliquant sur un bouton de type **EXIT**. Le paramètre fo_dostart indique sur quel objet devra être placé le curseur texte. S'il n'y a pas d'objet éditable, ce paramètre doit être égal à 0. La fonction renvoie le numéro de l'objet avec lequel l'utilisateur est sorti.

Données en entrée :

CONTRL[0] = 50

INT_IN[0] = fo_dostart (1er objet éditable. Zéro s'il n'y a aucun objet éditable).

ADDRIN[0] = fo_dotree (adresse de l'arbre).

Données en sortie :

INT_OUT[0] = Numéro de l'objet ayant provoqué la sortie du formulaire.

```
#51 (0×33) :
int form_dial(fo_diflag,fo_dixmin,fo_diymin,fo_diwmin,fo_dihmin,fo_dixmax,fo_
diymax,fo_diwmax,fo_dihmax);
int fo_diflag ;     /* type d'action 0 à 3 */
int fo_dixmin,fo_diymin,fo_diwmin,fo_dihmin ;
int fo_dixmax,fo_diymax,fo_diwmax,fo_dihmax ;
```

Cette fonction permet de réaliser plusieurs actions différentes en fonction du paramètre fo_diflag :

0 : Réserve une partie de l'écran pour un formulaire.

1 : Dessine une boîte qui s'agrandit pour passer de la taille minimale passée en paramètres jusqu'à la taille maximale également passée en paramètre.

2 : Dessine une boîte qui rétrécit pour passer de la taille maximale passée en paramètre jusqu'à la taille minimale également passée en paramètre.

3 : Libère la partie de l'écran qui a été réservée lors de l'appel de form_dial avec la valeur 0. Cela a pour effet de provoquer un retracé du bureau et des fenêtres qui étaient dans la zone réservée.



Données en entrée :

CONTRL[0] = 51

INT_IN[0] = fo_diflag. Variable spécifiant le type d'action.

INT_IN[1] = fo_dixmin.

INT_IN[2] = fo_diymin.

INT_IN[3] = fo_diwmin.

INT_IN[4] = fo_dihmin.

INT_IN[5] = fo_dixmax.

INT_IN[6] = fo_diymax.

INT_IN[7] = fo_diwmax.

INT_IN[8] = fo_dihmax.

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

```
#52 (0×34) :
int form_alert(fo_adefbttn,fo_astring);
int fo_adefbttn ;
char *fo_astring ;
```

Cette fonction affiche un message dans une boîte d'alerte. C'est un moyen très simple d'attirer l'attention de l'utilisateur. Une boîte d'alerte est constituée d'un icône représentatif du type de message, du texte dont la taille peut atteindre 5 lignes de 40 caractères et de la description des boutons permettant la sortie.

Le paramètre fo_adefbttn indique à AES quel est le bouton qui sera choisi par défaut si l'utilisateur appuie sur **RETURN**. Une valeur de 0 indique qu'aucun bouton ne sera pris par défaut, 1 pour le 1er bouton, 2 pour le second.

Le paramètre fo_astring contient une chaîne descriptive de l'alerte dont le format est le suivant.

[n] [texte1 | texte2 | texte3] [bouton1 | bouton2]

— n indique quel type de pictogramme sera affiché dans le coin supérieur gauche de l'alerte :

0 : pas de pictogramme
1 : pictogramme NOTE
2 : pictogramme WAIT
3 : pictogramme STOP

— texte1, texte2 et texte3 représentent 3 lignes du message à afficher. Les barres verticales sont significatives du retour à la ligne.

— bouton1 et bouton2 sont les deux boutons qui seront affichés. Remarquez qu'ils sont également séparés par des barres verticales.

La fonction renvoie le numéro du bouton utilisé pour sortir. 1 : premier bouton, 2 : second bouton, etc.

Données en entrée :

CONTRL[0] = 52

INT_IN[0] = fo_adefbttn. Bouton pris par défaut.

ADDRIN[0] = fo_astring. Chaîne de caractères décrivant l'alerte.

Données en sortie :

INT_OUT[0] = Bouton de sortie.

#53 (0×35) :
```
int form_error (fo_enum) ;
int fo_enum;
```

Cette fonction affiche un message standard du système à partir de son numéro. Ces numéros ne sont pas très documentés et doivent être découverts par la recherche. Nous y reviendrons.

Données en entrée :

CONTRL[0] = 53

INT_IN[0] = fo_enum. Numéro de l'erreur.

Données en sortie :

INT_OUT[0] = Bouton utilisé pour sortir. Voir form_alert.

#54 (0×36) :
```
int form_center (fo_ctree, fo_cx, fo_cy, fo_cw, fo_ch) ;
int *fo_ctree     /* Adresse de l'arbre*/
int *fo_cx, *fo_cy, *fo_cw, *fo_ch ;
```

Cette fonction permet de centrer un formulaire dans l'écran.

Données en entrée :

CONTRL[0] = 54

ADDRIN[0] = fo_ctree. Adresse de l'arbre à centrer.



Données en sortie :

INT_OUT[0] = 1

INT_OUT[1] = Coordonnée X de l'objet centré.

INT_OUT[2] = Coordonnée Y de l'objet centré.

INT_OUT[3] = Largeur de l'objet centré.

INT_OUT[4] = Hauteur de l'objet centré.

VI - Les fonctions graphiques

Cette ensemble de fonctions consiste en quelques fonctions graphiques simples ainsi que celles qui sont liées au VDI.

```
#70 (0×46) :
int graf_rubberbox (gr_rx, gr_ry, gr_rminwidth,
gr_rminheight, gr_rlastwidth, gr_rlastheight) ;
int gr_rx, gr_ry ;      /* Position de la boîte  */
int gr_rminwidth;       /* Largeur minimale      */
int gr_rminheight ;     /* Hauteur minimale      */
int *gr_rlastwidth ;    /* Largeur finale        */
int *gr_rlastheight ;   /* Hauteur finale        */
```

Cette fonction permet de dessiner une boîte élastique et modifiable par son coin inférieur droit.

Données en entrée :

CONTRL[0] = 70

INT_IN[0] = gr_rx. Coordonnée X du coin supérieur gauche.

INT_IN[1] = gr_ry. Coordonnée Y du coin supérieur gauche.

INT_IN[2] = gr_rminwidth. Largeur minimale.

INT_IN[3] = gr_rminheight. Hauteur minimale.

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

INT_OUT[1] = Largeur finale.

INT_OUT[2] = Hauteur finale.

```
#71 (0×47) :
int graf_dragbox(gr_dwidth, gr_dheight, gr_dstartx,
gr_dstarty, gr_dboundx, gr_dboundy, gr_dboundw,
gr_dboundh, gr_dfinishx, gr_dfinishy) ;
int gr_dwidth, gr_dheight ;          /* Taille rectangle   */
int gr_dstartx, gr_dstarty ;         /* Position de départ */
int gr_dboundx, gr_dboundy ;         /* Position rect. limite */
int gr_dboundw, gr_dboundh ;         /* Taille rect. limite */
int *gr_dfinishx, *gr_dfinishy;      /* Position finale    */
```

Cette fonction permet de déplacer un rectangle dans un autre rectangle. La fonction rend la main dès le relâchement de la souris.

Données en entrée :

CONTRL[0] = 71

INT_IN[0] = gr_dwidth. Largeur du rectangle à déplacer.

INT_IN[1] = gr_dheight. Hauteur du rectangle à déplacer.

INT_IN[2] = gr_dstartx. Coordonnée X de départ.

INT_IN[3] = gr_dstarty. Coordonnée Y de départ.

INT_IN[4] = gr_dboundx. Coordonnée X du rectangle limite.

INT_IN[5] = gr_dboundy. Coordonnée Y du rectangle limite.

INT_IN[6] = gr_dboundw. Largeur du rectangle limite.

INT_IN[7] = gr_dboundh. Hauteur du rectangle limite.

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

INT_OUT[1] = Coordonnée X finale.

INT_OUT[2] = Coordonnée Y finale.

```
#72 (0×48) :
int graf_movebox(gr_mwidth, gr_mheight, gr_msourcex,
gr_msourcey, gr_mdestx, gr_mdesty) ;
int gr_mwidth, gr_mheight ;          /* Taille du rectangle  */
int gr_msourcex, gr_msourcey ;       /* Position de départ   */
int gr_mdestx, gr_mdesty ;           /* Position d'arrivée   */
```

Cette fonction permet de déplacer un rectangle entre deux positions sans action de la souris



Données en entrée :

CONTRL[0] = 72

INT_IN[0] = gr_mwidth. Largeur du rectangle à déplacer.

INT_IN[1] = gr_mheight. Hauteur du rectangle à déplacer.

INT_IN[2] = gr_msourcex. Coordonnée X de départ.

INT_IN[3] = gr_msourcey. Coordonnée Y de départ.

INT_IN[4] = gr_mdestx. Coordonnée X d'arrivée.

INT_IN[5] = gr_mdesty. Coordonnée Y d'arrivée.

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

```
#73 (0x49) :
int graf_growbox (gr_gstx, gr_gsty, gr_gstw, gr_gsth,
gr_gfinx, gr_gfiny, gr_gfinw, gr_gfinh) ;
int gr_gstx, gr_gsty ;      /* Position de départ */
int gr_gstw, gr_gsth ;      /* Taille de départ   */
int gr_gfinx, gr_gfiny ;    /* Position finale    */
int gr_gfinw, gr_gfinh ;    /* Taille finale      */
```

Cette fonction permet de tracer une boîte qui se déplace d'une position à une autre en s'agrandissant.

Données en entrée :

CONTRL[0] = 73

INT_IN[0] = gr_gstx. Coordonnée X de départ.

INT_IN[1] = gr_gsty. Coordonnée Y de départ.

INT_IN[2] = gr_gstw. Largeur de départ.

INT_IN[3] = gr_gsth. Hauteur de départ.

INT_IN[4] = gr_gfinx. Coordonnée X d'arrivée.

INT_IN[5] = gr_gfiny. Coordonnée Y d'arrivée.

INT_IN[6] = gr_gfinw. Largeur d'arrivée.

INT_IN[7] = gr_gfinh. Hauteur d'arrivée.

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

```
#74 (0×4A) :
int graf_shrinkbox(gr_gfinx, gr_gfiny, gr_gfinw,
gr_gsth, gr_gstx, gr_gsty, gr_gstw, gr_gsth) ;
int gr_gfinx, gr_gfiny ;        /* Position finale    */
int gr_gfinw, gr_gfinh ;        /* Taille finale      */
int gr_gstx, gr_gsty ;          /* Position de départ */
int gr_gstw, gr_gsth ;          /* Taille de départ   */
```

Cette fonction permet de tracer une boîte qui se déplace d'une position à une autre en réduisant sa taille.

Données en entrée :

CONTRL[0] = 74
INT_IN[0] = gr_gfinx. Coordonnée X d'arrivée.
INT_IN[1] = gr_gfiny. Coordonnée Y d'arrivée.
INT_IN[2] = gr_gfinw. Largeur d'arrivée.
INT_IN[3] = gr_gfinh. Hauteur d'arrivée.
INT_IN[4] = gr_gstx. Coordonnée X de départ.
INT_IN[5] = gr_gsty. Coordonnée Y de départ.
INT_IN[6] = gr_gstw. Largeur de départ.
INT_IN[7] = gr_gsth. Hauteur de départ.

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

```
#75 (0×4B) :
int graf_watchbox(gr_wptree, gr_wobject, gr_winstate, gr_woutstate) ;
int *gr_wptree ;        /* Adresse de l'arbre  */
int gr_wobject ;        /* Objet à traiter     */
int gr_winstate ;       /* Etat si dedans      */
int gr_woutstate ;      /* Etat si dehors      */
```

Cette fonction permet de spécifier l'état d'un objet selon que la souris se trouve dans cet objet ou hors de cet objet.

Données en entrée :

CONTRL[0] = 75
INT_IN[0] = Réservé.
INT_IN[1] = Numéro de l'objet.
INT_IN[2] = Etat de l'objet lorsque la souris est à l'intérieur.



INT_IN[3] = Etat de l'objet lorsque la souris est à l'extérieur.

ADDRIN[0] = Adresse de l'arbre contenant l'objet.

Données en sortie :

INT_OUT[0] = Position de la souris lors du relâchement du bouton. 0 si hors de l'objet, 1 si elle est à l'intérieur.

```
#76 (0×4C) :
int graf_slidebox (gr_slptree, gr_slparent, gr_slobject, gr_slvh) ;
int *gr_sltree ;      /* Adresse de l'arbre          */
int gr_slparent ;     /* Père de l'objet à glisser   */
int gr_slobject ;     /* Objet à glisser             */
int gr_slvh ;         /* Sens du glissement 0 ou 1 */
```

Cette fonction permet de faire glisser une boîte à l'intérieur d'une autre. Pratiquement, c'est identique à l'ascenseur que l'on rencontre dans une fenêtre ou dans le sélecteur d'objets. Il faut indiquer la boîte dans laquelle se fera le glissement (l'objet père) et la boîte à glisser, ainsi que le sens du glissement. 0 pour horizontal et 1 pour vertical.

Données en entrée :

CONTRL[0] = 76

INT_IN[0] = gr_slparent. Objet père.

INT_IN[1] = gr_slobject. Objet à glisser.

INT_IN[2] = Sens du glissement (0 pour horizontal, 1 pour vertical).

ADDRIN[0] = gr_sltree. Adresse de l'arbre.

Données en sortie :

INT_OUT[0] = Position du centre de l'objet par rapport à son père. La valeur est comprise entre 0 et 1000.

```
#77 (0×4D) :
int graf_handle (gr_hwchar, gr_hhchar, gr_hwbox, gr_hhbox) ;
int *gr_hwchar, *gr_hhchar ;    /* Taille d'une cellule */
int *gr_hwbox, *gr_hhbox ;      /* Taille d'une boîte   */
```

Cette fonction réalise le lien avec le VDI et permet de connaître la taille maximale d'un caractère. Se référer au VDI, Partie 5, chapitre 2 pour plus de détails.

Données en entrée :

CONTRL[0] = 77

Données en sortie :

INT_OUT[0] = Identification de la station VDI.
INT_OUT[1] = Largeur d'un caractère.
INT_OUT[2] = Hauteur d'un caractère.
INT_OUT[3] = Largeur d'une boîte.
INT_OUT[4] = Hauteur d'une boîte.

```
#78 (0×4E) :
int graf_mouse (gr_monumber, gr_mofaddr) ;
int gr_monumber ;     /* Numéro de forme      */
int *gr_mofaddr ;     /* Description de forme */
```

Cette fonction permet de modifier l'aspect du pointeur de la souris. Certaines formes standard sont disponibles mais il est également possible de définir une forme propre.

Données en entrée :

CONTRL[0] = 78

INT_IN[0] = gr_monumber. Numéro de forme.

0 : flèche
1 : curseur texte
2 : abeille
3 : main fermée avec un doigt tendu
4 : main ouverte
5 : réticule fin
6 : réticule épais
7 : réticule encadré
255 : forme définie par gr_mofadr
256 : disparition de la souris
257 : apparition de la souris.

ADDRIN[1] = Adresse d'une forme dans un buffer de 35 octets. Ce paramètre n'est pris en compte que si gr_monumber est égal à 255.

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.



```
#79 (0×4F) :
int graf_mkstate (gr_mkmx, gr_mkmy, gr_mkmstate, gr_mkkstate) ;
int *gr_mkmx, *gr_mkmy ;        /* Position souris   */
int *gr_mkmstate ;              /* Etat des boutons  */
int *gr_mkkstate ;              /* Etat du clavier   */
```

Cette fonction permet de connaître à tout moment l'état de la souris et des touches spéciales du clavier.

Données en entrée :

CONTRL[0] = 79

Données en sortie :

INT_OUT[0] = 1

INT_OUT[1] = Coordonnée X de la souris.

INT_OUT[2] = Coordonnée Y de la souris.

INT_OUT[3] = Etat des boutons.
bit 0 : bouton gauche
bit 1 : bouton droit
1 si enfoncé, 0 si relâché

INT_OUT[4] = Etat des touches spéciales du clavier.
bit 0 : SHIFT droit
bit 1 : SHIFT gauche
bit 2 : CONTROL
bit 3 : ALTERNATE
1 si appuyé, 0 si relâché

VII - Les fonctions de gestion du presse-papier

Les deux fonctions de gestion du presse-papier, ou *clipboard* pour les anglophones, permettent de contrôler l'échange de données entre deux applications. L'intérêt du presse-papier est évident lorsque l'on utilise un utilitaire du type switcher. Il serait possible de couper un groupe de cellules dans un tableur et de les coller dans un traitement de texte. Malheureusement, très peu de programmes utilisent ces deux fonctions, qui sont d'ailleurs très limitées. Leur utilisation est relativement simple.

Tout d'abord, le disque est utilisé comme une zone de transit de l'information. L'application désirant écrire dans le presse-papier indique à l'AES avec la fonction scrp_write ( ) quel est le répertoire utilisé pour le presse-papier. L'application désirant lire le contenu du presse-papier doit demander à l'AES quel est le répertoire utilisé comme presse-papier. Ainsi le lien est fait. La dernière partie, et non la moindre, consiste en ce que les deux programmes puissent se comprendre dans le format utilisé pour les fichiers. Il existe pour cela quelques extensions de fichiers standards :

— TXT représente un fichier ASCII ;

— DIF est le format tableur ;

— GEM est le format metafile VDI ;

— IMG est un format bit-map compressé.

```
#80 (0×50) :
int scrp_read (sc_rpscrap) ;
char *sc_rpscrap ;      /* Répertoire du presse-papier */
```

Cette fonction permet de connaître quel est le répertoire utilisé pour le presse-papier.

Données en entrée :

CONTRL[0] = 80

ADDRIN[0] = sc_rpscrap. Buffer où sera stocké le nom du répertoire.

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

```
#81 (0×51) :
int scrp_write (sc_wpscrap);
char *sc_wpscrap ;      /* Répertoire du presse-papier */
```

Cette fonction permet de définir quel est le répertoire utilisé pour le presse-papier.

Données en entrée :

CONTRL[0] = 80

ADDRIN[0] = sc_rpscrap. Buffer où se trouve le nom du répertoire.

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

VIII - Les fonctions de sélection d'un fichier

On peut difficilement parler d'un groupe de fonctions dans ce cas puisqu'il n'y en a qu'une seule. D'un autre côté, elle est si importante, et si souvent utilisée, qu'elle mérite bien sa place ici. fsel_input ( ) permet de gérer le sélecteur de fichiers que vous connaissez tous. Son utilisation est très simple. Il suffit de lui passer le chemin complet du répertoire et un nom par défaut. En sortie, la fonction renverra le nouveau chemin



et le nom que l'utilisateur aura choisi. Il est possible d'indiquer, dans la dernière partie du chemin, des jokers permettant l'affichage d'une certaine catégorie de fichiers. Par exemple **A:*.PI?** n'affichera que les fichiers ayant l'extension débutant par PI.

```
#90 (0x5A) :
int fsel_input (fs_iinpath, fs_iinsel, fs_iexbutton) ;
char *fs_iinpath ;          /* Chemin complet */
char *fs_iinsel ;           /* Nom du fichier  */
int *fs_iexbutton ;         /* Bouton de sortie */
```

Cette fonction affiche le sélecteur d'objets avec comme chemin le contenu de la chaîne fs_iinpath et comme nom par défaut la chaîne fs_iinsel. En retour, la fonction renverra dans fs_iexbutton, une valeur représentant le bouton utilisé pour sortir : 1 pour le bouton OK et 0 pour le bouton ANNULER. Dans fs_iinsel se trouvera le nom choisi par l'utilisateur. Attention, il ne faut pas oublier de réserver suffisamment de place pour les deux chaînes (64 caractères pour le chemin et 12 pour le nom).

Données en entrée :

CONTRL[0] = 90

ADDRIN[0] = fs_iinpath. Chemin utilisé pour le sélecteur.

ADDRIN[1] = fs_iinsel. Nom par défaut.

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

INT_OUT[1] = Bouton de sortie. 0 pour ANNULER et 1 pour OK.

IX - Les fonctions de gestion des fenêtres

Le module de gestion des fenêtres permet de créer et de manipuler un des objets les plus importants de GEM. Ce module offre donc le moyen de créer, ouvrir, modifier une fenêtre. L'AES ne peut pas manipuler plus de 8 fenêtres et, de plus, la 1re fenêtre est réservée comme étant le bureau ou fond de l'écran. Une fenêtre est constituée de plusieurs parties bien distinctes :

- la barre de nom ;
- la barre d'informations ;
- les ascenseurs verticaux et horizontaux ;
- les flèches verticales et horizontales ;
- le bouton de fermeture de la fenêtre ;
- le bouton de zoom de la fenêtre ;
- le bouton de modification de taille.

L'utilisation des fenêtres nécessite un certain ordre :

— détermination de la taille disponible sur l'écran avec la fonction wind_get ( ) ;

— création de la fenêtre avec wind_create ( ) ;

— initialisation du nom, de la ligne d'information et de l'état des ascenseurs avec la fonction wind_set ( ) ;

— ouverture de la fenêtre à une taille quelconque mais inférieure à la taille de création avec la fonction wind_open ( ) ;

— utilisation de la fenêtre dans l'application. Toutes les fonctions de manipulation de fenêtres peuvent être envisagées ;

— en fin d'application ou dès que le besoin s'en fait sentir, il faut fermer la fenêtre avec la fonction wind_close ( ) et la supprimer avec wind_delete ( ). Là, l'ordre est très important.

```
#100 (0×64) :
int wind_create (wi_crkind, wi_crwx, wi_crwy, wi_crww, wi_crwh) ;
int wi_crkind ;
int wi_crwx ;
int wi_crwy ;
int wi_crww ;
int wi_crwh ;
```

Cette fonction crée une fenêtre. Attention, la fenêtre sera créée au niveau de l'AES mais ne sera pas affichée pour autant. Pour créer une fenêtre, il faut indiquer sa position et sa taille maximale (wi_crwx, wi_crwy, wi_crww, wi_crwh) mais aussi ses composantes définies par chacun des bits de wi_crkind :

bit 0 : nom de la fenêtre ;

bit 1 : bouton de fermeture de la fenêtre ;

bit 2 : bouton d'agrandissement de la fenêtre ;

bit 3 : déplacement de la fenêtre ;

bit 4 : ligne d'informations ;

bit 5 : changement de taille ;

bit 6 : flèche vers le haut ;

bit 7 : flèche vers le bas ;

bit 8 : ascenseur vertical ;

bit 9 : flèche vers la gauche ;

bit 10 : flèche vers la droite ;

bit 11 : ascenseur horizontal.



Données en entrée :

CONTRL[0] = 100

INT_IN[0] = wi_crkind. Composantes de la fenêtre.

INT_IN[1] = Coordonnée X de la fenêtre.

INT_IN[2] = Coordonnée Y de la fenêtre.

INT_IN[3] = Largeur de la fenêtre.

INT_IN[4] = Hauteur de la fenêtre.

Données en sortie :

INT_OUT[0] = Identificateur de la fenêtre ou une valeur négative si aucune fenêtre n'est disponible.

```
#101 (0×65) :
int wind_open (wi_ohandle, wi_owx, wi_owy, wi_oww, wi_owh)
int wi_ohandle ;
int wi_owx ;
int wi_owy ;
int wi_oww ;
int wi_owh ;
```

Cette fonction ouvre une fenêtre qui a été créée précédemment. La position et la taille d'ouverture peuvent être différentes de celles indiquées lors de la création. Il faut passer en paramètre wi_ohandle l'identificateur qui a été renvoyé par la fonction wind_create ( )

Données en entrée :

CONTRL[0] = 101

INT_IN[0] = wi_ohandle. Identificateur de la fenêtre.

INT_IN[1] = Coordonnée X de la fenêtre.

INT_IN[2] = Coordonnée Y de la fenêtre.

INT_IN[3] = Largeur de la fenêtre.

INT_IN[4] = Hauteur de la fenêtre.

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

```
#102 (0×66) :
int wind_close (wi_chandle) ;
int wi_chandle ;
```

Cette fonction ferme une fenêtre déjà ouverte. Attention la fermeture efface la fenêtre de l'écran mais ne la supprime pas pour autant de l'AES. Il reste possible de l'ouvrir à nouveau avec la fonction wind_open sans avoir à la recréer pour autant.

Données en entrée :

CONTRL[0] = 102

INT_IN[0] = wi_chandle. Identificateur de la fenêtre.

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

```
#103 (0×67) :
int wind_delete (wi_dhandle) ;
int wi_dhandle ;
```

Cette fonction supprime une fenêtre qui a déjà été fermée avec wind_close ( ). Attention la fenêtre ne peut plus être ouverte à nouveau, il faut la recréer avec wind_create ( ).

Données en entrée :

CONTRL[0] = 103

INT_IN[0] = wi_dhandle. Identificateur de la fenêtre.

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

```
#104 (0×68) :
    int wind_get (wi_ghandle, wi_gfield, wi_gw1, wi_gw2, wi_gw3, wi_gw4) ;
    int wi_ghandle ;
    int wi_gfield ;
    int *wi_gw1 ;
    int *wi_gw2 ;
    int *wi_gw3 ;
    int *wi_gw4 ;
```

Cette fonction, très importante, renseigne sur les caractéristiques d'une

fenêtre. Le paramètre wi_gfield détermine quelle information doit être retournée :

WF_WORKXYWH (4) : zone de travail de la fenêtre

wi_gw1 : coordonnée X
wi_gw2 : coordonnée Y
wi_gw3 : largeur
wi_gw4 : hauteur

WF_CURRXYWH (5) : taille totale de la fenêtre

La fonction renvoie les mêmes informations que pour WF_WORKXYWH

WF_PREVXYWH (6) : taille avant le dernier changement de taille

La fonction renvoie les mêmes informations que pour WF_WORKXYWH

WF_FULLXYWH (7) : taille la plus grande

La fonction renvoie les mêmes informations que pour WF_WORKXYWH

WF_HSLIDE (8) : position de l'ascenseur horizontal

wi_gw1 : position ascenseur entre 0 et 1 000
0 correspond à la gauche, 1 000 à la droite.

WF_VSLIDE (9) : position de l'ascenseur vertical

wi_gw1 : position ascenseur entre 0 et 1 000
0 correspond au haut, 1 000 au bas

WF_TOP (10) : Ident. de la fenêtre active

wi_gw1 : identificateur de la fenêtre la plus active.

WF_FIRSTXYWH (11) : premier rectangle de la fenêtre

wi_gw1 : coordonnée X du 1er rectangle
wi_gw2 : coordonnée Y du 1er rectangle
wi_gw3 : largeur
wi_gw4 : hauteur

WF_NEXTXYWH (12) : rectangle suivant la fenêtre

la fonction renvoie les mêmes informations que pour WF_FIRSTXYWH

WF_HSLIDE (15) : taille de l'ascenseur horizontal

wi_gw1 : taille de l'ascenseur. 1 à 1 000

WF_VSLIDE (16) : taille de l'ascenseur vertical

wi_gw1 : taille de l'ascenseur. 1 à 1 000

WF_WFSCREEN (17) : adresse et longueur du buffer menu/alertes

wi_gw1 : poids faible de l'adresse
wi_gw2 : poids fort de l'adresse
wi_gw3 : poids faible de la longueur
wi_gw4 : poids fort de la longueur

Données en entrée :

CONTRL[0] = 104

INT_IN[0] = wi_ghandle. Identificateur de la fenêtre.

INT_IN[1] = wi_gfield. Type d'information demandée.

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

INT_OUT[1] = wi_gw1

INT_OUT[2] = wi_gw2

INT_OUT[3] = wi_gw3

INT_OUT[4] = wi_gw4

```
#105 (0x69) :
int wind_set (wi_shandle, wi_sfield, wi_sw1, wi_sw2, wi_sw3, wi_sw4) ;
int wi_shandle ;
int wi_sfield ;
int wi_sw1 ;
int wi_sw2 ;
int wi_sw3 ;
int wi_sw4 ;
```

Cette fonction, très importante, permet de modifier les caractéristiques d'une fenêtre. Le paramètre wi_sfield détermine quelle caractéristique doit être modifiée :

WF_KIND (1) : composantes de la fenêtre

wi_sw1 : nouvelles composantes

WF_NAME (2) : nom de la fenêtre

wi_sw1 : poids faible du pointeur sur le nom
wi_sw2 : poids fort

WF_INFO (3) : texte d'informations

wi_sw1 : poids faible du pointeur sur l'information
wi_sw2 : poids fort

WF_WORKXYWH (3) : zone de travail de la fenêtre

wi_sw1 : coordonnée X
wi_sw2 : coordonnée Y
wi_sw3 : largeur
wi_sw4 : hauteur

WF_HSLIDE (8) : position de l'ascenseur horizontal

wi_sw1 : position ascenseur entre 0 et 1 000
0 correspond à la gauche, 1 000 à la droite



WF_VSLIDE (9) : position de l'ascenseur vertical

wi_sw1 : position ascenseur entre 0 et 1 000
0 correspond au haut, 1 000 au bas

WF_TOP (10) : identification de la fenêtre active

wi_sw1 : identificateur à activer.

WF_NEWDESK (14) : permet de modifier l'arbre décrivant le fond du bureau.

wi_sw1 : poids faible de l'adresse de l'arbre
wi_sw2 : poids fort

WF_HSLIDE (15) : taille de l'ascenseur horizontal

wi_sw1 : taille de l'ascenseur : 1 à 1 000

WF_VSLIDE (16) : taille de l'ascenseur vertical

wi_sw1 : taille de l'ascenseur : 1 à 1 000.

Données en entrée :

CONTRL[0] = 104

INT_IN[0] = wi_shandle. Identificateur de la fenêtre.

INT_IN[1] = wi_gfield. Type de la modification.

INT_IN[1] = wi_sw1

INT_IN[2] : wi_sw2

INT_IN[3] : wi_sw3

INT_IN[4] : wi_sw4

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

#106 (0×6A) :
int wind_find (wi_fmx, wi_fmy) ;
int wi_fmx, wi_fmy ;

Cette fonction détermine quelle est la fenêtre qui est sous une position de l'écran. Si aucune fenêtre n'est trouvée, la valeur 0 est retournée. Cela correspond à l'identificateur du bureau qui est aussi une fenêtre.

Données en entrée :

CONTRL[0] = 106

INT_IN[0] = wi_fmx. Coordonnée X.

INT_IN[1] = wi_fmy. Coordonnée Y.

Données en sortie :

INT_OUT[0] = Identificateur de la fenêtre.

 # 107 (0×6B) :
int wind_update (wi_ubegend) ;
int wi_ubegend ;

Cette fonction indique à AES que l'application commence à modifier le contenu d'une fenêtre. Ainsi qu'AES ne doit plus modifier quoi que ce soit à l'écran (les menus par exemple). La même fonction indique que l'application a terminé la modification de la fenêtre et qu'AES peut utiliser l'écran.

Données en entrée :

CONTRL[0] = 107

INT_IN[0] = wi_ubegend.

0 : réservation de l'écran

1 : réservation de la souris

2 : libération de la souris

3 : libération de l'écran

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

 # 108 (0×6C) :
int wind_calc (wi_ctype, wi_ckind, wi_cinx, wi_ciny, wi_cinw, wi_cinh, wi_coutx, wi_couty, wi_coutw, wi_couth) ;
int wi_ctype ;
int wi_ckind ;
int wi_cinx, wi_ciny, wi_cinw, wi_cinh ;
int wi_coutx, wi_couty, wi_coutw, wi_couth ;

Cette fonction renseigne sur la taille totale d'une fenêtre à partir de la taille de la zone de travail ou inversement.

Données en entrée :

CONTRL[0] = 108



INT_IN[0] = wi_ctype. Type de calcul.

0 : taille totale à partir de la taille de la zone de travail,
1 : taille de la zone de travail à partir de la taille totale.

INT_IN[1] = wi_ckind. Composantes de la fenêtre.

INT_IN[2] à INT_IN[5] = Position et taille initiales.

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

INT_OUT[1] à INT_OUT[4] : Position et taille finales.

X - Les fonctions de gestion des ressources

Les fonctions de gestion des ressources sont très liées aux fonctions de gestion des objets et formulaires que nous avons déjà vues. Un fichier ressource contient la description des menus et des formulaires d'un programme. Cette séparation entre le programme et la ressource est extrêmement importante et facilite la portabilité du programme. Cela ne va pas sans compliquer le travail du programmeur qui doit prendre en charge la mise en œuvre de ce nouvel environnement.

```
#110 (0×6E) :
int rsrc_load (re_lpname) ;
char *re_lpname ;      /* Nom du fichier ressource */
```

Cette fonction charge un fichier ressource. A cette occasion, AES le convertit pour qu'il soit utilisable directement.

Données en entrée :

CONTRL[0] = 110

ADDRIN[0] = re_lpname. Nom du fichier ressource.

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

```
#111 (0×6F) :
int rsrc_free( ) ;
```

Cette fonction libère le fichier ressource qui a été chargé par la fonction rsrc_load.

Données en entrée :

CONTRL[0] = 111

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

```
#112 (0×70) :
int rsrc_gaddr (re_gtype, re_gindex, re_gaddr) ;
int re_gtype ;          /* Type de structure      */
int re_gindex ;         /* Index de la structure  */
int ** re_gaddr ;       /* Variable de réception  */
```

Cette fonction sert à déterminer l'adresse d'une structure dans un fichier ressource. Il semble, malgré ce qu'affirme la littérature sur ce sujet, qu'il ne soit possible d'obtenir que l'adresse d'un arbre ou d'une structure libre telle qu'une chaîne de caractères définissant une boîte d'alerte. Ceci est dû au fait qu'il faille indiquer l'index de la structure que l'on recherche en fonction de son type. Ceci n'est pas vraiment aisé car le fichier INCLUDE, construit par un éditeur de ressource, ne donne que les index des arbres, des objets libres et des objets par rapport à l'arbre. Malgré tout, nous citerons l'ensemble des types possibles :

0 : arbre
1 : objet
2 : TEDINFO
3 : ICONBLK
4 : BITBLK
5 : chaîne de caractères
6 : données d'une image
7 : spécificité d'un objet
8 : te_ptext
9 : te_ptmplt
10 : te_pvalid
11 : ib_pmask
12 : ib_pdata
13 : ib_ptext
14 : bi_pdata
15 : pointeur sur une chaîne libre
16 : pointeur sur une image libre

Données en entrée :

CONTRL[0] = 112

INT_IN[0] = re_gtype. Type de structure recherchée.

INT_IN[1] = re_gindex. Index de la structure recherchée.

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

ADDROUT[0] = Adresse de la structure recherchée.



```
#113 (0x71) :
int rsrc_saddr (re_stype, re_sindex, re_saddr) ;
int re_stype ;        /* Type de structure */
int re_sindex ;       /* Index de la structure*/
int *re_saddr ;       /*Adresse              */
```

Cette fonction, à l'inverse de la fonction rsrc_gaddr, s'utilise pour définir l'adresse d'une structure.

Données en entrée :

CONTRL[0] = 113

INT_IN[0] = re_stype. Type de structure.

INT_IN[1] = re_sindex. Index de la structure.

ADDRIN[0] = re_saddr. Adresse de la structure.

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

```
#114 (0x72) :
int rsrc_obfix (re_otree, re_oobject) ;
int *re_otree ;       /* Adresse de l'arbre */
int re_oobject ;      /* Index de l'objet   */
```

Cette fonction convertit la taille et les coordonnées d'un objet du type caractère au type point. Cela permet d'être indépendant de la résolution utilisée sauf pour les icônes et les images qui ne pourraient l'être.

Données en entrée :

CONTRL[0] = 114

INT_IN[0] = re_oobject. Index de l'objet.

ADDRIN[0] = re_otree. Adresse de l'arbre.

Données en sortie :

INT_OUT[0] = 1. Il ne peut pas y avoir d'erreurs..

XI. Les fonctions de gestion des commandes

Ce groupe de fonctions réalise le lien entre l'AES et GEMDOS.

```
#120 (0×78) :
int shel_read (sh_rpcmd, sh_rptail) ;
char *sh_rpcmd ;      /* Commande d'appel */
char *sh_rptail ;     /* Paramètres       */
```

Cette fonction permet à l'application de savoir qui l'a appelée.

Données en entrée :

CONTRL[0] = 120

ADDRIN[0] = sh_rpcmd. Adresse d'un buffer où sera stocké le nom de l'appelant.

ADDRIN[1] = sh_rptail. Adresse d'un buffer où sera stocké le paramètre.

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

```
#121 (0×79) :
intshel_write (sh_wdoex, sh_wisgr, sh_wiscr, sh_wwpcmd, sh_wptail) ;
int sh_wdoex ;    /* Type d'action                */
int sh_wisgr ;    /* Application graphique ou non */
int sh_wiscr ;    /* Application GEM ou non       */
int sh_wptail ;   /* Paramètres                   */
```

Cette fonction lance une autre application en fonction de différents paramètres.

Données en entrée :

CONTRL[0] = 121

INT_IN[0] = sh_wdoex. Type d'action :
0 : retour au DOS (!!!)
1 : lancement d'une autre application.

INT_IN[1] = sh_wisgr. Type d'application :
0 : application non graphique
1 : application graphique.

INT_IN[2] = sh_wisc. Code de l'application :
0 : application non GEM
1 : application GEM.

ADDRIN[0] = sh_wpcmd. Commande d'appel de l'application.

ADDRIN[1] = sh_wptail. Ligne de commande.



Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

#122 (0×7A) :
char *sh_dfpbuff ;

Cette fonction un peu particulière permet de retrouver un fichier dans le répertoire courant et tous les répertoires définis par la variable d'environnement PATH=.

Données en entrée :

CONTRL[0] = 122

ADDRIN[0] = sh_fpbuff. Adresse du nom de fichier avant l'appel et adresse du nom complet après l'appel.

Données en sortie :

INT_OUT[0] = 0 en cas d'erreur, différent de 0 dans le cas contraire.

```
#123 (0×7B) :
int shel_envrn (sh_ename, sh_evar)
char *sh_ename ;      /* Buffer de réception */
char *sh_evar ;       /* Nom de la variable */
```

Cette fonction permet de connaître le contenu d'une variable d'environnement. Une telle variable est une chaîne de caractères passée à l'application lors de l'appel. Cela permet à une application de savoir où se trouvent certains paramètres si celle-ci a été appelée par un shell de type UNIX.

Données en entrée :

CONTRL[0] = 123

ADDRIN[0] = sh_ename. Buffer de réception.

ADDRIN[1] = sh_evar. Nom de la variable (ex. : PATH=).

3/6

Le Bureau GEM

La mise sous tension de l'Atari en présence de la disquette système provoque le chargement d'utilitaires accessibles en pointant le menu « Bureau ».

Ces utilitaires sont essentiellement des programmes de communication ou de configuration.

- L'émulateur VT 52 permet la transmission et la réception au travers d'un Modem.
- Le panneau de contrôle permet de régler certains paramètres utilisateur comme le temps de réponse du clavier, la vitesse de double-clic de la souris, l'heure et la date, etc.
- La configuration du port RS232.
- L'installation d'une imprimante.

3/6.1

L'émulateur de terminal VT 52

La version de base des Atari ST a la possibilité d'émuler la norme de visualisation VT 52. Cette norme permet d'éditer un texte sur l'écran :

- positionnement du curseur ;
- déplacement du curseur ;
- effacement :
  - d'une ligne,
  - d'une portion de ligne,
  - de l'écran,
  - d'une portion d'écran,
- insertion de lignes,
- scrolling,
- etc.

Voici la liste des commandes VT 52. Chacune d'entre elles commence par le code ESCape.

ESC A Curseur vers le haut.
Le curseur se déplace vers le haut d'une ligne. S'il se trouve sur la première ligne de l'écran, aucune action ne se produit.

ESC B Curseur vers le bas.
Le curseur se déplace vers le bas d'une ligne. S'il se trouve sur la dernière ligne de l'écran, aucune action ne se produit.

ESC C Curseur vers la droite.
Le curseur se déplace d'un caractère vers la droite.

ESC D Curseur vers la gauche.
Le curseur se déplace d'un caractère vers la gauche. S'il se trouve sur la première colonne de l'écran, aucune action ne se produit.

ESC E Effacement de l'écran.
L'écran est effacé et le curseur est positionné en haut et à gauche de l'écran (HOME).

ESC H Curseur à l'origine.
Le curseur est positionné en haut et à gauche de l'écran (HOME).

ESC I Curseur vers le haut.
Le curseur se déplace vers le haut d'une ligne. S'il se trouve sur la première ligne de l'écran, une ligne vierge est insérée, et tout l'écran est déplacé vers le bas d'une ligne.

ESC J Effacement de la fin de l'écran.
Les caractères qui se trouvent après le curseur (jusqu'à la fin de l'écran) sont effacés.
Le curseur ne change pas de position.

ESC K Effacement de la fin de la ligne.
Les caractères de la ligne courante qui sont situés après le curseur sont effacés.
Le curseur ne change pas de position.

ESC L Insertion d'une ligne.
Une ligne est insérée après la ligne courante. Les lignes suivantes sont décalées vers le bas, et le curseur est positionné au début de la nouvelle ligne.

ESC M Effacement d'une ligne.
La ligne courante est effacée et toutes les lignes qui suivent sont décalées vers le haut de l'écran. Une ligne vierge est insérée en bas de l'écran. Le curseur est positionné sur la première colonne de la nouvelle ligne courante.

ESC Y a b Positionnement du curseur.
Cette commande permet de positionner le curseur sur l'écran. a représente la ligne du curseur et b la colonne du curseur. Ces deux valeurs doivent être augmentées de 32. Par exemple, pour positionner le curseur en ligne 10 et colonne 11, il faudra envoyer la commande ESC Y * + (le code ASCII de * est 42 = 32+10, et le code ASCII de + est 43 = 32+11).

Pour un écran de 80 colonnes, les numéros de colonnes sont dans l'intervalle 0..79, et les numéros de lignes dans l'intervalle 0..24.

ESC b x Couleur d'écriture.
Cette commande permet de choisir la couleur d'écriture sur l'écran. Le paramètre x spécifie la couleur d'écriture : @ pour la couleur 0, A pour la couleur 1, B pour la couleur 2, ..., O pour la couleur 15.

ESC c Couleur de fond d'écran.
Cette commande permet de choisir la couleur du fond de l'écran. Le paramètre x spécifie cette couleur : @ pour la couleur 0, A pour la couleur 1, B pour la couleur 2, ..., O pour la couleur 15.

ESC d Effacement du début de l'écran.
Les données affichées entre le début de l'écran (en haut et à gauche) et la position courante du curseur sont effacées. Le curseur ne change pas de position.

ESC e Affichage du curseur.
Grâce à cette commande, le curseur est rendu visible.

ESC f Effacement du curseur.
Grâce à cette commande, le curseur est rendu invisible.

ESC j Mémorisation de la position du curseur.
La position courante du curseur est mémorisée. Ainsi, il sera possible de le déplacer pour effectuer une opération quelconque (par exemple un effacement partiel de l'écran), puis de le replacer à sa position d'origine en utilisant la commande ESC k.

ESC k Positionnement du curseur à la position mémorisée.
Cette commande positionne le curseur à la position mémorisée par la commande ESC j.

ESC l Effacement d'une ligne.
Cette commande efface la ligne courante. Les lignes qui suivent ne sont pas remontées vers le haut. Le curseur est positionné au début de la ligne effacée.

ESC o Effacement du début de la ligne.
Cette commande efface les caractères compris entre le début de la ligne et la position courante du curseur. Cette position est inchangée.

ESC p Vidéo inverse.
Après exécution de cette commande, les caractères seront affichés en vidéo inverse (encre de la couleur du fond, et fond de la couleur de l'encre).

ESC q Vidéo normale.
Cette commande annule la précédente. Après son exécution, les caractères sont affichés avec la couleur d'encre normale.

ESC v Insertion automatique de lignes.
Après exécution de cette commande, le curseur passe de la colonne 80 à la colonne 1 de la ligne suivante lorsqu'un déplacement vers la droite ou l'affichage d'un caractère est demandé.

ESC w Fin d'insertion automatique de lignes.
Cette commande annule la précédente : lorsque le curseur se trouve en colonne 80, il y reste même si d'autres caractères sont tapés. Le passage en début de ligne suivante n'est plus automatique.

3/7

La gestion des périphériques de base

3/7.1

La gestion du clavier

Le clavier est relié au 68000 à travers un microprocesseur 6301 (de la famille 6800) et un circuit série ACIA 6850. Le 6301 est un vrai microprocesseur 8 bits qui possède en interne des mémoires RAM et ROM, et une interface série. Le gestionnaire du clavier est un logiciel qui s'occupe non seulement du clavier, mais également de la souris, de la ou des manettes de jeu et de la maintenance de l'horloge système.

Dans ce paragraphe, nous allons décrire les commandes du 6301, classées par périphérique.

(Se reporter à la Partie 2, chapitre 1.4, paragraphe IV, pour une description détaillée de ce circuit.)

LE CLAVIER

Un code est renvoyé au microprocesseur par le 6301 à chaque appui et à chaque relâchement d'une touche du clavier. Ce code est indépendant du pays ou des touches spéciales pressées (**Shift, Ctrl, Alternate**) ou sélectionnées (**Caps Lock, Num Lock**). Il correspond en fait à un code géographique, comme le montre le schéma suivant :

```
   3B 3C 3D 3E 3F 40 41 42 43 44

01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 29 0E    62    61    63 64 75 66

0F 10 11 12 13 14 15 16 17 18 19 1A 1B    53    52 48 47    67 68 69 4A

1D 1E 1F 20 21 22 23 24 25 26 27 28    1C 2B    4B 50 4D    6A 6B 6C 4E

2A 60 2C 2D 2E 2F 30 31 32 33 34 35 36                     6D 6E 6F

   38                39                3A                70 71    72
```

Lors du relâchement d'une touche, le même code est réémis, mais avec le bit de poids fort (bit 7) à un.

Des codes compris entre $F6 et $FF peuvent également être émis. Ils correspondent aux événements particuliers suivants :

Code	Evénement
F6	Compte rendu d'état
F7	Enregistrement de la position absolue de la souris
F8 à FB	Enregistrement de la position relative de la souris
FC	Date système
FD	Etat des manettes
FE	Gestion de la manette 1
FF	Gestion de la manette 2

Des commandes peuvent être envoyées au gestionnaire de clavier sous la forme d'un octet de commande et éventuellement d'un ou de plusieurs octets de paramètres.

LA SOURIS

Le gestionnaire de clavier transmet un compte rendu dès que la souris est activée (déplacement et/ou appui sur un des deux boutons). Les commandes du gestionnaire de clavier dédiées à la souris ont des codes fonction compris entre #00 et #13. En voici la liste :

#00 Fonctionnement normal

#01 Après exécution de cette commande, la pression d'une des touches de la souris donne lieu à un compte rendu de position absolue



#02 Après exécution de cette commande, la pression d'une des touches de la souris donne lieu à un compte rendu de position relatif

#04 Après exécution de cette commande, les touches de la souris sont considérées comme des touches de déplacement du clavier

#08 Cette commande est sélectionnée par défaut à l'initialisation. Elle spécifie que la position de la souris doit être renvoyée de façon relative (par rapport à la position précédente).

Les données renvoyées à chaque déplacement de la souris sont composées des trois octets suivants :

Octet 1 = %111110DG
où D vaut un si le bouton de gauche est pressé,
et G vaut un si le bouton de droite est pressé.

Octet 2 = Déplacement sur l'axe horizontal (sous la forme d'un entier complémenté à deux).

Octet 3 = Déplacement sur l'axe vertical (sous la forme d'un entier complémenté à deux).

#09 Cette commande spécifie que la position de la souris doit être renvoyée de façon absolue. Elle est composée de cinq octets :

#09 MSBX LSBX MSBY LSBY

où MSBX et LSBX sont les poids fort et faible du déplacement maximal sur l'axe des abscisses,

et MSBY et LSBY sont les poids fort et faible du déplacement maximal sur l'axe des ordonnées.

Les données renvoyées à chaque déplacement de la souris sont composées des trois octets suivants :

Octet 1 = %111110DG
où D vaut un si le bouton de gauche est pressé,
et G vaut un si le bouton de droite est pressé.

Octet 2 = Déplacement sur l'axe horizontal (sous la forme d'un entier complémenté à deux) par rapport à une position absolue et un facteur d'échelle.

Octet 3 = Déplacement sur l'axe vertical (sous la forme d'un entier complémenté à deux) par rapport à une position absolue et un facteur d'échelle.

Remarque concernant les positionnements absolu et relatif :

Lorsque le déplacement de la souris est supérieur à 127 ou inférieur à 128, il donne lieu à autant de séries de trois octets que nécessaire.

#0A Cette commande positionne la souris en mode clavier : lors d'un déplacement de la souris, ce ne sont plus les coordonnées absolues ou relatives qui sont renvoyées, mais un ensemble de codes de déplacement du clavier.

Cette commande est composée de trois octets :

#0A IG ID

où IG est le nombre d'impulsions nécessaires pour produire un code de déplacement vers la gauche,
et ID est le nombre d'impulsions nécessaires pour produire un code de déplacement vers la droite.

#0B Cette commande permet de définir le seuil de déplacement en mode relatif, c'est-à-dire la distance au-delà de laquelle le gestionnaire doit considérer que la souris a été déplacée. Elle se compose de trois octets :

#0B SX SY

où SX est le seuil de déplacement sur l'axe des abscisses,
SY est le seuil de déplacement sur l'axe des ordonnées.

#0C Cette commande permet de définir le facteur d'échelle de la souris en mode absolu, c'est-à-dire la distance au-delà de laquelle le gestionnaire doit considérer que la souris a été déplacée. Elle se compose de trois octets :

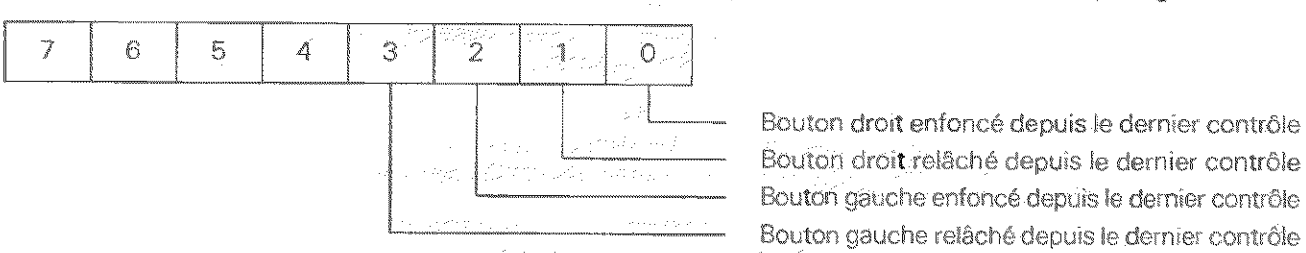
#0C SX SY

où SX est le seuil de déplacement sur l'axe des abscisses,
SY est le seuil de déplacement sur l'axe des ordonnées.

#0D Cette commande renvoie la position absolue de la souris. Cette information est codée sur 6 octets :

#F7 ET MSBX LSBX MSBY LSBY

où ET représente l'état des touches de la souris, selon le codage suivant :



MSBX ET LSBX sont les poids fort et faible du déplacement maximal sur l'axe des abscisses,

et MSBY et LSBY sont les poids fort et faible du déplacement maximal sur l'axe des ordonnées.

#0E Cette commande initialise la position absolue de la souris. Elle est composée des cinq octets suivants :

#0E #00 MSBX LSBX MSBY LSBY

où MSBX et LSBX sont les poids fort et faible du déplacement maximal sur l'axe des abscisses,

et MSBY et LSBY sont les poids fort et faible du déplacement maximal sur l'axe des ordonnées.



#0F Cette commande permet de définir l'origine de l'axe des ordonnées en bas de l'écran.

#10 Cette commande permet de définir l'origine de l'axe des ordonnées en haut de l'écran.

#11 Cette commande réactive la transmission des données concernant la souris à l'unité centrale. Cette transmission peut être suspendue à l'aide de la commande #12.

#12 Cette commande inhibe la souris. Les commandes #08, #09 et #0A réactivent la souris.

#13 Cette commande fait passer la souris en mode pause : les données en rapport avec la souris ne sont plus transmises à l'UC. La commande #11 permet de reprendre l'émission de données vers l'UC.

LA MANETTE (JOYSTICK)

Deux manettes peuvent être connectées aux ST. Le gestionnaire clavier transmet un compte rendu dès qu'une des deux manettes est activée. Les commandes du gestionnaire clavier dédiées aux manettes ont des codes fonction compris entre #14 et #1A. En voici la liste :

#14 Cette commande positionne le gestionnaire en compte rendu automatique : chaque déplacement de la manette donne lieu à un compte rendu de deux octets organisés comme suit :

%1111111x %b000pppp

x vaut 0 pour la manette 0, et 1 pour la manette 1,
b vaut 0 lorsque le bouton est relâché, et 1 quand il est appuyé,
pppp indiquent la position de la manette.

#15 Cette commande positionne le gestionnaire en compte rendu non automatique : les déplacements de la manette ne donnent pas lieu à un compte rendu automatique. Ce compte rendu peut être demandé en exécutant la commande #16.

#16 Cette commande renvoie la position et l'état du bouton d'une manette. Les données renvoyées sont au même format que celui explicité dans la commande #14.

#17 Cette commande définit le temps d'échantillonnage de la manette, c'est-à-dire la durée entre deux émissions successives d'informations en provenance de la manette.

La commande est codée sur trois octets :

#17 MSB LSB

où MSB et LSB sont les poids fort et faible du temps d'échantillonnage exprimé en centièmes de secondes.

Cette commande renvoie un compte rendu codé sur deux octets :

%000000ab %ccccdddd

où a et b représentent l'état des boutons des manettes 0 et 1. cccc et dddd représentent l'état des manettes 0 et 1.

#18 Cette commande met en fonction l'échantillonnage du bouton feu des manettes. Après son exécution, le gestionnaire renvoie à intervalles réguliers un octet qui correspond à 8 échantillonnages consécutifs du bouton feu.

#19 Cette commande permet de simuler les touches de déplacement du clavier pour la manette 0. Elle est composée des sept octets suivants :

19 TX TY DX DY FX FY

où TX et TY représentent le premier délai séparant deux codes de touches,

DX et DY représentent la durée d'émission des codes touches à l'intervalle TX ou TY,

FX et FY représentent le second délai séparant deux codes de touches.

Lors de la détection d'un déplacement de la manette, un code de déplacement du curseur est émis tous les DX ou DY dixièmes de seconde pendant TX ou TY dixièmes de seconde. Le code de déplacement est ensuite engendré tous les FX ou FY dixièmes de seconde.

#1A Cette commande désactive la gestion des manettes jusqu'à ce qu'une autre commande manette soit exécutée.

L'HEURE SYSTEME

Le gestionnaire du clavier permet également de maintenir et d'accéder en lecture et en écriture à l'heure système. Examinons les commandes qui sont dédiées à cette tâche.

#1B Cette commande initialise l'horloge système (interne au processeur clavier). Le codage de la date et de l'heure se fait sur 6 octets, en BCD (binaire codé décimal), selon le format suivant :

Octet	Information	Exemple
1	Année sur deux chiffres	89
2	Mois sur deux chiffres	02
3	Jour sur deux chiffres	21
4	Heure sur deux chiffres	09
5	Minutes sur deux chiffres	31
6	Secondes sur deux chiffres	00

Remarques :

- Chaque chiffre est codé sur un quartet (4 bits).
- Les octets qui ne contiennent pas des informations BCD correctes sont ignorés.

#1C Cette commande lit la valeur de l'horloge système et renvoie six octets d'information codés en BCD (comme pour la commande $1B) précédés de l'octet d'en-tête #FC.



#20 Cette commande permet d'envoyer un ou plusieurs octets dans la mémoire du contrôleur de clavier. Les données émises doivent respecter la structure suivante :

#21 MSBA LSBA NB O1 O2 O3...

où MSBA et LSBA sont le poids fort et le poids faible de l'adresse mémoire de début de stockage,
NB est le nombre d'octets à écrire,
et les Oi sont les octets à écrire.

#21 Cette commande permet de lire un ou plusieurs octets dans la mémoire du contrôleur de clavier. Le format de la commande est le suivant :

#21 MSBA LSBA

où MSBA et LSBA sont le poids fort et le poids faible de l'adresse mémoire de début de lecture.

Les données renvoyées sont au nombre de huit :

— deux octets d'en-tête : #F6 #20

— six octets de données correspondant au contenu des six mémoires consécutives de première adresse MSBA LSBA.

#22 Cette commande permet d'exécuter une routine interne du contrôleur de clavier. Elle est codée sur trois octets :

22 MSBA LSBA

où MSBA et LSBA sont les poids fort et faible du point d'entrée de la routine à exécuter.

LES MESSAGES D'ETAT

Il est possible d'obtenir un compte rendu de chacune des commandes que nous venons de décrire. Pour cela, il suffit de positionner le bit 7 dans le premier octet de la commande.

En retour, huit octets sont renvoyés :

Octet	Information
1	F6 : En-tête
2	Numéro de la commande
3	Paramètre 1
4	Paramètre 2
5	Paramètre 3
6	Paramètre 4
7	Paramètre 5
8	Paramètre 6

Les commandes qui permettent d'obtenir un compte rendu sont les suivantes :

Commande	Code premier octet
#7	#87
#8	#88
#9	#89
#A	#8A
#B	#8B
#C	#8C
#F	#8F
#10	#90
#12	#92
#14	#94
#15	#95
#19	#99
#1A	#9A



3/8

Les éditeurs de ressources

Tout puriste digne de ce nom vous dirait «Pour réaliser dans les règles de l'art un programme basé sur un menu, incluant des formulaires et des boîtes d'alerte, il faut utiliser un fichier ressource (xxx.RSC)».

Si développer directement en GFA un menu et des boîtes de dialogue ne pose pas trop de problèmes, la création de formulaires est beaucoup plus délicate et nécessite un éditeur de ressources.

Pour des raisons de clarté de programme, de simplicité des modifications et des traductions en d'autres langues, il faut développer à la fois les menus, les boîtes d'alerte et les formulaires à l'aide d'un éditeur de ressources.

Pour les lecteurs qui n'ont jamais entendu parler de ce type de logiciel, un éditeur de ressources est un outil graphique qui permet de composer les menus et les formulaires les plus complexes en utilisant uniquement la souris.

Les éditeurs de ressources les plus connus sont RCS(2).PRG (fourni avec le Basic GFA), WERCS.PRG, NRSC.PRG (le plus utilisé par les développeurs confirmés) et depuis peu, INTRFACE.PRG (certainement celui qui permet le plus de fantaisies).

Avant de donner le mode d'emploi d'un éditeur, il est intéressant d'avoir un peu plus d'informations sur la fonction et le contenu d'un fichier ressource.

Pour cela, revenons quelques années en arrière. Si les ordinateurs ont toujours été utilisés par des informaticiens, les micro-ordinateurs ont été développés pour atteindre une catégorie de consommateurs bien souvent néophytes. Si les plus jeunes générations ont très vite réussi à asservir la machine, les autres générations ont subi de nombreux échecs décourageants lors de leur première confrontation avec l'informatique. Qu'y a-t-il de plus déconcertant que de se retrouver, après allumage de la

machine, devant un curseur qui clignote en attendant une commande DOS dont la syntaxe est capricieuse ? Rappelez-vous les premiers APPLE et les premiers IBM PC (sans WINDOWS).

Pour que la micro-informatique puisse percer, notamment par l'intermédiaire de la bureautique, il a fallu développer un environnement facile d'emploi. Ces interfaces homme-machine sont les systèmes d'exploitation graphique que l'on trouve actuellement sur les Macintosh, les Atari et les Compatibles PC (sous WINDOWS uniquement). Ce type de système d'exploitation est appelé par les informaticiens, un environnement GEM (*Graphic Environment Manager*) dont la firme américaine DIGITAL possède la plupart des brevets.

Afin d'implanter sur l'Atari ST un environnement GEM, Jack Tramiel (l'inventeur du ST) et coll. ont développé un ensemble de routines qui sont regroupées dans l'AES (*Application Environment Services*). Ces routines, qui sont à la base du fonctionnement du bureau, peuvent heureusement être utilisées par les développeurs.

L'AES se compose de plusieurs bibliothèques :

— *Application Library* (bibliothèque de gestion des applications);

— *Event Library* (bibliothèque de gestion des évènements);

— *Menu Library* (bibliothèque de gestion des menus);

— *Object Library* (bibliothèque de gestion des objets);

— *Form Library* (bibliothèque de gestion des formulaires);

— *Graphics Library* (bibliothèque de fonctions graphiques);

— *File Selector Library* (bibliothèque du sélecteur de fichiers);

— *Windows Library* (bibliothèque de gestion des fenêtres);

— *Ressources Library* (bibliothèque de gestion des ressources);

— *Shell Library* (bibliothèque de gestion des programmes).

Les instructions présentes dans chaque bibliothèque sont définies plus précisément dans la partie 3/5.1 de cet ouvrage.

Ces différentes instructions peuvent sembler complexes. L'objectif de ce chapitre est de donner des éclaircissements sur leur emploi.

Mais, pour apprendre à utiliser ces fonctions, nous allons vous expliquer comment les fichiers ressource sont organisés et vous donner quelques précisions sur la structure des objets qui composent un fichier ressource.

Un objet est par définition une brique élémentaire qui peut avoir différentes propriétés. Une entrée de menu, le bouton OK d'une boîte d'alerte, une icône, un des champs de saisie d'un formulaire, sont des exemples d'objets. Leur organisation est faite sous forme d'arbres (Fig. 1).

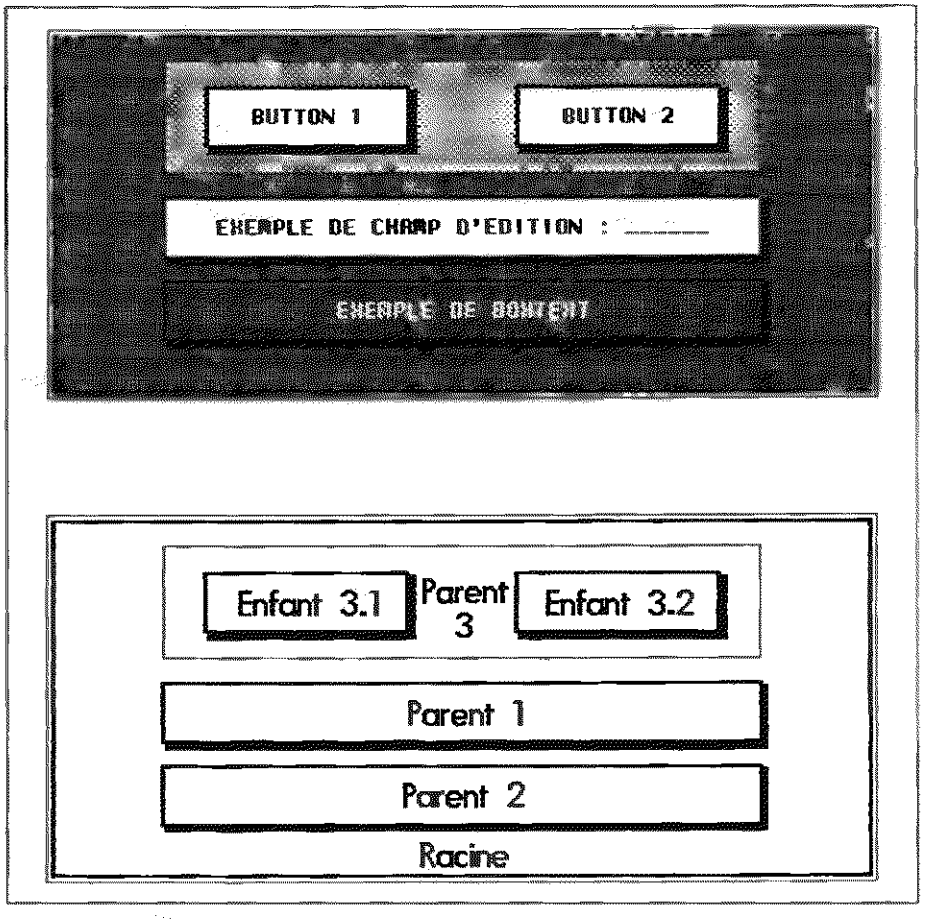


Fig. 1 : Organisation d'un arbre d'objets

Chaque objet est caractérisé par le nom de l'arbre dont il fait partie, et par un numéro qui lui est propre (offset dans l'arbre).

La structure principale a un format fixe pour tout type d'objet. En fonction de son type, un objet peut être défini par un ensemble de sous-structures (BOXINFO, TEDINFO, BITBLK,...) qui complètent la structure principale.

Voyons maintenant, la structure principale de chaque type d'objet ainsi que les sous-structures qui peuvent s'y rattacher.

STRUCTURE PRINCIPALE D'UN OBJET

WORD	ob_next	Numéro de l'objet suivant dans l'arborescence.
WORD	ob_head	Numéro du premier enfant (s'il existe).
WORD	ob_tail	Numéro du dernier enfant.
unsigned WORD	ob_type	Type de l'objet (voir chapitre 1).
unsigned WORD	ob_flag	Informations sur l'objet (voir chapitre 2).

unsigned WORD	ob_state	Etat de l'objet (voir chapitre 3).
LONG	ob_spec	Complément d'informations sur l'objet.
WORD	ob_x	Coordonnée X du coin supérieur gauche.
WORD	ob_y	Coordonnée Y du coin supérieur gauche.
WORD	ob_w	Largeur en pixels de l'objet.
WORD	ob_h	Hauteur en pixels de l'objet.

1. Fonction de ob_type

Valeur de ob_type	Type d'objet	Contenu de ob_spec
20	G_BOX : boîte simple	Structure BOXINFO
21	G_TEXT : texte graphique	Adresse d'une structure TEDINFO
22	G_BOXTEXT : boîte de type G_BOX contenant un texte de type G_TEXT	Adresse d'une structure TEDINFO
23	G_IMAGE : image de taille variable	Adresse d'une structure BITBLK
24	G_USERDEF : objet spécial dont la fonction peut être fixée par le programme	Adresse d'une structure ou d'une routine définie par l'utilisateur
25	G_IBOX : boîte de type G_BOX sans couleur et sans motif	Structure BOXINFO
26	G_BUTTON : boîte de type G_IBOX contenant un texte centré	Adresse du début de texte (format C)
27	G_BOXCHAR : lettre isolée dans une G_BOX	Structure BOXINFO
28	G_STRING : chaîne de caractères simple	Adresse du début de texte (format C)
29	G_FTEXT : objet analogue à un G_TEXT mais offrant un champ éditable et formattable	Adresse d'une structure TEDINFO



(suite)

30	G_FBOXTEXT : boîte de type G_BOX incluant un G_FTEXT	Adresse d'une structure TEDINFO
31	G_ICON : objet analogue à un G_IMAGE mais avec quelques particularités	Adresse d'une structure ICONBLK

a) Structure BOXINFO

WORD	te_color	Couleur de l'objet dont les bits ont les fonctions suivantes : — bits 0 à 3 : couleur à l'intérieur du cadre; — bits 4 à 6 : motif de remplissage (0 = pas de motif, 7 = densité maximale); — bit 7 : mode graphique du texte (0 = transparent, 1 = opaque); — bits 8 à 11 : couleur de texte; — bits 12 à 15 : couleur du cadre.
WORD	te_thickness	Epaisseur du cadre définie par : — valeur 0 : pas de cadre; — valeur comprise entre 1 et 127 : cadre s'enfonçant vers l'intérieur et dont l'épaisseur dépend de cette valeur; — valeur comprise entre -1 et -127 : cadre s'étendant vers l'extérieur et dont l'épaisseur dépend de cette valeur.

b) Structure TEDINFO

LONG	te_ptext	Adresse de début du texte.
LONG	te_ptmplt	Adresse de la chaîne de caractères définissant le masque de texte formatté.
LONG	te_valid	Adresse de la chaîne de caractères validant les types de données attendues dans le champ de saisie.
WORD	te_font	Taille du jeu de caractère (3 = taille normale, 5 = petite taille).
WORD	te_junk1	Inutilisé.
WORD	te_just	Type de justification (0 = texte aligné à gauche, 1 = texte aligné à droite, 2 = texte centré).
WORD	te_color	Voir structure BOXINFO.
WORD	te_junk2	Inutilisé.
WORD	te_thickness	Voir structure BOXINFO.
WORD	te_txtlen	Longueur du texte à l'adresse te_ptext.
WORD	te_tmplen	Longueur du masque à l'adresse te_ptmplt.

Les trois premiers pointeurs de cette structure nécessitent certainement quelques explications supplémentaires. Prenons un exemple.

Considérons le cas d'un champ de saisie présent dans un formulaire et dont la forme est la suivante :

"Veuillez entrer la date : 12/34/5678"

Avant qu'un tel champ de saisie soit activé par l'instruction FORM_DO (bibliothèque de gestion des formulaires), les trois LONGs te_ptext, te_ptmplt et te_pvalid pointent sur les trois chaînes de caractères suivantes :

te_ptext -> "12345678"
te_ptmplt -> "veuillez entrer la date : __/__/____"
te_pvalid -> "99999999"

Pendant l'exécution de l'instruction FORM_DO, le curseur ne peut que se déplacer sur les emplacements où figure le caractère "_" (fonction de te_ptmplt). De plus, les seuls caractères acceptés par le champ de saisie sont dans ce cas, des chiffres (fonction de te_pvalid).

Pour bien distinguer la provenance exacte de tout ce qui est affiché à l'écran, nous avons arbitrairement rempli la chaîne de caractères pointée par te_ptext par le nombre "12345678". Lors de la saisie de la bonne date, il va falloir, dans un premier temps, effacer ces huit chiffres, pour laisser apparaître les huit "_".

Après l'exécution de l'instruction FORM_DO, on retrouvera dans la chaîne de caractères pointés par te_ptext, la date qui aura été tapée. Les deux autres chaînes de caractère n'auront pas été modifiées.

La chaîne de caractères pointée par te_pvalid est dans notre exemple "999999999", parce qu'une réponse numérique était attendue par le champ de saisie. Pour répondre à d'autres types de besoins, une série de codes symboliques existe. A chaque code correspond une variante de la gamme des caractères qu'il est possible de taper dans un champ de saisie :

— 9 : uniquement les chiffres;

— A : les espaces et les majuscules;

— a : les espaces et toutes les lettres;

— N : les espaces, les majuscules et les chiffres;

— n : les espaces, les lettres et les chiffres;

— F : tous les caractères admis dans un nom de fichier valable avec en plus ? * et : ;

— P : tous les caractères admis dans un nom de fichier valable avec en plus ? * : et \ ;

— p : tous les caractères admis dans un nom de fichier valable avec en plus \ et : ;

— X : tous les types de caractères sont autorisés.



c) Structure BITBLK

LONG	bi_pdata	Adresse des données de l'image (chaque ligne doit contenir un nombre pair d'octets).
WORD	bi_wb	Largeur de l'image en octets.
WORD	bi_hl	Hauteur de l'image en pixels.
WORD	bi_x	Position X.
WORD	bi_y	Position Y.
WORD	bi_color	Couleur de l'image (entre 0 et 15).

d) Structure ICONBLK

LONG	ib_pmask	Adresse des données du masque (bi_pdata de la structure BITBLK).
LONG	ib_pdata	Adresse des données de l'image.
LONG	ib_ptext	Adresse du texte.
WORD	ib_char	Caractère à l'intérieur de l'icône.
WORD	ib_xchar	Position X de ib_char par rapport à l'icône.
WORD	ib_ychar	Position Y de ib_char par rapport à l'icône.
WORD	ib_xicon	Position X de l'icône.
WORD	ib_yicon	Position Y de l'icône.
WORD	ib_wicon	Largeur de l'icône en pixels.
WORD	ib_hicon	Hauteur de l'icône en pixels.
WORD	ib_xtext	Position X de ib_ptext par rapport à l'icône.
WORD	ib_ytext	Position Y de ib_ptext par rapport à l'icône.
WORD	ib_wtext	Largeur de ib_ptext en pixels.
WORD	ib_htext	Hauteur de ib_ptext en pixels.
WORD	ib_resvd	Inutilisé.

2. Fonctions des bits du descripteur ob_flags

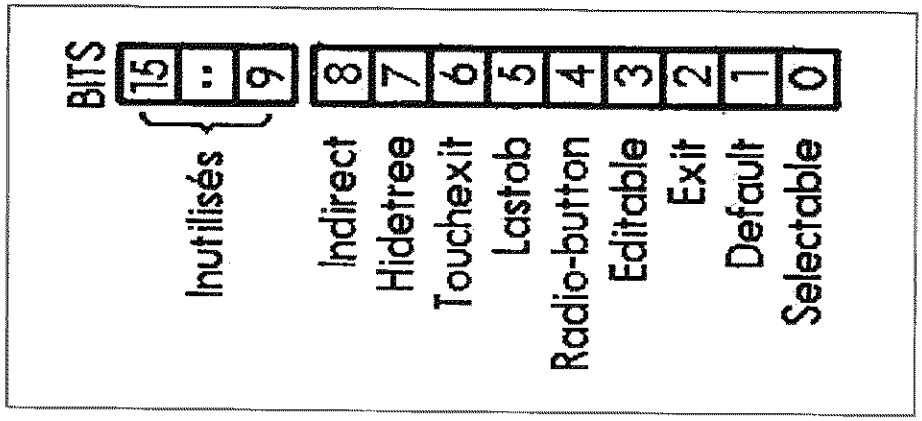


Fig. 2 : Illustration du descripteur ob_flags

SELECTABLE — L'objet peut être sélectionné.

DEFAULT — Enfoncer la touche RETURN est équivalent à cliquer sur l'objet possédant cette marque (un seul objet par arbre peut contenir ce flag).

EXIT — Activer un objet possédant cette marque a pour effet de terminer la session de travail dans l'arbre contenant cet objet.

EDITABLE — Flag permettant de transformer un G_TEXT en un G_FTEXT (champ éditable) et un G_BOXTEXT en un G_FBOXTEXT. Cette marque est impérative pour un G_BUTTON.

RADIO-BUTTON — Sur les vieux téléviseurs, le bouton de la chaîne qui avait été enfoncé, ressortait lorsque l'on pressait sur le bouton d'une autre chaîne.

Ce flag sert à produire le même effet. Mais cela fonctionne uniquement pour les objets situés au même niveau de l'arborescence.

LASTOB — Cette marque devrait être activée pour le dernier objet d'un arbre (la plupart des éditeurs de ressources le font automatiquement). Nous vous déconseillons d'avoir pour dernier objet de l'arbre un champ éditable. Vous risqueriez de perdre le contrôle de votre curseur!

TOUCHEXIT — A un détail près, ce flag a la même fonction que la marque EXIT. La différence réside dans la façon dont il est activé. Un objet EXIT est activé en cliquant dessus. Un objet TOUCHEXIT est activé en passant dessus avec la souris dont le bouton gauche est déjà enfoncé.

HIDETREE — Un objet ainsi que tous ses enfants ne seront pas affichés par l'instruction OBJC_DRAW s'il possèdent cette marque.

INDIRECT — Si un objet possède cette marque, cela signifie que son ob_spec contient une adresse de structure. Si l'objet ne possède pas cette marque, l'ob_spec contient directement des informations (cas de la BOXINFO).

Les autres bits du ob_flags ne sont pas utilisés.



3. Fonction des bits du descripteur ob_state

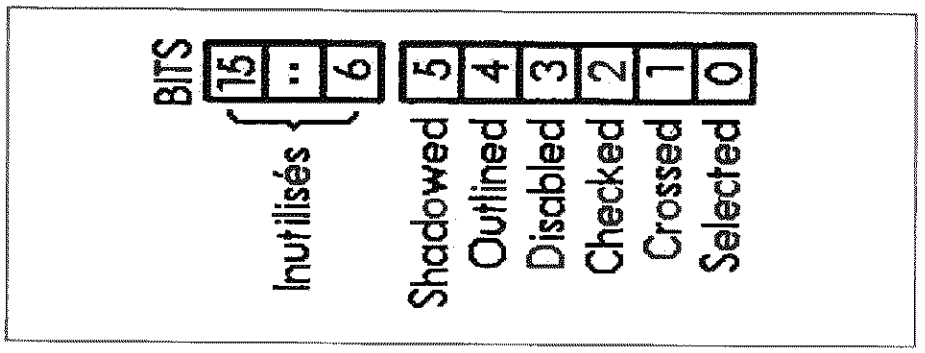


Fig. 3 : Illustration du descripteur ob_state

SELECTED — L'objet est soit pré-sélectionné avant la session de travail sur le formulaire, soit sélectionné au cours de cette session.

CROSSED — Les objets de type G_BOX, G_BOXTEXT, G_IBOX, G_BOXCHAR et G_FBOXTEXT sont biffés.

CHECKED — Un crochet est affiché à gauche de l'objet. Ce flag est traditionnellement utilisé dans les menus (c'est pour cette raison que le texte d'une entrée de menu doit toujours commencer par deux ou trois espaces).

DISABLED — L'objet possédant cette marque est affiché en grisé et ne peut pas être sélectionné, même si le flag SELECTABLE du ob_flags est activé.

OUTLINED — Les objets de type G_BOX, G_BOXTEXT, G_IBOX, G_BOXCHAR et G_FBOXTEXT possèdent un double cadre vers l'extérieur de l'objet.

SHADOWED — Le cadre des objets de type G_BOX, G_BOXTEXT, G_IBOX, G_BOXCHAR et G_FBOXTEXT possède une ombre portée dont la taille est fonction de l'épaisseur du cadre.

Comprendre comment s'organisent ces différentes structures n'est pas facile à la simple lecture de la liste des définitions figurant ci-dessus. Aussi nous proposons-vous un petit programme (un peu plus de 150 lignes) écrit en GFA qui permet de visualiser ces structures.

Auparavant, pour les lecteurs qui ne disposent pas encore d'un éditeur de ressources, nous vous fournissons un exemple simple de fichier ressources. Vous trouverez dans la suite le listing d'un premier programme capable de générer un fichier ressource (ESSAI.RSC) à partir de DATAs. Comme cette opération ne présente aucune difficulté, nous ne nous y attardons pas plus longtemps.

Une fois le fichier ressources créé, il sera possible de le décortiquer à l'aide du programme fourni dans le second listing. La fonction de ce second programme est d'isoler chaque objet afin de le raccrocher aux structures dont nous venons de parler. En fait ce programme ne s'intéresse qu'aux valeurs numériques que prennent les éléments de la structure principale. De plus, nous avons inclu une procédure d'affichage

pas à pas d'un arbre afin d'établir un lien entre les valeurs numériques des éléments de la structure principale et l'objet graphique.

Après avoir découvert tous les secrets du petit fichier ressource qui est fourni en guise d'exemple, vous pouvez vous "attaquer" aux fichiers ressources (.RSC) plus importants que l'on trouve habituellement avec les programmes exécutables. Un sélecteur de fichier situé au début du programme, vous permettra de fixer le nom du fichier ressources que vous souhaitez analyser.

Listing de MAKE_RSC permettant de générer un exemple de fichier ressources :

```
' ************************************************
' MAKE_RSC.GFA: Programme de génération d'un
' exemple de fichier ressources...
' ************************************************
' ------------------------------------------------
' Commencer par fixer (ci-dessous) le chemin
' d'enregistrement des deux fichiers qui
' seront crées...!!!
'
' La syntaxe est la suivante:
' exemple: chemin$="A:\DOSSIER1\DOSSIER2\"
' ------------------------------------------------
'
chemin$="E:\GFA\"        !    <= <= <= <= <=
'
test_data
gene_.rsc
gene_.b
PROCEDURE test_data
  DEFINT "a-z"
  RESTORE donnees_.rsc
  c=0
  FOR i=1 TO 297
    READ b
    ADD c,b
  NEXT i
  IF c<>3299063
    CLS
    PRINT "Erreur lors de la saisie des DATAs"
    PRINT "     (première série de DATAs)"
    PRINT
    PRINT "Tapez une touche..."
    ~INP(2)
    EDIT
  ENDIF
RETURN
PROCEDURE gene_.rsc
```



(suite)

```
  RESTORE donnees_.rsc
  txt$=""
  FOR i=1 TO 162
    READ valeur
    txt$=txt$+MKI$(valeur)
  NEXT i
  OPEN "o",#1,chemin$+"ESSAI.RSC"
  PRINT #1,txt$;
  CLOSE #1
RETURN
PROCEDURE gene_.b
  RESTORE donnees_.b
  txt$=""
  FOR i=1 TO 135
    READ valeur
    txt$=txt$+MKI$(valeur)
  NEXT i
  txt$=txt$+CHR$(26)
  OPEN "o",#1,chemin$+"ESSAI.B"
  PRINT #1,txt$;
  CLOSE #1
RETURN
PROCEDURE data
donnees_.rsc:
  DATA 0,176,120,120,120,0,36,120,0,320,6,1,2,0,0,0,0
  DATA 324,69,22597,19792,19525,8260,17696,17224,16717
  DATA 20512,17447,17732,18772,18767,20000,14880,24415
  DATA 24415,24415,57,14649,14649,14592,17752,17741,20556
  DATA 17696,17477,8258,20312,21573,22612,0,66,21844
  DATA 21583,20000,12544,16981,21588,20302,8242,0,0,36
  DATA 0,37,0,73,3,6,2,4481,0,65534,1,36,0,80,0,99,0
  DATA 100,3,6,2,4177,0,65534,19,1,65535,1,3,20,0,16
  DATA 2,4417,0,0,58,13,2,65535,65535,30,8,32,0,120,8
  DATA 6,42,2,3,65535,65535,22,7,32,0,148,8,9,42,2,0
  DATA 4,5,20,0,0,255,4369,8,1,42,4,5,65535,65535,26
  DATA 21,32,0,101,3,1,14,2,3,65535,65535,26,53,32,0
  DATA 110,25,1,14,2,0,176
donnees_.b:
  DATA 8224,8224,8224,21586,17733,12576,15648,12347,8224
  DATA 21061,19744,11565,11604,21061,17709,11565,3338
  DATA 8224,8224,8224,16981,21553,8253,8244,15136,8274
  DATA 17741,8237,11565,20290,19013,17236,8297,28192
  DATA 21586,17733,8227,12333,11565,3338,8224,8224,8224
  DATA 16981,21554,8253,8245,15136,8274,17741,8237,11565
  DATA 20290,19013,17236,8297,28192,21586,17733,8227
  DATA 12333,11565,3338,8224,8224,8224,16975,22594,20309
  DATA 21536,15648,13115,8224,21061,19744,11565,11599
  DATA 16970,17731,21536,26990,8276,21061,17696,9008
```

(suite)

```
  DATA 11565,11533,2592,8224,8224,8261,17481,21536,15648
  DATA 12603,8224,21061,19744,11565,11599,16970,17731
  DATA 21536,26990,8276,21061,17696,9008,11565,11533
  DATA 2592,8224,8224,8258,20312,21573,22612,8253,8242
  DATA 15136,8274,17741,8237,11565,20290,19013,17236
  DATA 8297,28192,21586,17733,8227,12333,11565,3338
RETURN
```

Listing de WORK_RSC permettant d'analyser et de visualiser les arbres d'un fichier ressources :

```
' ******************************************
' WORK_RSC.GFA: Programme d'analyse et de
' visualisation d'un fichier ressources
' ******************************************
init
detection
PROCEDURE init
  DEFINT "a-z"
  RESERVE FRE(0)-50000
  ON BREAK GOSUB fin
  CLS
  DIM txt$(2,33),tmp$(6)
  tmp$(0)=" "
  tmp$(1)="Struc. BOXINFO"
  tmp$(2)="Adr. struc. TEDINFO"
  tmp$(3)="Adr. struc. BITBLK"
  tmp$(4)="Adr. struc. USERBLK"
  tmp$(5)="Adr. chaine carac."
  tmp$(6)="Adr. struc. ICONBLK"
  RESTORE
  FOR i=19 TO 33
    READ txt1$
    READ txt2$
    txt$(1,i)=txt1$
    txt$(2,i)=tmp$(VAL(txt2$))
  NEXT i
  FILESELECT "E:\*.RSC","",fnom$
  '
  '  fnom$="E:\GFA\GFA_PROG\WEKA\RESOURCE\VISU_RSC\TEST1.RSC"
  '
  IF fnom$="" OR RIGHT$(fnom$,1)="\" OR RIGHT$(fnom$,3)<>"RSC"
    PRINT "ERREUR de FORMAT de FICHIER"
    PRINT "   --> Mauvais suffixe  ????"
    PRINT "tapez une touche"
    ~INP(2)
```



(suite)

```
    fin
  ENDIF
  ~RSRC_FREE()
  test_load=RSRC_LOAD(fnom$)
  IF test_load=0
    PRINT "ERREUR au CHARGEMENT"
    PRINT "    --> Fichier introuvable  ????"
    PRINT "    --> Mémoire insuffisante ????"
    PRINT
    PRINT "Si le .RSC que vous étudier est très grand (>20 Ko)"
    PRINT "Passez en edition manuelle (en pressant sur 'esc')"
    PRINT "et tapez ce qui est ci-dessous:"
    PRINT "     reserve fre()-50000"
    PRINT
    PRINT "tapez une touche"
    ~INP(2)
    fin
  ENDIF
  test_mem=RSRC_GADDR(0,0,adr_rsc)
  IF test_mem=0
    PRINT "ERREUR lors de la RECHERCHE DES ADRESSES"
    PRINT "    --> Mauvais  fichier Ressource ????"
    PRINT "tapez une touche"
    ~INP(2)
    fin
  ENDIF
RETURN
PROCEDURE detection
  num_objet=0
  flag!=FALSE
  CLS
  REPEAT
    affichage
    touch=INP(2)
    IF touch=8
      DEC num_objet
      num_objet=MAX(0,num_objet)
    ELSE IF touch=9
      CLS
      INPUT "Numéro de l'objet à atteindre (pas de valeur inconsidérée):
";num_move
      num_move=MAX(num_move,0)
      num_objet=num_move
    ELSE IF touch=226
      test_form
    ELSE IF touch=27
      flag!=TRUE
    ELSE
```

(suite)

```
      INC num_objet
    ENDIF
  UNTIL flag!
  fin
RETURN
PROCEDURE affichage
  CLS
  PRINT "------>  num_objet : ";num_objet;"  <------"
  adr_affi=ADD(adr_rsc,MUL(24,num_objet))
  PRINT "Adresse de la structure principale : ";adr_affi
  PRINT
  valeur=DPEEK(adr_affi)
  PRINT "   Numéro objet suivant       : ";valeur
  valeur=DPEEK(ADD(adr_affi,2))
  PRINT "   Numéro premier enfant      : ";valeur
  valeur=DPEEK(ADD(adr_affi,4))
  PRINT "   Numéro dernier enfant      : ";valeur
  valeur=DPEEK(ADD(adr_affi,6))
  typ_ob=valeur
  typ_ob=MAX(typ_ob,19)
  typ_ob=MIN(typ_ob,33)
  PRINT "   Type objet";TAB(30);": ";valeur;TAB(45);txt$(1,typ_ob)
  valeur=DPEEK(ADD(adr_affi,8))
  PRINT "   Info sur l'objet";TAB(30);": ";valeur;TAB(45);BIN$(valeur,16)
  valeur=DPEEK(ADD(adr_affi,10))
  PRINT "   Etat de l'objet";TAB(30);": ";valeur;TAB(45);BIN$(valeur,16)
  valeur=LPEEK(ADD(adr_affi,12))
  PRINT "   Info supplémentaires";TAB(30);": ";valeur;TAB(45);txt$(2,typ_ob)
  valeur=DPEEK(ADD(adr_affi,16))
  PRINT "   Pos coin sup. X (relatif)";TAB(30);": ";valeur
  valeur=DPEEK(ADD(adr_affi,18))
  PRINT "   Pos coin sup. Y (relatif)";TAB(30);": ";valeur
  valeur=DPEEK(ADD(adr_affi,20))
  PRINT "   Largeur";TAB(30);": ";valeur
  valeur=DPEEK(ADD(adr_affi,22))
  PRINT "   Hauteur";TAB(30);": ";valeur
  PRINT
  PRINT CHR$(27);"p";"tapez une touche pour continuer";CHR$(27);"q"
  PRINT
  PRINT "Backspace  : Retour à l'objet précedent"
  PRINT "Tab        : Aller à l'objet N° ?"
  PRINT "Help       : Test du formulaire"
  PRINT "Esc        : Quitter"
RETURN
PROCEDURE donnees
  DATA " ",0,G_BOX,1,G_TEXT,2,G_BOXTEXT,2,G_IMAGE,3,G_USERDEF,4
  DATA G_IBOX,1,G_BUTTON,5,G_BOXCHAR,1,G_STRING,5,G_FTEXT
  DATA 2,G_FBOXTEXT,2,G_ICON,6,G_TITLE,5," ",0
```



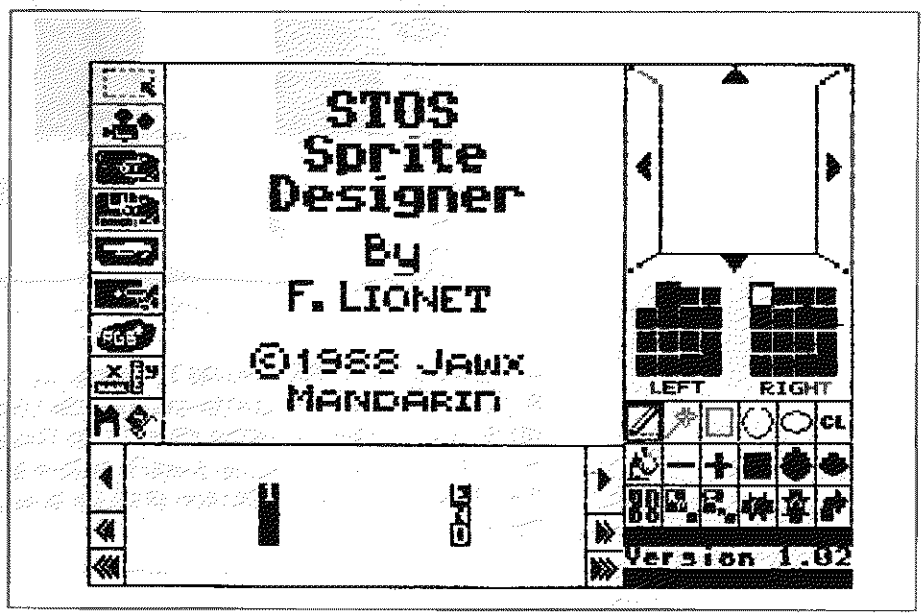
(suite)

```
RETURN
PROCEDURE test_form
  CLS
  HIDEM
  FOR i=num_objet TO 0 STEP -1
    adr_test=ADD(adr_rsc,MUL(24,i))
    IF DPEEK(adr_test)=65535
      num_affi_fich=i
      i=0
    ENDIF
  NEXT i
  ~FORM_CENTER(adr_test,x,y,l,h)
  num_affi_tree=0
  REPEAT
    test_affi=OBJC_DRAW(adr_test,num_affi_tree,0,x,y,l,h)
    adr_test1=ADD(adr_test,ADD(8,MUL(24,num_affi_tree)))
    IF DPEEK(adr_test1) AND 32
      LOCATE 25,23
      PRINT CHR$(27);"p";"Objet N°";RIGHT$("   "+STR$(num_affi_tree),3);"
(arbre)";RIGHT$("   "+STR$(num_affi_fich),3);" (fich.) ";CHR$(27);"q"
      LOCATE 25,24
      PRINT CHR$(27);"p";"Affichage de l'arbre terminé...";CHR$(27);"q"
      LOCATE 25,25
      PRINT CHR$(27);"p";"  Tapez une touche pour finir  ";CHR$(27);"q";
      ~INP(2)
      touch=27
    ELSE
      LOCATE 25,23
      PRINT CHR$(27);"p";"Objet N°";RIGHT$("   "+STR$(num_affi_tree),3);"
(arbre)";RIGHT$("   "+STR$(num_affi_fich),3);" (fich.) ";CHR$(27);"q"
      LOCATE 25,24
      PRINT CHR$(27);"p";"Tapez une touche pour continuer";CHR$(27);"q"
      LOCATE 25,25
      PRINT CHR$(27);"p";"     Esc          : Quitter     ";CHR$(27);"q";
      touch=INP(2)
      INC num_affi_tree
      INC num_affi_fich
    ENDIF
  UNTIL touch=27
RETURN
PROCEDURE fin
  ~MFREE(adr_rsc)
  EDIT
RETURN
```

4/2.3.3

Utilisation de l'éditeur de sprites (accessoires SPRITE)

Chargez l'accessoire SPRITE en procédant comme il a été dit plus haut et activez l'éditeur de sprites en appuyant sur les touches <Help> puis <F1>. L'écran suivant apparaît au bout de quelques secondes :



Appuyez sur une des touches de la souris. Le titre affiché au centre de l'écran s'efface et laisse la place à un crayon. C'est grâce à ce crayon que les sprites seront définis dans la fenêtre centrale de l'écran.

Sur l'écran, vous remarquez deux séries d'icônes, à gauche et à droite.

Nous utiliserons seulement quelques icônes parmi ceux affichés. Nous en parlerons en tant voulu. Pour l'instant, sachez simplement que :

— les icônes de droite permettent de choisir la fonction courante (tracé de points, de lignes, de rectangles, remplissage, etc.) ;

— les icônes de gauche permettent de charger et de sauvegarder des banques de sprites, de modifier la dimension des sprites, de redéfinir leurs couleurs, etc.

Le jeu DANGER BACTERIES utilise 4 sprites représentés dans 15 images. Nous allons les définir pas à pas.



4

Sommaire

4/1 La programmation structurée
4/1.1 Notions d'algorithmique
I. Définitions
II. Les structures de l'algorithme
III. L'algorigramme
IV. Représentation des structures algorithmiques complexes
V. Algorithmique et langages
4/1.2 Applications algorithmiques
4/1.2.1 Racine carrée entière
4/1.2.2 Nombres premiers
4/1.2.3 La récursivité

4/2 Les Basic sur Atari ST
4/2.1 Le ST Basic
I. Le lancement du ST Basic
II. Les fenêtres du ST Basic
III. Les menus du ST Basic
IV. Les instructions du ST Basic
4/2.2 Le Basic GFA
I. Historique et particularités
II. Le Basic GFA et la récursivité
III. L'implantation de routines assembleur dans le GFA (les liaisons dangereuses)
IV. Liste des mots clés du GFA Basic 2.02
V. Deux exemples de programme
A. Un solitaire (version monochrome)
B. Un simulateur d'explosions
4/2.2.1 Le Basic GFA 3.0
I. Les instructions du GFA Basic 3.0 en détail et leur utilisation
4/2.3 Le Basic STOS
4/2.3.1 L'éditeur du STOS - Généralités
4/2.3.2 Utilisation de l'éditeur de musique (MUSIC)
4/2.3.3 Utilisation de l'éditeur de sprites (SPRITE)
4/2.3.4 Le programme DANGER BACTERIES

4/2.4 Le Basic OMIKRON
4/2.4.1 Utilitaires OMIKRON
4/3 L'assembleur
I Introduction
II. La structure d'un ordinateur
III. La représentation des données en mémoire
IV. Le microprocesseur MC 68000
A. L'architecture interne du 68000
B. Les codes opératoires du 68000
4/3.1 Aide-mémoire des codes opératoires du 68000
4/3.2 L'assembleur par la pratique
4/3.2.1 Le fonctionnement de l'instruction MOVE
4/3.2.2 Le fonctionnement d'une pile

4/4 Le langage C
I. Origines et caractéristiques
II. Les types de données
III. Les expressions, variables et opérateurs
IV. La structure des programmes
V. Les pointeurs et les tableaux
VI. Les structures
4/4.1 Les compilateurs présents sur le marché
4/4.2 Le langage C par les exemples de programmation
4/4.3 Exemple de programme en C
4/4.3.1 Exemple de programme en C (suite)
4.4.3.2 Génération de motifs
4/4.3.3 Jeux musicaux en C
4/4.4 Utilitaires de programmation en C

4/5 Le langage Pascal

4/6 Le langage Logo

4/1

La programmation structurée

Ce chapitre s'adresse à tous les programmeurs (débutants ou confirmés) qui désirent être plus performants dans la conception puis la réalisation de leurs logiciels en employant une méthode simple et efficace...

Avant de parler de la méthode à mettre en œuvre, remarquons que l'évolution des langages de programmation a suivi de près celle des ordinateurs. Il y a une vingtaine d'années seulement, les informaticiens ne disposaient que du langage machine pour réaliser des programmes longs et souvent difficiles à mettre au point et à maintenir.

Mais ces temps héroïques sont bien lointains, et le programmeur moderne dispose d'outils qui lui facilitent grandement la tâche.

Ces outils sont de deux natures différentes :

— des méthodologies qui visent à optimiser les temps de conception/réalisation/maintenance des logiciels. Nous allons en présenter une dans ce chapitre ;

— des langages évolués qui permettent de manipuler des entités complexes à moindres frais.

Comment vous y prenez-vous lorsque vous décidez d'écrire un programme ?

— vous connectez l'ordinateur et vous commencez à taper ;

— vous réfléchissez d'abord au problème que vous voulez résoudre, puis vous passez à la programmation.

La première méthode peut certainement convenir si le programme à écrire est court et/ou simple. Mais, même dans ce cas, nous vous la déconseillons. A moins que vous n'ayez une grande habitude du langage de programmation et une connaissance à priori du problème que vous voulez résoudre, vous risquez :

— d'aligner des instructions de manière non optimale et désordonnée ;

— de perdre un temps non négligeable dans la phase de mise au point.

La deuxième méthode est de loin la meilleure. C'est celle dont nous allons parler dans la suite de l'exposé. Elle consiste à diviser la résolution du problème en quatre phases consécutives :

1) analyse du problème ;

2) recherche d'une structure ;

3) programmation ;

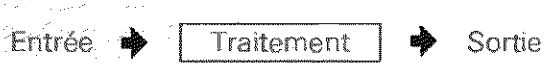
4) mise au point.

ANALYSE DU PROBLÈME

Le problème à résoudre doit être bien défini avant d'entamer la phase de programmation. C'est le point le plus important (et souvent, le plus négligé) de l'analyse.

En effet, comment prétendre écrire un programme bien organisé si l'on n'a pas, dès le départ, formalisé toutes les entrées, finalités et traitements ?

Une bonne méthode consiste à associer une boîte noire au problème à traiter. Dans cette boîte noire entrent et sortent des informations. L'intérieur de la boîte noire représente les traitements à appliquer aux entrées pour les transformer en sorties désirées.



Posez-vous les questions suivantes :

— quelles données dois-je entrer pour obtenir les sorties qui m'intéressent ?

— quels traitements appliquer aux entrées pour les transformer en résultats, en sortie du programme ?

RECHERCHE D'UNE STRUCTURE

La première phase définie, vous devez être en mesure d'énumérer :

— les entrées ;

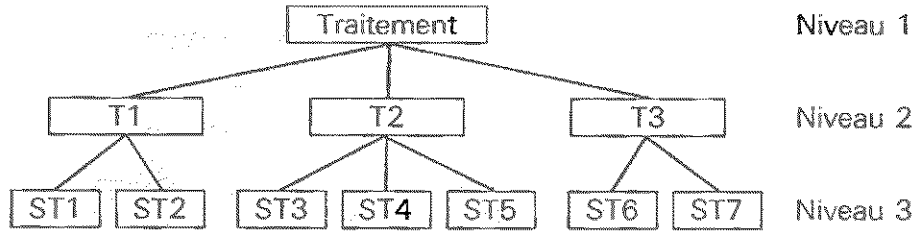
— les traitements ;

— les sorties.

Dans la mesure du possible, essayez de décomposer les traitements en sous-traitements plus élémentaires et indépendants. De même, décomposez les sous-traitements en tâches indépendantes et encore plus élémentaires, et ainsi de suite, pour arriver à de petits modules qui comportent seulement quelques dizaines d'instructions. Cette manière de décomposer un problème en sous-problèmes porte le nom d'*Analyse descendante*. Cette méthode a pour avantage d'obliger le concepteur

à se poser les bonnes questions dès le départ de l'analyse. Pour arriver à décomposer le problème en tâches de quelques dizaines d'instructions, il est effectivement nécessaire de l'avoir clairement défini.

Le traitement se décompose donc en une série de modules de niveaux hiérarchiques de plus en plus faibles et réalisant de moins en moins d'opérations.



Remarques :

- Essayez de limiter la décomposition au niveau 3.

Cette remarque se justifie pour de petits programmes. Descendez éventuellement au niveau 4 si le programme est très complexe, mais cela risque d'être inutile, et même de nuire à la bonne compréhension du fonctionnement du programme.

- Essayez de ne pas avoir plus de 5 sous-modules par module.

En effet, il est difficile de mémoriser plus de cinq concepts en même temps. Quand vous êtes à un niveau hiérarchique donné, pour un module donné, vous devez être capable d'avoir en tête les N sous-modules qui vont le composer.

Pour être sûr de ne rien oublier de la décomposition, limitez-vous à trois ou quatre sous-modules par module, et, au besoin, descendez d'un niveau supplémentaire si un ou plusieurs des sous-modules vous semble(nt) trop complexe(s).

PROGRAMMATION

Cette phase doit être quasi immédiate si vous avez respecté les consignes données dans les deux phases précédentes. A titre indicatif, elle ne devrait pas dépasser plus de 30 % du temps total que vous passerez à la totalité des trois étapes (définition, programmation, test).

Les modules et sous-modules seront organisés en procédures et/ou fonctions, et le programme principal fera appel aux procédures d'entrée, de traitement et de sortie.

A titre indicatif, prenons l'exemple suivant : tracé d'une courbe d'équation $Y = f(X)$.

L'analyse du problème nous montre que les entrées sont :

— domaine d'étude,

— équation de la courbe,

La sortie est :

— affichage de la courbe.

Le traitement est :

— calcul de Y en fonction de X dans le domaine d'étude,

— mémorisation du calcul.

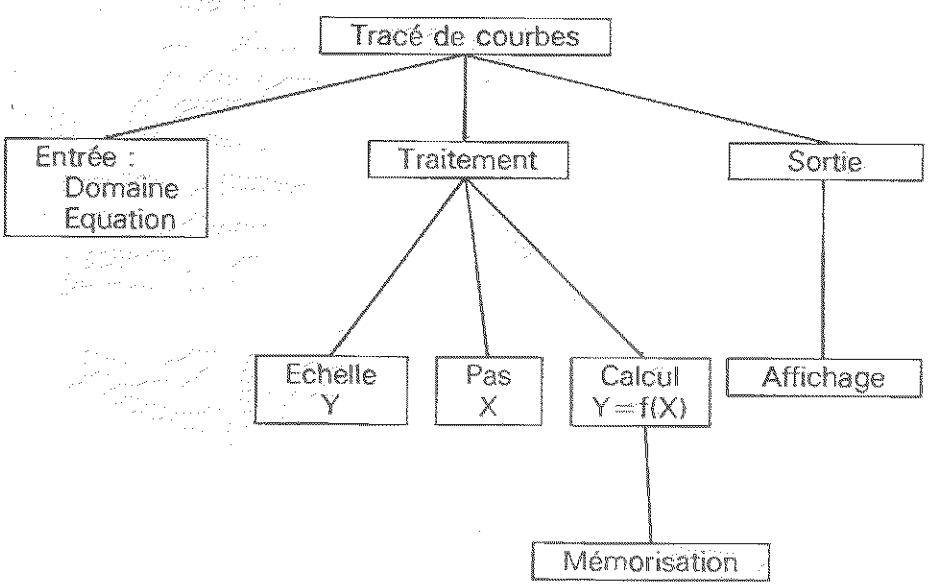
Si nous poursuivons l'analyse, le traitement peut se décomposer en :

— recherche de l'échelle en Y,

— recherche du pas du tracé,

— calcul et mémorisation des coordonnées Y en fonction de X sur le domaine donné.

Ce qui se représente par le graphe structuré suivant :





Le programme (écrit en ST BASIC) qui en résulte a la structure suivante :

```
10 REM Trace de courbes
20 REM
30 GOSUB 1000 : REM Entree
40 GOSUB 2000 : REM Traitement
50 GOSUB 3000 : REM Fin
60 REM
70 END
80 REM ------------------------------------------------------------
1000 REM Entree
1010 REM
1020 REM Saisie de l'equation et du domaine de definition
1030 REM
1040 RETURN
1050 REM ------------------------------------------------------------
2000 REM Traitement
2010 REM
2020 GOSUB 4000 : REM Calcul de l'echelle en Y
2030 GOSUB 5000 : REM Calcul du pas en X
2040 FOR I=1 TO NBPTS STEP PAS
2050   GOSUB 6000 : REM Memorisation des coordonnees
2060 NEXT I
2070 REM
2080 RETURN
2090 REM ------------------------------------------------------------
3000 REM Sortie
3010 REM
3020 REM Affichage de la courbe en lisant les donnees
3030 REM qui se trouvent dans le tableau Y
3040 REM
3050 RETURN
3060 REM ------------------------------------------------------------
4000 REM Echelle en Y
4010 REM
4020 REM Recherche de l'echelle
4030 REM
4040 RETURN
4050 REM ------------------------------------------------------------
5000 REM Pas en X
5010 REM
5020 REM Calcul du pas
5030 REM
5040 RETURN
5050 REM ------------------------------------------------------------
6000 REM Memorisation d'un point
6010 REM
6020 REM Memorisation de Y=f(X) dans le tableau Y
6030 REM
6040 RETURN
```

MISE AU POINT

Cette partie est quasi inexistante si vous avez bien suivi les conseils donnés précédemment. Si vous n'obtenez pas les résultats escomptés, déterminez dans quelle branche se trouve le problème, et descendez de niveau hiérarchique jusqu'à trouver le module ou sous-module responsable du mauvais fonctionnement.

4/1.1

Notions d'algorithmique

Il est admis, par la majorité des informaticiens, que la programmation, quel que soit le langage utilisé, recquiert une certaine rigueur.

Nous vous avons déjà évoqué (Partie 4, chapitre 1, pages 1 à 6) une des bases de la programmation structurée dans le traitement de cas qui s'approche de près à la structure arborescente.

Chacun aura remarqué que la structure arborescente n'est pas suffisante pour traiter tous les cas pouvant se produire, nous aurons ainsi besoin de structures complètes et bien définies, structures qui constituent le fondement de l'algorithmique, et que l'on décompose en deux thèmes :

— l'algorithme
— l'algorigramme

I - Définitions

L'ALGORITHME

Beaucoup s'entendent sur la définition de l'algorithme telle que nous vous la proposons ici :

Inventaire et mise en forme d'un ensemble fini d'actions permettant d'accomplir une tâche que l'on s'est fixée d'avance.

Définition que l'on pourrait encore formuler comme :

Une suite ordonnée d'opérations permettant d'obtenir un résultat déterminé.

La plupart des algorithmes utilisés par l'analyste informaticien sont associés à des programmes, mais rien n'empêche d'utiliser les structures que nous vous soumettons pour décrire d'autres fonctionnements. La recette de cuisine en est un exemple flagrant ; les électroniciens utilisent, de plus en plus, ce qu'ils nomment, un peu abusivement, des logigrammes, notamment pour sélectionner les différentes actions à effectuer lors d'un dépannage.

Dans notre cas, nous retiendrons que l'algorithme permet de décomposer un programme en actions élémentaires, d'introduire une logique dans

l'enchaînement de ces actions, ceci afin de proposer une solution pour résoudre un problème donné.

L'ALGORIGRAMME

Un algorigramme n'est fait que de la représentation graphique d'un algorithme, ce qui donne une vision d'ensemble plus rapide.

Pour la réalisation d'un algorigramme, les programmeurs ont à leur disposition des symboles graphiques qui proviennent de la norme AFNOR Z67-010.

L'ORDINOGRAMME

Devant le choix qui se présente au programmeur parmi tous les langages, il a été admis que l'on pourrait dissocier deux types de représentations graphiques de l'algorithme.

Nous utiliserons ainsi l'algorigramme pour expliciter le traitement dans sa généralité, cet algorigramme étant ensuite représenté, cette fois en utilisant les termes du langage dans lequel nous travaillerons.

Il sera ainsi possible de transposer un algorigramme donné, donc une solution retenue, dans différents langages, ce qui nous donnera des algorigrammes spécifiques, que nous appellerons ordinogrammes.

II - Les structures de l'algorithme

TOUT ALGORITHME A UN DÉBUT ET UNE FIN

D'après la définition, l'algorithme étant un ensemble fini d'actions, nous sommes dans l'obligation d'introduire les termes de DEBUT et FIN que nous présenterons ainsi :

— DEBUT

  — traitements ou opérations à effectuer.

— FIN

Pour plus de clarté, nous nous efforcerons de présenter en MAJUSCULES ET SOULIGNÉS tous les termes de l'algorithmique.

LA SÉQUENCE

La séquence est une structure linéaire de traitements ou d'opérations à effectuer, qui autorise donc un enchaînement séquenciel.



On pourrait noter les actions ainsi :

"effectuer le traitement numéro 1, puis effectuer le traitement numéro 2, puis le traitement numéro 3, puis ..., puis le traitement numéro n"

ce qui deviendra en algorithme :

- DEBUT
  - effectuer le traitement 1
  - effectuer le traitement 2
  - effectuer le traitement 3
  - ...
  - effectuer le traitement n
- FIN

On remarquera ici l'utilisation de l'indentation (décalage à droite) et le soulignement des verbes d'action (en minuscules), "effectuer", qui pourront être tout autre verbe selon le traitement à effectuer (additionner A et B, lire la touche frappée au clavier, afficher le résultat, ...).

LA SÉLECTION

La structure de sélection est ce que l'on appellera une structure ALTERNATIVE qui, comme son nom nous l'indique, permettra d'effectuer des choix. On la trouvera sous trois formes :

- la sélection simple générale,
- la sélection simple simplifiée,
- la sélection multiple ou structure de cas.

La sélection simple générale

Son utilisation correspond à la phrase suivante :

"si la condition fixée est vraie, alors il faut effectuer le traitement numéro 1, sinon il faut effectuer le traitement numéro 2, ..."

La forme algorithmique correspondante sera :

- SI condition vraie
  - ALORS
    - effectuer le traitement numéro 1
  - SINON
    - effectuer le traitement numéro 2
- FIN SI

On retiendra une fois encore l'indentation et le terme **FIN SI** signalant la fin de la structure alternative, sachant que les traitements 1 et 2 peuvent être composés d'autres structures de l'algorithmique.

La sélection simple simplifiée

Il arrive que, suite au test sur la condition, il n'y ait aucune action à effectuer dans le cas où la réponse est négative, nous sommes dans le cas de la sélection simple simplifiée que l'on décrit sous la forme :

```
— SI condition vraie
    — ALORS
        — effectuer le traitement Untel
— FIN SI
```

Le fait de supprimer le traitement dans la négative nous permet d'omettre le terme **SINON** et d'alléger l'algorithme.

La sélection multiple

La sélection multiple autorise des choix multiples, selon des cas bien précis, d'où son appellation : structure de cas.

Son utilisation correspond à la phrase suivante :

''si le cas numéro 1 se présente, donc répond à la condition numéro 1, alors effectuer le traitement numéro 1, sinon, étudier le cas numéro 2, si celui-ci se présente, alors effectuer le traitement numéro 2, sinon étudier..., sinon étudier le cas numéro n, si celui-ci se présente, alors effectuer le traitement numéro n, sinon effectuer le traitement numéro n + 1''.

Nous retiendrons la forme algorithmique, certainement plus lisible :

```
— SELON CAS
    — cas 1 : effectuer le traitement numéro 1
    — cas 2 : effectuer le traitement numéro 2
    — cas 3 : effectuer le traitement numéro 3
    — ...
    — cas n : effectuer le traitement numéro n
    — AUTREMENT
        — effectuer le traitement numéro n + 1
FIN CAS
```



LES RÉPÉTITIONS

Les structures de répétition, encore appelées structures itératives permettent de répéter des traitements.

Il existe deux types de structures itératives :

— la structure itérative contrôlée, encore appelée structure itérative arithmétique car elle fait intervenir une valeur numérique qui permettra de répéter le traitement un nombre de fois connu à l'avance ;

— la structure itérative non contrôlée, dans laquelle le nombre de fois qu'il faudra répéter le traitement est inconnu par avance et où seul un événement permettra de suspendre l'itération des opérations et de sortir ainsi de la boucle. On trouve aussi le nom de structure itérative logique, et il en existe trois formes algorithmiques : la structure itérative générale et deux cas particuliers très courants.

L'itération arithmétique

Enoncée sous la forme :

"pour I variant d'une valeur de départ à une valeur d'arrivée, par pas déterminé, effectuer le traitement Untel, ensuite...".

On s'aperçoit que cette structure est une structure de boucle évoluée qui va donc se répéter un nombre limité de fois, déterminé au préalable, et qu'un comptage sera fait. La façon de compter est déterminée par le terme « par pas déterminé », il est ainsi possible, par exemple, de compter de deux en deux, de cinq en cinq, ou même par valeurs négatives. Il est souvent admis, lorsque l'incrémentation s'effectue au pas de 1, de négliger le terme "par pas ...".

La forme algorithmique générale sera :

— POUR variable DEPUIS valeur initiale JUSQU'A valeur finale par PAS DE incrément

  — effectuer le traitement

— FIN POUR

Cette forme devient, lorsque l'incrémentation s'effectue de un en un :

— POUR variable DEPUIS valeur initiale JUSQU'A valeur finale

  — effectuer le traitement

— FIN POUR

L'itération logique sous sa forme générale

"effectuer le traitement numéro 1, si la condition du test est vérifiée, alors sortir de la boucle et effectuer la suite, sinon effectuer le traitement numéro 2, puis recommencer au traitement numéro 1."

La forme algorithmique pour cette phrase devient :

— ITERER

  — effectuer le traitement numéro 1

  — SORTIR SI la condition est vraie

  — effectuer le traitement numéro 2

— FIN ITERER

Comme nous l'avons annoncé précédemment, deux cas particuliers très courants peuvent se présenter :

— le traitement numéro 1 n'est pas présent, nous obtenons alors la forme d'itération TANT QUE ... FIN TANT QUE

— le traitement numéro 2 n'est pas présent, nous obtenons alors la forme d'itération REPETER ... JUSQU'A.

TANT QUE ... FIN TANT QUE

Lorsque le traitement numéro 1 est omis, l'algorithme devient alors :

''tant que la condition du test est fausse, effectuer le traitement (numéro 2), ensuite (sinon)...''

qui devient sous sa forme algorithmique :

— TANT QUE la condition est fausse

  — effectuer le traitement (numéro 1)

— FIN TANT QUE

On remarquera qu'il est possible, sous cette structure, de sortir de la boucle immédiatement, donc de ne pas effectuer le traitement : il suffit pour cela que la condition soit vraie dès l'entrée dans la structure.

REPETER ... JUSQU'A

Lorsque le traitement numéro 2 est omis, l'algorithme devient :

''répéter l'opération (numéro 1) jusqu'à ce que la condition du test soit vraie, ensuite...''

la forme algorithmique sera :

— REPETER

  — effectuer le traitement (numéro 1)

— JUSQU'A condition vraie

On remarquera cette fois-ci que le traitement est effectué au moins une fois car le test sur la condition est effectué en fin de la structure.



LES PROCÉDURES

Lors de l'étude de programmes relativement longs, les analystes informaticiens ne se lancent que très rarement dans l'écriture totale de l'algorithme du programme qu'il vont fournir.

Le travail est découpé en « morceaux » que bien souvent plusieurs analystes se partagent, chacun décrivant des morceaux d'algorithmes qui seront ensuite concaténés pour produire l'algorithme total, appelant chacun des algorithmes.

Il apparaît aussi des portions d'algorithmes qui pourront être utilisés plusieurs fois dans un algorithme complet (ce qui correspond aux sous-programmes ou aux procédures selon les langages).

Ces deux besoins ont appelé la notion de procédure algorithmique, qui facilite grandement la lisibilité des longs algorithmes. Chaque procédure ainsi créée portera un nom qui permettra de l'identifier et de l'appeler depuis l'algorithme principal, ou pourquoi pas d'une autre procédure, et commencera toujours par les mots réservés DEBUT et FIN.

L'algorithme dans son étude complète comportera donc l'algorithme principal suivant :

- DEBUT : ALGORITHME PRINCIPAL
  - Effectuer le traitement numéro 1
  - .....
  - PROCEDURE numéro 1
  - .....
  - effectuer les traitements suivants
  - .....
  - PROCEDURE numéro 2
  - .....
- FIN

Il faut savoir que les différents traitements peuvent s'intercaler, que le nombre de procédures n'est pas limité et que l'appel d'une procédure peut se trouver dans une structure algorithmique complexe.

On trouvera ensuite, sur des feuillets séparés les descriptions des algorithmes de chacune des procédures sous les formes suivantes :

- DEBUT : ALGORITHME DE LA PROCEDURE 1
  - .....
  - effectuer les traitements afférents à la procédure numéro 1
  - .....
- FIN

— DEBUT : ALGORITHME DE LA PROCEDURE 2
— .....
— effectuer les traitements afférents à la procédure numéro 2
— .....
— FIN

Le parcours de l'algorithme principal s'effectuera ainsi dans l'ordre suivant :

a/ réalisation des traitements précédant l'appel de la procédure numéro 1

b/ appel de la procédure 1

c/ réalisation des traitements de la procédure 1

d/ retour de l'algorithme principal à la ligne juste inférieure à l'appel de la procédure 1

e/ réalisation des traitements éventuels suivants

f/ appel de la procédure 2

g/ réalisation des traitements de la procédure 2

h/ retour à l'algorithme principal à la ligne juste inférieure à l'appel de la procédure 2

i/ ... etc ...

f/ jusque la fin de l'algorithme

LES COMMENTAIRES

La structure de ce que nous appellerons le « champ commentaires » n'est pas à proprement parlé une structure de l'algorithme, car elle n'effectue aucun traitement. Elle n'a pour raison d'exister que celle d'ajouter des explications au déroulement ou au pourquoi d'un traitement. Il est aussi possible d'introduire dans ce champ toute autre remarque, telle que la date ou le nom de l'analyste chargé de l'étude, car elle n'influence en aucun cas le parcours de l'algorithme.

La forme algorithmique retenue par beaucoup est :

— CO
je place ici mon commentaire
— FIN CO

On effectuera une indentation importante pour distinguer le commentaire des différents traitements.



CONSIDÉRATIONS D'ORDRE GÉNÉRAL

Nous résumons ici, dans un souci d'homogénéité entre analystes, ce qui est souvent accepté, en rappelant tout de même que les structures vues précédemment, même si elles sont admises par la plupart des gens de la profession informatique, et se retrouvent dans beaucoup de langages, ne font actuellement l'objet d'aucune norme officielle.

Donc :

— utiliser exclusivement les structures présentées,

— respecter la mise en MAJUSCULES et le soulignement des termes de l'algorithmique,

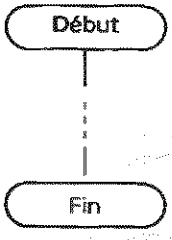
— utiliser et souligner des verbes d'action, placés en minuscules, pour les traitements,

— respecter une indentation visible dans les structures, et conserver celle-ci lors des changements de pages.

III - L'algorigramme

Nous ne distinguerons pas ici les termes d'algorigramme et d'ordinogramme, car ces deux termes, utilisés pour la représentation des algorithmes, font référence aux mêmes symboles graphiques, issus de la norme AFNOR.

Ces mêmes symboles sont utilisés par ce que certains appellent des logigrammes ou encore organigrammes.



Symboles de DEBUT et de FIN

LES SYMBOLES DE DEBUT ET DE FIN

Une seule forme graphique pour le début et la fin d'un algorithme, dans lequel on placera ces termes. Seule la flèche de continuité, unique sur le symbole, se trouve placée différemment : elle démarre du symbole et se dirige vers le bas pour le DEBUT, elle arrive sur le symbole pour la FIN.

On trouve parfois, à la place du terme DEBUT, le nom que l'on a pu donner à l'algorithme, ou encore le nom de la procédure concernée.

LES SYMBOLES DE TRAITEMENT

Quatre symboles principaux permettent d'effectuer des traitements : un symbole général, un symbole d'entrée ou sortie, un symbole indiquant la réalisation d'une opération manuelle de la part de l'utilisateur, et un symbole de préparation ou d'organisation.

Le symbole général

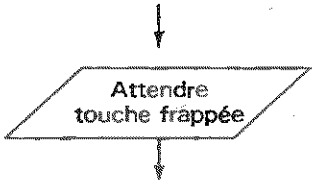


Représenté sous la forme d'un rectangle, il possède une entrée, toujours située en haut et une sortie, toujours située vers le bas

Ce symbole peut être utilisé à la place des autres symboles de traitement, si la phrase placée à l'intérieur est suffisamment précise pour décrire le traitement.

Il est bien sûr possible de placer plusieurs symboles de ce type l'un à la suite de l'autre, dans le cas de la structure séquencielle.

Le symbole d'entrée ou sortie



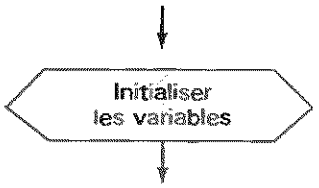
Nous avons à notre disposition un symbole unique pour représenter les entrées ou les sorties vers le monde extérieur (par exemple un affichage sur écran ou la lecture d'un fichier sur disquette). Il suffira de placer dans ce symbole une phrase suffisamment précise indiquant le sens de transfert.

Le symbole d'opération manuelle



Ce symbole, tel qu'il est représenté ci-contre, permettra d'insérer dans un algorigramme, le besoin d'une opération manuelle pour passer à la suite de son déroulement. Ce symbole pourra, par exemple, être utilisé pour un changement de disquette, ou l'allumage d'un périphérique oublié, voire l'attente de la frappe d'une touche au clavier.

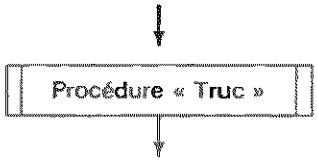
Le symbole de préparation ou organisation



Ce symbole permet de signaler dans le déroulement de l'algorigramme, la préparation ou l'organisation, par exemple de variables. Toute définition ou initialisation de variables numériques ou alphanumériques devrait être placée dans ce symbole.

Si vous avez eu l'occasion de lire certains algorigrammes, et bien souvent ceux des professionnels hélas, ce symbole est abusivement utilisé en lieu et place du symbole de branchement, prétextant un manque de place dans ce dernier. Nous vous donnerons un conseil pour éviter de corrompre la norme de cette façon, dans le paragraphe concernant le branchement.

LE SYMBOLE DE SOUS-PROGRAMME

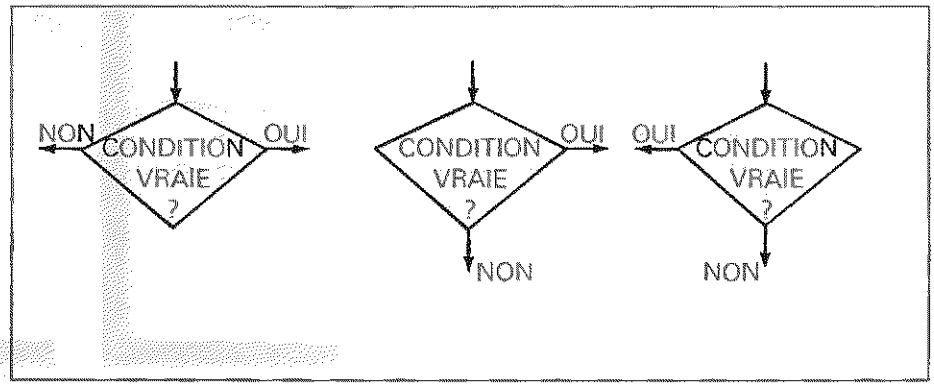


Lors de l'utilisation d'une procédure, on utilisera le symbole associé ci-contre. L'algorithme décrivant cette procédure se trouvera sur un ou plusieurs feuillets, placés en ajout, et qui comportera les symboles de DEBUT et de FIN.

Le symbole de branchement

Le symbole de branchement est le seul symbole de la norme AFNOR Z67-010 possédant une entrée et deux sorties. L'entrée, comme pour tous les symboles, s'effectuant par le haut, les sorties étant disposées au choix de l'utilisateur sur les trois emplacements restant (en général, selon la forme de l'algorigramme), mais jamais par le haut. Les réponses (au nombre de deux) à la condition soumise dans le symbole, peuvent être disposées indifféremment à droite, gauche ou en bas.

Ces réponses sont obligatoirement binaires : OUI ou NON.



Les trois représentations du symbole de branchement.

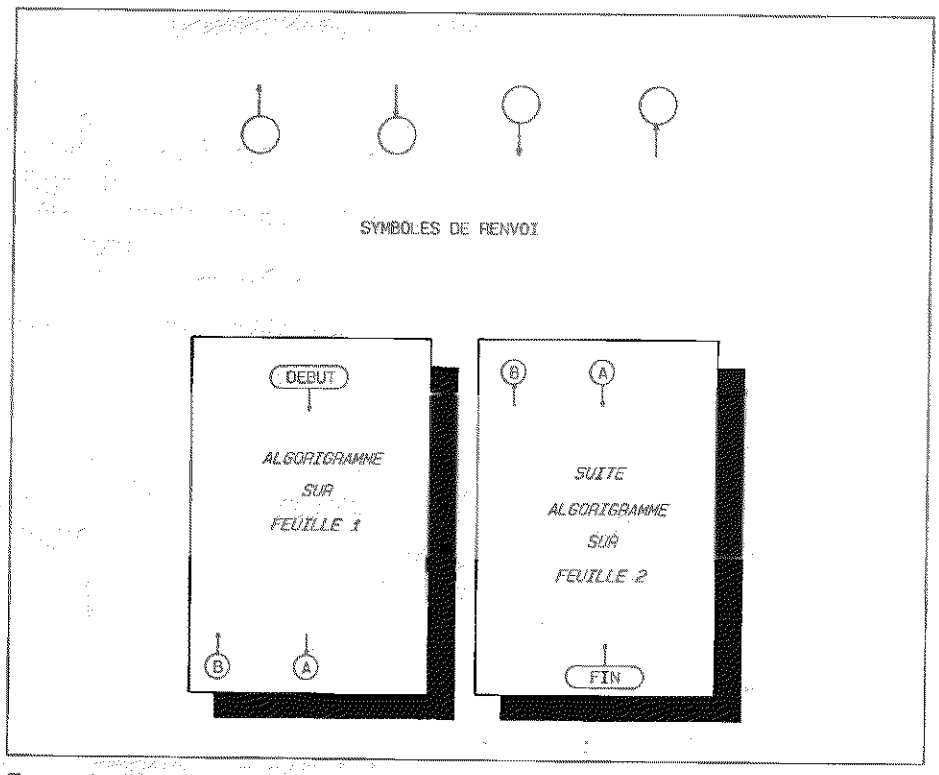
Il peut apparaître que la question posée nécessite plusieurs termes et que leur concaténation ne puisse pas s'insérer dans le symbole sous peine d'agrandir celui-ci considérablement, au détriment de la forme générale de l'algorigramme. On utilisera dans ce cas, uniquement le point d'interrogation, et à l'aide de la représentation du commentaire, la question sera posée en entier.

Les renvois

Vos algorigrammes seront en général d'une dimension plus qu'honorable, et nécessiteront certainement l'emploi de plusieurs feuilles de papier.

Or, si vous utilisez plusieurs fois le symbole de branchement, le nombre de flèches de liaison risque de s'accroître et de ne plus permettre de reconnaître la continuité entre les feuilles.

Aussi, il existe des symboles de renvoi, qui sont des cercles sur lesquels arrivent ou partent des flèches, selon leur emplacement et le sens de parcours. On placera dans ces cercles un repère.



Exemple d'utilisation des symboles de renvoi.

Remarques :

— Il est préférable de placer ces symboles uniquement en haut ou en bas des feuilles, pour éviter d'avoir à les rechercher dans l'algorigramme.

— Lorsque l'utilisation d'un renvoi doit permettre un traitement sur un feuillet qui ne succède pas directement à celui où il est utilisé, il est préférable d'assurer la continuité sur toutes les feuilles intermédiaires ; on pourra reprendre le même repère pour une même continuité.

LE COMMENTAIRE

Comme pour les algorithmes, les algorigrammes peuvent nécessiter des commentaires, qui seront placés sur les côtés, de préférence à droite, à l'aide du symbole représenté ci-contre.

Ici sera noté
le commentaire
concernant
le traitement

Symbole de commentaire

CONSEILS POUR LA CONSTRUCTION DE L'ALGORIGRAMME

La construction d'un algorigramme doit obéir à certaines règles pour permettre une lecture et un déroulement sans erreur, et une représentation correcte de l'algorithme :

— centrer l'algorigramme sur la feuille de papier,



— construire l'algorigramme de façon à ce que la lecture se déroule le plus verticalement possible,

— ne pas croiser les lignes de liaisons entre les symboles,

— toujours faire arriver une ligne de liaison sur le haut d'un symbole, et repartir par le bas (sauf cas particulier du branchement),

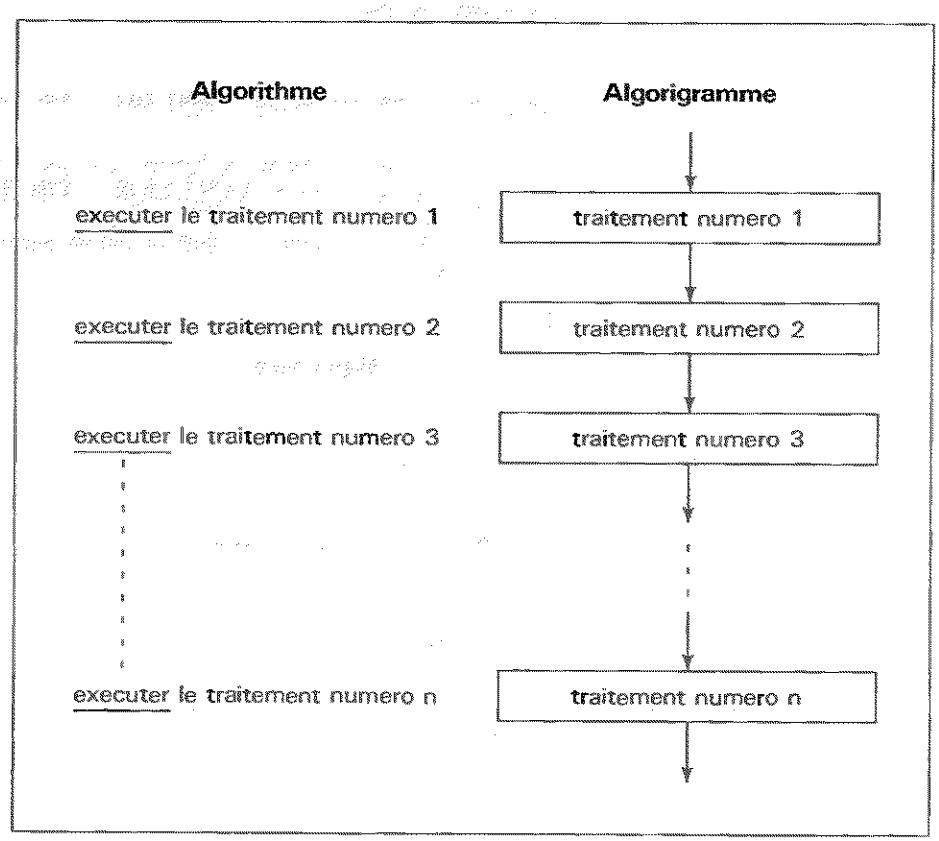
— les commentaires seront placés, si possible, à droite de l'algorigramme.

IV - Représentation des structures algorithmiques complexes

Les structures algorithmiques que nous avons étudiées nécessitent l'association de plusieurs symboles pour être entièrement réalisées, nous vous les représentons ci-après.

La séquence

Cette structure, étant linéaire, est la plus simple à représenter : elle sera composée d'une succession de symboles de traitements.

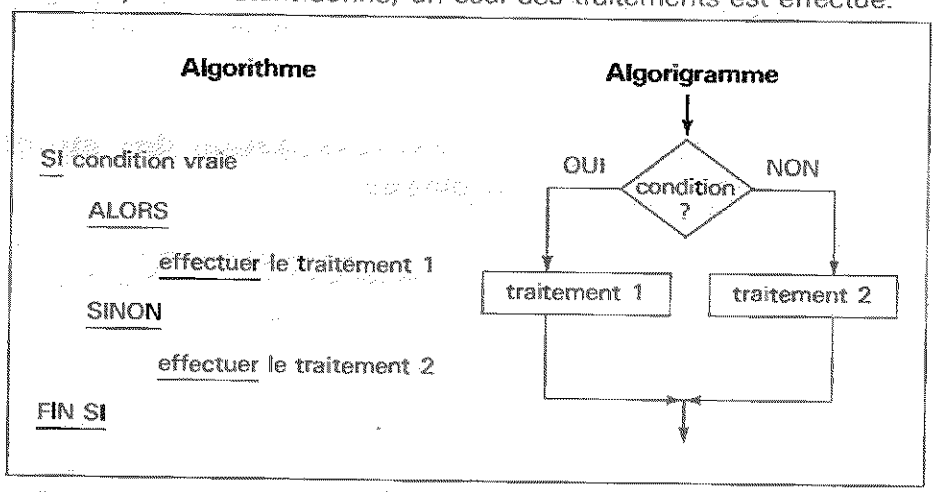


La séquence.

LA SÉLECTION SIMPLE

Cette structure utilise le symbole de branchement, duquel il est possible de sortir par les côtés, et, suite aux deux traitements possibles, la continuation se déroule avec un regroupement des deux branches.

Bien sûr, à un instant donné, un seul des traitements est effectué.

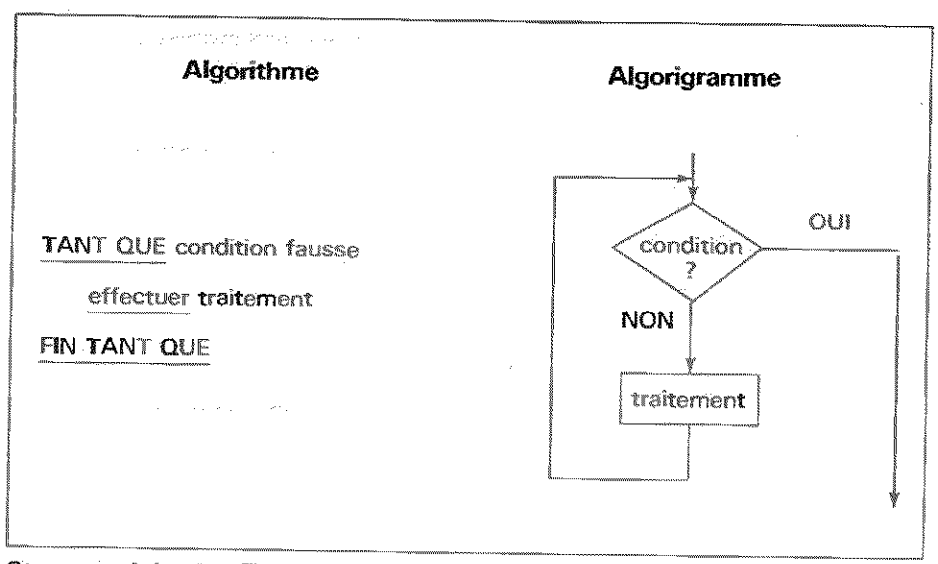


La sélection simple.

LA STRUCTURE ITERATIVE TANT QUE... FIN TANT QUE

Vous trouverez ci-après la forme que peut prendre la représentation de la structure de boucle TANT QUE... FIN TANT QUE.

On remarquera aisément que la sortie peut s'effectuer immédiatement à l'entrée.

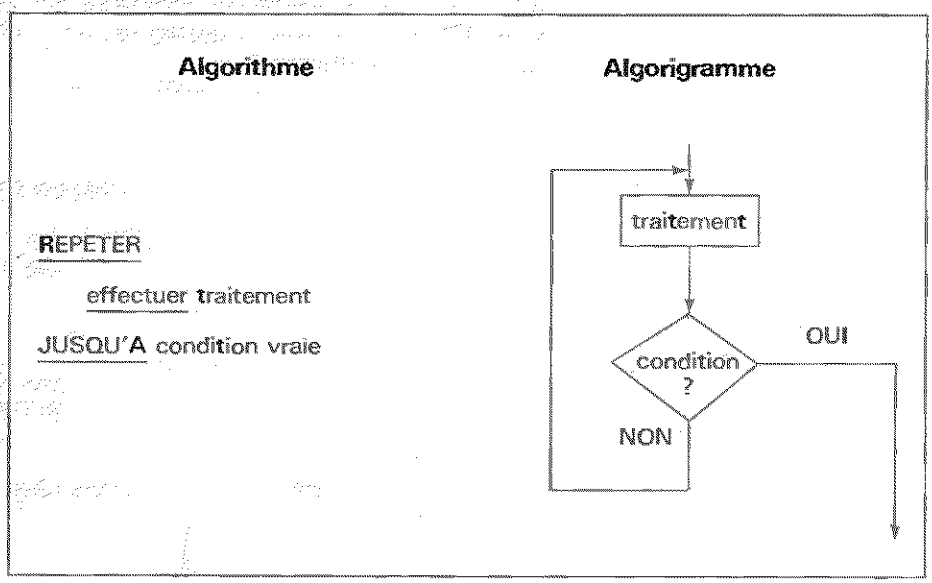


Structure itérative TANT QUE... FIN TANT QUE.



LA STRUCTURE ITÉRATIVE REPETER... JUSQU'A

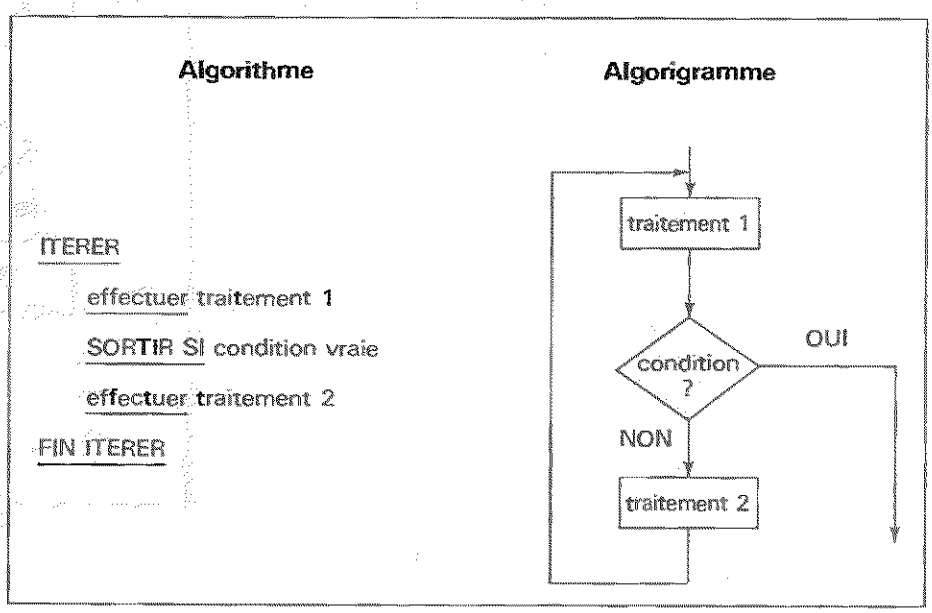
Cette représentation précise le fait que le traitement est effectué au moins une fois, avant de sortir de la boucle.



Structure itérative REPETER... JUSQU'A.

LA STRUCTURE ITÉRATIVE GÉNÉRALE

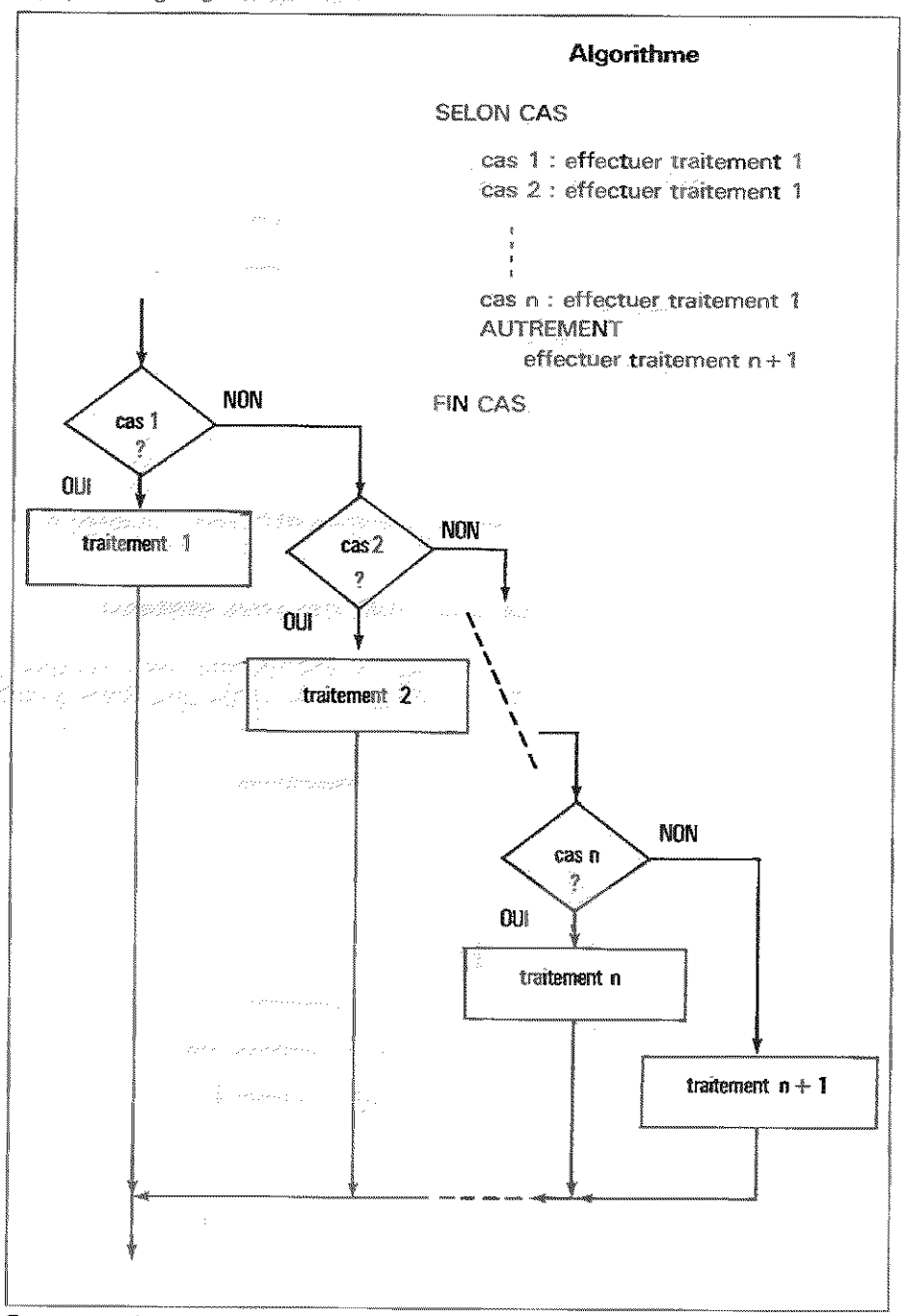
D'un usage moins courant, on s'aperçoit en fait qu'elle est constituée de l'imbrication des deux structures précédentes.



Structure itérative générale.

LA STRUCTURE DE CAS

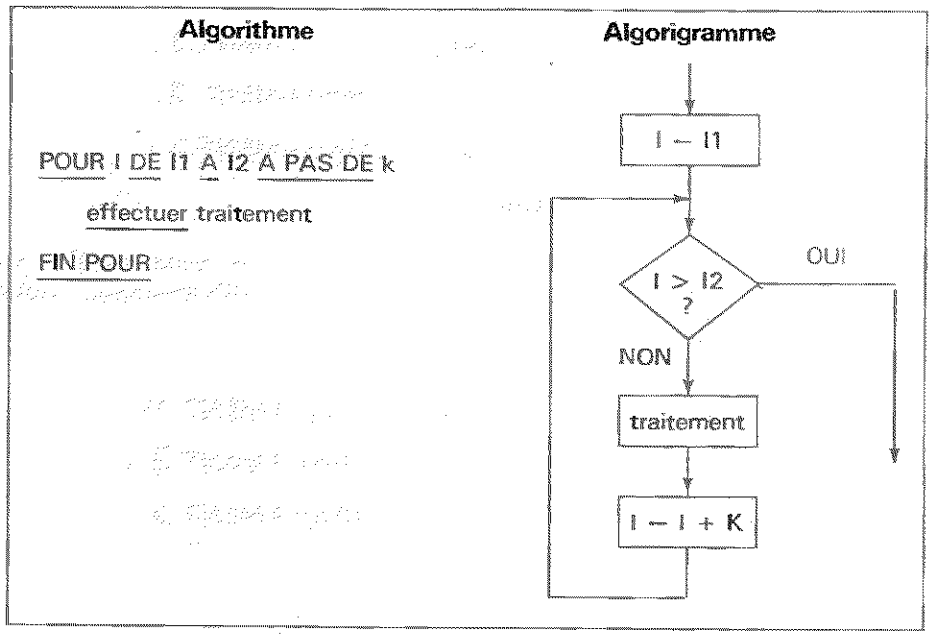
On remarque que la représentation par l'algorigramme de cette structure est mal adaptée si le nombre n de cas est particulièrement important. En cas d'utilisation intensive de la structure de cas dans un algorithme, on préférera garder la forme algorithmique écrite, sans passer par l'algorigramme.



Structure de cas

LA STRUCTURE DE RÉPÉTITION CONTROLÉE

Le schéma suivant représente l'algorithme et l'algorigramme de la structure boucle.



Structure de répétition contrôlée.

V - Algorithmique et langages

Nous allons par quelques exemples, découvrir que toutes les explications données précédemment se retrouvent dans tous les langages structurés, tels le langage Turbo-Pascal, ou le langage C, mais aussi dans des langages peu structurés comme le Basic.

LA SÉQUENCE

La séquence en Basic se traduit par une succession d'instructions ou de commandes notées sur des numéros de lignes successifs, ou séparés par le caractère :

```
10 PRINT "TRAITEMENT 1"
20 PRINT "TRAITEMENT 2"
30 PRINT "TRAITEMENT 3"
```

En langage Pascal, on déclare le début d'une séquence par l'instruction BEGIN, et la fin par l'instruction END;

```
BEGIN
    write ('TRAITEMENT 1')
    write ('TRAITEMENT 2')
    write ('TRAITEMENT 3')
END;
```

Le langage C utilise une syntaxe a peu près semblable, BEGIN et END étant remplacées par des accolades ouvertes et fermées.

```
{
    printf ("TRAITEMENT 1")
    printf ("TRAITEMENT 2")
    printf ("TRAITEMENT 3")
}
```

LA SÉLECTION SIMPLE

Dans les trois langages, la syntaxe du terme SI est la même et se traduit par le terme anglo-saxon IF. Seul Basic et Pascal connaissent le terme ALORS qui se traduit par THEN, ELSE traduira le terme SINON dans tous les cas.

— En Basic :

```
10 IF A = 2 THEN a = A + 1 ELSE A = A - 1
```

— En Pascal :

```
if A = 2
    then
        begin
        A := A + 1 ;
        end
    else
        begin
        A := A - 1 ;
        end ;
```



— En C :

```
if (A == 2)
    {
        A = A + 1
    }
    else
    {
        A = A - 1
    }
```

LA STRUCTURE DE CAS

La structure de cas n'est pas présente en Basic telle que nous l'avons présentée, mais peut sous certaines conditions être simulée, soit par une succession de tests, ou par l'instruction ON **variable GOSUB**, sachant que variable doit avoir pour valeurs 1, 2, 3, ... successivement.

— En Pascal :

```
case reponse of
    'L' : lecture
    'I' : insertion
    'R' : recherche
    else
        sauvegarde
```

— En C :

```
switch (reponse)
    case 'L' :
            lecture () ;
            break ;
    case 'I' :
            insertion() ;
            break;
    case 'R' :
            recherche () ;
            break ;
    default :
            sauvegarde () ;
```

LA RÉPÉTITION TANT QUE... FIN TANT QUE

Certains Basic, comme Pascal et C possèdent la structure TANT QUE... FIN TANT QUE, qui se traduit pas le terme WHILE, accompagne pour Basic du terme WEND.

— En Basic :

```
10 WHILE A < 10
20     A = A + 1
30 WEND
```

— En Pascal :

```
while A < 2 do
  begin
    A := A + 1 ;
  end
```

— En C :

```
while (A < 2)
  {
    ++ A ;
  }
```

LA RÉPETITION REPETER... JUSQU'A

En Basic, on trouve rarement cette forme, quelques uns acceptant tout de même une forme appelée DO ... LOOP, que l'on peut simuler avec des structures de tests.

Turbo-Pascal préconise la structure REPEAT... UNTIL et C a prévu la syntaxe DO ... WHILE.

— En Pascal :

```
reapeat
  write ('calcul en cours') ;
  A := A + 1 ;
until a = 10 ;
```



— En C :

```
do
{
    printf("calcul en cours") ;
    ++ A ;
}
while (A = 10) ;
```

LA RÉPÉTITION ARITHMÉTIQUE

La boucle POUR ... FIN POUR existe dans les trois langages, et c'est peut-être une des plus utilisée, surtout en Basic.

— En Basic :

```
10 FOR I = 1 TO 100
20     PRINT "NOUS SOMMES DANS LA BOUCLE"
30 NEXT I
```

— En Pascal :

```
for I := 1 to 100 do
  begin
    write ('NOUS SOMMES DANS LA BOUCLE') ;
  end;
```

— En C :

```
for (I = 1;I < = 100; ++ I)
  {
    printf ("NOUS SOMMES DANS LA BOUCLE") ;
  }
```

Bien sûr, il existe d'autres fonctions répondant à des structures algorithmiques étudiées, par exemple la procédure, mais nous vous les laisserons découvrir au fil des pages de notre ouvrage, citons cependant la structure de DEBUT et de FIN, que l'on peut ne pas retrouver en Basic, le numéro de la première ligne étant le début, et la dernière ligne, la fin, bien que l'on puisse ajouter le terme END à tout instant dans le programme. En Pascal, le début se signale par l'entête du programme (PROGRAM) et BEGIN, la fin par le terme END, attention à ne pas oublier le point. C propose comme début un nom suivi de deux parenthèses : programme (), et se termine par un retour au DOS, ici gemdos (0×1).

4/1.2

Applications algorithmiques

4/1.2.1

Racine carrée entière

HORNER, UN BEL EXEMPLE

Nous allons inaugurer cette nouvelle partie sur les algorithmes avec un calcul de racine carrée. Non pas que la racine carrée soit indispensable à tout logiciel digne de ce nom, mais plutôt qu'il s'agit là d'un bel exemple de fonction pour laquelle nous disposons d'une méthode tout à fait originale, et dont la mise en œuvre se prête bien à l'optimisation.

Car méthode et optimisation sont des mots clés en algorithmie. Prenons par exemple l'algorithme de Horner, utilisé dans le calcul des polynômes.

Soit $P(X) = a_0 + a_1.X + a_2.X^2 + \ldots + a_n.X^n$ un polynôme.

Lorsque X est un scalaire, P(X) nécessite n additions, n – 1 multiplications, et surtout n – 1 calculs de puissances. Mais le simple fait d'écrire P(X) sous la forme :

$$P(X) = ((\ldots((a_n.X + a_{n-1}).X + a_{n-2}).X + \ldots).X + a_1).X + a_0$$

montre que seules n additions et n – 1 multiplications suffisent à évaluer P(X). Ceci reste d'ailleurs vrai lorsque X n'est plus un scalaire mais une matrice, par exemple.

LA TROISIÈME MÉTHODE

Il existe en gros trois méthodes pour évaluer les fonctions numériques (exponentielle, logarithme, trigonométriques...). La première est la série entière ; la fonction est approximée par un polynôme dont le nombre de termes est d'autant plus élevé que l'on demande de la précision.

$$f(x) = \sum_{n=0}^{N} a_n X^n$$ La famille (a_n) est spécifique à f.

La deuxième est la suite récurrente ; un calcul simple, spécifique à chaque fonction, est effectué sur la variable de départ, puis on se sert du résultat comme nouvelle valeur de départ à ce calcul, de sorte que son résultat converge vers la valeur souhaitée au bout d'un nombre raisonnable d'itérations.

$$y_0 = x$$
$$y_1 = g(y_0)$$
$$y_2 = g(y_1)$$
$$\ldots$$
$$y_{n+1} = g(y_n) \quad \text{et} \quad \lim_{n \to \infty} y_n = f(x)$$
$$\ldots$$

Quant à la troisième, elle n'existe pas vraiment. En fait, pour certaines fonctions seulement, dont la racine carrée, on dispose de méthodes autres que calculatoires, ne nécessitant notamment pas de calculs flottants. C'est le cas du présent algorithme qui détermine la racine d'un entier positif de 32 bits, fournissant 1 bit de précision par itération.

Il arrive que l'on ait une longue série de racines carrées à effectuer, dans le cas d'applications graphiques notamment, pour calculer des distances entre objets par produit scalaire. En effet, la distance d qui sépare deux points de coordonnées (x1, y1, z1) et (x2, y2, z2) a pour expression :

$$d = \sqrt{(x2-x1)^2+(y2-y1)^2+(z2-z1)^2}$$

Si l'élévation au carré ne présente aucun problème, puisqu'elle se ramène à une multiplication, et que la multiplication d'entiers est câblée sur le microprocesseur 68000, l'extraction de la racine ne se fait en général qu'en virgule flottante, ce qui impose une quantité de conversions entiers-réels et un temps de calcul beaucoup trop long pour la précision demandée. En fait, bien souvent, une racine entière sur 16 bits suffit. Dans le cas des graphiques, cette précision permet sans problème de calculer toutes les distances dans un cube de 15 000 points de côté, avec la même précision que les coordonnées.

QUELLE EST LA RACINE DE 546238 ?

La méthode choisie est fondée sur une sorte de division. Le nombre à traiter tient lieu de dividende, sa racine de diviseur, la place du quotient servant à poser les calculs intermédiaires.



Il s'agit d'abord de grouper par deux les chiffres du nombre de départ. S'ils sont en nombre impair, on place un zéro à gauche du nombre, de façon à ne former que des doublets.

	54 62 38,00	r = 739,...	
t =	54	(0+z).z < 54 ?	→ z = 7
	−49		
t =	5 62	(140+z).z < 562 ?	→ z = 3
	−4 29		
t =	1 33 38	(1460+z).z < 13338 ?	→ z = 9
	−1 32 21		
t =	1 17 00		
	.		
	.		
	.		

On cherche d'abord l'entier de 1 chiffre dont le carré se rapproche le plus (par défaut) du premier doublet de chiffres. Soit, ici, 7 ($7 \times 7 = 49$, $8 \times 8 = 64$). On soustrait ensuite son carré : $54 - 49 = 5$. L'algorithme est maintenant en route. (Entre crochets se trouvent les résultats des opérations pour l'exemple ci-dessus).

A chaque itération sont effectuées les opérations suivantes :

— on aligne, à côté du reste de l'opération précédente, le doublet de chiffres suivant : [5 et 62 → 562]. Soit t ce nombre ;

— le nombre r est multiplié par 10, puis par 2, en case 3 : soit $y = 10 \times 2 \times r$.[$y = 10 \times 2 \times 7 = 140$] ;

— il faut alors trouver le plus grand chiffre z (de 0 à 9 et pas au-delà), tel que $(y+z) \cdot z$ reste inférieur ou égal à t.[$(140+3) \times 3 = 429$] ;

— on soustrait alors à t cette quantité [$562 - 429 = 133$], et on place z à droite de r : $r = 10 \times r + z$ [$r = 73$] ; la racine se construit chiffre par chiffre ;

— le reste de la soustraction [133] est prêt à recevoir le doublet suivant lors de la prochaine itération.

Remarque :

Cet algorithme permet d'obtenir des décimales à l'infini, car, lorsque le dernier doublet a été aligné (chiffres des dizaines et des unités), il suffit de poser le point décimal à la racine, et de continuer les calculs en alignant des doublets de zéros à chaque itération.

On obtiendra ainsi $\sqrt{546238} = 739{,}0791...$

PLUTÔT DEUX QUE DIX

C'est cet algorithme que nous allons mettre en œuvre sur l'Atari. Il donne un chiffre significatif de plus par itération, ce qui permet d'ajuster la quantité

de calculs à la précision requise. Mais il est ici présenté en base 10 : on cherche un chifre de 0 à 9 à chaque itération, et l'on sait à quel point les calculs en base 10 (codage BCD) sont pénibles dans un ordinateur. Le plus simple est de travailler en base 2, tout à fait adéquate. Certes, il y aura plus d'itérations, car on ne gagne qu'un bit à chaque tour, soit un facteur 2 en précision, au lieu de 10, mais nous allons voir qu'elles sont plus simples et surtout beaucoup plus rapides.

QUELLE EST LA RACINE DE 101101 ?

```
  10 11 01   | 110,...
             |-----------
  10         |  0+1 < 10 ?          → oui, z=1
- 1          |
  01 11      |  100+1 < 111 ?       → oui, z=1
- 1 01      |
     10 01   |  1100+1 < 1001 ?     → non, z=0
   -00 00    |
     10 01 00 |
        .    |
        .    |  (tous les nombres sont en base 2)
        .    |
```

Les itérations sont grandement simplifiées, car z ne peut prendre que deux valeurs : 0 ou 1. Elles se décomposent donc ainsi :

— on aligne le doublet sur le reste précédent :

— r n'est plus multiplié par 10 en base dix, mais 10 en base deux, c'est à dire 2, puis encore par deux (soit par quatre en tout) ;

— comme z vaut soit 0 soit 1, et que $(y+z) \times z = 0$, il suffit de comparer cette dernière quantité à t ; si elle est inférieure ou égale, c'est que $z=1$, et on fait la soustraction ; si elle est supérieure, $z=0$, et on ne soustrait rien.

Aucune multiplication, même entière, n'est nécessaire.

EN ASSEMBLEUR, SINON RIEN

Bien entendu, un tel algorithme n'a pas de raison d'être dans un langage évolué comme le C, le Pascal, ou le GFA. Seul l'assembleur le rend intéressant du point de vue de la rapidité d'exécution.

La routine reçoit le nombre (positif) sur 32 bits dans d0.l, et retourne sa racine (positive) sur 16 bits dans d1.w. d0, et d2 à d4 sont détruits. Une 17e itération, simplifiée, est effectuée. Elle permet de se renseigner sur le premier bit après la virgule, et donc d'arrondir correctement à l'entier le plus proche.



Peu de commentaires sur l'optimisation, si ce n'est l'emploi de moveq au lieu de move au début, du branchement courts bcs.s lors du test, et l'utilisation quasi systématique de mots (.w) au lieu de mots longs. Ces choix augmentent la vitesse de calcul. D'autre part, il est de bon augure de constater que 1 seule parmi les 13 instructions de la boucle est un transfert de type **move** ; toutes les autres sont des instructions actives.

```
; racine carrée
; → d0.l nombre entier positif sur 32 bits
;← d1.w racine sur 16 bits
racine:   moveq.l   #0,d2      ; reste : t
          moveq.l   #0,d1      ; racine : r
          moveq.l   #15,d4     ; 16 itérations
boucle:   add.l     d0,d0      ; bit de gauche de d0.l →
          addx.w    d2,d2      ; bit de droite de d2.w
          add.l     d0,d0      ; Ainsi, en 2 fois, d2.w reçoit
          addx.w    d2,d2      ; le nouveau débit sur sa droite
          add.w     d1,d1      ; r × 10b;
          move.w    d1,d3      ; y est placé dans d3 :
          add.w     d3,d3      ; y = (r × 10b) × 2 ...
          cmp.w     d2,d3      ; y >= t ?
          bcc.s     b10        ; si oui, z=0
          addq.w    #1,d3      ;
          sub.w     d3,d2      ; sinon, z=1, et t-(y+z) → r
          addq.w    #1,d1      ;
b10:      dbra      d4,boucle  ; itération suivante.
          cmp.w     d2,d1      ; pour arrondir à l'entier
          bcc.s     b20        ; le plus proche
          addq.w    #1,d1      ;
b20:      rts                  ; retour au programme appelant
```

ÇA VA VITE

Résultat : 142 μs, soit environ 1 200 cycles d'horloge, pour calculer une racine carrée entière. L'algorithme peut éventuellement être encore optimisé, en déroulant la boucle gérée par d4 notamment, ce qui permet de descendre à 116 μs. On peut par la suite optimiser la première itération, puis traiter d0 en deux fois sur 16 bits, au lieu de 32 systématiquement. On passe alors en dessous de 100 μs (98 précisément). A titre de comparaison, la racine carrée dure au mieux 370 μs en BASIC GFA 3.0 compilé, et 1 400 μs en Turbo Pascal 5.0 sur un compatible AT286 à 10 MHz. Certes le gain n'est pas extraordinaire, mais cet article n'était qu'une introduction aux algorithmes sur un exemple simple.

Le listing suivant est un chargeur Basic, toujours en GFA 3, usant des meilleures optimisations possibles pour cet algorithme. Il montre l'efficacité de la routine en présence d'une grande quantité de calculs. Celle-ci, appelée par un RCALL, calcule n racines dans le tableau d'entiers sur 32 bits **entree( )**, et les place dans le tableau **sortie( )**, également sur 32 bits.

```
.pl 50
' ------------------------------------------------------
' -     PROGRAMME DE CALCUL DE RACINE CARREE           -
' -          AVEC ROUTINE EN ASSEMBLEUR                -
' -                                                    -
' -     (c) Jean Guerard 1990 - editions WEKA          -
' ------------------------------------------------------
'
DEFINT "a-z"
DIM entree(100000),sortie(100000)                    ! tableaux de travail.
DIM registre(15)                                     ! registres pour RCALL.
prog$=SPACE$(500)                                    ! place memoire pour la
prog=V:prog$                                         ! routine en assembleur.
'
charge_prog                                          !execution
main                                                 !

PROCEDURE charge_prog
  RESTORE
  p=prog
  DO                                                 ! chargement de la routine
    READ code$                                       ! a partir des DATA.
    EXIT IF code$="#"
    CARDepe=VAL("&"+code$)
    ADD p,2
  LOOP
RETURN
PROCEDURE main
  DO
    INPUT "nombre de calculs : ";n                   ! nombre de calculs a faire
    EXIT IF n=0                                      ! d'un coup.
    CLS
    creation
    test_basic
    test_assem
    rapidite
    precision
  LOOP
RETURN
PROCEDURE creation
  FOR i=0 TO n-1                                     ! pour les tests, les nombres
    entree(i)=XBIOS(&H11)                            ! sont tires au hasard:
  NEXT i                                             ! XBIOS(&H11) = RANDOM
RETURN
PROCEDURE test_basic
  t=TIMER
  FOR i=0 TO n-1                                     ! calcul des racines par
    sortie(i)=SQR(entree(i))                         ! le basic, pour la reference
  NEXT i
  du_bas=(TIMER-t)/5
```



(suite)

```
  PRINT "duree en basic       : ";du_bas;" ms"
RETURN
PROCEDURE test_assem
  t=TIMER                                          ! parametres pour RCALL:
  registre(8+0)=V:entree(0)                        ! a0 : adresse tableau entree
  registre(8+1)=V:sortie(0)                        ! a1 : adresse tableau sortie
  registre(7)=n-1                                  ! d7.w : nombre de calculs -1
  RCALL prog,registre()
  du_ass=(TIMER-t)*5
  PRINT "duree en assembleur : ";du_ass;" ms"
RETURN
PROCEDURE rapidite
  PRINT "facteur ";
  IF du_ass                                        ! estimation du gain en
    gain#=du_bas/du_ass                            ! vitesse d'execution.
    PRINT USING ": ###.#",gain#
  ELSE
    PRINT "> ";du_bas/5
  ENDIF
RETURN
PROCEDURE precision
  saut=MAX(1,INT(n/20))
  CLR i
  FOR j=0 TO 19                                    ! affichage de quelques
    PRINT entree(i),                               ! points du tableau pour
    PRINT SQR(entree(i)),                          ! observer les arrondis.
    PRINT sortie(i),
    PRINT BIN$(sortie(i),16)
    ADD i,saut
  NEXT j
RETURN
'
DATA 2018,610A,4259,32C1,51CF,FFF6,4E75,7400,7200,4840,D040,D542
DATA D640,D542,4A42,6704,5342,5241,D040,D542,D040,D542,D241,3601
DATA D643,9642,6406,5243,9443,5241,D040,D542,D040,D542,D241,3601
DATA D643,B642,6406,5243,9443,5241,D040,D542,D040,D542,D241,3601
DATA D643,B642,6406,5243,9443,5241,D040,D542,D040,D542,D241,3601
DATA D643,B642,6406,5243,9443,5241,D040,D542,D040,D542,D241,3601
DATA D643,B642,6406,5243,9443,5241,D040,D542,D040,D542,D241,3601
DATA D643,B642,6406,5243,9443,5241,D040,D542,D040,D542,D241,3601
DATA D643,B642,6406,5243,9443,5241,4840,D040,D542,D040,D542,D241
DATA 3601,D643,B642,6406,5243,9443,5241,D040,D542,D040,D542,D241
DATA 3601,D643,B642,6406,5243,9443,5241,D040,D542,D040,D542,D241
DATA 3601,D643,B642,6406,5243,9443,5241,D040,D542,D040,D542,D241
DATA 3601,D643,B642,6406,5243,9443,5241,D040,D542,D040,D542,D241
DATA 3601,D643,B642,6406,5243,9443,5241,D040,D542,D040,D542,D241
DATA 3601,D643,B642,6406,5243,9443,5241,D040,D542,D040,D542,D241
DATA 3601,D643,B642,6406,5243,9443,5241,B242,6402,5241,4E75
DATA #
```

4/1.2.2

Nombres premiers

Après s'être intéressés à la racine carrée, nous allons aborder un domaine célèbre de l'arithmétique, et montrer comment y introduire l'assembleur de l'Atari.

Un nombre premier est un entier *p*, strictement plus grand que 1, et dont les seuls diviseurs sont 1 et lui-même. Les premiers nombres premiers sont :

2, 3, 5, 7, 11, 13, 17, 19...

UNE SUITE PARTICULIÈRE

Soit la suite des nombres premiers. La première propriété de cette suite est d'être infinie : Euclide l'a démontré dès le IIIe siècle. Autre propriété importante, les nombres premiers sont de plus en plus espacés, mais au sens statistique seulement, car on peut, semble-t-il, trouver des couples de nombres premiers distants de 2 (17 et 19, 29 et 31...) aussi loin qu'on veut. Mais ce point est aujourd'hui encore un problème ouvert. Si l'on considère la fonction $\pi(n)$, définie de la façon suivante :

$$\pi(n) = \sum_{p \leq n} 1, \text{ où } p \text{ est premier,}$$

c'est-à-dire donnant le nombre de nombres premiers inférieurs ou égaux à *n*, on dégage de son étude d'autres propriétés sur la répartition de ces nombres. Par exemple, Hadamard et de la Vallée Poussin ont démontré en 1896 (précédés par l'intuition de Legendre et Gauss), le Théorème des Nombres Premiers, l'un des grands théorèmes en mathématiques :

$$\text{lorsque } n \text{ tend vers l'infini, } \pi(n) \sim \frac{n}{\ln(n)}$$

On dispose aussi d'autres résultats fort intéressants. Grâce à la fonction *zeta* de Riemann, par exemple :

$$\zeta(s) = \sum_{n=1}^{\infty} n^{-s}$$

on montre

$$\zeta(s) = \prod_{p} (1 - p^{-s})^{-1}$$

une somme où figurent tous les entiers est transformée en un produit où seuls les nombres premiers apparaissent. Après quelques transformations

de cette expression, on parvient à montrer que

$$\lim_{n\to\infty} \sum_{p\leq n} \frac{1}{p} = +\infty, \text{ où } p \text{ est premier.}$$

GÉNÉRATION DE NOMBRES PREMIERS

Après cet aperçu du côté mathématique de la question, nous allons nous intéresser à la génération des nombres premiers par un ordinateur, et, puisque c'est l'objet du chapitre, à la mise au point d'algorithmes. Nous allons à cet effet considérer plusieurs approches.

LE TEST DES DIVISEURS

Partant de la définition, énoncée tout au début, on peut chercher, pour un candidat *p* donné, si oui ou non il admet d'autres diviseurs que 1 et lui-même. Il en résulte la version 1.0 de l'algorithme.

Ce premier programme, qui donne les nombres premiers inférieurs à 100 en 530 ms, est assez grossier (d'autant qu'il est erroné, puisque pour 2, le diviseur 2 est essayé, alors qu'il ne devrait pas, ce qui lui fait dire que 2 n'est pas premier). Un certain nombre d'optimisations évidentes pouvaient être faites, mais il importait de partir d'un programme dénué de toute réflexion.

Générateur de nombres premiers, version 1.0

L'optimisation peut commencer.

Tout d'abord, comme 2 est premier, il est inutile de tester les nombres pairs. La boucle de *p* peut alors aller de 2 en 2 (en veillant à commencer par 3), ce qui permet déjà d'aller 2 fois plus vite.

Il en va de même pour les diviseurs. La boucle de *d* peut aller de 2 en 2, mais à condition de commencer par 3, ce qui n'est pas gênant puisque désormais tous les candidats seront impairs. Et le programme de tourner encore deux fois plus vite.

Une autre amélioration, moins évidente, porte sur la quantité de diviseurs à confronter au candidat *p*. Il suffit en effet de tester les diviseurs jusqu'à $\sqrt{p}$. La raison en est simple : si *p* n'est pas premier, l'un au moins des termes de sa décomposition en facteurs premiers est plus petit que $\sqrt{p}$. Si tel n'était pas le cas, on ne pourrait écrire

$$p = m \cdot q,$$

avec *m* et *q* tous deux supérieurs à $\sqrt{p}$. Mais cela ne signifie pas que la racine de *p* soit à calculer systématiquement (erreur fréquente pour cet algorithme). Les diviseurs étant générés dans l'ordre croissant, le quotient est dans un premier temps inférieur au diviseur, mais dès que celui-ci devient supérieur au diviseur, cela signifie que les diviseurs seront dorénavant supérieurs à $\sqrt{p}$. On peut donc arrêter la boucle de *d* à ce moment-là. On y perd une division, mais on évite le calcul d'une racine carrée.



```
'
' ****************************************
' *   GENERATEUR DE NOMBRES PREMIERS     *
' *              version 1.0             *
' * (c) J. Guerard - Editions Weka 1990 *
' ****************************************
'
DEFINT "a-z"
DIM table(10000)                          !la table de stockage.
t=TIMER                                   !pour la duree d'execution.
n=300                                     !fixe la fin de l'exploration.
s=0                                       !compteur de nombres premiers.
FOR p=2 TO n                              !p est le candidat.
  flag!=FALSE                             !indicateur de primalite.
  d=2                                     !d est le diviseur.
  WHILE MOD(p,d)                          !tant que p n'est pas divisible par d,
    flag!=TRUE
    INC d
    EXIT IF d=p                           !on sort si d=p;
    flag!=FALSE                           !sinon on efface l'indicateur et
  WEND                                    !on passe au diviseur suivant.
  IF flag!=TRUE                           !si p est premier,
    table(s)=p                            !il est range dans la table,
    INC s                                 !et le compteur est incremente.
  ENDIF
NEXT p                                    !passage au candidat suivant.
du=(TIMER-t)*5                            !duree du calcul en millisecondes.
FOR i=0 TO s-1                            !affichage de la liste obtenue.
  PRINT table(i)'
NEXT i
PRINT
PRINT s;" nombres premiers, sur ";n;", en ";du;" ms"
PRINT nbdiv
```

Au passage, on justifie le sens de parcours de la boucle des diviseurs par le fait qu'un candidat *p* quelconque est plus fréquemment divisible par 3 (une fois sur trois !), 5, 7 ... que par de grands nombres. On sortira donc plus facilement de cette boucle dans le sens croissant que dans le sens décroissant.

D'où la version 1.1 de l'algorithme, qui tourne 10 fois plus vite.

Générateur de nombres premiers, version 1.1

```
'
' ****************************************
' *   GENERATEUR DE NOMBRES PREMIERS     *
' *              version 1.1             *
' * (c) J. Guerard - Editions Weka 1990 *
' ****************************************
'
DEFINT "a-z"
DIM table(10000)                          !la table de stockage.
t=TIMER                                   !pour la duree d'execution.
n=300                                     !fixe la fin de l'exploration.
s=0                                       !compteur de nombres premiers.
FOR p=3 TO n STEP 2                       !p est le candidat.
  flag!=TRUE                              !indicateur de primalite.
  d=3                                     !d est le diviseur.
  REPEAT
    q=p/d                                 !on effectue la division,
    EXIT IF q<d                           !on sort si d atteint la racine de p.
    IF q*d=p
      flag!=FALSE                         !si p n'est pas premier.
    ENDIF
    ADD d,2                               !les diviseurs vont de 2 en 2.
  UNTIL flag!=FALSE
  IF flag!=TRUE                           !si p est premier (sortie par EXIT IF)
    table(s)=p                            !il est range dans la table,
    INC s                                 !et le compteur est incremente.
  ENDIF
NEXT p
du=(TIMER-t)*5                            !passage au candidat suivant.
FOR i=0 TO s-1                            !duree du calcul en millisecondes.
  PRINT table(i)'                         !affichage de la liste obtenue.
NEXT i
PRINT
PRINT s;" nombres premiers, sur ";n;", en ";du;" ms"
```

A ce stade de l'optimisation, nous avons non seulement diminué le temps de calcul, mais de plus, la loi de progression de l'algorithme a été changée.



LES LOIS DE PROGRESSION

Elles sont définies pour des algorithmes itératifs : elles donnent la croissance de la quantité de calculs à effectuer en fonction du nombre d'itérations. Dans le cas des algorithmes 1.0 et 1.1, la quantité de calculs est le nombre de divisions (ou de modulo, puisque de façon entière une division est utilisée), car cette opération est de loin la plus longue à traiter par le processeur, par comparaison aux additions et aux manipulations d'indices.

L'algorithme de calcul de racine carrée présenté au chapitre précédent suivant une loi linéaire : pour une précision de *n* bits souhaitée sur le résultat, il fallait exécuter *n* fois le calcul principal.

L'algorithme de tri le plus simple qui soit, consistant à chercher dans un tableau de *n* valeurs le plus petit élément, le permuter avec le premier du tableau, chercher ensuite le plus petit élément parmi les *n – 1* restants, et ainsi de suite, nécessite $\frac{n^2-n}{2}$ recherches :

$$Q = n + n-1 + n-2 + n-3 + \ldots + 1$$

somme résolue par le génial Cauchy dès l'âge de 6 ans, et qui vaut :

$$Q = \sum_{i=1}^{n} n = \frac{n(n-1)}{2}$$

Dans la version 1.0 de l'algorithme générateur de nombres premiers, il se passe un peu la même chose : pour un candidat *p* donné, on essaye *a priori* *p(p – 1)/2* diviseurs ; à ceci près que ces diviseurs ne sont pas tous testés : on sort assez souvent de la boucle.

Statistiquement, une fois sur deux, on ne fait qu'une seule opération MOD (par 2). Une fois sur trois, on n'en fait que deux, si le candidat n'est pas divisible par 2, uniquement par 3. Mais là encore, une fois sur deux, un multiple de 3 est aussi un multiple de 2.

Par conséquent, gardant en tête le fait qu'un tiers des multiples de 5 est divisible par 3, $\frac{1}{5}$ des multiples de 7 est divisible par 5, etc..., on en déduit l'expression de la quantité d'opérations MOD nécessaires :

$$Q = n\left[\frac{1}{2}1+\frac{1}{3}\left(\frac{1}{2}\right)2+\frac{1}{5}\left(\frac{1}{2}\frac{2}{3}\right)4+\frac{1}{7}\left(\frac{1}{2}\frac{2}{3}\frac{4}{5}\right)6\ldots\right]$$

$$= n \cdot \sum_{p<n} \frac{p-1}{p}\left(\prod_{q<p} \frac{q-1}{q}\right), \text{ p et q premiers.}$$

Le calcul de cette somme en fonction de n est très complexe, mais on peut néanmoins le majorer :

$$Q_{1.0}(n) \approx n \sum \frac{p-1}{p} \cdot K$$

$$\boxed{Q_{1.0}(n) \approx k \frac{n^2}{\text{en } n}}$$

Du fait de la croissance démesurée de Q avec *n*, les algorithmes en n^2 sont toujours à proscrire, et *a fortiori* pour les puissances supérieures de m.

L'indéniable avantage de la version 1.1 par rapport à la 1.0, outre de s'affranchir des nombres pairs, ce qui va 4 fois plus vite, est d'arrêter les diviseurs à $\sqrt{p}$ au lieu de *p*. Ici, la quantité *Q(n)* a pour expression :

$$Q = \frac{n}{2}\left[\frac{1}{3}\frac{1}{2}1 + \frac{1}{5}\left(\frac{1}{2}\frac{2}{3}\right)2 + \frac{1}{7}\left(\frac{1}{2}\frac{2}{3}\frac{4}{5}\right)3 + \frac{1}{11}\left(\frac{1}{2}\frac{2}{3}\frac{4}{5}\frac{6}{7}\right).5...\right]$$

$$= \frac{n}{2}\sum_{p<\sqrt{n}}\frac{p-1}{2p}\left(\prod_{q<p}\frac{q-1}{q}\right)$$

Ce que l'on peut majorer par :

$$Q_{1.1}(n) \approx \frac{n}{4}\sum_{p<\sqrt{n}}\frac{p-1}{p}.K$$

$$\boxed{Q_{1.1}(n) \approx K\frac{n}{2}\frac{n\sqrt{n}}{en\,n}}$$

On peut encore faire mieux en n'utilisant que des diviseurs premiers ; en effet, pourquoi chercher à savoir si un candidat est divisible par 9, puisque si c'est le cas, il est donc divisible par 3, et on sort alors de la boucle avant d'arriver à 9. Cette nouvelle structure n'est pas si difficile à mettre en œuvre, grâce à la table dans laquelle sont rangés les nombres premiers au fur et à mesure de leur découverte. Il suffit de puiser les diviseurs dans cette table, au lieu d'utiliser le simple *ADD d,2*, lors du passage au diviseur suivant.

On parvient alors, pour la quantité de calculs, à l'expression suivante :

$$Q_{1.2}(n) = \frac{n}{2}\left[\frac{1}{3}\frac{1}{2}1 + \frac{1}{5}\left(\frac{1}{2}\frac{2}{3}\right)2 + \frac{1}{7}\left(\frac{1}{2}\frac{2}{3}\frac{4}{5}\right)3 + \frac{1}{11}\left(\frac{1}{2}\frac{2}{3}\frac{4}{5}\frac{6}{7}\right).4...\right]$$

$$= \frac{n}{2}\sum_{p<\sqrt{n}}\frac{\pi(p)-1}{p}\left(\prod_{q<p}\frac{q-1}{q}\right)$$

Cette fois-ci, la progression est pratiquement linéaire :

$$\boxed{Q_{1.2}(n) \approx K'\frac{n}{2}}$$

Les quelques macro-instructions utilisées dans le programme en assembleur sont dues à Pierre Bedrossian et John Faure. Celles qui ne sont pas listées font partie de la librairie du GFA assembleur.



Générateur de nombres premiers, version 1.2

```
;*********************************************
;*    GENERATEUR DE NOMBRES PREMIERS         *
;*                                           *
;* (c) Jean Guerard - editions WEKA 1990     *
;*********************************************

            .INCLUDE "JEAN_LI2.I8"

n           = 10000     ;fin de la recherche

            start       ;macro gemdos
main:
            print       'debut'             ;macro ecran: affichage d'une chaine

            cconin      ;macro gemdos
            super       #0                  ;macro gemdos : passage
            move.l      d0,savesp           ;en mode superviseur,

            move.l      $4ba,count          ;pour pouvoir lire le timer.

            movea.l     #table,a6           ;table des nombres premiers.
            move.w      #3,(a6)+            ;initialisation avec les
            move.w      #5,(a6)+            ;premiers nombres premiers.

            clr.l       a4                  ;a4 = compteur de nombres premiers.
            move.l      #7,d7               ;d7 est le candidat p.
prem:
            ext.l       d7                  ;on travaille sur d7.l.

            move.l      d7,d5               ;test simplifie par 3: pour 33%
            divu.w      #3,d5               ;des nombres, c'est plus rapide.
            swap.w      d5                  ;le reste d'une division est
            tst.w       d5                  ;dans le mot superieur du registre.
            beq.s       nonprem

            movea.l     #table+2,a5         ;on commence a 5.
div:
            move.w      (a6)+,d6            ;diviseur d dans d6.w.
            move.l      d7,d5               ;d5 = p
            divu.w      d6,d5               ;d5 = q = p/d
            cmp.w       d6,d5               ;si q < d, on a atteint sqr(p),
            blt.s       ouiprem             ;alors p est premier.
            swap.w      d5                  ;maintenant, d5 = r = p modulo d
            tst.w       d5                  ;(= reste de la divison)
            bne.s       div                 ;si r <> 0, test diviseur suivant.

nonprem:
            addq.w      #2,d7               ;si r = 0, p n'est pas premier.
            cmpi.l      #n,d7               ;alors, on passe au candidat
            ble.s       prem                ;suivant.
            bra.s       fin
ouiprem:
            move.w      d7,(a6)+            ;p premier ==> dans la table.
            addq.l      #1,a4               ;incremente le compteur pour pi(n).
            addq.w      #2,d7               ;puis, passage au candidat suivant.
            cmpi.l      #n,d7
            ble.s       prem
```

(suite)

```
fin:
            move.l      $4ba,d7             ;c'est fini; on mesure le timer,
            sub.l       count,d7            ;on en deduit la duree du calcul.

            print       'termine'
            aff_dec     d7                  ;duree du calcul.
            crlf        ;macro ecran: curseur a la ligne suivante.
            cconin      ;macro gemdos.

            movea.l     #table,a5
result:
            move.w      (a5)+,d7            ;affichage de la liste
            aff_dec     d7                  ;des nombres premiers.
            cmpa.l      a5,a6
            bgt.s       result
            crlf
            aff_dec     a4                  ;affichage de pi(n).
            super       savesp              ;retour au mode utilisateur.

            cconin      ;macro gemdos

            rts                             ;fin du programme

            .BSS
savesp:     .DS.l 1     ;sauvegarde de pile pour mode superviseur.
count:      .DS.l 1     ;valeur timer pour la duree du calcul
table:      .DS.w 50000                     ;table des nombres premiers

            .END



JEAN_LI2.IS, bloc, page 1, 29.11.1990, 14:08


            .MACRO print string             ; Sortir une chaine
; format d'appel:
; print 'String' ou
; print #Adresse

            .IF \'1>0 ;                     ; directement
            .SECTION 4,DATA                 ; mettre en mmoire toutes les chain
    es
;                                           ; des instructions print
.\~s:       .DC.b \1,0                      ;
            .TEXT
            movem.l   d0-d2/a0-a2,-(sp)
            movea.l   #.\~s,a0
            cconws    a0
            movem.l   (sp)+,d0-d2/a0-a2
            .ELSE     ;                     ; Adresse de la chaine
            cconws    \1
            .ENDIF
            .ENDM
```



(suite)

```
JEAN_LI2.IS, bloc, page 1, 29.11.1990, 14:10


            .MACRO start
            movea.l    sp,a5
            movea.l    4(a6),a5
            move.l     $c(a5),d0
            add.l      $14(a5),d0
            add.l      $1c(a5),d0
            addi.l     #$200,d0

            move.l     d0,d1
            add.l      a5,d1
            andi.l     #-2,d1
            movea.l    d1,sp

            mshrink    a6,d0
            tst.l      d0
            bne.s      .\~1
            bsr        main
.\~1:       pterm
            .ENDM



JEAN_LI2.IS, bloc, page 1, 29.11.1990, 14:11


            .MACRO aff_dec nombre,l,c
            movem.l    d0-d7/a0-a6,-(sp)
            .IF \?l
            .IF \?c
            cconout    #27
            cconout    #"Y"
            move.w     \l,d0
            addi.w     #32,d0
            cconout    d0
            move.w     \c,d0
            addi.w     #32,d0
            cconout    d0
            .ENDIF
            .ENDIF
            move.w     \l,d0
            move.l     #10000,d1              ; diviseur
.\~enc:     cmp.w      d1,d0
            blt        .\~ev
            ext.l      d0
            divu.w     d1,d0
            movem.l    d0/d1,-(sp)
            addi.w     #$30,d0
            cconout    d0
            movem.l    (sp)+,d0/d1
            swap.w     d0
            bra.s      .\~sui
```

(suite)

```
.\~av:      movem.l   d0/d1,-(sp)
            cconout   #"0"
            movem.l   (sp)+,d0/d1
.\~sui:     divu.w    #10,d1
            cmpi.w    #0,d1
            beq       .\~fi
            bra.s     .\~enc
.\~fi:      movem.l   d0/d1,-(sp)
            cconout   #" "
            movem.l   (sp)+,d0/d1
            movem.l   (sp)+,d0-d7/a0-a6
            .ENDM
```

LES GRECS ET L'INFORMATIQUE

La méthode du test des diviseurs pour un candidat étant maintenant épuisée, il faut nous orienter vers un autre algorithme si nous voulons aller encore plus vite. Précisément, il serait profitable de s'affranchir des divisions qui, en assembleur 68000, durent 100 à 150 cycles contre une dizaine en moyenne pour les autres instructions. L'idée est la suivante :

« Au lieu de chercher les diviseurs d'un candidat, essayons plutôt d'éliminer les multiples des nombres premiers déjà découverts. »

Cette idée, d'une simplicité déconcertante, Eratosthène l'a eue le premier voici plus de 2000 ans. Ce mathématicien contemporain d'Archimède (IIIe siècle avant Jésus-Christ) était en outre historien, philosophe, poète, critique de théâtre, géographe et astronome. C'est notamment lui qui, parcourant certains manuscrits de la grande bibliothèque d'Alexandrie, conclut à la rotondité de la Terre par l'observation suivante : en deux points de la Terre (à Syène et à Alexandrie), le soleil ne projetait pas les mêmes ombres le 21 juin, jour du solstice d'été. A Syène, ville tropicale, le soleil était au zénith, alors qu'à Alexandrie, plus au nord, il s'en écartait de 7°. La Terre était courbe, et comme le voulait l'idéal philosophique de l'époque, elle était même certainement sphérique ! Erathosthène en estima d'ailleurs avec une étonnante précision les 40 000 km de sa circonférence.

Mais pour ce qui nous intéresse, Erastosthène est aussi à l'origine du crible qui porte son nom. Son algorithme, très mécanique, est le suivant :



« Prenons un long ruban de papier, et inscrivons-y les entiers dans l'ordre croissant à partir de 2 (1 ne compte pas). 2 est le premier nombre de la suite. Nous l'entourons, car c'est un nombre premier. Par contre nous rayons tous les multiples de 2 : 4, 6, 8... jusqu'à N, la fin du ruban. Puis nous revenons au début et cherchons le premier nombre non rayé : 3, c'est donc un nombre premier ; nous l'entourons et rayons ses multiples 6, 9, 12... jusqu'à la fin. Ainsi, à chaque passage, nous découvrons un nombre premier qui n'a pas été rayé car multiple d'aucun nombre inférieur et nous rayons à leur tour ses multiples. »

Il est à noter que certains nombres sont rayés plusieurs fois. 55 est rayé lors du traitement des multiples de 5, mais aussi lors du traitement des multiples de 11.

Exercice classique de programmation, ce crible est d'une redoutable efficacité. Non seulement on s'affranchit des incontournables divisions présentes dans les algorithmes précédents, mais de plus les multiplications sont également superflues ! En effet, lorsqu'un nombre premier p est découvert, ses multiples sont calculés et rayés de p en p, soit en additionnant p au multiple précédent. Voici donc un algorithme d'arithmétique qui n'exploite rien de plus que des additions et des manipulations d'indices. Le rêve pour l'Assembleur !

ERASTOSTHÈNE PASSÉ AU CRIBLE

Passons donc à l'implantation d'un tel programme dans un Atari ST. A commencer par la structure du crible. L'algorithme impose de garder en mémoire la suite complète des entiers naturels, mais finalement une seule information est à retenir pour chaque nombre : s'il a été rayé, ou non, dans le crible. Un seul bit suffit. Il faudra donc adresser des bits isolément dans un octet, ce qui est heureusement aisé et rapide en assembleur 68000 grâce aux instructions **bset**, **bclr**, **btst**, **bchg**. Autrefois on utilisait un octet entier par nombre comme indicateur, pour conserver une relative rapidité, mais on utilisait 8 fois trop de mémoire... Sur un Atari ST disposant de 1 Mo de RAM, on peut espérer trouver les nombres premiers jusqu'à près de 8 millions, réservation faite d'un peu de place pour le programme. Et nous allons faire en sorte que ces millions de bits soient traités dans les meilleurs délais !

Mais avant de détailler le mécanisme du crible, c'est-à-dire les « itérations », il est nécessaire d'initialiser la zone mémoire représentative de la suite des entiers, à savoir le crible lui-même. Comme nous l'avons expliqué plus haut, il a été décidé d'allouer 1 bit à chaque entier naturel, et de mettre ce bit à 1 lorsque l'entier est décrété « non premier ». Les

nombres premiers seront donc tous ceux dont le bit indicateur est resté à zéro. Ainsi, il suffirait d'initialiser la zone mémoire avec des zéros, et commencer les recherches à partir de 2. Cependant le traitement des multiples de 2 est terriblement long, par rapport au reste : pensez donc, un nombre sur deux ! Mais alors, une fois rayés les multiples de 2, le crible ressemblerait à ceci :

	0	1	(2)	3	~~4~~	5	~~6~~	7	~~8~~	9	~~10~~	11	~~12~~	13	...
Crible :	0	0	0	0	1	0	1	0	1	0	1	0	1	0	...

Fig. 1 : Etat du crible.

Mise à part la séquence binaire spéciale correspondant à 0, 1, 2, le crible n'est ensuite qu'une alternance de 0 et de 1. Si simple et répétitive qu'il suffit en fait de la construire dès le début pour s'affranchir des multiples de 2. En fait, nous allons poursuivre cette idée et initialiser le crible avec les multiples de 2 et de 3. Pourquoi pas alors avec 5, 7, 11... ? Parce que pour initialiser notre crible, il nous faut déposer en mémoire un motif binaire périodique, pour pouvoir l'intégrer dans une boucle. Avec 2 et 3, il faut 3 octets ou mots pour constituer le motif périodique. Avec 2, 3, 5, 7, il faudrait déjà 105 octets pour constituer le motif...

Optimisations

Nous avons vu que le crible pouvait se passer des nombres pairs, et que le « marqueur » des multiples du nombre premier en cours pouvait sauter de 2 en 2. Ce n'est pas tout, pour cocher les multiples d'un nombre premier p, le marqueur peut commencer à p^2. En effet, il est inutile de cocher 3p, 5p, 7p,... (p-2)p, car ces nombres sont par essence des multiples de nombres inférieurs à p, premiers ou non, et ont dans tous les cas déjà été cochés.

Mais alors, les itérations peuvent cesser à $\sqrt{N}$, N désignant la fin de la recherche. En effet, une fois atteinte cette valeur, le marqueur commencera à cocher au delà de N, c'est déjà fini ! On est alors fixés sur les nombres non encore cochés entre $\sqrt{N}$ et N (donc encore candidats à la primalité) : ils sont bel et bien premiers, et il n'y a plus qu'à les ranger dans la table.



RÉSULTATS

Le tableau ci-après donne les lois de régression de la quantité d'itérations en fonction de N. Le crible est presque linéaire, ce qui est rare. Le programme présenté, en assembleur, sort les nombres premiers jusqu'à 150 000 dans la première seconde, et jusqu'à 1 million en 9"7.

Régressions sur la quantité de calculs	
Version	**Q : quantité d'itérations n : fin de la recherche**
1.0	$Q = 1\,020\ n^{1.87}$
1.1	$Q = 109\ n^{1.96}$
2.0	$Q = 120\ n^{1.23}$
3.0	$Q = 76\ n^{1.05}$

Nous pouvons maintenant nous permettre de jeter un regard hautain à la fonction π (n) présentée au début du chapitre, et avoir une pensée émue pour les mathématiciens nommés, qui ont passé leur vie à évaluer cette fonction, et tester des centaines de milliers de nombres premiers, aidés de nombreux assistants car tous les calculs s'effectuaient à la main. La courbe (fig. 2) donne une idée de la validité des résultats ; rappelons pour cela l'expression théorique de PI :

$$\pi(n) = \frac{n}{\ln n}$$

ainsi que l'approximation établie par Legendre sur une base statistique :

$$\pi(n) = \frac{n}{\ln n - 1{,}08366}$$

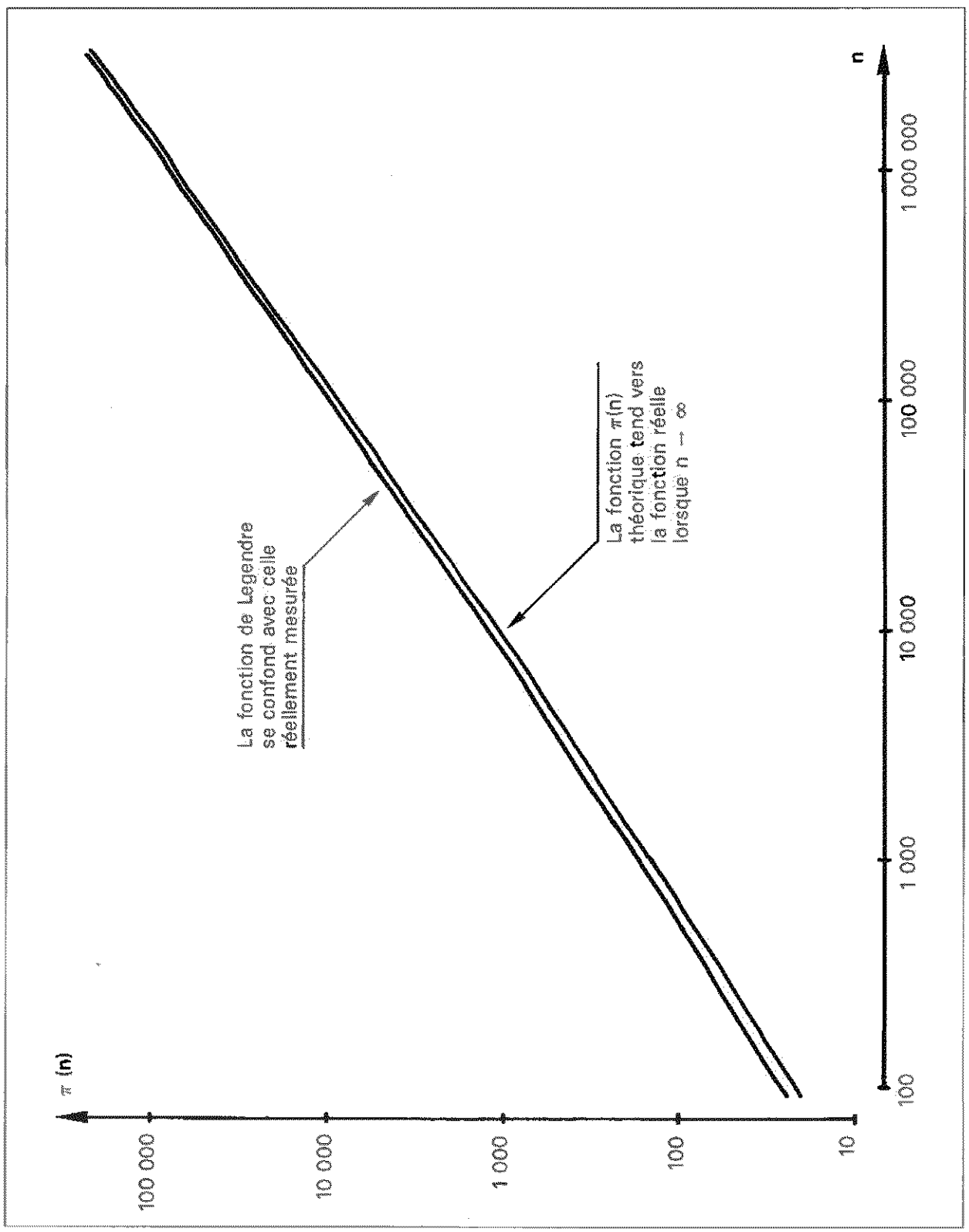


Fig. 2 : La fonction π (n) en fonction de n.



```
;
;*****************************************
;*     GENERATEUR DE NOMBRES PREMIERS    *
;*          crible d'Eratosthène         *
;*                                       *
;* (c) Jean Guérard - Editions WEKA 1991 *
;*****************************************
;
        .INCLUDE 'MYLIB2.IS'

dernier = 10000                         ;fin de la recherche
racine  = 100                           ;sqr(dernier)

        start                           ;macro: début d'un programme
main:
        super       #0                  ;macro gemdos: mode superviseur
        move.l      d0,savesp

        cls                             ;macro: effacer l'écran
        print       'début '            ;indicateur de début du calcul
        move.l      $4ba,count          ; TIMER: pour la durée d'éxécution

        move.l      #dernier,d7
        move.l      #racine,d5
        movea.l     #crible,a5          ;tableau du crible, bit par bit
        movea.l     #table,a4           ;tableau des nombres premiers, sur 32 bits

        movea.l     a5,a0               ;initialisation avec multiples de 2 et de 3
        move.l      d7,d3               ;calcul de la taille du crible:
        asr.l       #5,d3               ;d3 = nombre de longs mots dans le crible
        divu.w      #3,d3               ;car le masque est effectué par 3 octets
        addq.l      #1,d3               ;par sécurité

        move.l      #$5dd7755d,d0       ;masques binaires des multiples
        move.l      #$d7755dd7,d1       ;de 2 et de 3
        move.l      #$755dd775,d2       ;
remp:
        move.l      d0,(a0)+            ;remplissage
        move.l      d1,(a0)+
        move.l      d2,(a0)+
        dbra        d3,remp
        move.b      #$55,(a5)           ;au début du crible, 2 et 3 sont premiers

        move.l      #2,(a4)+            ;2 entre directement au début de la table
        move.l      #3,d6               ;3 aussi, mais on s'en servira plus tard
        move.l      d6,(a4)+
;        clr.l       a3                 ;##### compteur d'itérations
loop:
;-----------------------------------------
;recherche du prochain nombre premier
;entree: d6.l nombre premier actuel, a partir duquel il faut commencer
;         la recherche.
```

(suite)

```
;sortie: d6.l prochain nombre premier
suivant:
        addq.l      #2,d6               ;recherche de 2 en 2 sur les nombres impairs
scan:
        move.l      d6,d0               ;on cherche dans d0.l l'octet du crible qui
        asr.l       #3,d0               ; contient le bit pointé par d6.l: d0=d6/8
        btst.b      d6,0(a5,d0.l)       ;test de ce bit

;        addq.l      #1,a3               ;##### compteur d'itérations
        beq.s       trouve
        addq.l      #2,d6               ;si non trouvé, passage au candidat suivant
        cmp.l       d6,d7               ;mais sortie si la fin de la recherche
        bgt.s       scan                ;est atteinte
        bra.s       fin
trouve:
        move.l      d6,(a4)+            ;on place le nombre premier dans la table
        cmp.l       d5,d6               ;dès que les nombres sont > SQR(dernier),
        bgt.s       loop                ; on passe directement au remplissage
;                                       ; de la table
        move.l      d6,d4               ;sinon, appliquer le masque sur le crible
        mulu.w      d4,d4               ;on commence à partir du carré du nombre
        move.l      d6,d1               ;d4: pointeur masquage, d1: pas du masque:
        add.l       d1,d1               ; on masque de 2 en 2
mask:
        move.l      d4,d0               ;d0 = adresse relative de l'octet
        asr.l       #3,d0               ; concerné dans le crible
        bset.b      d4,0(a5,d0.l)       ;on met à 1 le bit correspondant

;        addq.l      #1,a3               ;##### compteur d'itérations

        add.l       d1,d4               ;on additionne le pas au pointeur
        cmp.l       d4,d7               ; et on masque le crible jusqu'à la fin
        bgt.s       mask
        bra.s       loop                ;puis retour à la recherche pour le
;                                       ; nombre premier suivant
; c'est fini!
;
fin:
        move.l      a4,d0               ;calcul de la quantité de nombres premiers
        movea.l     #table,a0           ; qui ont été introduits dans la table:
        sub.l       a0,d0               ; Q = (a4-#table)/4 car un nombre est
        asr.l       #2,d0               ; codé sur 4 octets
        move.l      d0,pi               ;pi(n): la fonction recherchée

        move.l      $4ba,count2         ;2e valeur du TIMER: pour la durée du calcul

        print       'fin'
        crlf

        move.l      pi,d5               ;les lignes qui suivent servent à
        movea.l     #table,a4           ; afficher les résultats:
```



(suite)

```
        divu.w      #6,d5               ; à raison de 6 nombres par lignes, on
        move.l      d5,d6               ; affiche les premiers et les derniers
        moveq.l     #9,d5               ; nombres premiers du crible, pour se
        bsr         affligne            ; convaincre du bon fonctionnement
        print       '...'               ; du programme !
        crlf
        movea.l     #table,a4
        subq.l      #3,d6
        mulu.w      #24,d6
        adda.l      d6,a4
        moveq.l     #3,d5
        bsr         affligne
        crlf

        print       'quantité  : '
        move.l      pi,d3
        bsr         aff_dec
        crlf

        print       'durée     : '
        move.l      count2,d3
        sub.l       count,d3
        bsr         aff_dec
        crlf

        print       'itérations: '
        move.l      a3,d3
        bsr         aff_dec
        crlf

        super       savesp              ;retour au mode utilisateur
        cconin

        rts                             ;fin du programme

;*************************************
affligne:                               ;routine pour l'affichage dune série de
        moveq.l     #5,d4               ; nombres premiers: le nombre de lignes
affli:                                  ; de 6 nombres est donné dans d5
        move.l      (a4)+,d3
        bsr         aff_dec
        cconout     #32
        cconout     #32
        dbra        d4,affli
        crlf
        dbra        d5,affligne
        rts

        .BSS
```

(suite)

```
pi:      .DS.l 1
savesp:  .DS.l 1
count:   .DS.l 1
count2:  .DS.l 1
crible:  .DS.l 10+dernier/32
table:   .DS.l 200000

         .DATA
;
; insérer ici, dans le segment DATA, la routine
; d'affichage d'un nombre en décimal:  aff_dec
;

         .END
; routine d'affichage d'un nombre en décimal
;
; entrée: nombre dans d3.l

aff_dec:
          movem.l   d0-d6/a0-a3,-(sp)   ;sauvegarde des registres
          lea.l     masque,a0           ;puissance de 2 codée en BCD
          clr.b     (a0)+               ;*
          clr.b     (a0)+               ;*
          clr.b     (a0)+               ;* initialisation : 00000001
          clr.b     (a0)+               ;*
          move.b    #$01,(a0)           ;*
          lea.l     affstr,a1           ;le nombre à afficher, en BCD
          clr.b     (a1)+               ;*
          clr.b     (a1)+               ;*
          clr.b     (a1)+               ;* initialisation : 00000000
          clr.b     (a1)+               ;*
          clr.b     (a1)                ;*

          tst.l     d3                  ;test du signe
          bpl.s     aff_d0              ;si N est négatif,
          cconout   #'-'                ; on affiche un - et on traite
          neg.l     d3                  ; la valeur positive de N
aff_d0:
          lea.l     finmasque,a2        ;sauvegarde du pointeur
          moveq.l   #0,d4               ;le masquage commence au bit 0
aff_d1:
          btst      d4,d3               ;test des bits dans l'ordre croissant
          beq.s     aff_d2              ;si le bit est à 1, on effectue
          movea.l   a2,a0               ; l'addition du masque (puissance
          lea.l     finaffstr,a1        ; de 2 correspondante):
          clr.w     d0                  ;(clr et addq servent à mettre à 0
          addq.w    #1,d0               ; le bit X du registre CCR)
          abcd.b    -(a0),-(a1)         ;*
          abcd.b    -(a0),-(a1)         ;* addition en BCD:
          abcd.b    -(a0),-(a1)         ;* source : a0 (puissance de 2)
          abcd.b    -(a0),-(a1)         ;* dest.  : a1 (total à afficher)
```



(suite)

```
            abcd.b      -(a0),-(a1)          ;*
aff_d2:
            movea.l     a2,a0                ;puis, multiplication du masque par 2
            moveq.l     #4,d1                ; sur 5 octets ( = 10 chiffres) en BCD
            clr.w       d0                   ;(clr et addq servent à mettre à 0
            addq.w      #1,d0                ; le bit X du registre CCR)
aff_d3:
            move.b      -(a0),d0             ;à cause du mode d'adressage de
            abcd.b      d0,d0                ; l'instruction ABCD, on est obligé
            move.b      d0,(a0)              ; de passer par un registre
            dbra        d1,aff_d3            ; d'où la boucle sur d1
            addq.w      #1,d4                ;passage au bit suivant
            cmpi.w      #32,d4               ;on boucle tant que les 32 bits
            bne         aff_d1               ; de N (d3.l) n'ont pas été traités
aff_d4:
            lea.l       affstr,a3            ;affichage de la chaine BCD:
            moveq       #4,d4                ;compteur pour 5 octets = 10 chiffres
aff_d5:
            move.b      (a3)+,d6             ;on isole un octet (2 chiffres)
            move.b      d6,d5                ;sauvegarde
            andi.w      #$f0,d5              ;le 1er chiffre est dans le quartet haut
            sub.w       d5,d6                ;1er chiffre dans d5; 2eme dans d6
            lsr.b       #4,d5                ;ramener le chiffre dans le quartet bas
            addi.b      #$30,d5              ;+ $30 : code ASCII
            cconout     d5                   ;affichage
            addi.b      #$30,d6              ;idem avec le 2eme chiffre
            cconout     d6
            dbra        d4,aff_d5            ;passage à l'octet suivant
aff_d6:
            movem.l     (sp)+,d0-d6/a0-a3    ;restitution des registres
            rts                              ;fin de la routine

            .BSS
masque:
            .DS.b 5     ;réservation de 5 octets pour la puissance de 2
finmasque:
affstr:
            .DS.b 5     ;de même pour le nombre à afficher
fineffstr:




;
; Quelques macro-instructions bien utiles:
;
; CRLF  : retour chariot et saut de ligne = RETURN
; CLS   : efface l'écran
;
```

(suite)

```
        .MACRO crlf
        movem.l     d0-d2/a0-a2,-(sp)
        cconout     #13
        cconout     #10
        movem.l     (sp)+,d0-d2/a0-a2
        .ENDM

        .MACRO cls
        movem.l     d0-d2/a0-a2,-(sp)
        cconout     #12
        cconout     #27
        cconout     #"Y"
        cconout     #33
        cconout     #33
        movem.l     (sp)+,d0-d2/a0-a2
        .ENDM
```

A noter à la fin du listing une intéressante routine d'affichage des nombres en décimal sur 32 bits. La version 2.0 pouvait s'en passer, puisqu'au delà de 20000 les temps de calcul devenaient prohibitifs, mais pour le crible d'Eratosthène ces mêmes temps ne deviennent significatifs qu'à partir de 100000 ! Il fallait donc concevoir une routine d'affichage décimal sur 32 bits, ce qui permet désormais d'exploiter le programme jusqu'à 30 millions si l'on dispose de 4 Mo de RAM. Cette routine peut être installée dans n'importe quel logiciel en assembleur : le nombre à afficher doit être dans d3.l (les nombres négatifs sont possibles), l'affichage se fait à la position courante du curseur, et aucun registre du 68000 n'est affecté par la routine, ce qui permet de l'insérer n'importe où comme contrôle, par exemple.



4/1.2.3

La récursivité

UN EXEMPLE DE PROGRAMME

La récursivité fait partie des quelques points délicats de la programmation en Basic GFA. Voici un exemple à caractère ludique qui illustre graphiquement un processus récursif. Pour mieux comprendre le fonctionnement de la récursivité, nous allons faire un peu d'histoire. Les premiers micro-ordinateurs «familiaux» sont apparus sur le marché vers 1980. La plupart des langages de programmation qui les accompagnaient, étaient généralement interprètes et rarement compilés. Nous vivions l'ère du «GOTO», symbole vivant de la programmation «bidouillée» ! C'est alors qu'IBM s'est intéressé au marché de la micro-informatique.

Nous avons pu assister à la naissance du PC (*Personnal Computer*). Les entreprises se sont très vite aperçues des intérêts que présentaient de telles machines :

— implantation d'une machine informatique indépendante sur chaque poste de travail ;

— réduction des surcharges d'utilisation des grosses machines informatiques (ordinateurs et mini-ordinateurs).

Mais si ces machines étaient intéressantes, il fallait des logiciels d'application performants. C'est la raison qui a motivé le développement de langages tels que le C, le Turbo Pascal, etc.

Contrairement au Basic originel (où toutes les cinq lignes, on pouvait trouver un GOTO qui ramenait quarante lignes en arrière ou cent lignes en avant), il fallait des langages lisibles par n'importe qui. Dans cette optique, des langages à syntaxe très stricte ont été développés. Les programmes «bidouille» ont disparu au profit de langages structurés utilisant :

— des procédures ;

— des fonctions ;

— des variables globales et locales ;

— des passages de paramètres.

Faisons quelques rappels sur les utilisations théoriques de ces notions.

Une procédure est un sous-programme susceptible de fournir à partir de données en entrée, des données en sortie sans perturber le reste du programme. Précisons cette notion à l'aide des exemples 1 et 2.

Exemple 1

```
REM ------------------------------------------------------------
REM   Illustration de la pollution du programme
REM   principal par une procédure
REM   Cas des variables GLOBALES.
REM ------------------------------------------------------------
REM ============  Programme Principal  =======================
i%=3
PRINT "i=";i%                    ! Affichage:   i=3
toto
PRINT "i=";i%                    ! Affichage:   i=5
END
REM ============  Procédure  =================================
PROCEDURE toto
  i%=5
RETURN
```

Exemple 2

```
REM ------------------------------------------------------------
REM   exemple de programmation structurée
REM   Cas des variables LOCALES
REM ------------------------------------------------------------
REM ============  Programme Principal  =======================
i%=3
PRINT "i=";i%                    ! Affichage:   i=3
titi
PRINT "i=";i%                    ! Affichage:   i=3
END
REM ============  Procédure  =================================
PROCEDURE titi
  LOCAL i%
  PRINT "i(local)=";i%           ! Affichage:   indéterminé (i=0)
  i%=5
  PRINT "i(local)=";i%           ! Affichage:   i(local)=5
RETURN
```

Contrairement à la procédure **toto**, la procédure **titi** ne perturbe pas le reste du programme. C'est la déclaration de la variable **i%** en variable locale qui évite la pollution du programme principal. Cette variable locale est déclarée au début de l'exécution de la procédure **titi** et est automatiquement libérée à sa fin. Vu du programme principal, la variable



locale i% n'existe pas. Lors de l'exécution de la procédure titi, il existe deux variables i% contenant des valeurs différentes ! Attention aux confusions !!!

Dans le monde des développeurs sur Atari, seuls les puristes respectent toutes ces conventions. Mais pour des sociétés qui commercialisent des logiciels à usage professionnel, ces notions sont très importantes. Prenons le cas d'une équipe de cinquante informaticiens. Lorsqu'un programmeur met au point une procédure, il ne va pas demander au quarante-neuf autres s'il peut utiliser la variable i%. Il l'utilisera sans se poser de questions, en la déclarant en variable locale. Sa seule préoccupation vis-à-vis de ses collègues sera de définir quels sont les paramètres d'entrée et de sortie de sa procédure. Ceci introduit la notion de passage de paramètre.

Il existe deux types de passage de paramètre :

— le passage par adresses ;

— le passage par valeurs.

Suivant que la procédure retourne ou non un résultat, on utilisera le premier ou le second type de passage de paramètres. Les deux exemples qui suivent illustrent ces deux techniques.

Exemple 3

```
REM ------------------------------------------------------------
REM   Illustration du passage de paramètre par valeurs
REM ------------------------------------------------------------
REM ==============  Programme Principal  ======================
i%=3
PRINT "i=":i%                    ! Affichage:   i=3
tutu(i%,5)
PRINT "i=":i%                    ! Affichage:   i=5
PRINT "x=":x%                    ! Affichage:   indéterminé (x=0)
PRINT "y=":y%                    ! Affichage:   indéterminé (y=0)
END
REM ==============  Procédure  ================================
PROCEDURE tutu(x%,y%)
  i%=5
  PRINT "x=":x%                  ! Affichage:   x=3
  PRINT "y=":y%                  ! Affichage:   y=5
  x%=4
  y%=6
RETURN
```

Lors de l'appel de la procédure **tutu**, les variables **x%** et **y%** sont déclarées en variables locales. Puis, le contenu de la variable **i%** et la valeur de la constante 5 sont transférés dans les variables **x%** et **y%**. Comme **x%** et **y%** sont effacées à la fin de la procédure **tutu**, **x%** et **y%** ont des valeurs indéterminées dans le programme principal.

Remarque :

tutu est définie de telle façon que la variable i% est globale. Une modification de cette variable engendrerait une pollution du programme principal.

Exemple 4

```
REM ------------------------------------------------------
REM   Illustration du passage de paramètres par adresses
REM ------------------------------------------------------
REM ============  Programme Principal  ====================
i%=3
x%=5
PRINT "i=":i%                    ! Affichage:   i=3
PRINT "x=":x%                    ! Affichage:   x=5
tata(i%)
PRINT "i=":i%                    ! Affichage:   i=3
PRINT "x=":x%                    ! Affichage:   indéterminé (x=0)
END
REM ============  Procédure  ==============================
PROCEDURE tata(VAR x%)
  PRçNT "x=":x%                  ! Affichage:   x=3
  x%=7
RETURN
```

Lors de l'appel de la procédure **tata**, contrairement à la procédure **tutu**, une nouvelle variable n'est pas créée; un second nom, en l'occurrence **x%**, est attribué à la variable **i%**. Mais cette double dénomination n'existe que durant l'exécution de la procédure **tata**.

Physiquement, les variables **x%** et **i%** ont la même adresse dans la mémoire (d'où le nom : «passage de paramètres par adresses»). Toutefois, une modification de **x%** dans la procédure **tata** entraîne uniquement une modification de **i%** dans le programme principal. La variable **x%** du programme principal n'est pas modifiée.

Le meilleur moyen de mesurer toutes les finesses de ces notions, est de faire quelques essais sur votre Atari en vous basant sur les quatre exemples précités. Pour refermer et justifier cette parenthèse de rappels historiques et théoriques, il faut préciser que sans les procédures avec passage de paramètres, un algorithme récursif sur un micro-ordinateur est impossible. L'exemple de programme qui suit, basé sur la récursivité, utilise ces techniques de programmation. Afin de bien attirer votre attention sur cela, nous avons décidé de prendre pour exemple un programme à caractère ludique : le Démineur.

Le but du jeu est de signaliser par l'intermédiaire d'un petit drapeau toutes les mines cachées dans une grille. Mais si par malchance, ou par erreur de jugement, vous tombez sur une mine, adieu !



A l'aide du bouton gauche de la souris, on découvre une case (par pur hasard au début du jeu). On peut tomber sur trois types de cases :

— une mine, et dans ce cas on a perdu ;

— une case blanche, ce qui a pour effet de découvrir toute une zone de cases blanches entourées par des cases à numéro ;

— une case à numéro contenant un nombre pouvant aller de 1 à 8.

Dans ce dernier cas, le chiffre découvert indique le nombre exact de mines qui se trouvent dans les huit cases adjacentes à la case que l'on vient de découvrir. C'est grâce à ces informations numériques que l'on peut localiser les mines. Lorsqu'on estime en avoir trouvé une, on applique à l'aide du bouton droit de la souris un petit drapeau sur la case suspecte. Le joueur gagne la partie lorsqu'il a réussi à placer un drapeau sur toutes les mines présentes dans la grille. En jouant quelques parties, vous comprendrez mieux les subtilités de ce jeu de réflexion.

FAISONS UNE ANALYSE DU PROBLÈME

Lorsque l'on découvre une case blanche, il est certain que les huit cases adjacentes ne contiennent aucune mine, sinon, la case découverte contiendrait un numéro. Pour simplifier le «travail» du joueur, il est souhaitable que le programme découvre automatiquement les huit cases adjacentes à celles que l'on vient de découvrir. Mais si d'autres cases blanches sont présentes parmi les huit cases découvertes, il faut réitérer l'opération. C'est la tâche de la procédure récursive. Dans la littérature spécialisée, ce type d'opération est appelé *region growing* ou croissance de région. La procédure doit s'arrêter lorsque la totalité de la région des cases blanches connexes à la case de départ est découverte. Cette région est alors entourée par une frontière de cases de type numéro qui fournit au joueur les informations nécessaires à la poursuite du jeu. Un exemple monochrome en figure 1.

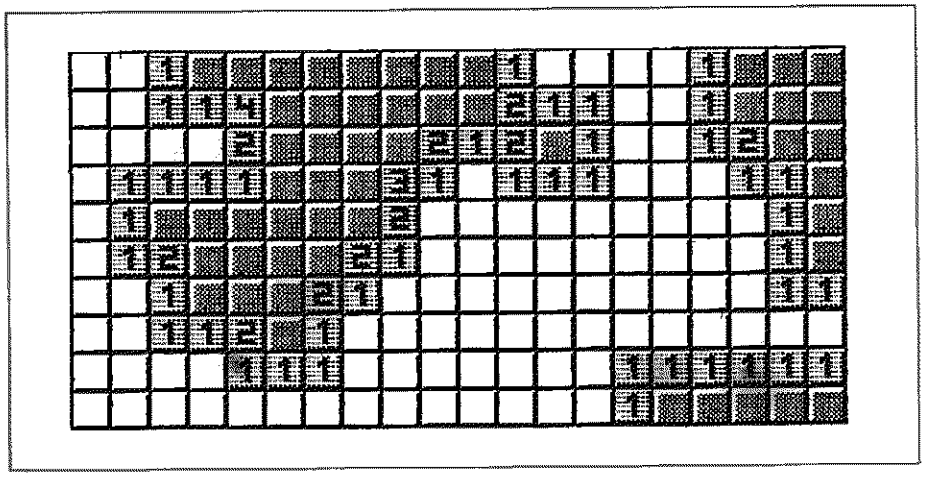


Fig. 1 : Exemple de région détectée.

DÉTAILLONS LE PROCESSUS DE CROISSANCE DE RÉGION

Dans la procédure **détection**, lorsqu'une case blanche est découverte, on l'affiche pour faire ensuite appel à la procédure de **region_growing (ptr_x,ptr_y)**. Celle-ci va rechercher successivement dans les huit directions: nord, nord-est, est, sud-est, sud, sud-ouest, ouest et enfin nord-ouest, la présence d'une autre case blanche. Si elle ne trouve pas de case blanche, c'est qu'il y a une case à numéro qu'elle affiche. Si elle trouve une case blanche, elle appelle la procédure **region_nord(x,y)** ou la procédure **region nord-est(x,y)**, etc. en passant comme paramètres, les valeurs des coordonnées de la nouvelle case blanche courante. Les procédures **region_XXXX(x,y)** testent, après affichage de la nouvelle case courante, celles qui lui sont adjacentes. Si les tests sont négatifs, la procédure affichera les numéros des cases. Par contre, si un des test est positif, les procédures **region_XXXX(x,y)** se rappelent mutuellement.

Prenons un exemple (figure 2).

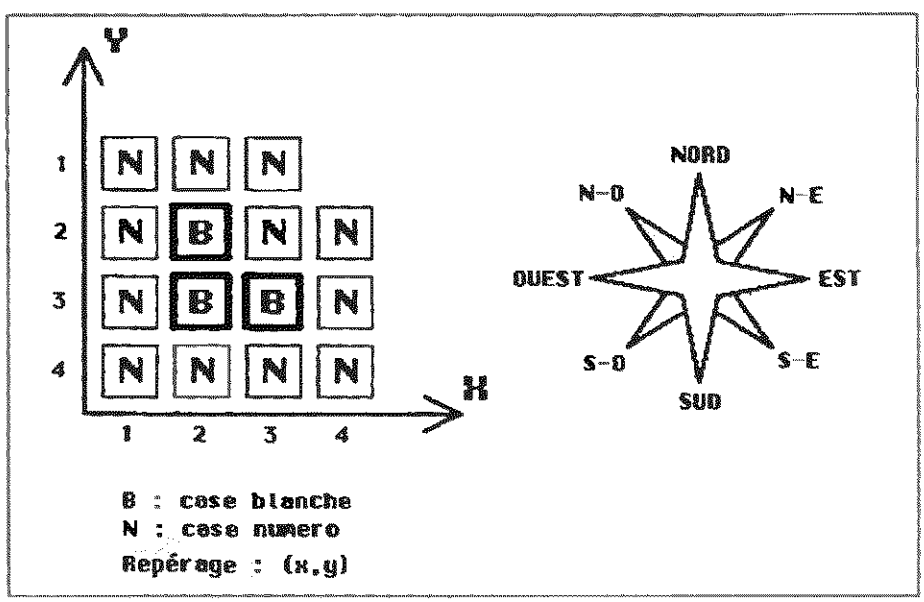


Fig. 2 : Illustration de l'algorithme de region-growing().

```
DETECTION => bouton gauche de la souris activé.
    ? case blanche                  OUI => affichage  (2,2)
    REGION_GROWING(2,2)
    |   ? TEST_NORD                 OUI => affichage  (2,3)
    |   REGION_NORD(2,3)
    |   |   ? TEST_NORD             NON => affichage  (2,4)
    |   |   ? TEST_NORD_EST         NON => affichage  (3,4)
    |   |   ? TEST_EST              NON => affichage  (3,3)
    |   |   ? TEST_SUD_EST          OUI => affichage  (3,2)
    |   |   REGION_SUD_EST(3,2)
```



(suite)

```
                ? TEST_NORD          DEJA TESTE
                ? TEST_NORD_EST      NON => affichage  (4,3)
                ? TEST_EST           NON => affichage  (4,2)
                ? TEST_SUD_EST       NON => affichage  (4,1)
                ? TEST_SUD           NON => affichage  (3,1)
                ? TEST_SUD_OUEST     NON => affichage  (2,1)
                ? TEST_OUEST         DEJA TESTE
                ? TEST_NORD_OUEST    DEJA TESTE
            RETURN de REGION_SUD_EST(3,2)
            ? TEST_SUD               DEJA TESTE
            ? TEST_SUD_OUEST         NON => affichage  (1,2)
            ? TEST_OUEST             NON => affichage  (1,3)
            ? TEST_NORD_OUEST        NON => affichage  (1,4)
        RETURN de REGION_NORD(2,2)
        ? TEST_NORD_EST              DEJA TESTE
        ? TEST_EST                   DEJA TESTE
        ? TEST_SUD_EST               DEJA TESTE
        ? TEST_SUD                   DEJA TESTE
        ? TEST_SUD_OUEST             NON => affichage  (1,1)
        ? TEST_OUEST                 DEJA TESTE
        ? TEST_NORD_OUEST            DEJA TESTE
    RETURN de REGION GROWING
RETURN de DETECTION
```

Il apparaît dans ce synoptique que les cases déjà testées doivent être mémorisées pour que le système ne reboucle pas sur lui-même. Cela est réalisé dans les deux tableaux **tab(x,y,1)** et **tab(x,y,2)**. Chaque fois qu'une case est affichée sur l'écran, son contenu est transféré de cette case du tableau 2 (tableau caché) dans la case correspondante du tableau 1 (tableau de ce qui est affiché sur l'écran). La lecture du tableau 1 permet de savoir si une case a déjà été testée. Une lecture attentive du listing permet de mieux comprendre tous les détails du processus.

Pour conclure ce chapitre, il reste à faire quelques remarques :

- Dans un souci de clarté de lecture du listing, nous avons décomposé le processus récursif en un grand nombre de procédures.

- Pour ne pas alourdir le listing nous n'avons pas réalisé une procédure de saisie des paramètres (largeur et hauteur de la grille, niveau de difficulté). Ces paramètres sont accessibles au début du programme en GFA.

- La gestion de l'affichage et de la détection de la position de la souris ne présentant aucune difficulté, nous n'avons pas jugé nécessaire de détailler cette partie. La seule astuce utilisée consiste à afficher les cases de la grille par l'intermédiaire de l'instruction PUT **x,y,image$** où les images sont des carrés (de seize par seize en monochrome et de dix par

dix en basse résolution) définis lors de la phase d'initialisation à l'aide des DATA situés en fin de listing.

- Le programme fonctionne en basse et en haute résolution. Si l'on se trouve en moyenne résolution, il faut passer en basse résolution.

Bonne chance sur les mines !

```
' ------------
' "LE DEMINEUR"
' ------------
DEFINT "a-z"
' ====================  Paramètres  ==========================
tail_x=20                ! Largeur de la grille
tail_y=15                ! Hauteur de la grille
niveau=1                 ! Niveau du jeu (1 à 10)
' ====================  Gestion affichage et souris  =========
' ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
som_data                 ! LIGNES A EFFACER APRES LE 1ER
~INP(2)                  ! LANCEMENT DU PROGRAMME...
' ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
main
PROCEDURE main
  initialisation
  REPEAT
    calcul_affichage
    tooche=0
    detection
    IF out!
      bouton=FORM_ALERT(1,"[1][|  |       RECOMMENCER ????       |  ][OUI|NON]")
      IF bouton=1
        out!=FALSE
      ENDIF
    ENDIF
  UNTIL out!=TRUE OR tooche=27
  END
RETURN
PROCEDURE initialisation
  rez=XBIOS(4)
  IF rez=0
    tx=320
    ty=200
    tail_x=MIN(30,tail_x)
    tail_x=MAX(10,tail_x)
    tail_y=MIN(18,tail_y)
    tail_y=MAX(8,tail_y)
    x0=SUB(152,MUL(5,tail_x))
    y0=SUB(97,MUL(5,tail_y))
    offset=10
```



(suite)

```
      RESTORE couleur
      FOR i=0 TO 15
        READ couleurs
        SETCOLOR i,couleurs
      NEXT i
      DEFFILL 0,1
      DEFTEXT 1,16,0,13
      max_data=43
      RESTORE low_rez
    ELSE IF rez=1
      ~FORM_ALERT(1,"[1][|  |  Le demineur ne fonctionne  |  qu'en HAUTE et
BASSE  |  resolution
 !!!][RETURN]")
      END
    ELSE IF rez=2
      tx=640
      ty=400
      tail_x=MIN(38,tail_x)
      tail_x=MAX(10,tail_x)
      tail_y=MIN(23,tail_y)
      tail_y=MAX(8,tail_y)
      x0=SUB(304,MUL(8,tail_x))
      y0=SUB(184,MUL(8,tail_y))
      offset=16
      DEFFILL 1,1
      DEFTEXT 1,16,0,13
      max_data=19
      RESTORE high_rez
    ENDIF
    PBOX 0,0,tx,ty
    DIM image$(13),tab(ADD(tail_x,1),ADD(tail_y,1),2)
    FOR i=0 TO 13
      FOR ii=1 TO max_data
        READ val_image
        image$(i)=image$(i)+MKI$(val_image)
      NEXT ii
    NEXT i
    niveau=MIN(10,niveau)
    niveau=MAX(1,niveau)
    nb_bombe=MUL(DIV(MUL(tail_x,tail_y),20),niveau)
  RETURN
  PROCEDURE calcul_affichage
    DEFMOUSE 2
    nb_bombe_trouve=0
    FOR i=0 TO ADD(tail_x,1)
      FOR ii=0 TO ADD(tail_y,1)
        tab(i,ii,2)=0
        tab(i,ii,1)=10
```

(suite)

```
    NEXT ii
  NEXT i
  i=0
  REPEAT
    x=ADD(RANDOM(tail_x),1)
    y=ADD(RANDOM(tail_y),1)
    IF tab(x,y,2)<9
      INC i
      tab(x,y,2)=9
      FOR xx=SUB(x,1) TO ADD(x,1)
        FOR yy=SUB(y,1) TO ADD(y,1)
          tab(xx,yy,2)=MIN(9,ADD(tab(xx,yy,2),1))
        NEXT yy
      NEXT xx
    ENDIF
  UNTIL i=nb_bombe
  FOR i=0 TO ADD(tail_x,1)
    tab(i,0,2)=10
    tab(i,ADD(tail_y,1),2)=10
  NEXT i
  FOR i=0 TO ADD(tail_y,1)
    tab(0,i,2)=10
    tab(ADD(tail_x,1),i,2)=10
  NEXT i
  PBOX 0,0,tx,ty
  FOR x=ADD(x0,offset) TO ADD(x0,MUL(offset,tail_x)) STEP offset
    FOR y=ADD(y0,offset) TO ADD(y0,MUL(offset,tail_y)) STEP offset
      PUT x,y,image$(10)
    NEXT y
  NEXT x
  DEFMOUSE 0
RETURN
PROCEDURE detection
  SHOWM
  REPEAT
    tooche=ASC(INKEY$)
    IF tooche<>27
      mk=MOUSEK
      IF mk<>0
        mx=DIV(SUB(MOUSEX,x0),offset)
        my=DIV(SUB(MOUSEY,y0),offset)
        IF mk=1 AND mx>0 AND my>0 AND mx=<tail_x AND my=<tail_y
          IF tab(mx,my,1)=10
            PUT ADD(x0,MUL(offset,mx)),ADD(y0,MUL(offset,my)),image$(12)
            REPEAT
            UNTIL MOUSEK=0
            IF tab(mx,my,2)=9
              affichage_bombes
```



(suite)

```
            ELSE IF tab(mx,my,2)=0
              PUT  DD(x0,MUL(offset,mx)),ADD(y0,MUL(offset,my)),
image$(tab(mx,my,2))
              tab(mx,my,1)=tab(mx,my,2)
              region_growing(mx,my)
            ELSE
              PUT ADD(x0,MUL(offset,mx)),ADD(y0,MUL(offset,my)),
image$(tab(mx,my,2))
              tab(mx,my,1)=tab(mx,my,2)
            ENDIF
          ENDIF
        ELSE IF mk=2
          IF tab(mx,my,1)=10
            tab(mx,my,1)=11
            IF tab(mx,my,2)=9
              INC nb_bombe_trouve
            ENDIF
          ELSE IF tab(mx,my,1)=11
            tab(mx,my,1)=10
            IF tab(mx,my,2)=9
              DEC nb_bombe_trouve
            ENDIF
          ENDIF
          PUT ADD(x0,MUL(offset,mx)),ADD(y0,MUL(offset,my)),image$(tab(mx,my,1))
          IF nb_bombe_trouve=nb_bombe
            FOR x=1 TO tail_x
              FOR y=1 TO tail_y
                xx=ADD(x0,MUL(offset,x))
                yy=ADD(y0,MUL(offset,y))
                PUT xx,yy,image$(tab(x,y,2))
              NEXT y
            NEXT x
            GRAPHMODE 3
            TEXT DIV(SUB(tx,260),2),13,"GAGNE...  Tapez une touche"
            GRAPHMODE 0
            ~INP(2)
            out!=TRUE
          ENDIF
          REPEAT
          UNTIL MOUSEK=0
        ENDIF
      ENDIF
    ELSE
      bouton=FORM_ALERT(2,"[1][|  |          ABANDON ????          |  |][OUI|NON]")
      IF bouton=2
        tooche=0
      ENDIF
    ENDIF
```

(suite)

```
  UNTIL out! OR tooche=27
RETURN
PROCEDURE affichage_bombes
  FOR x=1 TO tail_x
    FOR y=1 TO tail_y
      IF tab(x,y,2)=9
        PUT ADD(x0,MUL(offset,x)),ADD(y0,MUL(offset,y)),image$(tab(x,y,2))
      ELSE IF tab(x,y,1)=11
        PUT ADD(x0,MUL(offset,x)),ADD(y0,MUL(offset,y)),image$(13)
      ENDIF
    NEXT y
  NEXT x
  GRAPHMODE 3
  TEXT DIV(SUB(tx,260),2),13,"PERDU...  Tapez une touche"
  GRAPHMODE 0
  ~INP(2)
  out!=TRUE
RETURN
PROCEDURE affichage_case(ptr_xxx,ptr_yyy)
  IF tab(ptr_xxx,ptr_yyy,2)<10
    PUT ADD(x0,MUL(offset,ptr_xxx)),ADD(y0,MUL(offset,ptr_yyy)),
image$(tab(ptr_xxx,ptr_yyy,2))
    tab(ptr_xxx,ptr_yyy,1)=tab(ptr_xxx,ptr_yyy,2)
  ENDIF
RETURN
' ====================  Récursivité  ========================
PROCEDURE region_growing(ptr_x,ptr_y)
  DEFMOUSE 2
    affichage_case(ptr_x,ptr_y)
    IF tab(ptr_x,SUB(ptr_y,1),2)=0 AND tab(ptr_x,SUB(ptr_y,1),1)<>0
    region_nord(ptr_x,SUB(ptr_y,1))
  ENDIF
  IF tab(ADD(ptr_x,1),SUB(ptr_y,1),2)=0 AND tab(ADD(ptr_x,1),SUB(ptr_y,1),1)<>0
    region_nord_est(ADD(ptr_x,1),SUB(ptr_y,1))
  ENDIF
  IF tab(ADD(ptr_x,1),ptr_y,2)=0 AND tab(ADD(ptr_x,1),ptr_y,1)<>0
    region_est(ADD(ptr_x,1),ptr_y)
  ENDIF
  IF tab(ADD(ptr_x,1),ADD(ptr_y,1),2)=0 AND tab(ADD(ptr_x,1),ADD(ptr_y,1),1)<>0
    region_sud_est(ADD(ptr_x,1),ADD(ptr_y,1))
  ENDIF
  IF tab(ptr_x,ADD(ptr_y,1),2)=0 AND tab(ptr_x,ADD(ptr_y,1),1)<>0
    region_sud(ptr_x,ADD(ptr_y,1))
  ENDIF
  IF tab(SUB(ptr_x,1),ADD(ptr_y,1),2)=0 AND tab(SUB(ptr_x,1),ADD(ptr_y,1),1)<>0
    region_sud_ouest(SUB(ptr_x,1),ADD(ptr_y,1))
  ENDIF
  IF tab(SUB(ptr_x,1),ptr_y,2)=0 AND tab(SUB(ptr_x,1),ptr_y,1)<>0
    region_ouest(SUB(ptr_x,1),ptr_y)
```



(suite)

```
  ENDIF
  IF tab(SUB(ptr_x,1),SUB(ptr_y,1),2)=0 AND tab(SUB(ptr_x,1),SUB(ptr_y,1),1)<>0
    region_nord_ouest(SUB(ptr_x,1),SUB(ptr_y,1))
  ENDIF
  DEFMOUSE 0
RETURN
PROCEDURE region_nord(ptr_xx,ptr_yy)
  affichage_case(ptr_xx,ptr_yy)
  region_test
RETURN
PROCEDURE region_nord_est(ptr_xx,ptr_yy)
  affichage_case(ptr_xx,ptr_yy)
  region_test
RETURN
PROCEDURE region_est(ptr_xx,ptr_yy)
  affichage_case(ptr_xx,ptr_yy)
  region_test
RETURN
PROCEDURE region_sud_est(ptr_xx,ptr_yy)
  affichage_case(ptr_xx,ptr_yy)
  region_test
RETURN
PROCEDURE region_sud(ptr_xx,ptr_yy)
  affichage_case(ptr_xx,ptr_yy)
  region_test
RETURN
PROCEDURE region_sud_ouest(ptr_xx,ptr_yy)
  affichage_case(ptr_xx,ptr_yy)
  region_test
RETURN
PROCEDURE region_ouest(ptr_xx,ptr_yy)
  affichage_case(ptr_xx,ptr_yy)
  region_test
RETURN
PROCEDURE region_nord_ouest(ptr_xx,ptr_yy)
  affichage_case(ptr_xx,ptr_yy)
  region_test
RETURN
PROCEDURE region_test
  test_nord
  test_nord_est
  test_est
  test_sud_est
  test_sud
  test_sud_ouest
  test_ouest
  test_nord_ouest
RETURN
```

(suite)

```
PROCEDURE test_nord
  ptr_xxx=ptr_xx
  ptr_yyy=SUB(ptr_yy,1)
  IF tab(ptr_xxx,ptr_yyy,2)=0 AND tab(ptr_xxx,ptr_yyy,1)<>0
    region_nord(ptr_xxx,ptr_yyy)
  ELSE
    affichage_case(ptr_xxx,ptr_yyy)
  ENDIF
RETURN
PROCEDURE test_nord_est
  ptr_xxx=ADD(ptr_xx,1)
  ptr_yyy=SUB(ptr_yy,1)
  IF tab(ptr_xxx,ptr_yyy,2)=0 AND tab(ptr_xxx,ptr_yyy,1)<>0
    region_nord_est(ptr_xxx,ptr_yyy)
  ELSE
    affichage_case(ptr_xxx,ptr_yyy)
  ENDIF
RETURN
PROCEDURE test_est
  ptr_xxx=ADD(ptr_xx,1)
  ptr_yyy=ptr_yy
  IF tab(ptr_xxx,ptr_yyy,2)=0 AND tab(ptr_xxx,ptr_yyy,1)<>0
    region_est(ptr_xxx,ptr_yyy)
  ELSE
    affichage_case(ptr_xxx,ptr_yyy)
  ENDIF
RETURN
PROCEDURE test_sud_est
  ptr_xxx=ADD(ptr_xx,1)
  ptr_yyy=ADD(ptr_yy,1)
  IF tab(ptr_xxx,ptr_yyy,2)=0 AND tab(ptr_xxx,ptr_yyy,1)<>0
    region_sud_est(ptr_xxx,ptr_yyy)
  ELSE
    affichage_case(ptr_xxx,ptr_yyy)
  ENDIF
RETURN
PROCEDURE test_sud
  ptr_xxx=ptr_xx
  ptr_yyy=ADD(ptr_yy,1)
  IF tab(ptr_xxx,ptr_yyy,2)=0 AND tab(ptr_xxx,ptr_yyy,1)<>0
    region_sud(ptr_xxx,ptr_yyy)
  ELSE
    affichage_case(ptr_xxx,ptr_yyy)
  ENDIF
RETURN
PROCEDURE test_sud_ouest
  ptr_xxx=SUB(ptr_xx,1)
  ptr_yyy=ADD(ptr_yy,1)
```



(suite)

```
  IF tab(ptr_xxx,ptr_yyy,2)=0 AND tab(ptr_xxx,ptr_yyy,1)<>0
    region_sud_ouest(ptr_xxx,ptr_yyy)
  ELSE
    affichage_case(ptr_xxx,ptr_yyy)
  ENDIF
RETURN
PROCEDURE test_ouest
  ptr_xxx=SUB(ptr_xx,1)
  ptr_yyy=ptr_yy
  IF tab(ptr_xxx,ptr_yyy,2)=0 AND tab(ptr_xxx,ptr_yyy,1)<>0
    region_ouest(ptr_xxx,ptr_yyy)
  ELSE
    affichage_case(ptr_xxx,ptr_yyy)
  ENDIF
RETURN
PROCEDURE test_nord_ouest
  ptr_xxx=SUB(ptr_xx,1)
  ptr_yyy=SUB(ptr_yy,1)
  IF tab(ptr_xxx,ptr_yyy,2)=0 AND tab(ptr_xxx,ptr_yyy,1)<>0
    region_nord_ouest(ptr_xxx,ptr_yyy)
  ELSE
    affichage_case(ptr_xxx,ptr_yyy)
  ENDIF
RETURN
PROCEDURE som_data
  som=0
  RESTORE couleur
  FOR i=1 TO 884
    READ valeur
    ADD som,valeur
  NEXT i
  CLS
  IF som<>18130413
    PRINT "erreur de frappe lors de la saisie des data's"
  ELSE
    PRINT "aucune erreur détectée dans la liste des DATA's"
  ENDIF
RETURN
PROCEDURE data
couleur:          ! indispensable pour la basse résolution
  DATA 0,3003,546,6,48,1296,1649,1799
  DATA 1840,307,770,0,1365,1092,259,1911
low_rez:          ! indispensable pour la basse résolution
  DATA 9,9,4,65472,0,0,0,32768,0,32704,32704,32768,0,32704,32704,32768,0,32704,
32704,32768,0,32704,32704
  DATA 32768,0,32704,32704,32768,0,32704,32704,32768,0,32704,32704,32768,0,
32704,32704,32768,0,32704,32704
```

(suite)

```
   DATA 9,9,4,65476,0,0,0,32768,0,32704,32704,34304,1536,31168,31168,36352,3584,
29120,29120,34304,1536,31168,31168
   DATA 34304,1536,31168,31168,34304,1536,31168,31168,34304,1536,31168,31168,
34304,1536, 31168,31168,32768,0,32704,32704
   DATA 9,9,4,65476,0,0,0,32768,0,32704,32704,32768,0,32704,24768,32768,0,32704,
31936,32768,0,32704,31936
   DATA 32768,0,32704,24768,32768,0,32704,26560,32768,0,32704,26560,32768,0,
32704,24768,32768,0,32704,32704
   DATA 9,9,4,65476,0,0,0,32768,0,32704,32704,40704,0,32704,24768,33536,0,32704,
31936,33536,0,32704,31936
   DATA 34560,0,32704,30912,33536,0,32704,31936,33536,0,32704,31936,40704,0,
32704,24768,32768,0,32704,32704
   DATA 9,9,4,65476,0,0,0,32768,0,32704,32704,32768,6144,32704,26560,32768,6912,
32704,25792,32768,6912,32704,25792
   DATA 32768,7936,32704,24768,32768,768,32704,31936,32768,768,32704,31936,32768,
768,32704,31936,32768,0,32704,32704
   DATA 9,9,4,65476,0,0,0,32768,0,32704,32704,40704,7936,32704,24768,38912,6144,
32704,26560,38912,6144,32704,26560
   DATA 40704,7936,32704,24768,33536,768,32704,31936,33536,768,32704,31936,40704,
7936,32704,24768,32768,0,32704,32704
   DATA 9,9,4,65476,0,0,0,32768,0,32704,32704,32768,0,24768,32704,32768,0,26560,
32704,32768,0,26560,32704
   DATA 32768,0,24768,32704,32768,0,25792,32704,32768,0,25792,32704,32768,0,
24768,32704,32768,0,32704,32704
   DATA 9,9,4,65476,0,0,0,32768,0,32704,32704,40704,0,24768,32704,33536,0,31936,
32704,34560,0,30912,32704
   DATA 34304,0,31168,32704,36352,0,29120,32704,35840,0,29632,32704,35840,0,
29632,32704,32768,0,32704,32704
   DATA 9,9,4,65476,0,0,0,32768,0,32704,32704,32768,7936,24768,32704,32768,6912,
25792,32704,32768,6912,25792,32704
   DATA
32768,7936,24768,32704,32768,6912,25792,32704,32768,6912,25792,32704,32768,7936,24
768,32704,32768,0,32704,32704
   DATA 9,9,4,65476,0,0,0,52224,19456,13248,32704,45568,12800,19904,32704,36608,
3840,28864,32704,40832,8064,24640,32704
   DATA 40832,8064,24640,32704,40064,7296,25408,32704,36224,3456,25152,28608,
36608,3840,28864,32704,32768,0,32704,32704
   DATA 9,9,4,65476,0,32768,32768,32832,32704,32768,32832,32832,32704,32768,
32832,32832,32704,32768,32832,32832,32704,32768,32832
   DATA 32832,32704,32768,32832,32832,32704,32768,32832,32832,32704,32768,32832,
32832,32704,32768,32832,65472,32704,32768,65472
   DATA 9,9,4,65476,0,0,0,32768,0,32704,32704,32768,768,32704,32704,32768,3328,
32704,32192,32768,12544,32704,29120
   DATA 32768,3840,32704,32704,32768,256,32704,32704,32768,256,32704,32704,33664,
0,32704,31808,32768,0,32704,32704
   DATA 9,9,4,65476,65472,0,65472,32832,65408,64,32832,32832,65408,64,32832,
32832,65408,64,32832,32832,65408,64,32832
```



(suite)

```
   DATA 32832,65408,64,32832,32832,65408,64,32832,32832,65408,64,32832,32832,
65408,64,32832,65472,32768,64,32832
   DATA 9,9,4,65476,0,0,0,52224,19456,13248,32704,45696,4608,28096,24384,40704,
3584,29120,28352,40832,5504,27200,30144
   DATA 40832,7040,25664,31680,40576,5248,27456,30144,40320,3200,29504,28352,
44928,3840,28864,24384,32768,0,32704,32704
 high_rez:          ! indispensable pour la haute résolution
   DATA 15,15,1,1,3,3,3,3,3,3,3,3,3,3,3,3,32767,65535
   DATA 15,15,1,1,21847,387,22487,1923,21975,387,21975,387,21975,387,21975,3,
21847,32767,65535
   DATA 15,15,1,1,21847,4067,24567,99,21879,4067,24567,3075,23895,4067,24567,3,
21847,32767,65535
   DATA 15,15,1,1,21847,4067,24567,99,21879,995,22519,99,21879,4067,24567,3,
21847,32767,65535
   DATA 15,15,1,1,21847,3075,23927,3171,23927,4067,24567,99,21879,99,21879,3,
21847,32767,65535
   DATA 15,15,1,1,21847,4067,24567,3075,23895,4067,24567,99,21879,4067,24567,3,
21847,32767,65535
   DATA 15,15,1,1,21847,4067,24567,3075,23895,4067,24567,3171,23927,4067,24567,
3,21847,32767,65535
   DATA 15,15,1,1,21847,4067,24567,99,21879,995,22487,195,21975,387,21975,3,
21847,32767,65535
   DATA 15,15,1,1,21847,4067,24567,3171,23927,4067,24567,3171,23927,4067,24567,
3,21847,32767,65535
   DATA 15,15,1,1,19971,12547,22787,10179,4067,8179,8179,8179,8115,7987,4067,
1987,3,32767,65535
   DATA 15,15,1,1,3,5463,10923,5463,10923,5463,10923,5463,10923,5463,10923,5463,
10923,32767,65535
   DATA 15,15,1,1,3,195,1859,14659,5443,3395,579,451,67,227,339,499,3,32767,65535
   DATA 15,15,1,65534,65532,60072,54612,60072,54612,60072,54612,60072,54612,
60072,54612,60072,54612,32768,0
   DATA 15,15,1,1,19971,28931,30987,14291,2979,5459,6835,7539,6835,5459,2979,
6099,3,32767,65535
 RETURN
```

4/2

Les Basic sur Atari ST

4/2.1

Le ST Basic

Le langage ST Basic est fourni avec la machine. Il n'est pas rapide, mais il permet d'écrire et de mettre au point assez rapidement de petits programmes. De plus, il tire pleinement parti de l'environnement graphique GEM de l'Atari ST, et de ses possibilités de fenêtrage. Dans les paragraphes qui suivent, nous allons voir comment :

— lancer ST Basic,

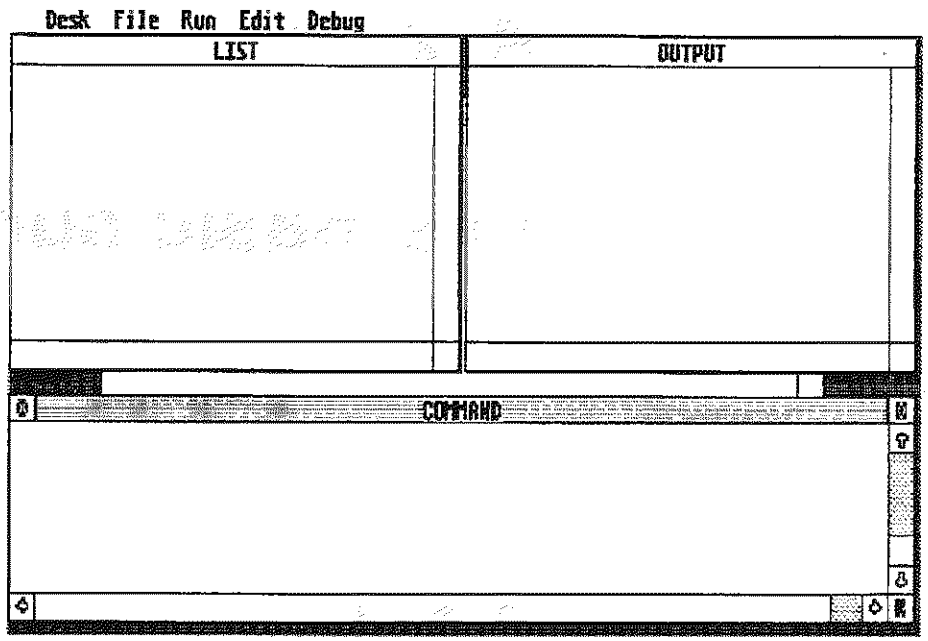
— utiliser les trois fenêtres qu'il met en œuvre,

— accéder aux options de menu,

— utiliser les ordres du langage.

I - Le lancement du ST Basic

Placez la disquette ST Basic dans le drive A:. « Double-cliquez » sur l'icône **Disquette A:**. La liste des fichiers de la disquette apparaît sur l'écran. « Double-cliquez » sur **BASIC.PRG**. Quelques secondes après, l'environnement de développement ST BASIC se trouve en mémoire, prêt à être utilisé. L'écran (voir page suivante) est affiché.



Cet écran laisse apparaître trois fenêtres. En réalité, il en contient quatre. Nous allons les analyser en détails dans le paragraphe suivant.

II - Les fenêtres du ST Basic

Le ST Basic met en œuvre quatre fenêtres dont trois sont apparentes lors de son lancement. Il s'agit des fenêtres LIST, OUTPUT et COMMAND. La fenêtre EDIT apparaît très partiellement entre les fenêtres LIST, OUTPUT et COMMAND.

Après le lancement de ST BASIC, la fenêtre COMMAND est active. Le curseur apparaît après le message OK, et le système est en attente d'une commande de l'utilisateur.

COMMAND

Cette fenêtre permet d'entrer des commandes exécutables directement et des lignes de programme.

Lorsqu'une commande ou une ligne de programme n'est pas interprétable par ST BASIC, le message **Something is wrong** est affiché dans la fenêtre COMMAND, et un signe ^ est placé sous le mot clé mis en doute.

Il est possible de faire occuper tout l'écran à cette fenêtre en cliquant dans le petit carré réservé à cet effet en haut et à droite de la fenêtre, ou en utilisant l'ordre **ST BASIC FULLW 3**.

La fenêtre COMMAND peut être effacée en cliquant sur le carré réservé en haut et à gauche de la fenêtre, et rappelée à partir de la fenêtre EDIT en tapant **command**.



LIST

La fenêtre LIST est utilisée pour afficher les instructions du programme qui se trouve en mémoire. Ces instructions apparaissent lorsque l'ordre LIST PRINT a été tapé par l'utilisateur dans la fenêtre COMMAND, EDIT ou a été exécuté dans un programme.

Il est possible de faire occuper tout l'écran à cette fenêtre en cliquant dans le petit carré réservé à cet effet en haut et à droite de la fenêtre, ou en utilisant l'ordre **ST BASIC FULLW 1**.

OUTPUT

La fenêtre OUTPUT est utilisée pour afficher toutes les informations à destination de l'écran (ordres **PRINT, INPUT,** ordres d'affichage graphique comme **CIRCLE** ou **ELLIPSE**).

Il est possible de faire occuper tout l'écran à la fenêtre OUTPUT en cliquant dans le petit carré réservé à cet effet en haut et à droite de la fenêtre, ou en utilisant l'ordre **ST BASIC FULLW 2.**

EDIT

La fenêtre EDIT permet d'accéder aux lignes du programme en mémoire en mode pleine page. Cette fenêtre est activée en tapant le mot clé **EDIT** dans la fenêtre COMMAND ou en sélectionnant l'option **START EDIT** dans le menu EDIT.

Il est possible de faire occuper tout l'écran à cette fenêtre en cliquant dans le petit carré réservé à cet effet en haut et à droite de la fenêtre, ou en utilisant l'ordre **ST BASIC FULLW 0**.

Reportez-vous au menu EDIT pour avoir de plus amples renseignements sur cette fenêtre.

III - Les menus du ST Basic

Sur la première ligne de l'écran se trouvent les options principales du menu : **DESK, FILE, RUN, EDIT, DEBUG**.

Pour activer une option principale de ce menu, déplacez l'indicateur de la souris sur le mot clé qui vous intéresse. Un menu déroulant est automatiquement affiché. Il vous suffit alors de sélectionner une des options de ce menu en cliquant sur le bouton de gauche.

Voici la liste détaillée des options du menu ST Basic :

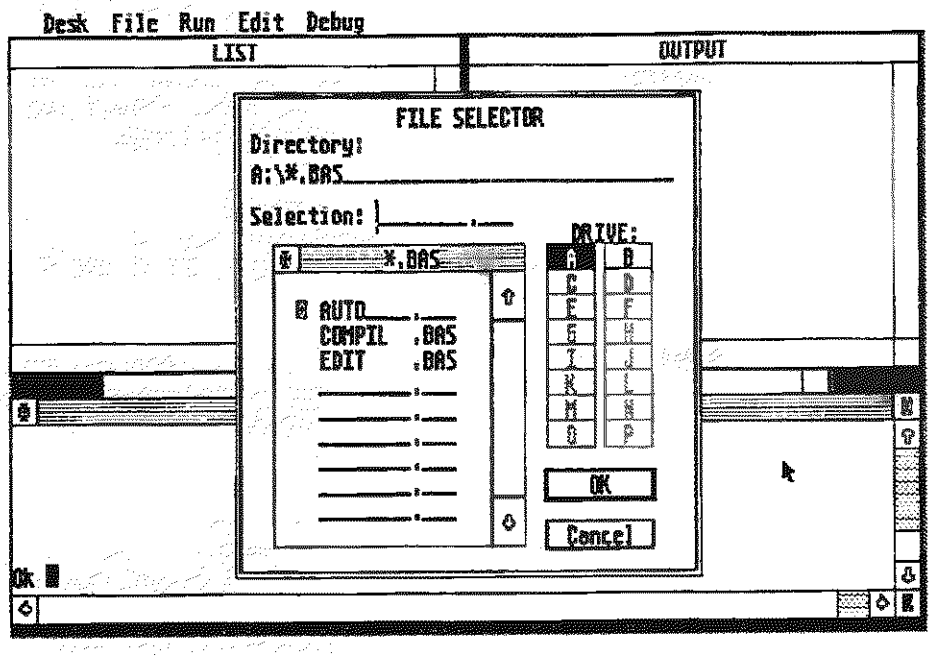
DESK

L'option principale **DESK** donne accès à un menu qui comporte une ou plusieurs options, en fonction des accessoires installés. La première option (About ST Basic) indique l'origine de ST Basic (concepteur et année de conception). Les autres options sont les accessoires installés lors du démarrage ou du changement de résolution.

FILE L'option principale **FILE** concerne toutes les entrées/sorties sur disque. Elle donne accès à un menu dont les options sont les suivantes :

LOAD

Cette option permet de charger un programme ST Basic en mémoire. Le sélecteur d'ITEM dont un exemple suit est affiché.



Si vous désirez changer de répertoire ou de disque, procédez comme suit :

— positionnez l'indicateur de la souris sur le champ Directory ;

— cliquez sur le bouton de gauche ;

— entrez le nom du nouveau répertoire ou de disque ;

— cliquez sur le carré de sélection en haut et à gauche de la fenêtre. Les fichiers du répertoire ou du disque défini apparaissent dans la fenêtre réservée à cet effet.

Pour choisir le programme à charger, procédez comme suit :

— tapez **NEW** dans la fenêtre COMMAND ;

— déroulez le menu File ;

— sélectionnez l'option **LOAD** dans ce menu ;

— affichez la liste des fichiers du disque/répertoire désiré ;

— positionnez la flèche de la souris sur l'entrée désirée et double-cliquez sur le bouton de gauche. Autre possibilité : entrez le nom du programme



et son extension dans le champ « Sélection », et appuyez sur <Return>.

Il est également possible de charger un programme en utilisant l'instruction **LOAD**. Reportez-vous à cette instruction pour avoir plus de détails.

SAVE AS

Cette option permet de sauvegarder le programme ST Basic qui se trouve en mémoire dans un fichier disque (le même que celui d'où il est issu, ou un autre). Déroulez le menu File, sélectionnez l'option **SAVE AS** dans ce menu, puis choisissez un nom de fichier dans la fenêtre, en procédant comme pour l'option **LOAD**.

Il est également possible de sauvegarder le programme en mémoire en utilisant l'instruction **SAVE**. Reportez-vous à cette instruction pour avoir plus de détails.

DELETE FILE

Cette option permet d'effacer un fichier de la disquette. Déroulez le menu File, sélectionnez l'option **DELETE FILE** dans ce menu, puis choisissez un nom de fichier dans la fenêtre, en procédant comme pour l'option **LOAD**.

Il est également possible d'utiliser l'instruction **ST BASIC ERA** pour effacer un fichier sur disque. Reportez-vous à cette instruction pour avoir des renseignements complémentaires.

MERGE

Cette option permet de mixer deux ou plusieurs fichiers. Attention : Les lignes du fichier en mémoire dont les numéros sont les mêmes que celles du fichier mixé sont systématiquement effacées et remplacées par les nouvelles lignes.

Déroulez le menu File, sélectionnez l'option **MERGE** dans ce menu, puis choisissez un nom de fichier dans la fenêtre, en procédant comme pour l'option **LOAD**.

Il est également possible d'utiliser l'instruction **ST BASIC MERGE** pour mixer deux programmes. Reportez vous à cette instruction pour avoir des renseignements complémentaires.

QUIT

Cette option permet de quitter ST BASIC et de retourner sous GEM. Pour activer cette option, déroulez le menu File puis sélectionnez l'option **QUIT** dans ce menu.

Il est également possible d'utiliser les instructions **ST BASIC QUIT** ou **SYSTEM** pour retourner sous GEM. Reportez-vous à ces instructions pour avoir des renseignements complémentaires.

RUN L'option principale **RUN** concerne toutes les opérations qui peuvent être effectuées pendant l'exécution d'un programme. Elle donne accès à un menu dont les options sont les suivantes :

RUN

L'option **Run** permet d'exécuter le programme en mémoire. Pour activer cette option, déroulez le menu Run puis sélectionnez l'option **Run** dans ce menu.

L'instruction **ST BASIC RUN** peut également être utilisée pour exécuter un programme. Reportez-vous à cette instruction pour avoir des informations complémentaires.

BREAK

L'option **BREAK** arrête le programme en cours d'exécution. Dans le cas où la fenêtre EDIT était active avant l'exécution du programme, le contrôle est redonné à cette fenêtre. Dans tous les autres cas, le prompt **BR ?** est affiché dans la fenêtre COMMAND. Cette commande vous permet d'exécuter directement des instructions pour, par exemple, vérifier la valeur des variables du programme. Pour activer cette option, déroulez le menu Run puis sélectionnez l'option **BREAK** dans ce menu.

L'instruction **ST BASIC BREAK** peut également être utilisée pour arrêter un programme en cours d'exécution. Reportez-vous à cette instructions pour avoir des informations complémentaires.

STOP

L'option **STOP** permet de mettre hors fonction le mode Break activé par l'option **BREAK** du menu Run.

Pour activer cette option, déroulez le menu Run puis sélectionnez l'option **STOP** dans ce menu.

L'instruction **ST BASIC STOP** peut également être utilisée pour désactiver le mode BREAK. Reportez-vous à cette instruction pour avoir des informations complémentaires.

CONTINUE

L'option **CONTINUE** permet de reprendre l'exécution du programme à l'endroit où il a été stoppé avec l'option **BREAK** du menu « Run ».

Pour activer cette option, déroulez le menu Run puis sélectionnez l'option **CONTINUE** dans ce menu.

L'instruction **ST BASIC CONT** peut également être utilisée pour reprendre l'exécution d'un programme. Reportez-vous à cette instruction pour avoir des informations complémentaires.

STEP

L'option **STEP** permet d'exécuter un programme ligne par ligne. Pour activer cette option, déroulez le menu Run puis sélectionnez l'option **STEP** dans ce menu. Le message **Hit <CR> to execute each line** apparaît sur l'écran lorsque cette option est sélectionnée. Avant qu'une ligne ne



soit exécutée, elle est affichée dans la fenêtre COMMAND, et le prompt **Br** apparaît juste en dessous, dans l'attente d'une commande de l'utilisateur.

L'instruction **ST BASIC STEP** peut également être utilisée pour exécuter un programme ligne par ligne. Reportez-vous à cette instruction pour avoir des informations complémentaires.

BUF GRAPHICS

L'option **BUF GRAPHICS** doit être sélectionnée si vous voulez réserver un espace mémoire de 32 KOctets pour les éventuels graphismes affichés dans la fenêtre OUTPUT.

Pour activer cette option, déroulez le menu Run puis sélectionnez l'option **BUF GRAPHICS** dans ce menu.

EDIT

L'option principale **EDIT** concerne toutes les facilités d'édition du ST BASIC. Elle donne accès à un menu dont les options sont les suivantes :

START EDIT

L'option **START EDIT** active le mode EDIT (passage dans la fenêtre Edit).

Pour activer cette option, déroulez le menu Edit puis sélectionnez l'option **START EDIT** dans ce menu.

Il est également possible d'utiliser les instructions **ST BASIC EDIT** pour activer cette fenêtre. Reportez-vous à cette instruction pour avoir des renseignements complémentaires.

EXIT EDIT

L'option **EXIT EDIT** désactive le mode EDIT (passage dans la fenêtre COMMAND).

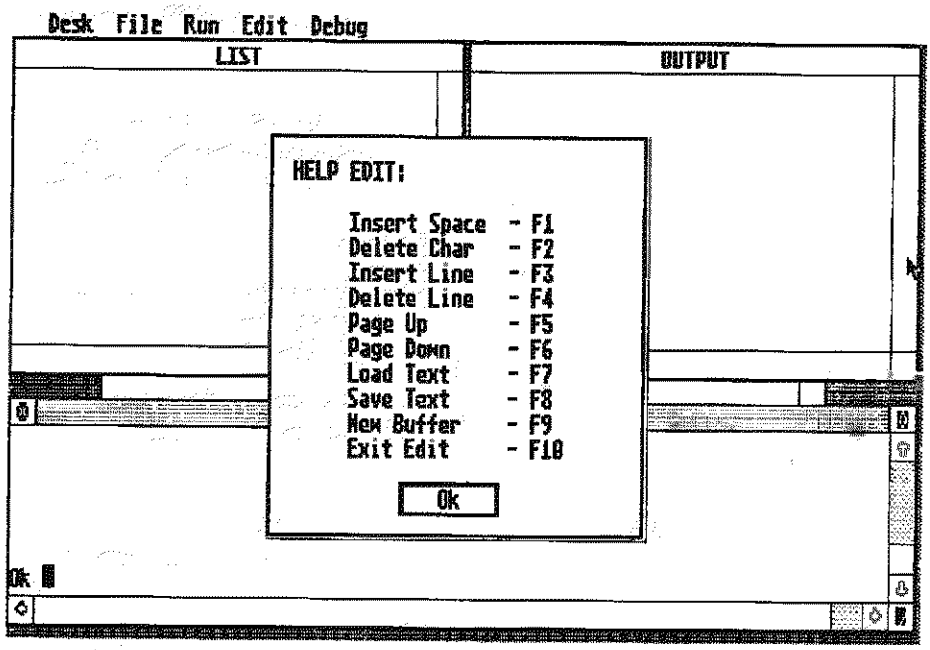
Pour activer cette option, déroulez le menu EDIT puis sélectionnez l'option **EXIT EDIT** dans ce menu.

HELP EDIT

L'option **HELP EDIT** affiche l'affectation des touches de fonction dans la fenêtre EDIT.

Pour activer cette option, déroulez le menu File puis sélectionnez l'option **QUIT** dans ce menu.

Il est également possible d'utiliser les instructions **ST BASIC QUIT** ou **SYSTEM** pour retourner sous GEM. Reportez-vous à ces instructions pour avoir des renseignements complémentaires.



GOTO LINE..

L'option **GOTO LINE..** permet de déplacer le curseur sur une ligne de programme de votre choix. Pour activer cette option, déroulez le menu Edit, sélectionnez l'option **GOTO LINE..** dans ce menu, et entrez le numéro de ligne désiré dans la boîte de dialogue qui est affichée.

DELETE LINES

L'option **DELETE LINES** permet d'effacer une ou plusieurs lignes du programme en mémoire. Pour activer cette option, déroulez le menu Edit, sélectionnez l'option **DELETE LINES** dans ce menu, et entrez les numéros de lignes à effacer : le premier après l'indication **FROM LINE** et le dernier après l'indication **TO LINE**.

INSERT SPACE

L'option **INSERT SPACE** permet d'insérer un caractère d'espacement à la position courante du curseur.

DELETE CHAR

L'option **DELETE CHAR** efface le caractère courant (position curseur) et décale vers la gauche les caractères qui suivent.

Pour activer cette option, déroulez le menu Edit puis sélectionnez l'option **DELETE CHAR** dans ce menu.



Il est également possible d'utiliser la touche **Backspace** pour produire le même effet.

INSERT LINE

L'option **INSERT LINE** insère une ligne vierge à la position courante du curseur.

Pour activer cette option, déroulez le menu EDIT puis sélectionnez l'option **INSERT LINE** dans ce menu.

Il est également possible d'utiliser la touche **Insert** pour produire le même résultat.

REMOVE LINE

L'option **REMOVE LINE** efface la ligne courante.

Pour activer cette option, déroulez le menu **EDIT** puis sélectionnez l'option **REMOVE LINE** dans ce menu.

PAGE UP

L'option **PAGE UP** positionne le curseur sur le premier caractère (en haut et à gauche de la page précédant celle qui est affichée).

Pour activer cette option, déroulez le menu EDIT puis sélectionnez l'option **PAGE UP** dans ce menu.

PAGE DOWN

L'option **PAGE DOWN** positionne le curseur sur le premier caractère (en haut et à gauche de la page qui suit celle qui est affichée).

Pour activer cette option, déroulez le menu EDIT puis sélectionnez l'option **PAGE DOWN** dans ce menu.

LOAD TEXT

L'option **LOAD TEXT** charge le contenu du fichier « BASIC.BUF » à la position courante du curseur.

Pour activer cette option, déroulez le menu **EDIT** puis sélectionnez l'option **LOAD TEXT** dans ce menu.

SAVE TEXT

L'option **SAVE TEXT** sauvegarde le texte (programme ou non) qui se trouve dans la fenêtre EDIT dans le fichier BASIC.BUF.

Pour activer cette option, déroulez le menu EDIT puis sélectionnez l'option **SAVE TEXT** dans ce menu.

NEW BUFFER

L'option **NEW BUFFER** affiche le programme ST BASIC en mémoire dans la fenêtre EDIT.

Pour activer cette option, déroulez le menu EDIT puis sélectionnez l'option **NEW BUFFER** dans ce menu.

LIST

L'option **LIST** affiche le programme ST BASIC en mémoire dans la fenêtre LIST.

Pour activer cette option, déroulez le menu EDIT puis sélectionnez l'option **LIST** dans ce menu.

Il est également possible d'utiliser l'instruction **ST BASIC LIST** pour produire le même effet. Reportez-vous à cette instruction pour avoir des renseignements complémentaires.

DEBUG

L'option principale **DEBUG** concerne les outils de mise au point de l'environnement de programmation ST Basic. Elle donne accès à un menu dont les options sont les suivantes :

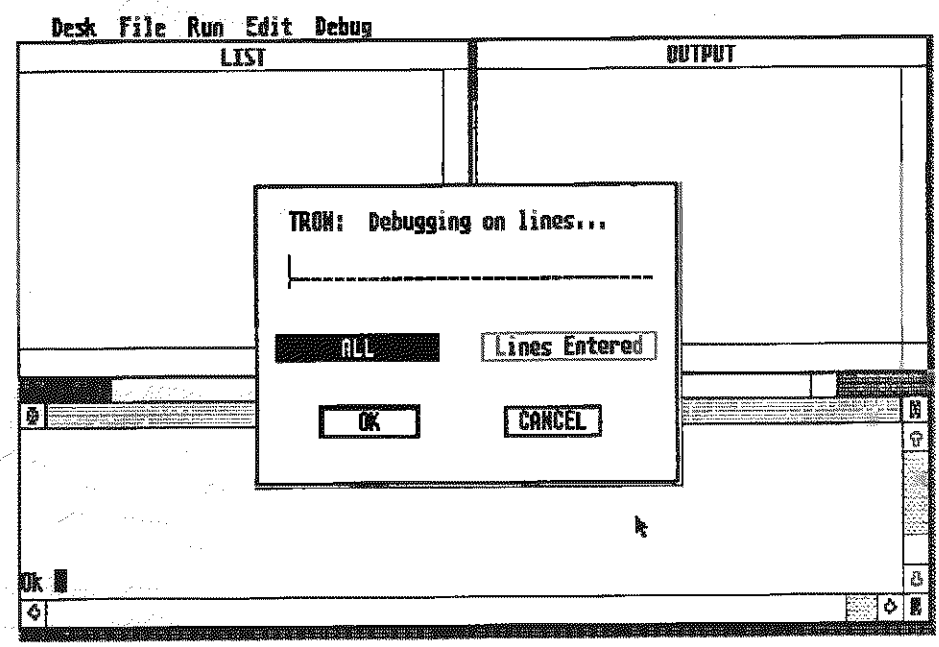
TRON

L'option **TRON** permet de suivre l'exécution d'une, de plusieurs ou de toutes les lignes du programme en mémoire pendant son exécution.

Pour activer cette option, déroulez le menu DEBUG puis sélectionnez l'option **TRON** dans ce menu.

Il est également possible d'utiliser l'instruction **ST BASIC TRON** pour activer le mode trace. Reportez-vous à cette instruction pour avoir des renseignements complémentaires.

La boîte de dialogue suivante est affichée :



Sélectionnez **ALL** dans cette boîte de dialogue pour avoir un écho de tous les numéros de lignes du programme pendant leur exécution. Sélectionnez **Lines Entered** pour avoir un écho d'un ou de plusieurs numéros de lignes pendant leur exécution. Si vous choisissez cette commande, vous devez indiquer le ou les numéros de lignes dans l'espace réservé à cet effet.

Quel que soit votre choix, sélectionnez **OK** pour valider le mode trace selon le paramétrage qui apparaît dans la boîte de dialogue, et **CANCEL** pour ne pas prendre en considération les données affichées dans la boîte de dialogue.

TROFF

L'option **TROFF** permet de dévalider le mode trace activé par l'option **TRON** du menu DEBUG.

Pour activer cette option, déroulez le menu DEBUG puis sélectionnez l'option **TROFF** dans ce menu.

Il est également possible d'utiliser l'instruction **ST BASIC TROFF** pour annuler le mode TRON. Reportez-vous à cette instruction pour avoir des renseignements complémentaires.

Une boîte de dialogue similaire à celle de l'option **TRON** est affichée. Sélectionnez **ALL** dans cette boîte pour dévalider le mode trace sur tous les numéros de lignes du programme. Sélectionnez **Lines Entered** pour dévalider le mode trace sur certains numéros de lignes seulement, puis entrez les numéros de ligne dans l'espace réservé à cet effet.

Dans tous les cas, sélectionnez **OK** pour dévalider les options du mode trace selon le paramétrage qui apparaît dans la boîte de dialogue, ou **CANCEL** pour ne pas prendre en considération les données affichées dans la boîte de dialogue.

TRACE

L'option **TRACE** permet de visualiser une, plusieurs ou toutes les lignes du programme en mémoire pendant leur exécution. Pour activer cette option, déroulez le menu DEBUG puis sélectionnez l'option **TRACE** dans ce menu.

Il est également possible d'utiliser l'instruction **ST BASIC TRACE** pour activer le mode trace. Reportez-vous à cette instruction pour avoir des renseignements complémentaires.

Une boîte de dialogue du même type que celle de l'option **Tron** est affichée. Sélectionnez **ALL** dans cette boîte de dialogue pour avoir un écho de toutes les lignes du programme pendant leur exécution. Sélectionnez **Lines Entered** pour avoir un écho d'une ou de plusieurs lignes pendant leur exécution. Si vous choisissez cette commande, vous devez indiquer le ou les numéros de lignes dans l'espace réservé à cet effet.

Quel que soit votre choix, sélectionnez **OK** pour valider le mode trace selon le paramétrage qui apparaît dans la boîte de dialogue, et **CANCEL** pour ne pas prendre en considération les données affichées dans la boîte de dialogue.

UNTRACE

L'option **UNTRACE** permet de dévalider le mode trace activé par l'option **TRACE** du menu DEBUG.

Pour activer cette option, déroulez le menu DEBUG puis sélectionnez l'option **UNTRACE** dans ce menu.

Il est également possible d'utiliser l'instruction **ST BASIC UNTRACE** pour désactiver le mode TRACE. Reportez-vous à cette instruction pour avoir des renseignements complémentaires.

Une boîte de dialogue similaire à celle de l'option **Tron** est affichée. Sélectionnez **ALL** dans cette boîte pour dévalider le mode trace sur toutes les lignes du programme. Sélectionnez **Lines Entered** pour dévalider le mode trace sur certaines lignes seulement, puis entrez les numéros de ligne dans l'espace réservé à cet effet.

Dans tous les cas, sélectionnez **OK** pour dévalider les options du mode trace selon le paramétrage qui apparaît dans la boîte de dialogue, ou **CANCEL** pour ne pas prendre en considération les données affichées dans la boîte de dialogue.



IV - Les instructions du ST Basic

Nous avons choisi de présenter les instructions du langage dans l'ordre alphabétique, car c'est sans doute la disposition la plus pratique pour accéder à l'information qui vous manque. De plus, la plupart de nos lecteurs ne doivent pas en être à leur première version de Basic. Les instructions du ST Basic sont très voisines de celles de la plupart des Basic ; aussi, la répartition des instructions par groupes fonctionnels offre-t-elle plus d'inconvénients que d'avantages.

ABS Valeur absolue

Format

<Variable numérique> = ABS(<Expression numérique>)
où <Expression numérique> est un nombre de type quelconque ou une expression mathématique
et <Variable numérique> est la valeur absolue de l'expression numérique.

Fonction

Cette instruction calcule la valeur absolue de l'expression numérique qui lui est passée (valeur numérique privée de son signe).

Exemple :

```
? ABS(-12.34)
12.34
? ABS(12.34)
12.34
? ABS(-3*5.1)
15.3
```

ASC Code ASCII

Format

<Variable entière> = ASC(<Expression chaîne>)
où <Expression chaîne> est une chaîne alphanumérique ou une expression chaîne
et <Variable entière> contient le code ASCII du premier caractère de l'expression chaîne passée.

Fonction

Cette fonction renvoie le code ASCII du premier caractère de l'expression chaîne à laquelle on l'applique.

Reportez-vous à l'annexe 1 pour prendre connaissance de la liste des codes ASCII.

Exemple :

```
? ASC("123")
49                  → Code ASCII de 1
? ASC(CHR$(85))
85                  → ASC et CHR$ s'annulent
? ASC("A")
65
```

ATN Arc tangente

Format

<Variable numérique> = ATN(<Expression numérique>)
où <Expression numérique> contient un angle en radians,
et <Variable numérique> l'arc tangente de l'angle passé.

Fonction

Cette instruction calcule l'arc de la tangente (fonction inverse de la tangente) de l'angle en radians (entre −PI/2 et PI/2) qui lui est passé. Le résultat est de type réel.

Exemple :

```
? ATN(1)
.785398         (soit sensiblement PI/4)
? ATN(0)
0
```



AUTO Numérotation automatique

Format

AUTO[<Première ligne>][,<espace entre 2 lignes>]
où <Première ligne> est le numéro de la première ligne à numéroter,
et <espace entre 2 lignes> est le nombre à ajouter au numéro de ligne N pour obtenir le numéro de ligne N+1.

Fonction

Cette instruction évite la saisie des numéros de ligne lors de l'écriture de programmes. Lorsque le numéro de première ligne n'est pas spécifié, il est par défaut égal à 10. Lorsque l'espacement entre deux lignes consécutives n'est pas spécifié, il est par défaut égal à 10.

Exemple :

AUTO
Numérotation de 10 en 10 à partir de la ligne 10
AUTO 1000
Numérotation de 10 en 10 à partir de la ligne 1000
AUTO 1000,20
Numérotation de 20 en 20 à partir de la ligne 1000

BLOAD Chargement d'un fichier binaire en mémoire

Format

BLOAD <Nom de fichier>[,<Adresse>]
où <Nom de fichier> est le nom du fichier à charger
et <Adresse> est l'adresse d'implantation en mémoire.

Fonction

Cette instruction permet de charger en mémoire un fichier binaire sauvegardé par l'instruction **BSAVE**. Lorsque l'adresse de chargement n'est pas spécifiée, l'adresse indiquée dans l'instruction **BSAVE** qui a permis de sauvegarder le fichier est prise par défaut.

Exemple :

BLOAD "F_BIN",400
Implantation à l'adresse 400 du fichier F_BIN

BREAK Arrêt du programme

Format

BREAK [<ligne 1>,...<ligne N>]
où les <Ligne I> sont des numéros de lignes.

Fonction

Cette instruction permet d'arrêter automatiquement le programme pendant son exécution aux numéros de ligne spécifiés. Dans le cas où aucun numéro de ligne n'est spécifié, l'exécution est interrompue à chaque ligne.
Remarque : Cette instruction est équivalente à l'option **BREAK** du menu RUN.

Exemple :

```
BREAK10       → point d'arrêt ligne 10
BREAK 10,50   → points d'arrêt lignes 10 et 50
BREAK 10-50   → points d'arrêt sur toutes les lignes entre 10 et 50
```

BSAVE Sauvegarde d'une partie de la mémoire dans un fichier

Format

BSAVE <Nom de fichier>,<Adresse de départ>,<Longueur>
où <Nom de fichier> est un nom de fichier TOS,
<Adresse de départ> est l'adresse du premier octet à sauvegarder
et <Longueur> le nombre d'octets à sauvegarder.

Fonction

Cette instruction sauvegarde un bloc de mémoire dans un fichier. Ce bloc pourra être rechargé en utilisant l'instruction **BLOAD**, à la même adresse ou à une adresse différente.

Exemple :

```
BSAVE "F.BIN",3000,500
```
Sauvegarde du bloc de mémoire commençant en 3000 et de longueur 500 dans le fichier F.BIN.



CALL Appel à une routine Assembleur

Format

CALL <Expression numérique>[(<liste de paramètres>)]
où <liste de paramètres> représente le ou les paramètres à passer à la routine.
et <Expression numérique> est l'adresse d'implantation de la routine en mémoire.

Fonction

Cette instruction donne le contrôle à une routine en langage machine. Cette routine peut être chargée en mémoire avec l'instruction **BLOAD**.

Lorsqu'un ou plusieurs paramètres sont précisés dans l'instruction, ils sont récupérés dans la routine assembleur par l'intermédiaire du registre A7 (pile utilisateur). Les deux premiers octets de ce registre contiennent le nombre de paramètres passés à la routine par ST Basic. Les quatre octets suivants contiennent un pointeur vers un tableau dans lequel se trouvent les adresses où sont stockés les paramètres.

Exemple :

```
LM=1000
Remplissage=500
CALL LM(X,Y,R%)
CALL Remplissage
```

CHAIN Chaînage et mixage de deux programmes

Format

CHAIN <Nom de fichier>[, <Numéro de ligne>][,ALL]
où CHAIN MERGE <Nom de fichier>[,<Numéro de ligne>]
[,DELETE<liste de numéros de lignes>]
Dans sa première forme :
<Nom de fichier> est le nom du fichier à exécuter,
et <Numéro de ligne> est le numéro de la première ligne à exécuter.

Dans sa seconde forme :
<Nom de fichier> est le nom du fichier à mixer,
<Numéro de ligne> est le numéro de la première ligne à exécuter
et <liste de numéros de lignes> représente une ou plusieurs lignes à effacer avant de mixer le nouveau programme.

Fonction

Dans sa première forme, CHAIN permet de lancer l'exécution d'un autre programme, à partir de la ligne spécifiée ou de la première ligne dans le cas où aucune ligne n'est spécifiée. Les variables des deux programmes peuvent être partagées grâce à l'instruction optionnelle **ALL**. Si vous désirez passer seulement quelques variables, vous devez utiliser l'ins-

truction **COMMON**. Reportez-vous à cette instruction pour avoir des renseignements complémentaires.

Dans sa seconde forme, CHAIN MERGE permet de mixer le programme en mémoire et un autre programme dont le nom est indiqué. Le programme résultant est exécuté, à partir du numéro de ligne spécifié, ou à partir du premier numéro de ligne dans le cas où aucun numéro de ligne n'est spécifié. Une ou plusieurs lignes du programme en mémoire peuvent également être effacées en utilisant le mot clé optionnel **DELETE**.

Exemple :

Chaînage simple :
CHAIN "Suite"

Chaînage simple et exécution à partir de la ligne 100 :
CHAIN "Suite",100

Chaînage simple avec passage de toutes les variables de l'ancien programme au programme chaîné :
CHAIN "Suite",ALL

Mixage avec le programme en mémoire et exécution du résultat du mixage.
CHAIN MERGE "Suite"

Mixage avec le programme en mémoire et exécution du résultat du mixage à partir de la ligne 100.
CHAIN MERGE "Suite",100

Mixage avec le programme en mémoire, effacement des lignes comprises entre 10 et 90 et exécution du résultat du mixage à partir de la ligne 100.
CHAIN MERGE "Suite",100,DELETE 10-90

CHR$ Conversion d'un code ASCII en un caractère

Format

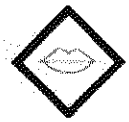
<Variable Chaîne> = CHR$(<Expression numérique>)
où <Expression numérique> est une valeur entière ou un calcul dont le résultat est entier,
et <Variable Chaîne> contient la conversion caractère de l'expression numérique passée.

Fonction

L'instruction **CHR$** renvoie le caractère dont le code ASCII est passé. Reportez-vous à l'annexe 1 pour avoir la liste des codes ASCII (Partie 1, Chapitre 3).

Exemple :

? CHR$(68) → D
? CHR$(200000) → @ car 200000 MOD 256 = 64 et CHR$(64) = @



CINT Arrondi d'un réel

Format

<Variable entière> = CINT(<Expression numérique>)
où <Expression numérique> est un nombre réel, entier, ou le résultat d'un calcul,
et <Variable entière> est la conversion entière de l'expression passée.

Fonction

Cette instruction renvoie l'entier le plus proche (entre −32768 et 32767) de l'expression numérique passée.

Exemple :

```
? CINT (100.86)
101
? CINT (100.34)
100
? CINT (-100.86)
-101
? CINT (-100.34)
-100
```

CIRCLE Tracé de cercles et arcs de cercles

Format

CIRCLE <Abscisse centre>,<Ordonnée centre>,<Rayon>
[<Angle de départ>,<Angle de fin>]
où <Abscisse centre> et <Ordonnée centre> contiennent les coordonnées du centre du cercle,
<Rayon> est le rayon du cercle à tracer,
<Angle de départ> et <Angle de fin> délimitent (éventuellement) l'arc de cercle à tracer.

Fonction

L'instruction **CIRCLE** permet de tracer des cercles ou arcs de cercles.

Pour tracer un arc de cercle, vous devez spécifier l'angle de départ et l'angle de fin de tracé. Ces arguments sont exprimés en degrés et multipliés par 10 (1800 pour 180 degrés par exemple).

Exemple :

```
Cercle complet de rayon 30
CIRCLE 100,100,30
Arc de cercle de 180 degrés de rayon 30
CIRCLE 200,100,30,0,1800
```

CLEAR Effacement des variables

Format CLEAR

Fonction Cette fonction efface le contenu des variables en mémoire : mise à zéro des variables numériques et mise à vide des variables chaînes.

CLEARW Effacement d'une fenêtre du ST Basic

Format CLEARW <Expression numérique>
où <Expression numérique> est un entier ou le résultat d'un calcul dont la valeur est une des suivantes :
0 pour la fenêtre EDIT,
1 pour la fenêtre LIST,
2 pour la fenêtre OUTPUT,
et 3 pour la fenêtre COMMAND.

Fonction Cette instruction efface le contenu de la fenêtre spécifiée dans l'expression numérique passée en entrée.

CLOSE Fermeture de fichiers

Format CLOSE [#]<Numéro de fichier>
où <Numéro de fichier> est un numéro affecté à un fichier par la commande **OPEN**.

Fonction Cette instruction permet de fermer un fichier séquentiel ouvert avec l'instruction **OPEN**.

Exemple :

```
CLOSE #3
CLOSE
CLOSE 1,2
```



CLOSEW Fermeture d'une fenêtre ST Basic

Format

CLOSEW <Numéro de fenêtre>
où <Expression numérique> est un entier ou le résultat d'un calcul dont la valeur est une des suivantes :
0 pour la fenêtre EDIT,
1 pour la fenêtre LIST,
2 pour la fenêtre OUTPUT,
et 3 pour la fenêtre COMMAND.

Fonction

Cette instruction ferme la fenêtre spécifiée dans l'expression numérique passée en entrée. Les fenêtres du ST Basic peuvent également être fermées en cliquant dans le carré situé en haut et à gauche de chaque fenêtre.

COLOR Choix des couleurs de l'écran

Format

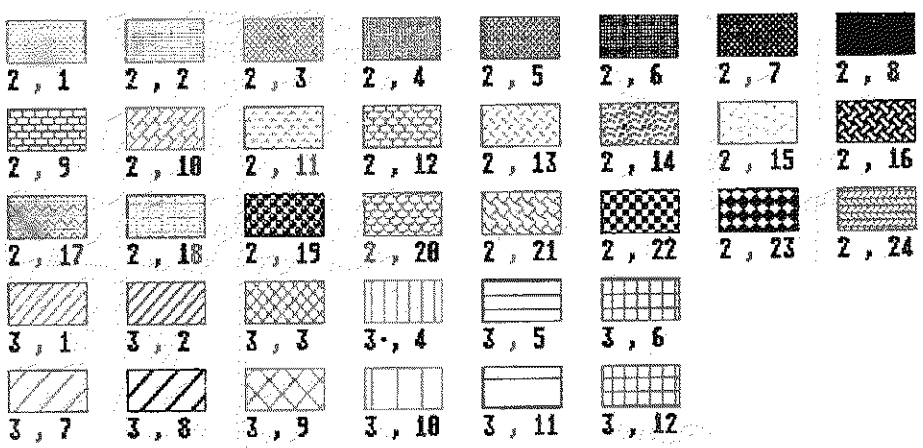
COLOR [<Texte>,[<Fond>,[<Graphique>,
[<Index>,<Style>]]]]
où <Texte> est la couleur du texte affiché,
<Fond> la couleur du fond de l'écran,
<Graphique> la couleur de tracé graphique,
<Index> et <Style> définissent le type de remplissage.

Les couleurs des arguments <Texte>, <Fond> et <Graphique> sont l'une des suivantes, en fonction du mode de résolution :

Résolution	Couleurs possibles
haute	0 ou 1
moyenne	0 à 3
basse	0 à 15

Code	Couleur
0	Blanc
1	Noir
2	Rouge
3	Vert
4	Bleu
5	Bleu foncé
6	Marron
7	Vert foncé
8	Gris
9	Gris foncé
10	Bleu clair
11	Gris bleu
12	Violet clair
13	Violet foncé
14	Jaune clair
15	Jaune foncé

Le type de remplissage est l'un des suivants, en fonction des paramètres <Index> et <Style> :



Fonction Cette instruction permet de définir la couleur des caractères affichés, du fond, des lignes graphiques tracées, et de choisir un type de remplissage.

Exemple :

```
COLOR 1,2,3
COLOR 1,2,3,2,2
```

COMMON Déclaration de variables globales

Format COMMON <Variable 1>[,<Variable 2>...[,<Variable N>]...] où les <Variable I> sont les noms des variables communes.

Fonction Cette instruction permet de déclarer une ou plusieurs variables à passer à un programme chaîné par l'instruction **CHAIN**. Toutes les variables peuvent également être passées au programme chaîné en utilisant le mot clé **ALL**. Reportez-vous à l'instruction COMMON pour avoir de plus amples renseignements.

Exemple :

```
COMMON A,B$,C!
```



CONT — Suite d'un programme suspendu par un BREAK

Format

CONT

Fonction

Cette instruction permet de poursuivre l'exécution d'un programme stoppé :

- par l'instruction **STOP**,
- par l'appui simultané sur les touches **Ctrl** et **G**,
- par l'option **BREAK** du menu ''Run''.

COS — Cosinus d'un angle

Format

<Variable numérique>=COS<Expression numérique>
où <Expression numérique> contient un angle en radians,
et <Variable numérique> contient le cosinus de l'angle passé.

Fonction

Cette instruction calcule le cosinus de l'angle en radians (entre 0 et 2 PI) qui lui est passé. Le résultat est de type réel.

Exemple :

```
PI=3.141592654
? COS(PI/4)      → .707106
? COS(PI/3)      → .5
? COS(PI/6)      → .866025
```

CVD — Conversion d'une chaîne de 8 octets en une valeur numérique réelle double précision

Format

CVD(<Chaîne>)
où <Chaîne> est une chaîne extraite d'un fichier à accès direct.

Fonction

Cette instruction convertit une chaîne de 8 octets extraite d'un fichier à accès direct en un nombre réel double précision. Ce nombre réel avait été stocké dans le fichier avec l'instruction **MKD$**.

Exemple :

```
10 OPEN "R",#1,"FICH"
20 FIELD #1,8 AS A$
30 GET #1,1
40 V#=CVD(A$)        → Conversion en un réel
                     → double précision
50 CLOSE #1
```

CVI Conversion d'une chaîne en une valeur numérique entière

Format CVI(<Chaîne>)
où <Chaîne> est une chaîne extraite d'un fichier à accès direct.

Fonction Cette instruction convertit une chaîne de 2 octets extraite d'un fichier à accès direct en un nombre entier. Ce nombre réel avait été stocké dans le fichier avec l'instruction **MKI$**.

Exemple :

```
10 OPEN "R",#1,"FICH"
20 FIELD #1,2 AS A$
30 GET #1,1
40 V%=CVI(A$)        → Conversion en un entier
50 CLOSE #1
```

CVS Conversion d'une chaîne en une valeur réelle simple précision

Format CVS(<Chaîne>)
où <Chaîne> est une chaîne extraite d'un fichier à accès direct.

Fonction Cette instruction convertit une chaîne de 4 octets extraite d'un fichier à accès direct en un nombre réel simple précision. Ce nombre réel avait été stocké dans le fichier avec l'instruction **MKS$**.



Exemple :

```
10 OPEN "R",#1,"FICH"
20 FIELD #1,4 AS A$
30 GET #1,1
40 V!=CVS(A$)        → Conversion en un réel
                     → simple précision
50 CLOSE #1
```

DATA Définition d'une ou de plusieurs constantes

Format DATA <constante 1>[,<constante 2>...[,<constante N>]...]
où les <constante I> sont des données de type quelconque.

Fonction Cette instruction permet de définir des données (entières, réelles ou chaînes) qui pourront être lues par des instructions **READ**. Chaque constante doit être séparée de la suivante par un caractère virgule.

Exemple :

```
DATA 1,10.5,Bonjour, "Une seule donnée"
```

DEF FN Définition d'une fonction utilisateur

Format DEF FN<Nom>[(<Liste de paramètres>)]=<Définition>
où <Nom> est le nom à donner à la fonction, <Liste de paramètres> représente 0, 1 ou plusieurs paramètres (non tableaux) à passer à la fonction,
et <Définition> est une expression mathématique qui dit comment calculer le résultat de la fonction à partir des paramètres qui lui sont passés.

Fonction L'instruction DEF FN permet de définir une fonction utilisateur.
Remarque : Lorsqu'un ou plusieurs paramètres sont passés à la fonction, leur valeur est stockée dans des variables locales à la fonction qui n'interfèrent, en aucune manière, avec les éventuelles variables du même nom en dehors de la fonction.

Exemple :

```
10 PI=3.141592654
20 DEF FNSEC(X)=1/COS(X)      → Fonction sécante
30 ? FNSEC(PI/3)              → 1.9999
```

DEF SEG Définition de l'adresse de départ du segment

Format

DEF SEG[=<Expression numérique>]
où <Expression numérique> peut avoir une valeur positive ou nulle.

Fonction

Cette instruction définit l'adresse de base du segment dans lequel les instructions de lecture et d'écriture en mémoire (**PEEK** et **POKE**) vont opérer.

Si DEF SEG vaut zéro, les instructions **PEEK** et **POKE** opèrent sur deux octets, à moins que l'adresse ne soit spécifiée avec un réel double précision. Dans ce cas, les instructions **PEEK** et **POKE** opèrent sur 4 octets.

Si DEF SEG a une valeur positive, les instructions **PEEK** et **POKE** opèrent sur un octet, et DEF SEG représente l'adresse de base du segment de données.

Exemple :

```
10 DEF SEG=0
20 ? PEEK(500)        → Résultat sur 1 octet
30 AD#=500
40 ? PEEK(#AD)        → Résultat sur 2 octets
50 DEF SEG=100
60 ? PEEK(400)        → Résultat sur 4 octets
```

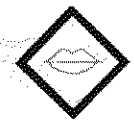
DEFDBL Initiales des variables double précision

Format

DEFDBL <lettre 1>[-<lettre 2>]
où <lettre 1> et <lettre 2> sont deux lettres de l'alphabet.

Fonction

Cette instruction spécifie que les variables dont le nom commence par <lettre 1>, ou est compris entre <lettre 1> et <lettre 2> sont par défaut de type réel double précision.



Exemple :

DEFDBL A-Z → toutes les variables sont des nombres réels en double précision par défaut

DEFDBL C → les variables d'initiale C sont des nombres réels en double précision par défaut

DEFINT Initiales des variables entières

Format DEFINT <lettre>[– <lettre>]
où <lettre 1> et <lettre 2> sont deux lettres de l'alphabet.

Fonction Cette instruction spécifie que les variables dont le nom commence par <lettre 1>, ou est compris entre <lettre 1> et <lettre 2> sont par défaut de type entier.

Exemple : **Voir DEFDBL**

DEFSNG Initiales des variables simple précision

Format DEFSNG <lettre>[– <lettre>]
où <lettre 1> et <lettre 2> sont deux lettres de l'alphabet.

Fonction Cette instruction spécifie que les variables dont le nom commence par <lettre 1>, ou est compris entre <lettre 1> et <lettre 2> sont par défaut de type réel simple précision.

Exemple : **Voir DEFDBL**

DEFSTR Initiale des variables chaînes

Format DEFSTR <lettre>[– <lettre>]
où <lettre 1> et <lettre 2> sont deux lettres de l'alphabet.

Fonction Cette instruction spécifie que les variables dont le nom commence par <lettre 1>, ou est compris entre <lettre 1> et <lettre 2> sont par défaut de type chaîne.

Exemple : **Voir DEFDBL**

DELETE Effacement de lignes dans le programme

Format

DELETE <Liste de numéros de lignes>
où <liste de numéros de lignes> représente un, plusieurs ou une plage de numéros de lignes.

Fonction

Cette instruction permet d'effacer la ou les lignes dont les numéros sont spécifiés ou appartiennent à la plage spécifiée.

Exemple :

```
DELETE 30       → Efface la ligne 30
DELETE 30,50    → Efface les lignes 30 et 50
DELETE 30-      → Efface la ligne 30 et toutes les lignes qui suivent
DELETE 30-100   → Efface les lignes comprises entre 30 et 100
```

DIM Dimensionnement d'un tableau

Format

DIM <Nom 1> (<Dim 1>, ..., <Dim N>)
[...[<Nom P>(<Dim 1>,...,<Dim K>)]...]
où les <Nom I> représentent le nom d'un ou de plusieurs tableaux, et les <Dim I> les dimensions de ces tableaux.

Fonction

L'instruction DIM permet de dimensionner un ou plusieurs tableaux. Le premier indice de chaque dimension est 0 ou 1 en fonction de la valeur définie dans OPTION BASE.

Remarque : Il est possible d'utiliser un tableau sans le déclarer tant que les indices qu'il met en œuvre n'atteignent pas le nombre 10.

Exemple :

Définition du tableau bidimensionnel A et du tableau de réels double précision monodimensionnel B# :

```
DIM A(10,15), B#(12)
```



DIR Liste des fichiers qui se trouvent sur un disque

Format

DIR [<Nom de lecteur:>][<Nom de fichier>] [<Type de fichier>
où <Nom de lecteur> est le nom d'unité d'un lecteur de disquettes
<Nom de fichier> est le préfixe ou le modèle de préfixe du ou de: fichier(s) recherché(s),
et <Type de fichier> est l'extension ou le type d'extension du ou des fichier(s) recherché(s).

Fonction

L'instruction DIR affiche le nom du ou des fichiers spécifié(s) sur le répertoire spécifié. Les jokers **?** et ***** peuvent être utilisés pour remplacer (respectivement) un caractère ou un ensemble de caractères quelconque.

Exemple :

Affichage du répertoire du disque courant :
DIR
Affichage des fichiers d'extension PRG du disque A:
DIR "A:*.PRG"
Affichage des fichiers du disque B dont la première lettre de l'extension est P et la troisième 2:
DIR "B:P?2"

EDIT Edition d'une ligne de programme

Format

EDIT [<Numéro de ligne>]
où ED <Numéro de ligne>
où <Numéro de ligne> est le numéro de la ligne de début d'édition.

Fonction

L'instruction EDIT permet d'activer la fenêtre EDIT. Lorsque aucun numéro de ligne n'est spécifié, l'édition commence à la première ligne du programme. Elle commence au numéro de ligne spécifié dans le cas contraire.

Remarque : pour quitter le mode d'édition et redonner le contrôle à la fenêtre COMMAND, il faut utiliser l'option « **Exit Edit** » de la fenêtre EDIT.

Exemple :

ED 30 → Edition à partir de la ligne 30
EDIT → Edition à partir de la première ligne

ELLIPSE Tracé d'ellipses et d'arcs d'ellipses

Format

ELLIPSE <Abcisse centre>, <Ordonnée centre>,
<Rayon horizontal>, <Rayon vertical>
[<Angle de départ>, <Angle de fin>]
où <Abcisse centre> et <Ordonnée centre> contiennent les coordonnées du centre de l'ellipse,
<Rayon horizontal> et <Rayon vertical> contiennent les rayons de l'ellipse,
<Angle de départ> et <Angle de fin> délimitent (éventuellement) l'arc d'ellipse à tracer.

Fonction

L'instruction ELLIPSE permet de tracer des ellipses ou des arcs d'ellipses.

Pour tracer un arc d'ellipse, vous devez spécifier l'angle de départ et l'angle de fin de tracé. Ces arguments sont exprimés en degrés et multipliés par 10 (400 pour 40 degrés par exemple).

Exemple :

Pour vous rendre compte des effets des différents paramètres, tapez les instructions suivantes :

```
ELLIPSE 100,100,50,30
ELLIPSE 100,100,100,30
ELLIPSE 10,100,30,50
ELLIPSE 50,50,30,30,0,1800
```

END Fin du programme

Format

END

Fonction

Cette instruction termine l'exécution du programme en mémoire, ferme les fichiers ouverts et redonne le contrôle à la fenêtre COMMAND sans passer en mode BREAK.

EOF Indicateur de fin de fichier séquentiel

Format

<Variable entière> = EOF (<Numéro de fichier>)
où <Numéro de fichier> est le numéro d'un fichier séquentiel défini dans une instruction OPEN précédente,
et <Variable entière> contient le résultat du test.



Fonction Cette instruction permet de tester la fin d'un fichier séquentiel afin d'éviter des erreurs de lecture. Le résultat retourné est 0 si la fin du fichier n'est pas atteinte, et −1 dans le cas contraire.

Exemple :

```
10 INPUT "Nom du fichier à lire";N$
20 OPEN "I",1,N$
30 WHILE NOT EOF (1)
40 INPUT #1, A$
50 PRINT A$
60 WEND
70 CLOSE 1
80 END
```

ERA Effacement de fichiers sur disque

Format ERA[<Nom de lecteur:>]<Nom de fichier>
où <Nom de fichier> est le nom complet du fichier ou le modèle des fichiers à effacer,
et <Nom de lecteur> est le nom du lecteur sur lequel se trouve le fichier à effacer.

Fonction Cette commande efface le (ou les) fichier(s) spécifié(s). Lorsqu'aucun nom de lecteur n'est précisé, le lecteur courant est pris par défaut.

Exemple :

```
ERA B:*.TXT
Efface tous les fichiers du disque B: d'extension TXT
ERA FOC.DWN
Efface le fichier FOC.DWN du disque par défaut.
```

ERASE Effacement de tableaux en mémoire

Format ERASE <Tableau>[...,<Tableau>]
où les <Tableau> sont des noms de tableaux.

Fonction Cette instruction libère l'espace occupé par un ou plusieurs tableaux déclarés par l'instruction DIM. Le contenu du ou des tableaux est bien entendu inaccessible après l'exécution de l'instruction.

Exemple :

ERASE A,C$
Efface les tableaux A et C$.

ERL Ligne où s'est produite la dernière erreur

Format

<Variable> = ERL
où <Variable> est une variable entière quelconque.

Fonction

Cette instruction renvoie le numéro de ligne sur laquelle s'est produite la dernière erreur. Elle permet de prendre la décision adéquate dans les programmes qui traitent les erreurs d'une façon qui leur est propre.

Exemple :

A = ERL
A contient le numéro de la dernière ligne sur laquelle s'est produite une erreur.
IF ERL = 1000 THEN GOTO 2000 ELSE GOTO 3000
Branchement à la ligne 2000 si l'erreur s'est produite en ligne 1000, et en ligne 3000 dans le cas contraire.

ERR Numéro de la dernière erreur

Format

<Variable> = ERR
où <Variable> est une variable entière quelconque.

Fonction

Cette instruction renvoie le numéro de la dernière erreur. Elle permet de prendre la décision adéquate dans les programmes qui traitent les erreurs d'une façon qui leur est propre.
Reportez-vous à la Partie 1, chapitre 2.2 pour avoir la liste des erreurs possibles.

Exemple :

A = ERR
A contient le numéro de la dernière erreur.
IF ERR = 7 THEN GOTO 2000 ELSE GOTO 3000
Branchement à la ligne 2000 si l'erreur est de type « Espace mémoire insuffisant », et en ligne 3000 dans le cas contraire.



ERROR Simulation d'une erreur ST Basic

Format

ERROR <Expression numérique>
où <Expression numérique> est un numéro d'erreur.

Fonction

Cette instruction déclenche l'erreur spécifiée.

Exemple :

ERROR 61 → Disque plein

EXP Exponentielle

Format

<Variable réelle> = EXP(<Expression numérique>)
où <Expression numérique> est une valeur entière ou réelle inférieure à 43.6682,
et <Variable réelle> est l'exponentielle de <Expression numérique>.

Fonction

Cette fonction retourne l'exponentielle (mise à la puissance e) de l'argument qui lui est passé en entrée.

Exemple :

```
a=exp(0)
1
PRINT EXP(1)
2.7182
```

FIELD Définition de la dimension des champs d'un fichier à accès direct

Format

FIELD #<Numéro de fichier>,<Champ>AS<variable chaîne 1>
[...,<Champ> AS <variable chaîne N>]
où <Numéro de fichier> est le numéro logique du fichier affecté dans l'instruction OPEN,
<Champ> sont les nombres d'octets réservés pour chaque champ du fichier,
et <Variable chaîne i> sont les noms des variables qui représentent les champs du fichier.

Fonction

Cette instruction définit la structure d'un fichier à accès direct ouvert par

l'instruction OPEN. Les enregistrements du fichier sont partitionnés en plusieurs champs alphanumériques dont les longueurs sont définies dans l'instruction.

Exemple :

```
FIELD #1,2 AS A$,4 AS B$,10 AS C$
```

Les enregistrements du fichier comportent trois champs. Le premier a une longueur de 2 caractères, le second de 4 caractères et le troisième de dix caractères.

FILL — Remplissage de contours fermés

Format

FILL <Abscisse>,<Ordonnée>
où <Abscisse> et <Ordonnée> représentent les coordonnées d'un point à l'intérieur de la figure à remplir.

Fonction

Cette instruction remplit une figure délimitée par un contour fermé. Le contour doit être de la couleur définie dans l'instruction COLOR. La couleur de remplissage correspond au second paramètre de l'instruction COLOR. Le point spécifié dans l'instruction doit se trouver à l'intérieur du contour fermé.

Exemple :

Remplissage du cercle avec la couleur 2

```
COLOR 1,2,3
CIRCLE 100,100,50
FILL 100,100
```

FIX — Partie entière d'un réel

Format

<Variable entière> = FIX(<Expression réelle>)
où <Expression réelle> contient un nombre réel quelconque.
et <Variable entière> la partie entière de l'<expression réelle>.

Fonction

Cette fonction renvoie la partie entière de l'expression réelle qui lui est passée, sans l'arrondir à l'entier le plus proche.

Exemple :

```
PRINT FIX(4.89)
4
PRINT FIX(4.11)
4
```



FLOAT Conversion d'un entier en un réel

Format

<Variable réelle> = FLOAT(<Variable entière>)
où <Variable entière> contient un nombre entier quelconque,
et <Variable réelle> est la version simple précision de l'entier passé à la fonction.

Fonction

Cette fonction renvoie le réel simple précision correspondant à l'entier qui lui est passé en entrée.

Exemple :

```
? FLOAT(12)
12
```

FOLLOW Mémorisation de la valeur des variables

Format

FOLLOW <Variable 1>[...,<Variable N>]
où les <Variables I> sont les noms des variables à suivre.

Fonction

Cette instruction permet de suivre l'évolution d'une ou de plusieurs variables pendant l'exécution du programme. Les numéros de ligne et nouvelles valeurs des variables suivies sont affichés pour chaque changement. Reportez-vous également à l'instruction UNFOLLOW qui permet de désactiver le suivi d'une ou de plusieurs variables.

Exemple :

```
FOLLOW A,B$
```

FOR Boucle dont le nombre de parcours est fixé

Format

FOR <Compteur> = <Début> TO <Fin> [STEP <Pas>]
où <Compteur> est le nom de la variable utilisée pour parcourir la boucle,
<Début> et <Fin> sont les première et dernière valeurs de <Compteur>,
et <Pas> est le pas de progression de <Compteur> dans l'intervalle <Début>..<Fin>.

Fonction Cette instruction est utilisée pour exécuter répétitivement une ou plusieurs instructions.

Exemple :

```
10 FOR I=1 TO 100:NEXT          → Boucle vide : temporisation
10 FOR I=1 TO 10
20 PRINT SIN(I)                 → Affichage des 10 premiers sinus
30 NEXT I
```

FRE — Taille de la mémoire libre

Format <Variable entière> = FRE(<Argument>)
où <Argument> est un argument quelconque,
et <Variable entière> le nombre d'octets non utilisés.

Fonction Cette fonction renvoie le nombre d'octets disponibles.

Exemple :

```
10 PRINT FRE(0)                 → Affichage de l'espace RAM disponible.
```

FULLW — Taille maximale des fenêtres du ST Basic

Format FULLW(<Expression numérique>)
où <Expression numérique> est un numéro de fenêtre.

Fonction Cette fonction alloue tout l'écran à la fenêtre spécifiée : 0 = Edition, 1 = List, 2 = Sortie, 3 = Commande.

Exemple :

```
FULLW 0                         → Edition sur tout l'écran.
```



GEMSYS Accès à l'interface AES (se reporter en 3/5)

Format

GEMSYS (<Code AES>)
où <Code AES> est la fonction AES à exécuter.

Fonction

Cette instruction permet d'accéder aux fonctions de l'AES. La variable GB contient les adresses des tables CONTROL, GLOBAL, INTIN, INTOUT, ADDRIN et ADDROUT. Chacune de ces adresses est codée sur quatre octets. Les paramètres à passer à l'AES doivent être initialisés à l'aide d'une suite de POKE.

Exemple :

```
10 A# = GB
10 CONTROL = PEEK(A#)
20 GLOBAL = PEEK(A# + 4)
30 GINTIN = PEEK(A# + 8)
40 GINTOUT = PEEK(A# + 12)
50 ADDRIN = PEEK(A# + 16)
60 ADDROUT = PEEK(A# + 20)
70 GEMSYS (10)              → Initialisation AES
80 PRINT PEEK (INTOUT)      → Identificateur retourné
```

GET Lecture d'un enregistrement dans un fichier à accès direct.

Format

GET [#]<Numéro de fichier>[,<Numéro d'enregistrement>]
où <Numéro de fichier> est le numéro logique du fichier défini dans l'instruction OPEN,
et <Numéro d'enregistrement> est le numéro de l'enregistrement à lire.

Fonction

Cette instruction lit l'enregistrement spécifié. Lorsque le numéro d'enregistrement n'est pas spécifié, l'enregistrement qui suit l'enregistrement courant est accédé.

Exemple :

Accès à l'enregistrement numéro 12 dans le fichier logique #3:

```
GET #3,12
```

GOSUB Exécution d'un sous-programme

Format

GOSUB <Numéro de ligne>
ou GOSUB <Label>
où <Numéro de ligne> représente le numéro de la première ligne du sous-programme,
et <Label> l'étiquette du sous-programme à activer.

Fonction

Cette instruction exécute un sous-programme identifié par son numéro de première ligne ou par une étiquette.

Exemple :

```
GOSUB 100          → Activation du S/P ligne 100
GOSUB AFFICH       → Activation du S/P AFFICH
```

GOTO Débranchement au numéro de ligne spécifié

Format

GOTO <Numéro de ligne>
ou GOTO <Label>
où <Numéro de ligne> représente le numéro de la ligne à accéder,
et <Label> l'étiquette de la ligne à accéder.

Fonction

Cette instruction donne le contrôle à la ligne spécifiée (numéro ou étiquette).

Exemple :

```
GOTO 100           → Activation de la ligne 100
GOTO AFFICH        → Activation de la ligne AFFICH
```

GOTOXY Positionnement du curseur sur l'écran

Format

GOTOXY <Abscisse>,<Ordonnée>
où <Abscisse> est la nouvelle abscisse du curseur (0 à 39 ou 0 à 79 selon le mode d'affichage),
et <Ordonnée> la nouvelle ordonnée du curseur (0 à 24).

Fonction

Cette instruction positionne le curseur texte aux coordonnées spécifiées.



Exemple :

Affichage d'un texte ligne 10, colonne 10:

```
GOTOXY 10,10
PRINT "Bonjour"
```

HEX$ Conversion hexadécimale d'un nombre

Format

<Variable chaîne> = HEX$(<Expression numérique>)
où <Expression numérique> est un nombre entier exprimé en décimal, et <Variable chaîne> contient la conversion hexadécimale de <Expression numérique>.

Fonction

Cette fonction renvoie sous un format chaîne la conversion hexadécimale de la valeur entière passée.

Exemple :

```
PRINT HEX$(10)                    → 10 en hexadécimal
A
```

IF Test logique

Format

IF <Expression logique> THEN <Instruction><:Instruction>
[ELSE <Instruction><:Instruction>]
où les <Instruction> sont des instructions quelconques du BASIC ST, et <Expression logique> est une condition logique.

Fonction

Cette instruction permet d'exécuter une ou plusieurs instructions si une condition logique est réalisée, et éventuellement une ou plusieurs autres instructions dans le cas contraire.

Exemple :

```
IF A<100 THEN PRINT "A est inférieur à 100"
         ELSE PRINT "A est supérieur ou égal à 100"
```

INP Lecture d'une valeur sur un port

Format

<Variable entière> = INP(<Port>)
où <Port> est un numéro de port,
et <Variable entière> la valeur lue sur le port spécifié.

Fonction

Cette fonction lit un octet sur le port d'entrée/sortie dont le numéro est spécifié. Le numéro du port peut être compris entre 0 et 65535. Les quatre premiers numéros représentent les périphériques suivants :
0 : imprimante parallèle
1 : port RS 232
2 : système clavier/écran
3 : interface MIDI
La lecture de l'état d'un port se fait en passant un numéro de port négatif. Si la valeur retournée est nulle, aucun caractère n'est disponible. Une valeur −1 indique par contre la disponibilité d'un caractère.

Exemple :

Affichage d'un caractère lu sur le port série
PRINT INP(−1)

INPUT Lecture de données au clavier

Format

INPUT[;][<prompt><{; ou ,}><variable>[...<variable>]
où <prompt> est une variable alphanumérique qui contient le texte à afficher avant l'acquisition de données,
et les <Variable> représentent les noms des variables dans lesquelles stocker les données entrées au clavier.

Fonction

Cette instruction permet de lire une ou plusieurs données entrées au clavier et de les stocker dans les variables spécifiées. Si un point virgule est spécifié après le prompt, un point d'interrogation et un espace sont affichés entre le prompt et la réponse. Si une virgule est spécifiée après le prompt, un espace est affiché entre le prompt et la réponse.

Exemple :

INPUT "Entrez un nombre entier",N



INPUT# Lecture de données dans un fichier séquentiel

Format

INPUT # <Numéro de fichier>,<Variable>,... <Variable>

où <Numéro de fichier> est le numéro logique d'un fichier séquentiel ouvert en lecture,

et les <Variable> sont les noms des variables dans lesquelles stocker les valeurs lues.

Fonction

Cette instruction permet de lire une ou plusieurs valeurs dans un fichier séquentiel ouvert en lecture. La ou les valeurs lue(s) est (sont) stockée(s) dans la(les) variable(s) spécifiée(s).

Exemple :

Lecture d'une valeur alphanumérique et d'une valeur numérique dans le fichier référencé # 1
```
INPUT #1,A$,B
```

INPUT$ Lecture de n caractères au clavier

Format

<Variable chaîne> = INPUT$(<Nombre>[,[#]<Numéro de fichier>])

où <Nombre> est le nombre de caractères à lire, <Numéro de fichier> le numéro logique du fichier à accéder,

et <Variable chaîne> la variable dans laquelle doit être stockée la valeur lue.

Fonction

Cette fonction renvoie un certain nombre de caractères lus au clavier (lorsqu'aucun numéro de fichier n'est spécifié), ou dans le fichier dont le numéro est spécifié.

Exemple :

```
A$=INPUT$(4)
A$=INPUT$(4,#1)
```

— Lecture de 4 caractères au clavier
— Lecture de 4 caractères dans le fichier référencé # 1

INSTR Recherche de la position d'une chaîne dans une autre chaîne.

Format

<Variable entière> = INSTR([<Départ>,]
<Expression chaîne>, <Chaîne source>)
où <Chaîne source> est la chaîne dans laquelle s'effectue la recherche,

<Expression chaîne> est la chaîne recherchée,

<Départ> est la position de début de recherche dans la chaîne source, et <Variable entière> la position de <Expression chaîne> dans <Chaîne source>.

Fonction

Cette fonction recherche une sous-chaîne dans une chaîne. Elle renvoie la valeur 0 si la sous-chaîne n'est pas trouvée, et la position de la sous-chaîne dans la chaîne dans le cas contraire. Dans le cas où le début de la recherche n'est pas précisé, il s'effectue à partir du premier caractère de la chaîne source.

Exemple :

```
PRINT INSTR("COU", "COUCOU")
1
PRINT INSTR (2, "COU", "COUCOU")
4
```

INT Valeur entière d'une expression

Format

<Variable entière> = INT(<Expression numérique>)
où <Expression numérique> est une variable ou valeur numérique quelconque,
et <Variable entière> la valeur entière de <Expression numérique>.

Fonction

Cette fonction renvoie la valeur entière de l'argument numérique qui lui est passé.

Exemple :

```
PRINT INT(12.6)
12
```



KILL Effacement d'un fichier

Format
KILL <Expression chaîne>
où <Expression chaîne> est le modèle du fichier à effacer.

Fonction
Cette instruction efface le ou les fichiers spécifié(s) dans le modèle.

Exemple :

Effacement de tous les fichiers de nom A et d'extension quelconque :

```
KILL "A.*"
```

LEFT$ Extraction des n caractères les plus à gauche dans une chaîne alphanumérique

Format
<Variable chaîne> = LEFT$(<Chaîne source>,<Longueur>)
où <Chaîne source> est le nom de la chaîne dans laquelle des caractères vont être extraits,
et <Longueur> est le nombre de caractères à extraire.

Fonction
Cette fonction extrait les n caractères les plus à gauche de la chaîne alphanumérique spécifiée.

Exemple :

```
PRINT LEFT$("ABCDEFGHIJ",5)
ABCDE
```

LEN Longueur d'une chaîne

Format
<Variable entière> = LEN(<Expression chaîne>)
où <Expression chaîne> est une variable ou une valeur chaîne alphanumérique,
et <Variable entière> contient la longueur de la chaîne passée.

Fonction Cette fonction renvoie le nombre de caractères de la chaîne passée en entrée.

Exemple :

```
PRINT LEN("ABCDEF")
6
```

LET — Affectation d'une valeur à une variable ou un tableau

Format

LET <Variable> = <Expression>
où <Expression> est une expression numérique ou chaîne qui correspond au type de <Variable>
et <Variable> est la variable dans laquelle <Expression> doit être stockée.

Fonction Cette instruction affecte une valeur à une variable. Elle peut être supprimée (voir exemple).

Exemple :

```
LET A=12
est équivalent à
A=12
```

LINE INPUT — Lecture de données au clavier

Format

LINE INPUT [;][<prompt>[{, ou ; }]<Variable chaîne>
où <prompt> est une variable alphanumérique qui contient le texte à afficher avant l'acquisition de données,
et <Variable chaîne> est un nom de variable dans laquelle doit être stockée la ligne de données.

Fonction Cette instruction permet de lire une ligne entrée au clavier. Cette ligne peut éventuellement comporter un ou plusieurs séparateurs (virgule, point-virgule, etc.). Si un point-virgule est spécifié après le prompt, un point d'interrogation et un espace sont affichés entre le prompt et la réponse. Si une virgule est spécifiée après le prompt, un espace est affiché entre le prompt et la réponse.



Exemple :

```
LINE INPUT "Entrez une ligne de texte", L$
```

LINE INTPUT # Lecture de données dans un fichier séquentiel

Format

LINE INPUT #<Numéro de fichier>,<Variable chaîne>
où <Numéro de fichier> est le numéro logique d'un fichier séquentiel ouvert en lecture,
et <Variable chaîne> est un nom de variable dans laquelle doit être stockée la ligne de données.

Fonction

Cette instruction permet de lire une ligne de données dans un fichier séquentiel ouvert en lecture. La valeur lue est stockée dans la variable spécifiée.

Exemple :

```
LINE INPUT #1,A$
```

LINEF Tracé d'une ligne

Format

LINEF <AD,OD>,<AA,OA>
où <AD,OD> sont les abscisse et ordonnée du point de départ,
et <AA,OA> sont les abscisse et ordonnée du point d'arrivée.

Fonction

Cette instruction trace une ligne entre les coordonnées absolues spécifiées, selon les données de couleur spécifiées dans l'instruction COLOR.

Cette instruction doit également être utilisée pour allumer ou éteindre un point sur l'écran.

Exemple :

Tracé d'une ligne entre les coordonnées 100,100 et 150,150

```
LINEF 100,100,150,150
```

Affichage d'un pixel en 100,100

```
LINEF 100,100,100,100
```

LIST — Affichage du programme dans la fenêtre LIST

Format

LIST [<Lignes à afficher>]
où <Lignes à afficher> est une ligne ou une liste de lignes à afficher, selon les conventions suivantes :

— aucun numéro de ligne : affichage de toutes les lignes du programme,

— un numéro de ligne : affichage de la ligne spécifiée,

— plusieurs numéros de lignes séparés par des virgules : affichage des lignes spécifiées,

— une plage dont les extrêmes sont séparés par un tiret : affichage des lignes dont les numéros sont compris entre les deux extrêmes. Lorsque la première valeur de la plage n'est pas spécifiée, le listage se fait à partir de la première ligne du programme. De même, lorsque la dernière valeur de la plage n'est pas spécifiée, le listage se fait jusqu'à la dernière ligne du programme.

Fonction

Affichage d'une, de plusieurs ou de toutes les lignes du programme en mémoire dans la fenêtre LIST.

Exemple :

LIST	liste complète du programme
LIST 10	affichage de la ligne 10
LIST 10,50,100	affichage des lignes 10, 50 et 100
LIST 10–50	affichage des lignes comprises entre 10 et 50
LIST –100	affichage des lignes précédant la ligne 100
LIST 100–	affichage depuis la ligne 100 jusqu'à la fin.

LLIST — Impression du programme

Format

LLIST [<Lignes à imprimer>]
où <Lignes à imprimer> est une ligne ou une liste de lignes à imprimer, selon les mêmes conventions que pour l'instruction LIST.

Fonction

Affichage d'une, de plusieurs ou de toutes les lignes du programme en mémoire dans la fenêtre LIST.

Exemple :

Voir **LIST**



LOAD Chargement d'un programme

Format

LOAD <Nom de fichier>
où <Nom de fichier> est le nom du programme à charger en mémoire.

Fonction

Cette instruction charge le programme ST BASIC spécifié en mémoire. Attention, l'éventuel programme qui se trouvait en mémoire est perdu.

Exemple :

```
LOAD MONPROG
```

LOC Numéro d'enregistrement ou nombre d'octets échangés avec un fichier

Format

<Variable entière> = LOC(<Numéro de fichier>)
où <Numéro de fichier> est le numéro logique associé au fichier dans une précédente instruction OPEN
et <Variable entière> contient une information qui dépend du type du fichier spécifié (voir ci-dessous).

Fonction

Cette fonction renvoie :

— pour un fichier à accès direct, le dernier numéro d'enregistrement accédé,

— pour un fichier séquentiel, le nombre d'octets lus ou écrits dans le fichier.

Exemple :

Si #1 est un fichier à accès direct, **PRINT LOC(1)** affichera le dernier numéro d'enregistrement accédé dans ce fichier.
Si #1 est un fichier séquentiel, **PRINT LOC(1)** affichera le nombre d'octets lus ou écrits dans le fichier.

LOF Taille d'un fichier

Format

<Variable entière> = LOF(<Numéro de fichier>)
où <Numéro de fichier> est le numéro logique associé au fichier dans une précédente instruction OPEN,
et <Variable entière> contient la taille du fichier après exécution de la fonction.

Fonction Cette fonction renvoie la taille d'un fichier repéré par son numéro logique.

Exemple :

```
PRINT LOF(#1)
affiche la taille du fichier #1
```

LOG Logarithme népérien

Format

<Variable réelle> = LOG(<Expression numérique>)
où <Expression numérique> est une valeur entière ou réelle supérieure à 0,
et <Variable réelle> est le logarithme népérien de <Expression numérique>.

Fonction Cette fonction renvoie le logarithme népérien de l'expression numérique qui lui est passée.

Exemple :

```
PRINT LOG(1)
0
PRINT LOG(10)
2.3025
```

LOG10 Logarithme décimal

Format

<Variable réelle> = LOG10(<Expression numérique>)
où <Expression numérique> est une valeur entière ou réelle supérieure à 0,
et <Variable réelle> est le logarithme décimal de <Expression numérique>.

Fonction Cette fonction renvoie le logarithme décimal de l'expression numérique qui lui est passée.

Exemple :

```
PRINT LOG10(1)
0
PRINT LOG10(10)
1
```



LPOS Position de la tête d'impression

Format

<Variable entière> = LPOS(X)
où X est une variable fictive (elle n'a aucun intérêt),
et <Variable entière> contient la position de la tête d'impression après exécution de la fonction.

Fonction

Cette fonction renvoie la position de la tête d'impression depuis le dernier retour chariot, sans prendre en compte les éventuels caractères de tabulation utilisés. L'impression d'un Backspace diminue de 1 la valeur renvoyée par la fonction.

Exemple :

```
PRINT LPOS(X)
```

LPRINT Impression de variables et de constantes

Format

LPRINT [<Expression>,...,<Expression>]
où les <Expression> sont du type entier, réel ou chaîne.

Fonction

Cette instruction imprime la ou les expression(s) spécifiées.

Exemple :

```
J=12 : M=08 : A=89
LPRINT "Nous sommes le ";J;M;A
```

LSET Initialisation d'un champ d'un fichier à accès direct et justification à gauche

Format

LSET <Variable chaîne> = <Expression chaîne>
où <Expression chaîne> est une valeur ou une variable chaîne,
et <Variable chaîne> est un des champs d'un fichier à accès direct.

Fonction

Cette instruction affecte une valeur chaîne à un champ d'un fichier à accès direct en le justifiant à gauche.

Exemple :

Affectation du nom MARTIN à la variable champ NOM$ et justification à gauche :

```
LSET NOM$=''MARTIN''
```

MERGE Mixage de deux programmes ST Basic

Format

MERGE <Nom de programme>
où <Nom de programme> est le nom du programme à mélanger avec le programme en mémoire.

Fonction

Cette instruction mélange les lignes du programme en mémoire et les lignes du programme spécifié. Attention, si une ligne du fichier spécifié a la même valeur qu'une des lignes en mémoire, elle la remplace.

Exemple :

Ajout des lignes du programme SECPROG au programme en mémoire :

```
MERGE''SECPROG''
```

MID$ Extraction d'une sous-chaîne dans une chaîne

Format

<Variable chaîne>=MID$(<Chaîne source>,<Début>[,<Longueur>])
où <Chaîne source> est le nom de la chaîne dans laquelle des caractères vont être extraits,
<Début> est la position du premier caractère à extraire dans la chaîne, et <Longueur> est le nombre de caractères à extraire.

Fonction

Cette fonction extrait <Longueur> caractères à partir du caractère <Début> dans <Chaîne source>.

Exemple :

```
A$=''L'été est là''
PRINT MID$(A$,5,7)
é est l
```



MKD$ Conversion d'une valeur réelle double précision (8 octets) en une chaîne

Format

<Variable chaîne> = MKD$(<Réel DP>)
où <Réel DP> est un nombre réel double précision
et <Variable chaîne> est une variable alphanumérique.

Fonction

Cette fonction convertit un réel double précision en une chaîne de 8 caractères. Elle doit être utilisée pour stocker des réels double précision dans des fichiers à accès direct. Reportez-vous également aux fonctions MKI$ et MKS$.

Exemple :

Initialisation de la variable champ A$ avec la valeur de la variable double précision VDP#

```
LSET A$=MKD$(VDP#)
```

MKI$ Conversion d'une valeur entière (2 octets) en une chaîne

Format

<Variable chaîne> = MKI$(<Entier>)
où <Entier> est un nombre entier,
et <Variable chaîne> est une variable alphanumérique.

Fonction

Cette fonction convertit un entier en une chaîne de 2 caractères. Elle doit être utilisée pour stocker des entiers dans des fichiers à accès direct. Reportez-vous également aux fonctions MKD$ et MKS$.

Exemple :

Initialisation de la variable champ B$ avec la valeur de la variable entière VE%

```
LSET B$=MKD$(VE%)
```

MKS$ Conversion d'une valeur réelle simple précision (4 octets) en une chaîne

Format

<Variable chaîne> = MKS$(<Réel simple précision>)
où <Réel SP> est un nombre réel simple précision,
et <Variable chaîne> est une variable alphanumérique.

Fonction

Cette fonction convertit un réel simple précision en une chaîne de 4 caractères. Elle doit être utilisée pour stocker des réels simple précision dans des fichiers à accès direct. Reportez-vous également aux fonctions MKI$ et MKD$.

Exemple :

Initialisation de la variable champ C$ avec la valeur de la variable réelle simple précision VSP !
LSET C$ = MKD$(VSP!)

NAME — Modification du nom d'un fichier

Format NAME <Expr chaîne 1> AS <Expr chaîne 2>
où <Expr chaîne 1> est le nom du fichier à renommer,
et <Expr chaîne 2> est le nouveau nom du fichier.

Fonction Cette instruction change le nom d'un fichier disque.

Exemple :

Récupération d'un fichier .BAK :
NAME "STAT.BAK" AS "STAT.DOC"

NEW — Effacement du fichier en mémoire

Format NEW [<Nom de programme>]
où <Nom de programme> est le nom du programme à effacer.

Fonction Cette fonction efface le programme en mémoire. Dans le cas où un nom de programme est précisé, l'instruction SAVE pourra être par la suite utilisée sans préciser le nom du programme à sauvegarder.

Exemple :

NEW
NEW MONPROG

NEXT — Marque de fin de boucle FOR

Format NEXT [<Compteur>...[,<Compteur>]...]
où les <Compteur> sont des noms d'index utilisés dans des boucles FOR.

Fonction Cette instruction marque la fin d'une ou de plusieurs boucles FOR.

Exemple :

```
10 FOR I=1 TO 10
20 PRINT I
30 NEXT I
```

OCT$ Conversion octale

Format <Variable chaîne> =OCT$(<Expression numérique>)
où <Expression numérique> est exprimé en décimal ou en hexadécimal, et <Variable chaîne> est la conversion octale de <Expression numérique>.

Fonction Cette fonction convertit le nombre décimal ou hexadécimal passé en une chaîne octale (base 8).

Exemple :

```
PRINT OCT$(10)
12
```

OLD Lecture d'un programme

Format OLD <Nom de fichier>
où <Nom de fichier> est le nom du programme à charger en mémoire.

Fonction Cette instruction charge le programme ST Basic spécifié en mémoire. Elle est équivalente à l'instruction LOAD. Attention, l'éventuel programme qui se trouvait en mémoire est perdu.

Exemple :

```
OLD MONPROG
```

ON Débranchement en fonction de la valeur d'une variable

Format

ON <Expression numérique> GOTO <Ligne 1>[, ... <Ligne N>]
où <Expression numérique> est une variable entière,
et les <Ligne I> sont des numéros de lignes.

Fonction

Cette instruction effectue un débranchement en fonction de la valeur de la variable spécifiée après le mot clé ON. Si cette variable vaut 1, le débranchement se fait au premier numéro de ligne spécifié. Si cette variable vaut 2, le débranchement se fait au second numéro de ligne spécifié, etc. Si cette variable vaut 0 ou un nombre supérieur aux lignes spécifiées, il n'y a aucun débranchement.

Exemple :

Débranchement à trois lignes différentes en fonction de la valeur de la variable A :
ON A GOTO 1000,2000,3000
Activation de deux procédures différentes en fonction de la valeur de la variable A :
ON A GOSUB 1000,2000

ON ERROR Débranchement en cas d'erreur

Format

ON ERROR GOTO <Ligne>
où <Ligne> est le numéro de la première ligne de la procédure de traitement des erreurs.

Fonction

Cette instruction définit le numéro de la ligne à laquelle le programme doit donner le contrôle lorsqu'une erreur se produit.

Exemple :

Définition du point d'entrée de la procédure de traitement d'erreurs en ligne 1000 :
ON ERROR GOTO 1000

OPEN Ouverture d'un fichier ou début de dialogue avec un périphérique

Format

OPEN <mode>,[#]<Numéro de fichier>,<Nom de fichier>
[,<Longueur d'enregistrement>]
où <mode> est le mode d'utilisation du fichier :
O pour un fichier séquentiel ouvert en écriture,



I pour un fichier séquentiel ouvert en lecture,
R pour un fichier à accès direct ouvert en lecture/écriture,
<Numéro de fichier> est un entier compris entre 1 et 15 qui sera utilisé par toutes les instructions de lecture et/ou d'écriture futures,
<Nom de fichier> est le nom du fichier à ouvrir et <Longueur d'enregistrement> est la taille d'un enregistrement pour les fichiers à accès direct.

Fonction

Cette instruction permet d'affecter un numéro logique à un fichier, et de l'ouvrir en lecture (accès séquentiel), en écriture (accès séquentiel) ou en lecture/écriture (accès direct).

La taille d'un enregistrement d'un fichier à accès direct est égale par défaut à 128 octets. Cette valeur lui est affectée lorsque le paramètre <Longueur d'enregistrement> n'est pas spécifié.

Exemple :

Ouverture d'un fichier à accès direct de nom TOTO et dont chaque enregistrement fait 12 octets :
OPEN "R",#1,"TOTO",12
Ouverture du fichier séquentiel TATA en lecture :
OPEN "I",#2,"TATA"

OPENW — Ouverture d'une fenêtre de ST Basic

Format

OPENW <Numéro de fenêtre>
où <Numéro de fenêtre> est le numéro de la fenêtre à ouvrir : 0=Edition, 1=List, 2=Sortie, 3=Commande.

Fonction

Cette fonction ouvre une fenêtre ST Basic précédemment fermée à l'aide de l'instruction CLOSEW.

Exemple :

Ouverture de la fenêtre LIST
OPENW 1

OPTION BASE Premier indice des tableaux par défaut

Format OPTION BASE <{1,0}>.

Fonction Cette instruction définit l'indice du premier élément des tableaux. Cette valeur peut être égale à zéro ou à un.

Exemple :

```
OPTION BASE 0
```

OUT Envoi d'un octet sur un port de sortie

Format OUT <Expression entière 1>,<Expression entière 2>
où <Expression entière 1> est le numéro du port de sortie,
et <Expression entière 2> la valeur octet à envoyer sur le port de sortie.

Fonction Cette fonction envoie un octet sur le port d'entrée/sortie dont le numéro est spécifié. Le numéro du port doit être compris entre 0 et 65535. Les quatre premiers numéros représentent les périphériques suivants :
0 : imprimante parallèle
1 : port RS 232
2 : système clavier/écran
3 : interface MIDI.

Exemple :

```
Envoi du caractère A vers l'imprimante
OUT 0,65
```

PCIRCLE Tracé de cercles et arcs de cercle pleins

Format PCIRCLE <Abscisse centre>,<Ordonnée centre>,<Rayon>,
[<Angle de départ>,<Angle de fin>]
où <Abscisse centre> et <Ordonnée centre> représentent les coordonnées du centre du cercle,
<Rayon> est le rayon du cercle,
<Angle de départ> et <Angle de fin> sont les angles de l'arc de cercle à tracer. Ils sont exprimés en degrés multipliés par 10.



Fonction Cette instruction trace des cercles ou arcs de cercle pleins sur l'écran. Elle peut également être utilisée pour allumer ou éteindre un pixel sur l'écran (au même titre que les instructions CIRCLE et LINEF). La couleur de remplissage est définie dans l'instruction COLOR.

Exemple :

Tracé d'un cercle de rayon 30 et de centre 100,100
PCIRCLE 100,100,30
Tracé d'un arc de cercle de rayon 30, de centre 100,100 et d'angles de début et de fin 30 et 60 degrés
PCIRCLE 100,100,30,300,600

PEEK Lecture du contenu d'une adresse en mémoire

Format <Variable entière> = PEEK(<Adresse mémoire>)
où <Adresse mémoire> est l'adresse de la mémoire à lire,
et <Variable entière> est la valeur qui se trouve à l'adresse spécifiée.

Fonction Cette fonction renvoie le contenu de l'adresse spécifiée. Reportez-vous également à l'instruction DEF SEG qui définit l'offset de l'instruction POKE et de la fonction PEEK.

Exemple :

PRINT PEEK(100)

PELLIPSE Tracé d'ellipses pleines et d'arcs d'ellipse pleins

Format PELLIPSE <Abscisse centre>,<Ordonnée centre>,
<Rayon horizontal>,<Rayon vertical>,
[<Angle de départ>,<Angle de fin>]
où <Abscisse centre> et <Ordonnée centre> contiennent les coordonnées du centre de l'ellipse,
<Rayon horizontal> et <Rayon vertical> contiennent les rayons de l'ellipse,
<Angle de départ> et <Angle de fin> délimitent (éventuellement) l'arc d'ellipse à tracer.

Fonction L'instruction ELLIPSE permet de tracer des ellipses ou des arcs d'ellipse

pleins. Reportez-vous à la Partie 5 pour prendre connaissance des coordonnées graphiques de l'écran.

Pour tracer un arc d'ellipse, vous devez spécifier l'angle de départ et l'angle de fin de tracé. Ces arguments sont exprimés en degrés et multipliés par 10 (120 pour 12 degrés par exemple).

Exemple :

Pour vous rendre compte des effets des différents paramètres, tapez les instructions suivantes :

```
PELLIPSE 100,100,50,30
PELLIPSE 100,100,100,30
PELLIPSE 10,100,30,50
PELLIPSE 50,50,30,30,0,1800
```

POKE — Stockage d'un octet à une adresse mémoire

Format

POKE <Adresse>,<Donnée>
où <Adresse> est l'adresse de la mémoire à écrire,
et <Donnée> est la valeur à stocker à l'adresse spécifiée.

Fonction

Cette instruction stocke la valeur spécifiée à l'adresse mémoire spécifiée. Reportez-vous également à l'instruction DEF SEG qui définit l'offset de l'instruction POKE et de la fonction PEEK.

Exemple :

```
POKE 100,20
A=125
POKE 2312,A
```

POS — Position du curseur sur l'écran ou de la tête de l'imprimante

Format

<Variable entière>=POS(<Argument>)
où <Argument> est un argument quelconque mais nécessaire,
et <Variable entière> est la position du curseur ou de la tête de l'imprimante.

Fonction

Cette fonction renvoie la colonne courante du curseur ou de la tête d'impression. La position la plus à gauche est égale à zéro.



Exemple :

```
PRINT POS(1)
```

PRINT Impression de données dans la fenêtre OUTPUT

Format

PRINT[<Expression>{,ou;}[<expression>...
<Expression>[{, ou ;}]...]

où les <Expression> sont les données à afficher.

Fonction

Cette instruction affiche une ou plusieurs expressions sur l'écran. Deux expressions séparées par un caractère point-virgule seront affichées à la « queue leu leu ». Deux expressions séparées par une virgule seront séparées d'une tabulation.

Exemple :

```
PRINT 10
10
RESULTAT=12.5
PRINT "Le résultat est égal à" ; RESULTAT
Le résultat est égal à 12.5
```

PRINT # Sauvegarde de données dans un fichier séquentiel

Format

PRINT #<Numéro de fichier>,<Expression>...,<Expression>
où les <Expression> sont les données à stocker dans le fichier de numéro logique <Numéro de fichier>.

Fonction

Cette instruction stocke une ou plusieurs expressions dans un fichier séquentiel ouvert en écriture.

Exemple :

```
A$="Bonjour" : B=12.8
PRINT #1,A$,B
```

PRINT USING Affichage de données formatées dans la fenêtre OUTPUT

Format

PRINT USING <Expression chaîne>;<Expression>[,...
<Expression>]

où <Expression chaîne> contient un type du formatage, et les <Expression> sont des variables ou valeurs quelconques qui seront affichées selon le format spécifié dans <Expression chaîne>.

Fonction

Cette instruction affiche une ou plusieurs données formatées dans la fenêtre OUTPUT. Les données de la chaîne de formatage peuvent être composées d'un ou de plusieurs des éléments suivants :

!	affichage du premier caractère de chaque expression,
\N espaces \	affichage de N+2 caractères de chaque expression,
&	affichage sans formatage,
#	un chiffre,
.	point décimal,
+	affichage du signe,
−	signe négatif après le nombre (le cas échéant),
**	remplir les espaces vides avec des astérisques,
$$	remplir les espaces vides avec des dollars,
**$	remplir les espaces vides avec des astérisques et le dernier espace vide avec un dollar,
,	virgule tous les milliers,
^	notation exponentielle.

Exemple :

```
RESULTAT=1234.5678
PRINT USING "Le résultat est : -####.##",RESULTAT
1234.56
```

PRINT # USING Sauvegarde de données formatées dans un fichier séquentiel

Format

PRINT #<Numéro de fichier>,USING<Expression chaîne>;
<Expression>[...<Expression>]

où <Numéro de fichier> est un numéro logique de fichier,
<Expression chaîne> contient un type du formatage,
et les <Expression> sont des variables ou valeurs quelconques qui seront affichées selon le format spécifié dans <Expression chaîne>.

Fonction

Cette instruction permet de sauvegarder des données formatées dans des fichiers séquentiels ouverts en écriture. Reportez-vous à l'instruc-



cédente pour prendre connaissance des divers types de formatage possibles.

Exemple :

```
RESULTAT=1234.5678
PRINT #1,USING "Le résultat est : -#####.##", RESULTAT
1234.56
```

PUT Sauvegarde d'un enregistrement dans un fichier à accès direct.

Format

PUT #<Numéro de fichier>,<Numéro d'enregistrement>
où <Numéro de fichier> est le numéro logique du fichier défini dans l'instruction OPEN,
et <Numéro d'enregistrement> est le numéro de l'enregistrement à écrire.

Fonction

Cette instruction écrit l'enregistrement spécifié. Lorsque le numéro d'enregistrement n'est pas spécifié, l'enregistrement qui suit l'enregistrement courant est accédé.

Exemple :

Accès à l'enregistrement numéro 12 dans le fichier logique #3 :

```
PUT #3,12
```

QUIT Sortie de ST Basic, et retour sous GEM

Format

QUIT.

Fonction

Cette instruction ferme tous les fichiers ouverts, quitte le ST Basic et retourne sous GEM.

Exemple :

```
QUIT
```

RANDOMIZE Initialisation du générateur pseudo-aléatoire

Format

RANDOMIZE [<Expression numérique>]
où <Expression numérique> est un entier compris entre −32768 et 32767.

Fonction

Cette instruction fixe la séquence des nombres pseudo-aléatoires tirés par la fonction RND. L'argument est obligatoire. S'il n'est pas spécifié, le ST Basic le demande lors de l'exécution.

Exemple :

```
RANDOMIZE 34
```

READ Lecture de données dans des instructions DATA

Format

READ <Variable>[,...<Variable>]
où les <Variable> sont des noms de variable de tous types.

Fonction

Cette instruction affecte les données situées dans des lignes de DATA aux variables spécifiées. Les données et les variables doivent être du même type.

Exemple :

```
READ A,B$,C
FOR I=1 TO 10
READ A[I]
NEXT I
```

REM Insertion de remarques dans un programme

Format

REM <Remarque>
où <Remarque> est une chaîne alphanumérique quelconque.

Fonction

Cette instruction permet d'insérer des commentaires dans un programme. Elle peut être abrégée en ' (voir exemples).

Exemple :

```
REM Ceci est un commentaire
'Ceci est un commentaire
```

RENUM Renumérotation d'un programme

Format

RENUM [<Nouvelle première ligne>],[<Première ligne>]
[,<Incrémentation>]
où <Nouvelle première ligne> est le numéro de la première ligne après renumérotation,
<Première ligne> est le numéro de la première ligne à renuméroter,
et <Incrémentation> est l'écart entre deux lignes consécutives après renumérotation.

Fonction

Cette instruction permet de renuméroter les lignes (ou une partie des lignes) du programme en mémoire. Lorsqu'aucun paramètre n'est spécifié, les données suivantes sont prises par défaut : 10, 10, 10.

Exemple :

```
Avant renumérotation
10 REM Première ligne
11 REM Seconde ligne
12 REM Troisième ligne

RENUM 1000,10,100
1000 REM Première ligne
1100 REM Seconde ligne
1200 REM Troisième ligne
```

REPLACE Sauvegarde du programme en mémoire

Format

REPLACE [<Nom de fichier>[,<Liste de lignes>]]
où <Nom de fichier> est le nom de la nouvelle version du programme en mémoire,
et <Liste de lignes> est la ou les ligne(s) à stocker dans le fichier.

Fonction

Cette instruction sauvegarde dans un fichier disque le programme en mémoire. Si aucun nom de fichier n'est précisé, le programme est recopié dans le même fichier que celui chargé par l'instruction LOAD ou OLD

précédente. Si un ou plusieurs numéros de lignes sont spécifiés, la ou les lignes correspondantes sont les seules à être sauvegardées. Cette instruction est équivalente à l'instruction SAVE.

Exemple :

```
REPLACE
REPLACE TOTO.BAS
REPLACE TOTO.BAS,10-100
```

RESET Fenêtre OUTPUT dans le buffer graphique

Format RESET.

Fonction Cette instruction copie le contenu de la fenêtre OUTPUT dans le tampon graphique. Il pourra par la suite être sauvegardé sur disque ou réaffiché par l'instruction OPENW 2.

Exemple :

```
RESET
```

RESTORE Relecture des données dans les instructions DATA

Format RESTORE[<Numéro de ligne>]
où <Numéro de ligne> est la ligne à partir de laquelle sera lue la prochaine donnée.

Fonction Cette instruction définit la ligne à partir de laquelle la prochaine instruction READ lira une donnée. Lorqu'aucun numéro de ligne n'est précisé, la première ligne de données est prise par défaut.

Exemple :

```
RESTORE
RESTORE 1000
```



RESUME Poursuite de l'exécution du programme après une erreur

Format

RESUME (0)
ou RESUME NEXT
ou RESUME <Numéro de ligne>
où <Numéro de ligne> est le numéro de la ligne à exécuter après erreur.

Fonction

Cette instruction définit le numéro de la ligne à exécuter après l'occurrence d'une erreur. Si le numéro de ligne est omis ou est égal à zéro, l'instruction exécutée est celle qui a provoqué l'erreur. Si NEXT est spécifié après RESUME, l'instruction exécutée est celle qui suit l'instruction qui a provoqué l'erreur.

RETURN Fin de sous-programme

Format

RETURN.

Fonction

Cette instruction donne le contrôle à l'instruction qui suit le GOSUB ou le ON GOSUB précédemment exécuté. Elle est utilisée pour rendre la main au programme appelant.

RIGHT$ Extraction de n caractères à droite dans une chaîne

Format

<Variable chaîne> = RIGHT$(<Chaîne source>,<Longueur>)
où <Chaîne source> est le nom de la chaîne dans laquelle des caractères vont être extraits,
et <Longueur> est le nombre de caractères à extraire.

Fonction

Cette fonction extrait les n caractères les plus à droite de la chaîne alphanumérique spécifiée.

Exemple :

```
PRINT RIGHT$("ABCDEFGHIJ",5)
FGHIJ
```

RND Extraction d'un nombre pseudo-aléatoire

Format

<Variable réelle> = RND[(<Expression numérique>)]
où <Expression numérique> vaut :
— 0 pour obtenir à nouveau le dernier nombre tiré,
— un nombre négatif pour fixer une nouvelle séquence de nombres aléatoires et renvoyer le premier nombre de cette séquence,
— un nombre positif pour renvoyer le prochain nombre de la séquence de nombres aléatoires,
<Variable réelle> est un nombre aléatoire compris entre 0 et 1.

Fonction

Cette fonction renvoie un nombre réel aléatoire compris entre 0 et 1, en accord avec l'instruction RANDOMIZE précédente.

Exemple :

```
100 FOR I=1 TO 10
110 IF RND>0.5 then PRINT "sup" ELSE PRINT "inf"
120 NEXT I
sup
sup
sup
sup
```

RSET Initialisation d'un champ d'un fichier à accès direct et justification à droite

Format

RSET <Variable chaîne> = <Expression chaîne>
où <Expression chaîne> est une valeur ou une variable chaîne,
et <Variable chaîne> est un des champs d'un fichier à accès direct.

Fonction

Cette instruction affecte une valeur chaîne à un champ d'un fichier à accès direct en le justifiant à droite.

Exemple :

Affectation du nom MARTIN à la variable champ NOM$ et justification à droite :

```
RSET NOM$="MARTIN"
```



RUN Exécution du programme

Format

RUN
ou RUN <Numéro de ligne>
ou RUN <Nom de fichier>
où <Numéro de ligne> est le numéro de la ligne à exécuter dans le fichier en mémoire,
<Nom de fichier> est le fichier disque à exécuter.

Fonction

Cette fonction permet :
— d'exécuter le programme qui se trouve en mémoire, éventuellement à partir d'une ligne particulière,
— de charger un programme en mémoire et de l'exécuter.

Exemple :

Exécution du programme en mémoire à partir de sa première ligne :
RUN
Exécution du programme en mémoire à partir de la ligne 100 :
RUN 100
Chargement et exécution du programme MONPROG à partir de sa première ligne :
RUN MONPROG

SAVE Sauvegarde du programme

Format

SAVE [<Nom de fichier>][,<Liste de numéros de lignes>]
où <Nom de fichier> est le nom de la nouvelle version du programme en mémoire,
et <Liste de lignes> est la ou les ligne(s) à stocker dans le fichier.

Fonction

Cette instruction sauvegarde dans un fichier disque le programme en mémoire. Si aucun nom de fichier n'est précisé, le programme est recopié dans le même fichier que celui chargé par l'instruction LOAD ou OLD précédente. Si un ou plusieurs numéros de lignes sont spécifiés, la ou les lignes correspondantes sont les seules à être sauvegardées. Cette instruction est équivalente à l'instruction REPLACE.

Exemple :

```
SAVE
SAVE TOTO.BAS
SAVE TOTO.BAS,10-100
```

SGN Signe d'une expression

Format

<Variable numérique> = SGN(<Expression numérique>)
où <Expression numérique> est une variable ou une donnée numérique quelconque,
et <Variable numérique> est le résultat de la fonction SGN.

Fonction

Cette fonction renvoie la valeur :
1 si l'expression passée est positive,
0 si l'expression passée est nulle,
−1 si l'expression passée est négative.

Exemple :

```
PRINT SGN(-12)
-1
PRINT SGN(0)
0
PRINT SGN(156.2)
1
```

SIN Sinus d'un angle

Format

<Variable réelle> = SIN(<Expression numérique>)
où <Expression numérique> est une variable ou une valeur numérique quelconque,
et <Variable réelle> est le sinus de l'angle passé.

Fonction

Cette fonction renvoie le sinus de l'angle en radians passé en entrée.

Exemple :

```
PRINT SIN(3.1415)
0
```

SOUND Emission d'un son

Format

SOUND <Voix>,<Volume>,<Note>,<Octave>,<Durée>
où <Voix> est compris entre 1 et 3,
<Volume> est compris entre 0 et 15,



<Note> est compris entre 1 et 12,
<Octave> est compris entre 1 et 8,
et <Durée> est exprimé en 1/50 secondes.

Fonction Cette instruction envoie une directive au générateur sonore de l'Atari. Il est possible de déterminer la voix, le volume, la note, l'octave et la durée (voir ci-dessus).

Exemple :

```
SOUND 1,8,10,4,20
SOUND 1,0,0,0,0
```

SPACE$ Chaîne d'espaces

Format <Variable chaîne> = SPACE$(<Expression numérique>)
où <Expression numérique> contient le nombre d'espaces désirés,
et <Variable chaîne> est la chaîne d'espaces en sortie de la fonction.

Fonction Cette fonction renvoie une chaîne de blancs dont la longueur est spécifiée en entrée.

Exemple :

```
PRINT SPACE$(10);"bonjour"
          bonjour
```

SPC Affichage d'espaces dans une instruction PRINT

Format PRINT SPC(<Expression numérique>)
où <Expression numérique> contient le nombre d'espaces désirés.

Fonction Cette fonction permet d'insérer des espaces dans des données affichées ou imprimées avec les instructions PRINT, LPRINT ou PRINT#.

Exemple :

```
PRINT "Test de SPC"
PRINT "1"SPC(3) "2"SPC(2) "3"
Test de SPC
1    2  3
```

SQR Racine carrée

Format

<Variable numérique> = SQR(<Expression numérique>)
où <Expression numérique> est une variable ou une donnée numérique positive ou nulle,
et <Variable numérique> contient la racine carrée de <Expression numérique> en sortie de la fonction.

Fonction

Cette fonction renvoie la racine carrée de l'argument positif qui lui est passé.

Exemple :

```
PRINT SQR(2)
1.41421
```

STEP Pas de progression dans l'exécution d'un programme

Format

STEP
ou STEP <,Numéro de ligne>
ou STEP <Nom de fichier>
où <Numéro de ligne> est le numéro de la première ligne à exécuter,
et <Nom de fichier> est le nom du programme sur disque à exécuter.

Fonction

Cette instruction exécute ligne par ligne le programme en mémoire, éventuellement à partir de la ligne spécifiée, ou un programme qui se trouve sur disque. L'instruction CONT annule le mode d'exécution ligne par ligne.

Exemple :

```
STEP ,10
STEP MONPROG.BAS
```

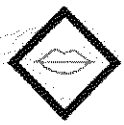
STOP Arrêt du programme courant

Format

STOP

Fonction

Cette instruction arrête le programme en cours d'exécution et donne le contrôle à la fenêtre COMMAND. L'instruction CONT permet de reprendre l'exécution du programme à l'endroit où elle a été interrompue.



Exemple :

```
STOP
```

STR$ Conversion d'une expression numérique en chaîne

Format

<Variable chaîne> = STR$(<Expression numérique>)
où <Expression numérique> est une variable ou une valeur numérique quelconque,
et <Variable chaîne> une chaîne qui contient la version alphanumérique de <Expression numérique>.

Fonction

Cette fonction convertit en chaîne une donnée numérique quelconque.

Exemple :

```
PRINT STR$(12345);"francs"
12345francs
```

STRING$ Répétition multiple d'une chaîne

Format

<Expression chaîne> = STRING$(<Expression numérique>, <ASCII ou chaîne>)
où <Expression numérique> contient le nombre de répétitions,
<ASCII ou chaîne> contient le caractère ASCII ou la chaîne à répéter,
et <Expression chaîne> est le résultat de la fonction.

Fonction

Cette fonction renvoie la répétition multiple du caractère dont le code ASCII est spécifié ou de la chaîne spécifiée.

Exemple :

```
PRINT STRING$(3,"Coucou")
Coucou Coucou Coucou
PRINT STRING$(4,65)
AAAA
```

SWAP Echange de deux variables

Format

SWAP (<Variable 1>,<Variable 2>)
où <Variable 1> et <Variable 2> sont les deux variables à échanger.

Fonction

Cette instruction échange les valeurs stockées dans les deux variables spécifiées. Bien entendu, les deux variables doivent être du même type.

Exemple :

```
10 A$="Premier":B$="Second"
20 SWAP A$,B$
30 PRINT A$,B$
Second  Premier
```

SYSTAB Début de la table des paramètres système

Format

<Variable entière>=PEEK(SYSTAB+<Offset>)
POKE SYSTAB+<Offset>,<Expression numérique>
où <Variable entière> est la valeur extraite de la table,
<Offset> est le numéro de l'élément à accéder dans SYSTAB,
et <Expression numérique> est la valeur à stocker dans cet élément.

Fonction

SYSTAB est une variable interne qui contient l'adresse de la table des paramètres systèmes. Cette table donne accès aux données suivantes :

Offset	Fonction
0	Résolution (1=haute, 2=medium, 4=basse)
2	Style de la ligne (entre 1 et 17)
4	Gestionnaire fenêtre EDIT
6	Gestionnaire fenêtre LIST
8	Gestionnaire fenêtre OUTPUT
10	Gestionnaire fenêtre COMMAND
12	Flag d'ouverture de la fenêtre EDIT (0=fermé, 1=ouvert)
14	Flag d'ouverture de la fenêtre LIST (0=fermé, 1=ouvert)
16	Flag d'ouverture de la fenêtre OUTPUT (0=fermé, 1=ouvert)
18	Flag d'ouverture de la fenêtre COMMAND (0=fermé, 1=ouvert)
20	Adresse du tampon graphique
24	Flag du GEM (0=ON, 1=OFF)

Exemple :

```
Mise OUT de GEM
POKE SYSTAB+24,1
Lecture et affichage de l'état de la fenêtre LIST
PRINT PEEK(SYSTAB+14)
```

SYSTEM Retour sous GEM

Format SYSTEM

Fonction Cette instruction est utilisée pour quitter le ST Basic et retourner au GEM. Tous les fichiers ouverts sont fermés.

Exemple :

```
SYSTEM
```

TAB Tabulation

Format PRINT TAB(<Position>)
où <Position> est la nouvelle colonne du curseur.

Fonction Cette instruction positionne le curseur à la colonne indiquée. Elle est souvent utilisée avant une instruction PRINT.

Exemple :

```
PRINT "coucou"
PRINT TAB(2) "coucou"
PRINT TAB(5) "coucou"
coucou
  coucou
    coucou
```

TAN Tangente d'un angle

Format

<Variable réelle> = TAN(<Angle>)
où <Angle> est une variable ou une valeur numérique non multiple de PI/2,
et <Variable réelle> est la tangente de l'angle passé.

Fonction

Cette fonction renvoie la tangente de l'angle en radians passé en entrée.

Exemple :

```
PRINT TAN(3.1415)
0
```

TRACE Suivi de l'exécution d'un programme

Format

TRACE [<Numéros de lignes>]
où <Numéros de lignes> spécifient les lignes à exécuter en mode TRACE.

Fonction

Cette instruction permet d'exécuter le programme en mémoire (ou seulement certaines de ses lignes) en mode TRACE: les numéros de lignes sont affichés sur l'écran pendant leur exécution. Le mode TRACE peut être annulé par l'instruction UNTRACE.

Exemple :

```
TRACE
TRACE 100-200
```

TROFF Mode TRACE OFF

Format

TROFF [<Numéros de lignes>]
où <Numéros de lignes> représentent le domaine d'action de l'instruction.

Fonction

Cette instruction dévalide entièrement ou localement le mode TRACE.



Exemple :

```
TROFF
TROFF 100-200
```

TRON Mode TRACE ON

Format TRON [<Numéros de lignes>]
où <Numéros de lignes> représentent le domaine d'action de l'instruction.

Fonction Cette instruction valide entièrement ou localement le mode TRACE.

Exemple :

```
TRON
TRON 100-200
```

UNBREAK Annulation d'une commande BREAK

Format UNBREAK [<Numéros de lignes>]
où <Numéros de lignes> représentent le domaine d'action de l'instruction.

Fonction Cette instruction dévalide entièrement ou localement le mode BREAK.

Exemple :

```
UNBREAK
UNBREAK 100-200
```

UNFOLLOW Annulation de la commande FOLLOW

Format UNFOLLOW [<Variable>][,<Variable>...[,<Variable>]...]
où <Numéros de lignes> représentent le domaine d'action de l'instruction.

Fonction Cette instruction dévalide le mode FOLLOW pour la ou les variables spécifiées.

Exemple :

```
UNFOLLOW
UNFOLLOW A$,C
```

UNTRACE Annulation de la commande TRACE

Format UNTRACE [<Numéros de lignes>]
où <Numéros de lignes> représentent le domaine d'action de l'instruction.

Fonction Cette instruction dévalide entièrement ou localement le mode TRACE.

Exemple :

```
UNTRACE
UNTRACE 10,20
UNTRACE 100-200
```

VAL Conversion d'une chaîne en nombre

Format <Variable réelle> = VAL(<Expression chaîne>)
où <Expression chaîne> est une chaîne alphanumérique qui contient la donnée numérique à convertir.

Fonction Cette fonction renvoie la conversion numérique de la chaîne alphanumérique qui lui est passée.

Exemple :

```
PRINT VAL ("123ABC")
123
PRINT VAL ("ABC123")
0
```



VARPTR Adresse d'une variable

Format

<Variable entière> = VARPTR(<Variable>)
ou <Variable entière> = VARPTR(# <Numéro de fichier>)
où <Variable> est une variable ST Basic quelconque,
<Numéro de fichier> est le numéro logique d'un fichier,
et <Variable entière> contient l'adresse de la variable ou du début du buffer de fichier.

Fonction

Cette fonction renvoie l'adresse d'une variable ou de la zone tampon utilisée pour les entrées/sorties dans un fichier.

Exemple :

```
PRINT VARPTR(A)
15236
```

VDISYS Accès à l'interface VDI

Format

VDISYS(<Argument>)
où <Argument> est sans signification mais nécessaire.

Fonction

Cette instruction permet d'accéder aux fonctions VDI. L'utilisation du VDI suppose l'initialisation préalable des tableaux suivants : INTIN, INTOUT, PTSIN, PTSOUT et CONTRL.
Le tableau CONTRL doit contenir les données suivantes :

Offset	Valeur
0	Fonction VDI
2	Paramètres dans PTSIN
4	Paramètres dans PTSOUT
6	Paramètres dans INTIN
8	Paramètres dans INTOUT
10	Numéro (éventuel) de sous-fonction
12	Sémaphore de périphérique

Les autres tableaux doivent être initialisés conformément aux exigences de la fonction activée.

Exemple :

Voir Partie 5, chapitre 2

WAIT Attente d'une donnée sur un port

Format

WAIT <Numéro de port>, <Expression entière 1>
[,<Expression entière 2>].

Fonction

Cette instruction est utilisée pour attendre une configuration donnée sur un port d'entrée.
Lorsque <Expression entière 2> est présente, l'instruction effectue un OU EXCLUSIF entre cette expression et les données qui arrivent sur le port. L'instruction effectue ensuite un ET logique entre <Expression entière 1> et le résultat de la première opération logique (si <Expression entière 2> est spécifiée), ou les données qui parviennent sur le port d'entrée (dans le cas contraire).
L'attente se poursuit jusqu'à ce que le résultat de l'opération logique (ou des opérations logiques) soit égal à zéro.

Exemple :

```
10 WAIT 6,&H12,&H7E
```
Attente d'une donnée (<D>) sur le port 6 qui satisfait la condition logique suivante :
(<D> XOR &H7E) AND &H12 = 0

```
10 WAIT 6,&H12
```
Attente d'une donnée (<D>) sur le port 6 qui satisfait la condition logique suivante :
<D> AND &H12 = 0

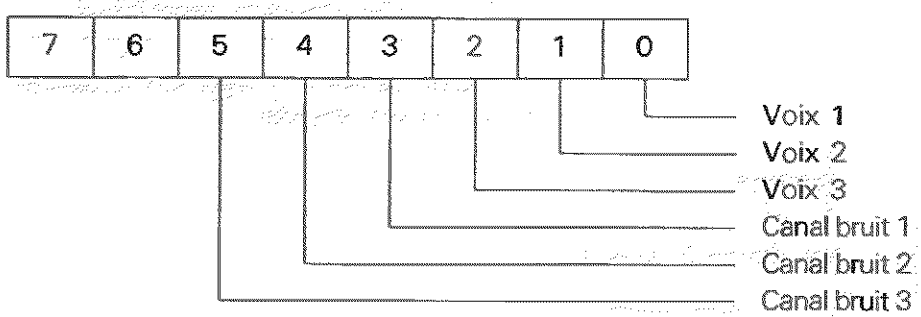
WAVE Forme d'onde de l'instruction SOUND

Format

WAVE <Validation>, <Enveloppe>, <Contrôle>, <Période>, <Délai>

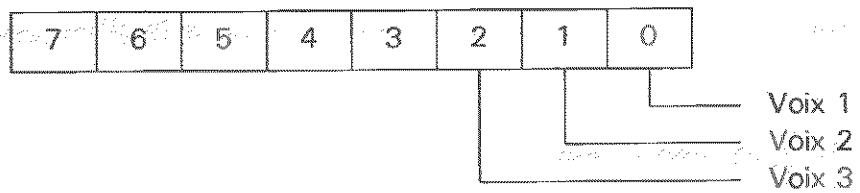
Fonction

Cette fonction est utilisée pour déterminer la forme de l'onde utilisée dans une instruction SOUND. Ses paramètres ont la signification suivante :
<Validation>
Ce paramètre octet valide les voix de son et de bruit selon les conventions suivantes :



<Enveloppe>

Ce paramètre octet valide une ou plusieurs enveloppes pour les voix 1 à 3 selon les conventions suivantes :



<Contrôle>

Les bits 0 à 3 de ce paramètre permettent de choisir la forme de l'enveloppe selon les conventions suivantes :

B3	B2	B1	B0	Forme d'onde
0	0	X	X	
0	1	X	X	
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

<Période>

Ce paramètre contient la période de l'enveloppe.

<Délai>

Ce paramètre contient la durée d'exécution de l'instruction en 1/50 secondes.

Exemple :

```
WAVE 1,1,8,1000,20
WAVE 2,2,4,2000,10
```

WEND Fin d'une boucle WHILE

Format WEND

Fonction Cette instruction marque la fin d'une boucle WHILE.

Exemple :

Reportez-vous à l'instruction WHILE

WHILE Boucle en fonction de la valeur d'une variable

Format WHILE <Expression logique>

Fonction Cette instruction exécute une ou plusieurs instructions comprises entre les mots clés WHILE et WEND tant qu'une condition logique est vraie.

Exemple :

Affichage répétitif à l'aide d'une boucle WHILE WEND

```
10 INPUT ''Entrez un nombre : '';N
20 I=0
30 WHILE I< > N
40 I=I+1
50 PRINT I
60  WEND
```

WIDTH Dimensionnement d'une ligne

Format WIDTH [LPRINT] <Expression entière>

Fonction Cette instruction détermine la largeur (en nombre de caractères) d'une ligne sur l'écran ou sur l'imprimante. La largeur par défaut est de 72 caractères.

Exemple :

Initialisation de la largeur de l'écran à 20 caractères :

```
WIDTH 20
```

Initialisation de la largeur des lignes imprimées à 60 caractères

```
WIDTH LPRINT 60
```



WRITE Affichage de données sur l'écran

Format WRITE [<Expression>[, <Expression>...[, <Expression>]...]]

Fonction Cette instruction est comparable à l'instruction PRINT. Elle permet donc d'afficher des données sur l'écran. La différence réside dans le fait que WRITE affiche automatiquement des virgules entre chaque valeur et des guillemets (") de part et d'autre de chaque chaîne.

Exemple :

```
10 A=10
20 B=20.5
30 C$="Coucou"
40 WRITE A,B,C$
```

Ces instructions affichent les données suivantes sur l'écran :

```
10 20.5 Coucou
```

WRITE # Envoi de données dans un fichier séquentiel

Format WRITE #[<Expression>[,<Expression>...[,<Expression>]...]]

Fonction Cette instruction est comparable à l'instruction PRINT #. Elle permet donc d'affecter des données à un fichier texte ouvert en écriture. La différence réside dans le fait que WRITE # insère automatiquement des virgules entre chaque valeur et des guillemets (") de part et d'autre de chaque chaîne.

Exemple :

```
10 OPEN "O",#1, "ESSAI.DAT",128
20 A=10
30 B=20.5
40 C$="Coucou"
50 WRITE #1,A,B,C$
```

Ces instructions affectent les valeurs suivantes au fichier "ESSAI.DAT" :

```
10 20.5 Coucou
```



4/2.2

Le Basic GFA

I - Historique et particularités

Comme tout programme ayant marqué son ordinateur hôte, le Basic GFA a une histoire. Celle-ci commence quand Franck Ostrowski s'achète un Atari XL. Passionné d'informatique, il s'attaque à la programmation et conçoit un nouveau Basic pour cette machine. Le temps passe... En 1985, il entend parler de l'Atari ST, machine au rapport qualité-prix révolutionnaire pour son époque et s'intéresse à elle. Il est alors contacté par la société GFA, qui a eu vent de son très bon Basic pour XL, et qui lui demande d'en écrire un autre pour le ST. Six mois plus tard, le Basic GFA est prêt ; il n'a plus qu'à être fignolé.

Lors de sa sortie, il est considéré comme le Basic interprété le plus rapide du monde. Après une réédition sensée corriger quelques bugs et l'adjonction d'un compilateur, il entreprend la programmation d'une version 3.0 qui lui apporte désormais la puissance d'un outil de développement professionnel.

Ce chapitre est consacré au Basic version 2, le compilateur de la version 3 n'étant pas encore disponible. Malgré tout, la compatibilité ascendante étant conservée entre deux versions, les exemples donnés fonctionnent aussi sur la 3.0, même s'ils ne tirent pas toujours parti de ses possibilités.

C'est bien connu : le Basic, même s'il reste le langage d'initiation à l'informatique par excellence, n'a pas que des qualités. Ce n'est pas sans raison s'il est actuellement délaissé dans les facultés françaises, au profit du Pascal. Cependant, le Basic conserve encore bien des avantages par rapport au Pascal : grande simplicité de mise au point des programmes (contrairement au Pascal qui est la plupart du temps compilé), facilité de programmation, etc. Le GFA, s'il ne prétend pas être parfait, offre pourtant une réponse à la plupart des détracteurs du Basic. Ce dernier offre en effet les avantages cumulés du Pascal et du Basic, sans en présenter la plupart des inconvénients :

— une programmation structurée (pas de numéros de lignes) ;
— une seule instruction par ligne, pour une plus grande lisibilité ;
— grand nombre d'instructions ;
— grande rapidité de l'interpréteur ;
— compilateur disponible ;

— mise au point de programmes facilitée par un contrôle syntaxique de toute ligne entrée après l'appui sur **RETURN** ;

— éditeur disposant de nombreuses fonctions puissantes (copie ou déplacement de blocs, recherche-remplacement, etc.).

AVANTAGES DE LA PROGRAMMATION STRUCTURÉE

Un des grands plus d'une programmation structurée est la possibilité d'avoir de nombreuses routines indépendantes, donc plus faciles à mettre au point, et réutilisables. Vous pouvez aussi vous concocter une librairie de routines qui vous servira par la suite.

LES PARTICULARITÉS DU GFA

La version « RUN-ONLY »

Sur la disquette d'origine, se trouve aussi une version bridée du langage ne permettant que l'exécution d'un programme et non sa modification. L'avantage est que vous pouvez ainsi donner autour de vous vos œuvres et en faire profiter vos amis, même si ceux-ci ne possèdent pas le GFA.

Le mode DIRECT

Vous venez d'apporter la touche finale à votre merveilleux programme de 200 Ko, celui qui vous rendra millionnaire et plus célèbre que Bill Gates (pour ceux qui ne le sauraient pas, Bill Gates est le fondateur de Microsoft, une des plus grandes sociétés de logiciels au monde).

Vous sauvez fébrilement votre source sur disquette quand, brusquement, le message **Disquette pleine** s'affiche à l'écran. Horreur, aucune disquette vierge à l'horizon ! Eh bien, grâce au GFA, il existe désormais une alternative au suicide pur et simple. En effet, le GFA possède un mode direct : appuyez sur la touche **ESCAPE** et l'écran s'efface, laissant apparaître un curseur. Des commandes telles que l'affichage d'un directory, l'effacement d'un fichier ou la consultation de l'état d'une variable sont disponibles.

PRINCIPALES DIFFÉRENCES ENTRE LES VERSIONS 2 ET 3 DU GFA

Mis à part plus de 100 nouvelles instructions dont il serait trop long de faire le tour, une des grandes nouveautés de la version 3.0 est le repliage des procédures : lorsqu'une procédure est tout à fait au point, vous n'avez a priori plus besoin de la conserver à l'écran et vous la repliez en l'effaçant. Seul son nom reste apparent à l'écran. Avantage : vous pouvez visualiser l'intégralité d'un programme en quelques pages, et ainsi en avoir une vue plus générale.



La rapidité a elle aussi été augmentée. Le GFA : la vitesse facile !...

Tests de rapidité

Boucle de 10000 à vide :

```
FOR I=1 To 10000
NEXT I
```

GFA BASIC 2.02 : 1.17 secondes
GFA BASIC 3.03 : 0.52 secondes

Boucle de 10000 à vide (I est en valeurs entières) :

```
FOR I%=1 To 10000
NEXT I%
```

GFA BASIC 2.02 : 0.4 secondes
GFA BASIC 3.03 : 0.25 secondes

Ecriture de 1000 "Bonjour" à l'écran

```
For I=1 To 1000
Print At(10,10) ; "Bonjour"
NEXT I
```

GFA BASIC 2.02 : 3.4 secondes
GFA BASIC 3.03 : 3.24 secondes

Ecriture de 1000 "Bonjour" à l'écran
(I est en valeurs entières) :

```
For I%=1 To 1000
  Print At(10,10) ;"Bonjour"
Next I%
```

GFA BASIC 2.02 : 3.32 secondes
GFA BASIC 3.03 : 3.21 secondes

Voici un autre exemple de l'exceptionnelle rapidité du GFA. Cette petite routine charge une image au format DEGAS (un standard sur ST), puis la découpe en un morceau et l'assimile à la souris : le morceau se déplace selon ses mouvements.

```
Bload "*.pi?",Xbios(3)-34
Sget Decor$
Get 0,0,300,300,P$
E$=Space$(32256)
E%=Int(Varptr (E$)/256)*256
Ad1%=E%
Ad2%=Xbios(2)
Hidem
Do
  Void Xbios(5,L:Ad1%, L:Ad2%, W:-1)
  Vsync
  Swap Ad 1%, Ad2%
  Exit If Mousek <>0
```

```
  Sput Decor$
  Put Mousex,Mousey,P$
Loop
Void Xbios (5,L:1015808, L:1015808, W:-1)
Setcolor 0,6,6,6
```

Description de la routine :

Sur ST, l'adresse de RAM vidéo peut être changée à condition que son début se positionne à un multiple de 256 octets.

L'astuce consiste à avoir deux pages vidéo. La première sert à la visualisation de l'image actuelle, et la seconde est la page de travail où se construit la prochaine image. Ainsi, l'ordinateur ne travaille jamais sur la page visualisée (ceci pour éviter un scintillement de l'écran). Lorsque la page suivante est prête, l'ordinateur l'assimile comme nouvelle RAM vidéo et construit l'image suivante sur la page devenue inutile.

Ces quelques temps d'exécution peuvent vous donner une petite idée de l'exceptionnelle rapidité de cet interpréteur.

LES BASIC PRÉSENTS SUR LE MARCHÉ

Le GFA n'est pas le seul Basic sur l'Atari ST. Existent aussi l'Omikron Basic, le Fast Basic, le compilateur LDW, Basalg, et le ST Basic (pour ne citer que les principaux).

L'Omikron Basic est lui aussi un phénomène de rapidité et de puissance. Il a d'ailleurs été choisi par plusieurs filiales Atari en Europe (dont l'Allemagne) comme langage fourni en standard à l'achat de la machine. Il est légèrement plus rapide que le GFA, et un compilateur pour sa nouvelle version 3.0 est déjà disponible. Pourtant, son implantation en France n'est pas encore très importante.

Le Fast Basic est anglais et présente la particularité d'être vendu en cartouche. De plus, il possède un assembleur intégré et un synthétiseur vocal.

Le compilateur LDW est, comme son nom l'indique, un compilateur Basic générant un code rapide et comptact, mais lent à la compilation ; de plus, il ne compile ni le GFA, ni l'Omikron.

Quant à Basalg (Basic Algébrique), ce Basic reste un cas particulier. Sa spécialité est le calcul mathématique. Des transformations de matrices aux fonctions hyperboliques, en passant par les calculs d'imaginaires, sa rapidité va tout simplement vous foudroyer. Ce n'est pas sans raison puisqu'il a été écrit par un chercheur du CNRS (disponible à Pressimage, 210, rue du Faubourg-Saint-Martin, 75010 Paris).

Le ST Basic fait quant à lui figure de pauvre. Ses nombreux bugs, sa lenteur et son manque total d'ergonomie le placent définitivement en dernière place.



II - Le Basic GFA et la récursivité

Le GFA est un langage récursif. Cela signifie qu'une routine peut elle-même s'appeler. Du point de vue théorique, tout cela est bien joli, mais quel avantage pratique peut en tirer un programmeur ? Voici un exemple répondant à cette question...

```
DEFFN f(t)=EXP(t)                    ; fonction à tracer
cx=20                                ; coefficient d'agrandissement
                                       en x
3cy=10                               ; coefficient d'agrandissement
                                       en y
dist0=1/(cx^2+cy^2)                  ; distance minimum entre deux
                                       points
ti=-5                                ; borne inférieure de la courbe
tf=3                                 ; borne supérieure de la courbe
GOSUB trace (ti,tf)                  ; on appelle la routine de tracé
ALERT 1,"  C'est
fini",1,"Return",void                ; boîte d'alerte
END                                  ; et fin

PROCEDURE trace(x0,x1)               ; routine de tracé
  PLOT  x0*cx+320,  -(FN f(x0))*cy+200
  IF (x0-x1)^2+(FN f(x0)-FN f(x1))^2>dist0
    trace (x0, (x0+x1)/2)
    trace ((x0+x1)/2,x1)
  ENDIF
RETURN
```

Ce petit programme trace une courbe pouvant être définie dans la première ligne. Ici, c'est une exponentielle toute simple. Mais où est donc la fameuse récursivité ?

En lançant le programme, vous ne manquerez sûrement pas de remarquer que la courbe tracée est parfaite. On ne trouve jamais de trou entre deux points.

Etudions attentivement la routine de tracé. Elle reçoit en paramètres d'entrée deux valeurs : les bornes inférieure et supérieure de la courbe.

La première instruction est :

```
PLOT x0*cx+320,  -(FN f(x0))*cy+200
```

qui trace le point initial de la courbe (les constantes 320 et 200 servent à centrer la courbe au milieu de l'écran et peuvent être modifiées en cas de test sur un moniteur couleur).

La seconde ligne est très importante :

```
IF (x0-x1)^2+(FN f(x0)-FN f(x1))^2>dist0
```

Elle teste si la distance entre les deux points, envoyés comme paramètres, est inférieure ou non à la distance minimum exigée. Si oui, c'est que le tracé est fini. Si non, la routine s'appelle elle-même avec de nouveaux paramètres à la valeur plus proche, et ceci autant de fois qu'il est nécessaire.

Remarques :

La distance minimum exigée devrait normalement être 1. Mais, comme le tracé s'effectue avec des coefficients d'agrandissement pour x et y, celle-ci est en fait 1/(Cx^2+Cy^2) (Cx et Cy étant les coefficients).

Cette routine permet de déterminer intelligemment le pas de la fonction et ainsi s'adapte automatiquement suivant qu'elle prend des valeurs très proches ou très éloignées.

III - L'implantation de routines assembleur dans le GFA (les liaisons dangereuses)

Rassurez-vous (ou désolez-vous), ce chapitre ne traite, contrairement à ce que pourrait laisser penser le titre, que d'informatique.

Le GFA est un Basic très puissant, mais il reste tout de même un Basic. Ainsi, certaines applications particulières sont tout de même inexploitables, car encore trop lentes (le graphisme en trois dimensions, les jeux d'arcade, etc.).

Vient alors tout naturellement à l'esprit l'idée d'écrire tout l'environnement du logiciel en GFA, tout en gardant certaines routines de calcul ou d'affichage en C ou assembleur.

Et là, les problèmes commencent...

Deux grandes méthodes existent.

La première intègre des routines assembleur directement dans le GFA, en datas. Le grand défaut de cette façon de procéder est que, dès la moindre modification de la routine en datas, il faut entièrement la retaper. De plus, seule la saisie de petites routines est humainement envisageable.

Exemple :

```
Gosub Assembleur          ; on initialise la routine
Void C:Routine%()         ; on appelle la routine

Procedure Assembleur      ; procédure d'initialisation
  Routine$=""
  Restore Donnees
  For I=1 To 1
    Read A%
    Routine$=Routine$+Mki$(A%)
  Next I
  Routine%=Varptr(Routine$)
  Donnees:
  Data &h4e75             ; datas en assembleur (ici, c'est
                          ; simplement un RTS)
Return
```

On prend une variable et on la remplit avec des datas. Pour appeler la routine, il suffit d'appeler l'adresse de la variable.



Remarque :

Des passages de paramètres sont possibles dans l'appel **'Void C: Routine%( )'.** Ils doivent être placés entre les parenthèses et sont alors stockés sur la pile utilisateur (USER STACK ou USP).

La seconde méthode consiste à avoir des appels entre les deux langages, tout en conservant des fichiers séparés. Il faut, pour cela, écrire des routines de communication entre les deux fichiers. C'est la méthode la plus longue à mettre en œuvre, mais finalement la plus pratique et la plus fiable (essayez donc de rentrer 10 Ko de routines en datas Basic sans faire une erreur).

IV - Liste des mots clés du GFA Basic 2.02

ABS, ADD, ALERT, ARRAYFILL, ARRPTR, ASC, ATN.

BASEPAGE, BGET, BIN$, BIOS, BITBLT, BLOAD, BMOVE, BOX, BPUT, BSAVE.

C:, CALL, CHAIN, CHDIR, CHDRIVE, CHR$, CIRCLE, CLEAR, CLEARW, CLOSE, CLOSEW, CLR, CLS, COLOR, CONT, CONTRL, COS, CRSCOL, CRSLIN, CVD, CVF, CVI, CVL, CVS.

DATA, DATE$, DEC, DEFFILL, DEFFN, DEFLINE, DEFLIST, DEFMARK, DEFMOUSE, DEFNUM, DEFTEXT, DFREE, DIM, DIM?, DIR, DIR$, DIV, DO, DOWNTO, DPEEK, DPOKE, DRAW.

EDIT, ELLIPSE, ELSE, END, ENDIF, EOF, ERASE, ERR, ERROR, EVEN, EXEC, EXIST, EXIT IF, EXP.

FALSE, FATAL, FIELD, FILES, FILESELECT, FILL, FIX, FN, FOR, FORM INPUT, FORM INPUT AS, FRAC, FRE.

GB, GCONTRL, GEMDOS, GEMSYS, GET, GET#, GINTIN, GOSUB, GOTO, GRAPHMODE.

HARDCOPY, HEX$, HIDEM, HIMEM.

IF, INC, INFOW, INKEY$, INP, INP?, INPUT, INPUT$, INSTR, INT, INTIN, INTOUT.

KILL.

LEFT$, LEN, LET, LINE, LINE INPUT, LIST, LLIST, LOAD, LOC, LOCAL, LOF, LOG, LOG10, LOOP, LPEEK, LPOKE, LPOS, LPRINT, LSET.

MAX, MENU KILL, MENU, MENU OFF, MID$, MIN, MKD$, MKDIR, MKF$, MKI$, MKL$, MKS$, MONITOR, MOUSE, MOUSEK, MOUSEX, MOUSEY, MUL.

NAME, NEW, NEXT.

OCT$, ODD, ON BREAK GOSUB, ON ERROR GOSUB, ON GOSUB, ON MENU, ON MENU BUTTON GOSUB, ON MENU GOSUB, ON MENU IBOX GOSUB, ON MENU KEY GOSUB, ON MENU MESSAGE GOSUB, ON MENU OBOX GOSUB, OPEN, OPENW, OPTION, OPTION BASE, OUT.

PAUSE, PBOX, PCIRCLE, PEEK, PELLIPSE, PI, PLOT, POINT, POKE, POLYFILL, POLYLINE, POLYMARK, POS, PRBOX, PRINT, PRINT USING, PROCEDURE, PSAVE, PTSIN, PTSOUT, PUT.

QUIT.

RANDOM, RBOX, READ, RELSEEK, REM, REPEAT, RESERVE, RESTORE, RESUME, RETURN, RIGHT$, RMDIR, RND, RSET, RUN.

SAVE, SDPOKE, SEEK, SETCOLOR, SETTIME, SGET, SGN, SHOWM, SIN, SLPOKE, SOUND, SPACE$, SPC, SPOKE, SPRITE, SPUT, SQR, STEP, STOP, STR$, STRING$, SUB, SWAP, SYSTEM.

TAB, TAN, TEXT, TIME$, TIMER, TITLEW, TO, TROFF, TRON, TRUE, TRUNC, TYPE.

UNTIL, UPPER$.

VAL, VARPTR, VDIBASE, VDISYS, VOID, VSYNC.

WAVE, WEND, WHILE, WINDTAB, WRITE.

XBIOS.

Cette petite liste vous fera, sans nul doute, changer d'avis si vous hésitiez encore à acheter ce merveilleux langage. N'oubliez pas que la version 3 possède de nombreuses instructions supplémentaires.

V - Deux exemples de programme

A. UN SOLITAIRE (VERSION MONOCHROME)

Ce logiciel est séparé en deux parties : la première initialise un tableau et l'écran, la seconde (entre le DO et le LOOP) est le cœur du jeu.

Procédure **Init_tab** :

Elle crée un tableau à deux dimensions (6 par 6), le remplit entièrement avec la valeur −1 (ARRAYFILL), puis met la croix du tableau de jeu avec la valeur 1. Au fur et à mesure que les pions disparaissent les valeurs correspondantes du tableau passeront à 0.

Chaque case du tableau peut ainsi avoir 3 valeurs différentes :

0 : pion déjà enlevé,
1 : pion toujours pas enlevé,
−1 : pion n'appartenant pas à la surface de jeu.

Procédure **Dessine** :

Simple routine de tracés de droites de l'aire de jeu (coordonnées des droites à modifier pour l'adaptation du programme en moyenne ou basse résolution).



```
GOSUB init_tab  ! initialisation du tableau et dessin des pions
GOSUB dessine   ! dessin de la grille
GOSUB ote_first ! enlevement du premier pion
'
DO
  GOSUB get_coord(1) ! attente d'appui sur ESC ou de clic
  aa%=a%
  bb%=b%
  '
  GOSUB inverse(aa%,bb%) ! inversion du premier pion
  GOSUB get_coord(0) ! attente d'appui sur ESC ou de clic
  GOSUB inverse(aa%,bb%) ! inversion du second pion
  '
  b1=(b%+bb%)/2
  a1=(a%+aa%)/2
  '
  IF t%(a1,b1)=1 AND a1=INT(a1) AND b1=INT(b1) AND (a%=aa% OR b%=bb%) ! test de validité du choix
    GOSUB aff(aa%,bb%) ! restauration
    t%(aa%,bb%)=0
    '
    GOSUB aff(a%,b%)
    t%(a%,b%)=1
    '
    GOSUB aff(a1,b1)
    t%(a1,b1)=0
  ELSE
    ALERT 1," ùPetit filoux !",1,"Return",void ! choix impossible
  ENDIF
LOOP ! boucle infinie
'
PROCEDURE dessine ! procédure de dessin de la grille
  LINE 200,50,300,50
  LINE 200,100,300,100
  LINE 200,300,300,300
  LINE 200,350,300,350
  FOR i%=200 TO 300 STEP 50
    LINE i%,50,i%,350
  NEXT i%
  FOR i%=150 TO 250 STEP 50
    LINE 100,i%,400,i%
  NEXT i%
  LINE 100,150,100,250
  LINE 150,150,150,250
  LINE 400,150,400,250
  LINE 350,150,350,250
RETURN
```

(suite)

```
PROCEDURE init_tab ! procédure d'init du tableau
  DIM t%(6,6)
  ARRAYFILL t%(),-1
  '
  FOR i%=2 TO 4
    FOR j%=0 TO 6
      t%(i%,j%)=1
      aff(i%,j%)
    NEXT j%
  NEXT i%
  '
  FOR i%=0 TO 6
    IF i%<2 OR i%>4
      FOR j%=2 TO 4
        t%(i%,j%)=1
        aff(i%,j%)
      NEXT j%
    ENDIF
  NEXT i%
  '
RETURN
PROCEDURE aff(i%,j%) ! procédure d'affichage ou d'effacement d'un pion
  GRAPHMODE 3
  x%=(i%+2)*50
  y%=(j%+1)*50
  CIRCLE x%,y%,8
  CIRCLE x%,y%,10
  CIRCLE x%,y%,12
  GRAPHMODE 1
RETURN
PROCEDURE inverse(i%,j%) ! procédure d'inversion d'un pion
  GRAPHMODE 3 ! choix du mode graphique
  x%=(i%+2)*50
  y%=(j%+1)*50
  PCIRCLE x%,y%,8
  GRAPHMODE 1
RETURN
PROCEDURE ote_first
  ALERT 2," |Indiquez le pion à retirer|pour commencer la partie.",1,"Return",r%
  GOSUB get_coord(1)
  t%(a%,b%)=0
  aff(a%,b%)
RETURN
PROCEDURE get_coord(r%) ! récupération d'un clic souris  ou d'appui sur ESCAPE
  REPEAT
  UNTIL MOUSEK=0
```



(suite)

```
      REPEAT
        REPEAT
          MOUSE x%,y%,z%
          a%=(x%-75)ç50 ! conversion des coordonnées du clic
          b%=(y%-25)ç50 ! en coordonnées tableau
          IF ASC(INKEY$)=27 ! appui sur ESCAPE
            ALERT 0,"ùAlors, qu'est-ce qu'on fait ?",1,"quitùnewùcont",r%
            IF r%=1
              QUIT ! fin du programme
            ELSE IF r%=2
              RUN ! on recommence une nouvelle partie
            ENDIF
          ENDIF
        UNTIL a%>=0 AND b%>=0 AND a%<7 AND b%<7 ! clic sur un pion du tableau
      UNTIL t%(a%,b%)=r% AND z%<>0 ! existant
    RETURN
```

Procédure **Ote_first** :

Affiche une boîte d'alerte demandant d'enlever le premier pion, puis met la case du tableau correspondant au pion choisi à zéro (état pion effacé).

Procédure **Aff** :

Affiche ou efface un pion du tableau de jeu.

Procédure **Inverse** :

Sélectionne un pion (celui sur lequel on vient de cliquer).

Procédure **Get_coord** :

C'est la plus complexe de toutes les routines.

Elle teste deux choses :

— l'appui sur la touche **ESCAPE**, qui permet de sortir ou de recommencer une partie depuis le début ;

— le clic sur un des pions.

Deux boucles sont effectuées en REPEAT...UNTIL.

La première pour tester la position de la souris sur la surface de jeu et la seconde pour tester un éventuel clic de sélection d'un pion.

Dans la première partie du programme, on trouve ensuite un test sur la validité du choix de l'utilisateur :

— les deux points sélectionnés doivent se trouver sur la même ligne ou la même colonne ;

— un seul d'entre eux doit être présent et ils doivent être séparés par un autre pion.

B. UN SIMULATEUR D'EXPLOSIONS

```
nbr%=20
g%=9.81*5  ! fausse valeur de g mais plus realiste
DO
  x0%=RANDOM(200)+200
  y0%=RANDOM(50)+20
  ERASE t%()
  DIM t%(nbr%,2)
  k%=0
  FOR i%=1 TO nbr%
    t%(i%,1)=RANDOM(150)-50
    t%(i%,2)=RANDOM(100)-70
    k%=MAX(k%,2/g%*(SQR(t%(i%,2)*t%(i%,2)+2*g%*y0%)-t%(i%,2)))
  NEXT i%
  CLS
  FOR t=0 TO k% STEP 0.05
    FOR i%=1 TO nbr%
      x%=t%(i%,1)*t+x0%
      y%=(g%/2*t+t%(i%,2))*t+y0%
      PSET x%,y%,1
    NEXT i%
  NEXT t
  EXIT IF INP(2)=27
LOOP
QUIT
```

Ce petit exemple est donné uniquement pour vous montrer la rapidité du Basic GFA. Testez-le...



4/2.2.1

Le Basic GFA 3.0

I - Les instructions du GFA Basic 3.0 en détails et leur utilisation

Dans ce paragraphe, nous allons passer en revue la liste des instructions et fonctions du GFA Basic 3.0 classées par ordre alphabétique. Elles seront toujours présentées sous le même modèle pour faciliter leur apprentissage :

Mot-clé Signification résumée
Format
Fonction
Exemple

ABS Valeur absolue

Format

<x>=ABS(<y>)
où <y> est une expression numérique,
et <x> vaut :
−1 si le signe de <y> est négatif,
0 si <y> est égal à zéro,
1 si le signe de <y> est positif.

Fonction

Cette fonction renvoie la valeur absolue de l'argument numérique qui lui est passé.

Exemple :

```
PRINT ABS(-3),ABS(0),ABS(454.3)
```
affiche les valeurs suivantes : 3, 0 et 454.3

ABSOLUTE Modification de l'adresse d'une variable

Format

ABSOLUTE <x>, <y>
où <x> est une variable de type quelconque,
et <y> est l'adresse d'une variable de même type que <x>.

Fonction Cette instruction modifie l'adresse d'une première variable pour la faire coïncider avec celle d'une seconde variable.

Exemple :

```
a=3
b=-5
ABSOLUTE b,*a
PRINT a,b,*a,*b
```

L'instruction PRINT ci-dessus affiche les valeurs suivantes :

```
3     3     234076     234076
```

ACHAR Affichage graphique d'un caractère

Format ACHAR <code>,<x>,<y>,<fonte>,<style>,<angle>
où <code> est le code ASCII du caractère à imprimer,
<x> et <y> sont les coordonnées du caractère sur l'écran,
<fonte> définit le type d'écriture :
0 = Ecriture d'icône,
1 = Ecriture normale en mode couleur,
2 = Ecriture normale en mode monochrome,
>2 = Adresse (MSB, LSB) d'un header de fonte,
<style> représente le style du texte (entre 0 et 31),
<angle> est l'angle de sortie du texte. Il est exprimé en degrés multipliés par 10.

Fonction Cette instruction permet d'afficher un caractère sur l'écran en précisant : son code ASCII, sa position, le type d'écriture utilisée (icône ou normale), son style et son angle de sortie exprimé en degrés × 10.

Exemple :

```
ACHAR 45,100,100,0,1,900
ACHAR 63,0,0,1,5,1800
```

ACLIP Définition de la zone d'affichage

Format ACLIP <flag>,<AbsMin>,<OrdMin>,<AbsMax>,<OrdMax>
où <flag> permet d'activer (< >0) ou de désactiver (=0) la délimitation,
<AbsMin> et <OrdMin> sont les abscisse et ordonnée minimales de la zone à délimiter,



<AbsMax> et <OrdMax> sont les abscisse et ordonnée maximales de la zone à délimiter.

Fonction

Cette instruction permet de définir une zone rectangulaire à l'intérieur de laquelle les graphismes utilisant LINE-A seront affichés.

Attention :
ACLIP n'a aucun effet sur les instructions ALINE, BITBLT, HLINE, PSET et PTST.

Exemple :

```
ACLIP 1,0,0,100,100
ACLIP 0,0,0,0,0
```

ACOS Arc-Cosinus

Format

<x>=ACOS(<y>)
où <y> est une mesure d'angle exprimée en radians,
et <x> l'angle de cosinus <y>.

Fonction

Cette fonction renvoie l'arc-cosinus de l'arc qui lui est passé.

Remarque :
Les fonctions DEG et RAD peuvent être utilisées pour convertir des angles (respectivement) en degrés et en radians.

Un cosinus étant compris entre −1 et +1, la frappe de PRINT ACOS(1.1) provoquera une erreur.

Exemple :

```
PRINT ACOS(0.5)        → -1.047197551197
PRINT ACOS(0.816)      → -0.6163392897064
```

ADD Addition

Format

ADD <x>,<y>
où <x> est une variable arithmétique,
et <y> est une expression arithmétique.

Fonction Cette instruction additionne la valeur d'une variable arithmétique au résultat d'une expression arithmétique et stocke la valeur résultante dans la variable arithmétique.

Exemple :

```
a%=1
b%=3
ADD a%,4*b%
PRINT a%          → affiche 13 (1 + 4*3)
```

ADDRIN Tableau d'entrée d'adresses AES (se reporter en Partie 3, chapitre 5.1)

Format

ADDRIN(<x>)=<y>
où <x> est l'index du tableau ADDRIN à accéder,
et <y> est la valeur (mot long) à affecter à l'élément désigné dans le tableau.

<x>=ADDRIN(<y>)
où <y> est l'index du tableau ADDRIN à accéder,
et <x> est la valeur (mot long) lue dans ce tableau.

Fonction Cette fonction permet d'accéder au tableau d'entrée d'adresses de l'AES. Les données écrites sont des mots longs codés sur 4 octets.

Exemple :

```
ADDRIN(3)=a
```

ADDROUT Tableau de sortie d'adresses AES

Format

ADDROUT(<x>)=<y>
où <x> est l'index du tableau ADDROUT à accéder,
et <y> est la valeur (mot long) à affecter à l'élément désigné dans le tableau.

<x>=ADDROUT(<y>)
où <y> est l'index du tableau ADDROUT à accéder,
et <x> est la valeur (mot long) lue dans ce tableau.

Fonction Cette fonction permet d'accéder au tableau d'entrée d'adresses de l'AES. Les données lues sont des mots longs codés sur 4 octets.



Exemple :

```
a=ADDROUT(2)
```

AFTER Activation d'une procédure après un délai

Format

AFTER<délai> GOSUB <proc>
où <délai> est un délai d'attente exprimé en multiples de 1/200 secondes,
et <proc> est le nom de la procédure à exécuter après écoulement de ce délai.

Fonction

Cette instruction active une procédure lorsque le délai spécifié en 1/200 secondes est écoulé. Le décomptage se fait dès l'exécution de l'instruction AFTER.

Exemple :

Activation de la procédure « beep » toutes les secondes :

```
AFTER 200 GOSUB beep
```

AFTER CONT Réactivation de l'instruction AFTER

Format

AFTER CONT

Fonction

Cette instruction réactive le décomptage du temps précisé dans une instruction AFTER précédemment exécutée. Elle n'a de sens que si l'instruction AFTER STOP a été exécutée. Dans le cas contraire, elle n'a aucun effet.

AFTER STOP Désactivation de l'instruction AFTER

Format

AFTER STOP

Fonction

Cette instruction désactive le décomptage du temps précisé dans une instruction AFTER précédemment exécutée. Son effet peut être annulé en utilisant l'instruction AFTER CONT.

Exemple :

```
PRINT "Appuyez sur la touche a"
AFTER 100 GOSUB bad
REPEAT
UNTIL INKEY$< > " "
IF INKEY$="a"
   AFTER STOP
ENDIF
'
'
PROCEDURE bad
  PRINT "peut mieux faire..."
RETURN
```

ALERT Affichage d'une boîte d'alerte

Format

ALERT <symbole>,<texte>,<défaut>,<bouton>,<choix>
où <symbole> est une expression entière qui permet de définir l'icône de la boîte d'alerte selon les conventions suivantes :
0 = pas d'icône,
1 = point d'exclamation,
2 = point d'interrogation,
3 = panneau stop.
<texte> est une expression chaîne qui doit contenir le texte à afficher dans la boîte d'alerte (jusqu'à 4 lignes de 30 caractères),
<defaut> est le choix par défaut des options de la boîte d'alerte,
<bouton> est une expression chaîne qui doit contenir le texte de chaque bouton de la boîte d'alerte. Chaque texte a une longueur maximale de 8 caractères et est séparé du suivant par une barre verticale |,
<choix> est une variable arithmétique dans laquelle est renvoyé le numéro du bouton sélectionné.

Fonction

Cette instruction permet de préciser tous les paramètres et d'afficher une boîte d'alerte sur l'écran.

Exemple :

Affichage d'un icône point d'exclamation dans une boîte d'alerte. Les deux possibilités de choix sont Bis et Stop, et le résultat est placé dans la variable x% :

```
ALERT 1, "Erreur",1,"Bis|Stop",x%
```



ALINE Tracé d'une ligne

Format

ALINE <AbsMin>, <OrdMin>, <AbsMax>, <OrdMax>, <coul>, <motif>, <mode>
où <AbsMin> et <OrdMin> sont les abscisse et ordonnée du point de départ de la ligne,
<AbsMax> et <OrdMax> sont les abscisse et ordonnée du point d'arrivée de la ligne,
<coul> est la couleur de tracé (entre 0 et 15),
<motif> contient une information binaire qui représente le motif du tracé,
et <mode> définit le mode d'affichage (0 = surimpression, 1 = transparent, 2 = inversion, 3 = inversion transparente).

Fonction

Cette instruction permet de tracer une ligne graphique entre deux points, en précisant la couleur, le motif et le mode de tracé.

Exemple :

```
ALINE 0,0,100,100,1,&HAFAF,0
ALINE 0,10,10,50,2,&HAAAA,2
```

AND ET logique

Format

<x> = <y> AND <z>
où <y> et <z> sont des expressions entières,
et <x> est le résultat bit à bit du ET logique entre <y> et <z>.

Fonction

Cette fonction renvoie le résultat d'un ET logique entre ses deux arguments entiers.

Pour mémoire, voici la table de vérité du ET logique :

X	0	0	1	1
Y	1	0	0	1
X ET Y	0	0	0	1

Exemple :

```
PRINT &HF0F0 AND &HA000
A000
```

APOLY Tracé et remplissage d'un polygone

Format

APOLY <AT1>, <NP>, <y0> TO <y1>, <coul>, <mode>, <AT2>, <TM>
où <AT1> est l'adresse d'un tableau qui contient les coordonnées (x et y) des arêtes du polygone,
<NP> est une expression entière qui contient le nombre d'arêtes du polygone,
<y0> et <y1> sont les ordonnées entre lesquelles le polygone doit être rempli,
<coul> est la couleur de tracé et de remplissage (entre 0 et 15),
<mode> définit le mode d'affichage (0 = surimpression, 1 = transparent, 2 = inversion, 3 = inversion transparente),
<AT2> est l'adresse d'un tableau qui contient des informations binaires représentant le motif de chaque ligne,
<TM> est l'index dans le tableau des motifs de ligne AT2.

Fonction

Cette instruction trace et remplit totalement ou partiellement un polygone. Le tracé est effectué à partir d'un ensemble de points dont les coordonnées x et y sont définies dans un tableau. Le remplissage est effectué entre les deux ordonnées spécifiées. La couleur, le mode d'affichage et le motif d'affichage sont spécifiés dans l'instruction.

Exemple :

Si a% contient les coordonnées de 10 points et b% le motif de tracé, l'instruction suivante :
APOLY a%,10,0 TO 50,1,1,b%,1
tracera un polygone à 10 côtés rempli entre les ordonnées 0 et 50 avec le motif défini dans b%.

APPL_EXIT Annulation d'un programme pour l'AES

Format

APPL_EXIT( )

Fonction

Cette instruction annule la déclaration d'un programme pour l'AES. Ainsi, son numéro d'identification redevient disponible pour d'autres programmes.

APPL_FIND

Format

<a> = APPL_FIND(<Nom>)
où <Nom> est une expression chaîne qui contient un nom de fichier écrit en majuscules, sans extension et codé sur 8 caractères (éventuellement complété avec des espaces),
et <a> est l'identificateur renvoyé.



Fonction Cette fonction renvoie l'identificateur correspondant à une application donnée, ou −1 dans le cas où l'application ne peut être trouvée.

Exemple :

```
PRINT APPL_FIND("CONTROL")
```

APPL_WRITE Ecriture de données dans un buffer d'événement

Format

APPL_WRITE (<ident>, <longueur>, <AB>)
où <ident> est une expression entière qui contient le numéro de l'application,
<longueur> est une expression entière qui contient le nombre d'octets à écrire,
<AB> est l'adresse du début du buffer d'écriture.

Fonction Cette instruction permet d'écrire un ou plusieurs octets dans un buffer d'événement.

ARECT Dessin et remplissage d'un rectangle

Format

ARECT <x1>, <y1>, <x2>, <y2>, <coul>, <mode>, <AT2>, <TM>
où <x1> et <y1> sont les abscisse et ordonnée d'un des coins du rectangle,
<x2> et <y2> les abscisse et ordonnée du coin opposé du rectangle,
<coul> est la couleur de tracé et de remplissage (entre 0 et 15),
<mode> définit le mode d'affichage (0 = surimpression, 1 = transparent, 2 = inversion, 3 = inversion transparente),
<AT2> est l'adresse d'un tableau qui contient des informations binaires représentant le motif de chaque ligne,
<TM> est l'index dans le tableau des motifs de ligne AT2.

Fonction Cette fonction permet de dessiner et de remplir un rectangle. Le rectangle est défini par les coordonnées de deux de ses angles opposés. La couleur de remplissage, le mode d'affichage et le motif de remplissage doivent être définis dans les paramètres passés à l'instruction.

Exemple :

Si a% le motif de remplissage, l'instruction suivante :
ARECT 0,0,50,100,1,1,a%,1
tracera un rectangle plein de couleur 1, en mode transparent et avec le motif décrit dans a%.

ARRAYFILL Remplissage d'un tableau

Format

ARRAYFILL <T> ( ),<R>
où <T> est le nom d'un tableau,
et <R> une expression numérique qui fixe la valeur de remplissage.

Fonction

L'instruction ARRAYFILL permet d'initialiser en une seule opération tous les éléments d'un tableau.

Exemple :

```
DIM a(15)
ARRAYFILL a( ),13
PRINT a(5)        → 13
```

ARRPTR Adresse d'une variable ou d'un tableau

Format

ARRPTR(<V>)
où <V> est le nom d'une variable ou d'un tableau (suivi de parenthèses vides dans ce dernier cas).

Fonction

L'instruction ARRPTR(x) (équivalente à *x) renvoie l'adresse d'une variable ou d'un descripteur de tableau ou de chaîne.

Exemple :

```
a=12
DIM b (10)
c$="Bonjour"
PRINT ARRPTR(a)      → 234094
PRINT ARRPTR(b( ))   → 234108
PRINT ARRPTR(c$)     → 234102
```

ASC Valeur ASCII

Format

<x>=ASC(<y>)
où <y> est une expression chaîne,
et <x> une variable entière qui contient la valeur ASCII du premier caractère de <y>.



Fonction Cette fonction renvoie la valeur ASCII du premier caractère de la chaîne qui lui est passée.

Exemple :

```
PRINT ASC ("E"), ASC ("e")    → 69    101
PRINT ASC ("Coucou")          → 67
```

ASIN Arc Sinus

Format <x>=ASIN(<y>)
où <y> est une mesure d'angle exprimée en radians,
et <x> l'angle de sinus <y>.

Fonction Cette fonction renvoie l'arc-sinus de l'arc qui lui est passé.

Remarque :
Les fonctions DEG et RAD peuvent être utilisées pour convertir des angles (respectivement) en degrés et en radians. Un sinus étant compris entre −1 et +1, la frappe de PRINT ASIN (1.1) provoquera une erreur.

Exemple :

```
PRINT ASIN (0.5)      → 0.5235987755983
PRINT ASIN (0.816)    → 0.9544570370885
```

ATEXT Affichage de chaînes de caractères

Format ATEXT <Abs>, <Ord>, <fonte>, <Ch>
où <Abs> et <Ord> sont les abscisse et ordonnée d'affichage du premier caractère de la chaîne,
<fonte> définit le type d'écriture :
0 = Ecriture d'icône,
1 = Ecriture normale en mode couleur,
2 = Ecriture normale en mode monochrome,
>2 = Adresse (MSB, LSB) d'un header de fonte
<Ch> contient la chaîne à afficher.

Fonction Cette instruction permet d'afficher un texte alphanumérique sur l'écran en précisant : sa position et le type d'écriture à utiliser (icône ou normale).

Exemple :

```
ATEXT 10,10,2,"Bonjour"
```

ATN — Arc Tangente

Format

<x> = ATN(<y>)
où <y> est une mesure d'angle exprimée en radians,
et <x> l'angle de tangente <y>.

Fonction

Cette fonction renvoie l'arc-tangente de l'arc qui lui est passé.

Remarque :
Les fonctions DEG et RAD peuvent être utilisées pour convertir des angles (respectivement) en degrés et en radians.

Exemple :

```
PRINT ATN (0.5)      — 0.4636476090008
PRINT ATN (0.816)    — 0.6844211819539
```

BASEPAGE — Adresse de la « base page »

Format

<x> = {BASEPAGE + <Depl>}
où <Depl> est un déplacement dans la « base page »,
et <x> est une variable entière qui contient l'octet pointé par la partie à droite du signe égale.

Fonction

Cette adresse permet d'accéder aux 256 premiers octets de l'interpréteur GFA. Les données suivantes sont accessibles par son intermédiaire :

Offset	Signification
0-3	Adresse de début de TPA
4-7	Adresse de fin de TPA
8-11	Adresse du segment TEXT
12-15	Longueur du segment TEXT
16-19	Adresse du segment DATA
20-23	Longueur du segment DATA
24-27	Adresse du segment BSS
28-31	Longueur du segment BSS
32-35	Adresse de la DTA
36-39	Adresse de la « base page » du programme d'appel
40-43	Réservé
44-47	Adresse du « string d'environnement »
48-127	Réservé
128-255	Ligne d'instruction



Exemple :

```
x% = (BASEPAGE + 128)
```

BCHG Inversion d'un bit

Format BCHG(<x>,<y>)
où <y> est un numéro de bit (entre 0 et 31),
et <x> une expression entière.

Fonction Cette instruction permet d'inverser la valeur d'un bit de l'argument entier qui lui est passé. Le numéro du bit à inverser doit être compris entre 0 et 31.

Exemple :

```
a% = &HFA00
PRINT BCHG (5,a%)
```
affiche **4**

BCLR Mise à zéro d'un bit

Format BCLR (<x>, <y>)
où <y> est un numéro de bit (entre 0 et 31),
et <x> une expression entière.

Fonction Cette instruction permet d'annuler un bit de l'argument entier qui lui est passé. Le numéro du bit à annuler doit être compris entre 0 et 31.

Exemple :

```
a% = &HFA00
PRINT BCLR(5,a%)
```
affiche **4**

BGET Lecture d'une zone mémoire

Format

BGET #<Canal>, <Adresse>, <Longueur>
où <Canal> est le numéro du fichier à accéder,
<Adresse> est l'adresse de chargement,
et <Longueur> le nombre d'octets à charger.

Fonction

Cette instruction permet de lire un ou plusieurs octets dans un fichier binaire sauvegardé par l'instruction BSET. Les données de ce fichier sont stockées en mémoire à partir de l'adresse spécifiée.

Exemple :

Lecture de 16000 octets dans le fichier #1 et stockage en mémoire d'écran :

```
BGET #1,XBIOS(2),16000
```

Lecture de 100 octets dans le fichier #2 et stockage dans la variable a$:

```
BGET #2,a$,100
```

BIN$ Conversion en binaire

Format

BIN$(<x>[,<y>])
où <x> est l'expression entière à convertir,
et <y> contient éventuellement le nombre de chiffres significatifs dans la conversion (entre 1 et 32).

Fonction

Cette instruction convertit un nombre entier codé en décimal en binaire.

Exemple :

```
PRINT BIN$(64)     — 1000000
PRINT BIN$(253,8)  — 11111101
```

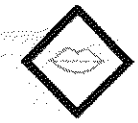
BIOS Appel d'une routine du BIOS

Format

BIOS (<Numéro>[, <x>[...,<t>]...])
où <Numéro> est le numéro de la fonction BIOS à activer,
et <x> à <t> sont les éventuels paramètres à transmettre à la fonction BIOS activée.

Fonction

Cette fonction permet d'appeler une routine du BIOS. Reportez-vous en Partie 3, chapitre 2 pour prendre connaissance de la liste des routines du BIOS.



BITBLT Activation de la routine Line A BITBLT

Format

BITBLT <Adresse>
où <Adresse> contient l'adresse d'un tableau de 76 octets qui doit être initialisé avec les valeurs suivantes :

Offset	Signification
0	Largeur de la grille en pixels
2	Hauteur de la grille en pixels
4	Nombre de plans de couleurs
6	Couleur du premier plan
8	Couleur de l'arrière-plan
10	Combinaison logique premier plan/arrière-plan
14	Déplacement en X dans la grille source
16	Déplacement en Y dans la grille source
18	Adresse de la grille source
22	Déplacement jusqu'au prochain mot du même niveau
24	Déplacement jusqu'à la prochaine ligne de la grille source
26	Déplacement jusqu'au prochain niveau de couleur
28	Déplacement en X dans la grille destination
30	Déplacement en Y dans la grille destination
32	Adresse de la grille destination
36	Déplacement jusqu'au prochain mot du même niveau
38	Déplacement jusqu'à la prochaine ligne de la grille destination
40	Déplacement jusqu'au prochain niveau de couleur
42	Pointeur sur la table des motifs de remplissage
46	Déplacement jusqu'à la prochaine ligne de la grille de masque
48	Déplacement jusqu'à la prochaine couleur du motif de remplissage
50	Masque d'affichage
52	Buffer utilisé par le blitter

Autre format possible :
BITBLT <T> ( )
où <T> est le nom d'un tableau entier qui contient les informations énumérées ci-dessus.

Fonction

Cette instruction permet d'effectuer un transfert du bloc de bits définis par une table de structure.

BLOAD Chargement d'un fichier binaire

Format

BLOAD <Nom>[,<Adresse>]
où <Nom> est une expression chaîne qui contient le nom du fichier binaire à charger,
et <Adresse> une expression arithmétique optionnelle qui contient l'adresse de chargement du fichier binaire.

Fonction Cette instruction permet de charger un fichier binaire sauvegardé par l'instruction BSAVE. L'implantation se fait par défaut à l'adresse de sauvegarde. Il est néanmoins possible de préciser une autre adresse d'implantation.

Exemple :

Chargement du fichier binaire « image » :
BLOAD "image"
Chargement du fichier binaire « image » à partir de l'adresse a% :
BLOAD "image",a%

BMOVE Copie d'une zone mémoire

Format BMOVE <AZS>, <AZD>, <NO>
où <AZS> est une expression entière qui contient l'adresse de la zone source,
<AZD> est une expression entière qui contient l'adresse de la zone destination,
et <NO> est une expression entière qui contient le nombre d'octets à déplacer.

Fonction Cette instruction permet de copier une zone mémoire définie par l'adresse de son premier octet et sa longueur en octets à l'emplacement spécifié.

Exemple :

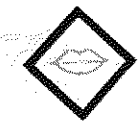
Sauvegarde de l'écran en mémoire dans le tableau ad% :

```
DIM ad%(16000)
A%=VARPTR(ad%(0))
BMOVE XBIOS(2),a%,32000
```

BOUNDARY Encadrement de la surface de remplissage

Format BOUNDARY <Indic>
où <Indic> est une expression entière qui permet d'activer (< > 0) ou de désactiver (0) l'encadrement automatique de la surface de remplissage.

Fonction Cette instruction permet d'activer ou de désactiver l'encadrement automatique de la surface de remplissage.



Exemple :

Affichage d'un rectangle plein sans encadrement :

```
DEFFILL 1,2,2
BOUNDARY 0
PBOX 0,0,50,50
```

Affichage d'un rectangle plein avec encadrement :

```
BOUNDARY 45
PBOX 100,100,150,150
```

BOX Affichage d'un rectangle

Format

BOX <x1>, <y1>, <x2>, <y2>
où <x1> et <y1> sont les abscisse et ordonnée d'un des coins du rectangle à tracer,
et <x2> et <y2> sont les abscisse et ordonnée du coin opposé du rectangle à tracer.

Fonction

Cette instruction affiche un rectangle vide sur l'écran, à partir des coordonnées de deux angles opposés.

Exemple :

```
BOX 10,10,100,100
BOX 50,60,70,90
```

BPUT Sauvegarde d'une zone mémoire

Format

BPUT #<Canal>, <Adresse>, <Longueur>
où <Canal> est le numéro du fichier à accéder,
<Adresse> est l'adresse de début de sauvegarde,
et <Longueur> le nombre d'octets à sauvegarder.

Fonction

Cette instruction permet de sauvegarder un ou plusieurs octets dans un fichier binaire sauvegardé. Les données sont lues en mémoire à partir de l'adresse spécifiée.

Exemple :

Sauvegarde des 16000 octets de la mémoire d'écran dans le fichier #1 :

```
BPUT #1,XBIOS(2),16000
```

Sauvegarde des 100 premiers octets de la variable a$ dans le fichier #2 :

```
BPUT #2,a$,100
```

BSAVE Sauvegarde dans un fichier binaire

Format

BSAVE <Nom>, <Adresse>, <Longueur>
où <Nom> est une expression chaîne qui contient le nom du fichier binaire à sauvegarder,
<Adresse> est une expression arithmétique qui contient l'adresse du premier élément mémoire à sauvegarder,
et <Longueur> est une expression arithmétique qui contient le nombre d'octets consécutifs à sauvegarder.

Fonction

Cette instruction permet de sauvegarder des données qui se trouvent en mémoire dans un fichier binaire. L'adresse de la première adresse mémoire sauvegardée ainsi que le nombre d'octets à sauvegarder doivent être précisés dans l'instruction.

Exemple :

Sauvegarde du fichier binaire « image » de longueur 32000 octets et dont la première adresse est donnée par XBIOS (2) :

```
BSAVE "image",XBIOS(2),32000
```

BSET Mise à un d'un bit

Format

BSET (<x>, <y>)
où <y> est un numéro de bit (entre 0 et 31),
et <x> une expression entière.

Fonction

Cette instruction permet de mettre à un un des bits de l'argument entier qui lui est passé. Le numéro du bit à positionner doit être compris entre 0 et 31.

Exemple :

```
a%=&H0000
PRINT BSET (2,a%)
```

affiche **3**



BTST Test d'un bit

Format

<Result> = BTST (<x>,<y>)
où <y> est un numéro de bit (entre 0 et 31),
et <x> une expression entière,
et <Result> est une expression entière qui contient le résultat du test.

Fonction

Cette fonction permet de tester un des bits de l'argument entier qui lui est passé. Le numéro du bit à tester doit être compris entre 0 et 31. La fonction renvoie la valeur −1 si le bit <y> de <x> est à un. Elle renvoie la valeur 0 dans le cas contraire.

Exemple :

```
a%=&H0000
PRINT BTST (1,a%)
```
affiche 0

BYTE Extraction des 8 bits de poids faibles

Format

Premier format :

<x>=BYTE(<y>)
où <y> est l'expression entière source,
et <x> est une expression entière qui contient les 8 bits de poids faibles de <y> après l'appel.

Deuxième format :

<y>=BYTE{<x>} ou BYTE {<x>}=<y>
où <x> est une adresse mémoire,
et <y> est un octet.

Fonction

Dans sa première forme, la fonction BYTE renvoie les huit bits de poids faibles de l'expression entière qui lui est passée.

Dans sa seconde forme, la fonction BYTE stocke ou lit un octet à l'adresse mémoire spécifiée.

Exemple :

```
PRINT BYTE(260),BYTE(45)
```
affiche 4 45
Stockage de la valeur &HAB à l'adresse 1234 :
```
BYTE{123456}=&HAB
```

C

Appel d'un sous-programme en C ou en assembleur

Format

C:<Adresse>([<x1>,...<xn>])
où <Adresse> est l'adresse d'implantation d'un sous-programme en C ou en assembleur,
et <x1> à <xn> les éventuels paramètres à passer au sous-programme.

Fonction

Cette instruction permet d'activer un sous-programme écrit en langage C ou en assembleur. Pour ce faire, il suffit de préciser l'adresse d'implantation de la routine et le ou les éventuels paramètres (mots de 16 ou 32 bits) entre parenthèses. Les paramètres de 32 bits doivent être précédés d'un « L: ».

CALL

Appel d'un sous-programme en C ou en assembleur

Format

CALL <Adresse>([<x1>,...<xn>])
où <Adresse> est l'adresse d'implantation d'un sous-programme en C ou en assembleur,
et <x1> à <xn> les éventuels paramètres à passer au sous-programme.

Fonction

Cette fonction permet d'activer un sous-programme écrit en langage C ou en assembleur. Pour ce faire, il suffit de préciser l'adresse d'implantation de la routine et le ou les éventuels paramètres (mots de 16 ou 32 bits, ou chaînes de caractères) entre parenthèses. Le sous-programme doit se terminer par un RTS car l'exécution du CALL empile l'adresse de retour.

CARD

Extraction des 16 bits de poids faible

Format

Premier format :

<x>=CARD(<y>)
où <y> est l'expression entière source,
et <x> est une expression entière qui contient les 16 bits de poids faibles de <y> après l'appel.

Deuxième format :

<y>=CARD{<x>} ou CARD{<x>}=<y>
où <x> est une adresse mémoire,
et <y> est une suite de deux octets.



Fonction Dans sa première forme, la fonction CARD renvoie les seize bits de poids faibles de l'expression entière qui lui est passée.

Dans sa seconde forme, la fonction CARD stocke ou lit deux octets à partir de l'adresse mémoire spécifiée.

Exemple :

```
PRINT HEX$(CARD(&H12345678))
affiche 5678
Stockage de la valeur &HABCD à partir de l'adresse 1234 :
CARD{123456}=&HABCD
```

CASE

Voir SELECT et CONT

CFLOAT

Conversion d'un entier en un réel

Format <x> = CFLOAT (<y>)
où <y> est une expression entière,
et <x> est une variable réelle.

Fonction Cette fonction convertit une variable entière en son équivalent réel.

Exemple :

```
a%=12
b=CFLOAT (a%)
PRINT a%,b
affiche    12     12
```

CHAIN

Chargement et exécution d'un programme

Format CHAIN "<Nom>"
où <Nom> est le nom complet du programme à exécuter.

Fonction Cette instruction permet de charger en mémoire et d'exécuter un programme écrit en GFA-Basic. Le nom du programme peut (si besoin est) être exprimé de manière extensive (nom d'unité, de sous-répertoire, de fichier et d'extension).

Exemple :

```
CHAIN "B:\MONPROG.GFA"
```

CHAR Lecture/copie d'une chaîne

Format <y>=CHAR{<x>} ou CHAR{<x>}=<y>
où <x> est une adresse mémoire,
et <y> est une chaîne de caractères.

Fonction La fonction CHAR stocke ou lit une chaîne de caractères terminée par un octet nul à partir de l'adresse mémoire spécifiée.

Exemple :

Stockage de la chaîne "coucou" à partir de l'adresse 1234 :

```
CHAR{123456}="coucou"
```

CHDIR Définition du répertoire courant

Format CHDIR "<Chemin>"
où <Chemin> est le chemin du nouveau répertoire courant.

Fonction Cette fonction permet de définir le nouveau répertoire courant. Notez qu'elle n'autorise pas le changement de lecteur.

Exemple :

```
CHDIR "\MONREP"
```

Le répertoire « MONREP » devient le répertoire courant.

CHDRIVE Définition du lecteur par défaut

Format CHDRIVE <n> ou CHDRIVE <n$>
où <n> est une expression entière qui contient le numéro du lecteur,
et <n$> est une expression chaîne qui contient l'identificateur du lecteur.



Fonction Cette fonction permet de définir le lecteur par défaut. Le lecteur peut être spécifié :

— de manière numérique (0 à 16 : 0 = lecteur standard, 1 = A, 2 = B, ..., 16 = P),

— de manière alphanumérique « A: » à « P: ».

Exemple :

```
CHDRIVE (2)      → Lecteur B:
CHDRIVE "C:\"    → Lecteur C:
```

CHR$ Caractère correspondant à un code ASCII

Format

<x> = CHR$(<y>)
où <y> est un code ASCII,
et <x> est une variable chaîne qui contient le caractère correspondant au code ASCII <x>.

Fonction Cette fonction transforme le code ASCII qui lui est passé en un caractère et le renvoie à l'appelant.

Exemple :

```
PRINT CHR$(56),CHR$(65)+CHR$(66)
Affiche 8      AB
```

CINT Conversion d'un réel en un entier

Format

<y> = CINT(<x>)
où <y> est une expression entière,
et <x> est une variable réelle.

Fonction Cette fonction convertit une variable réelle en son équivalent entier.

Exemple :

```
a=12.56
b%=CINT(a)
PRINT a,b%
affiche      12.56      13
```

CIRCLE Tracé d'un cercle ou d'un arc de cercle

Format

CIRCLE <x>,<y>,<r>[,<Deb>,<Fin>]
où <x> et <y> sont les coordonnées (abscisse et ordonnée) du centre du cercle,
<r> est le rayon du cercle,
<Deb> et <Fin> sont les angles de début et de fin de tracé de l'arc de cercle. Ces deux paramètres sont optionnels.

Fonction

Cette instruction permet de tracer un cercle ou un arc de cercle vide. Pour dessiner un cercle, il suffit de passer les arguments suivants : coordonnées du centre et rayon du cercle. Pour dessiner un arc de cercle, il faut ajouter les deux paramètres suivants : angle de début et de fin de l'arc (en degrés multipliés par 10).

Exemple :

```
CIRCLE 100,100,50
CIRCLE 300,200,50,0,900
```

CLEAR Purge des variables et tableaux

Format

CLEAR

Fonction

Cette instruction purge toutes les variables et tous les tableaux définis dans le programme.

CLEARW Vidage d'une fenêtre d'écran

Format

CLEARW [#]<NF>
où <NF> est un numéro de fenêtre.

Fonction

Cette instruction efface le contenu de la fenêtre spécifiée. Pour que l'opération soit possible, il est nécessaire que la fenêtre ait été ouverte à l'aide de l'instruction OPENW.



CLIP Fenêtre d'affichage graphique pour les instructions VDI

Format

CLIP <x>,<y>,<L>,<l> [OFFSET <x0>,<y0>]
ou CLIP <x1>,<y1> TO <x2>,<y2> [OFFSET <x0>,<y0>]
ou CLIP #<F> [OFFSET <x0>,<y0>]
ou CLIP OFFSET <x0>,<y0>
ou CLIP OFF.

Dans la première forme, <x> et <y> désignent les abscisse et ordonnée du coin supérieur gauche de la fenêtre, <L> désigne la largeur de la fenêtre, et <l> sa hauteur.
Dans la seconde forme, <x1>, <y1>, <x2> et <y2> désignent les abscisse et ordonnée de deux coins opposés de la fenêtre.

Dans la troisième forme, <F> désigne un numéro de fenêtre.

Le suffixe optionnel OFFSET permet de définir l'origine des tracés graphiques aux coordonnées <x0>,<y0>.

La cinquième forme désactive la fenêtre d'affichage (tout l'écran est désormais accessible).

Fonction

Cette instruction permet de définir la zone rectangulaire dans laquelle les instructions VDI permettront d'afficher des éléments graphiques.

Exemple :

```
CLIP 0,0,100,100 OFFSET 100,100
CLIP #1 OFFSET 10,40
CLIP OFF
```

CLOSE Fermeture d'un canal

Format

CLOSE [#<n>]
où <n> est un numéro de canal affecté par l'instruction OPEN.

Fonction

Cette instruction permet de fermer un canal de données (fichier disque ou périphérique) ouvert à l'aide de l'instruction OPEN.

Exemple :

```
CLOSE #1
```

CLOSEW Fermeture d'une fenêtre d'affichage

Format CLOSEW [#]<n>
où <n> est le numéro de la fenêtre à fermer.

Fonction Cette instruction ferme la fenêtre d'affichage dont le numéro est spécifié.

Exemple :

```
CLOSEW #2
```

CLR Purge de variables

Format CLR <v1>[,...[,<vn>]...]
où les <vi> sont les noms des variables à purger.

Fonction Cette instruction purge la ou les variables dont les noms sont spécifiés lors de son exécution. Elle ne permet pas d'effacer des tableaux. Si vous désirez effacer un tableau, reportez-vous aux instructions CLEAR et ERASE.

Exemple :

```
CLR a%,v
```

CLS Effacement d'écran

Format CLS [#<n>]
où <n> est une expression entière optionnelle qui spécifie l'identificateur d'un fichier.

Fonction Cette instruction vide l'écran ou le fichier spécifié.

COLOR Couleur d'écriture

Format COLOR <Couleur>
où <Couleur> est la couleur d'encre utilisée pour l'écriture (entre 0 et 15).



Fonction Cette instruction définit la couleur d'encre à utiliser pour écrire du texte sur l'écran.

CONT Fin de test dans une structure SELECT

Format CONT

Fonction Cette instruction est utilisée pour inhiber les tests dans une structure SELECT. Pour comprendre son utilisation, exécutez les deux programmes ci-dessous.

Exemple :

```
a$="a"
SELECT a$
  CASE "a"
    PRINT "a$ vaut a"
  CASE "b"
    PRINT "a$ vaut b"
  CASE "a", "b"
    PRINT "a$ vaut a ou b"
ENDSELECT
```

Ce premier programme affiche le texte suivant :

```
a$ vaut a
```

```
a$="a"
SELECT a$
  CASE "a"
    PRINT "a$ vaut a"
    CONT
  CASE "b"
    PRINT "a$ vaut b"
  CASE "a", "b"
    PRINT "a$ vaut a ou b"
ENDSELECT
```

Ce second programme affiche seulement les textes suivants :

```
a$ vaut a
a$ vaut b
```

Le test CASE "b" est donc inhibé.

CONTRL Adresse du tableau de contrôle VDI. (Se reporter à la Partie 5, chapitre 2.)

Format

CONTRL (<Idx>)=<Valeur>
ou <Valeur>=CONTRL(<Idx>)
où <Idx> représente le paramètre à lire ou à écrire dans le tableau de contrôle,
et <Valeur> est le mot (16 bits) lu ou à écrire dans le tableau de contrôle.

Fonction

Cette fonction permet d'accéder en lecture ou en écriture au tableau de contrôle VDI. Elle est équivalente (selon qu'elle est utilisée en lecture ou en écriture) à un DPEEK ou à un DPOKE.

COS Cosinus

Format

<x>=COS(<y>)
où <y> est un angle exprimé en radians,
et <x> le cosinus de <y>.

Fonction

Cette fonction renvoie le cosinus de l'angle qui lui est passé.

Remarque :
La fonction COSQ est équivalente à la fonction COS, mais elle demande d'exprimer les angles en degrés.

Exemple :

```
PRINT COS(0.707)      → 0.7603139617445
PRINT COS(0.818)      → 0.6836821335303
```

COSQ Cosinus

Format

<x>=COSQ(<y>)
où <y> est un angle exprimé en degrés,
et <x> le cosinus de <y>.

Fonction

Cette fonction renvoie le cosinus de l'angle qui lui est passé.

Remarque :
La fonction COS est équivalente à la fonction COSQ, mais elle demande d'exprimer les angles en radians.



Exemple :

```
PRINT COSQ(45)      → 0.7071067811921
PRINT COSQ(60)      → 0.5
```

Les instructions du GFA BASIC 3.0 en détails et leur utilisation suite...

CRSCOL Abscisse du curseur

Format <x>=CRSCOL
où <x> est la colonne du curseur sur l'écran.

Fonction Cette instruction renvoie l'abscisse du curseur sur l'écran (entre 1 et 80).

Exemple :

```
? CRSCOL
x=CRSCOL
```

CRSLIN Ordonnée du curseur

Format <x>=CRSLIN
où <x> est la ligne du curseur sur l'écran.

Fonction Cette instruction renvoie l'ordonnée du curseur sur l'écran (entre 1 et 24).

Exemple :

```
? CRSCOL
x=CRSCOL
```

CVD Conversion double précision

Format <Vr>=CVD(<Vc>)
où <Vr> est une variable réelle,
et <Vc> une variable chaîne.

Fonction

Le GFA Basic stocke en mémoire les valeurs affectées aux variables d'une manière qui lui est propre. L'instruction ARRPTR permet de connaître l'adresse d'implantation des variables. En fonction de leur type, le codage est effectué sur :

— 8 octets (réels double précision GFA 3.0) ;

— 6 octets (réels simple précision GFA 3.0) ;

— 4 octets (réels simple précision GFA ou entier 32 bits) ;

— 2 octets (entier 16 bits).

L'instruction CVD convertit une chaîne de 8 octets au format GFA Basic 3.0 en un nombre réel double précision.

Exemple :

Notez la précision parfaite de cet exemple :

```
PRINT CVD(MKD$(12.34567890123))
12.34567890123
```

La boucle FOR et le dernier PRINT affichent le même résultat :

```
c=12.34567890123
FOR i=ARRPTR(c) TO ARRPTR (c)+7
  PRINT CHR$(PEEK(i)) ;
NEXT I
PRINT
PRINT MKD$(12.34567890123)
```

CVF Conversion simple précision

Format

<Vr>=CVD(<Vc>)
où <Vr> est une variable réelle,
et <Vc> une variable chaîne.

Fonction

L'instruction CVF convertit une chaîne de 6 octets au format GFA Basic 1.0 et 2.0 en un nombre réel double précision.

Exemple :

Notez la perte de précision sur cet exemple :

```
PRINT CVF(MK$(12.34567890123))
12.34567890118
```



CVI Conversion entière 2 octets

Format

<Ve> = CVD(<Vc>)
où <Ve> est une variable entière,
et <Vc> une variable chaîne.

Fonction

L'instruction CVI convertit une chaîne de 2 octets en un nombre entier codé sur 16 bits.

Exemple :

```
c& = 12
PRINT CVI (MKI$(c&))
12

c& = 100000
PRINT CVI(MKI$(c&))
provoque une erreur (Le nombre n'est pas un mot -32768 .. 32767).
```

CVL Conversion entière 4 octets

Format

<Ve> = CVL(<Vc>)
où <Ve> est une variable entière,
et <Vc> une variable chaîne.

Fonction

L'instruction CVL convertit une chaîne de 4 octets en un nombre entier codé sur 32 bits.

Exemple :

```
c% = 65000
PRINT CVL (MKL$(c%))
65000

Notez l'erreur sur le résultat dès que la valeur à convertir dépasse 65534 (2^16) :
c% = 69000
PRINT CVL(MKL$(c%))
3464
```

CVS Conversion simple précision

Format <Vr>=CVS(<Vc>)
où <Vr> est une variable réelle,
et <Vc> une variable chaîne.

Fonction L'instruction CVS convertit une chaîne de 4 octets en un nombre réel simple précision.

Exemple :

Notez la perte de précision de cet exemple :

```
PRINT CVS(MKD$(12.34567890123))
12.34567832947
```

La boucle FOR et le dernier PRINT affichent le même résultat :

```
c=12.34567890123
FOR i=ARRPTR(c) TO ARRPTR(c)+4
  PRINT CHR$(PEEK(i)) ;
NEXT I
PRINT
PRINT MKS$(c)
```

DATA Déclaration de constantes

Format DATA <ct1>[, <ct2>[,...<ctN>]...]
où les ctI sont des constantes de tous types (entier, réel, chaîne).

Fonction Cette instruction permet de définir des données constantes à l'intérieur d'un programme. Ces données peuvent être lues par l'instruction READ, et le pointeur de données (qui indique la prochaine donnée accédée par READ) peut être ajusté avec l'instruction RESTORE.

Exemple :

```
READ a,b$,c&
DATA 12.345, "coucou",56
```

DATE$ Date système

Format <vc>=DATE$
où <vc> est une variable chaîne.



Fonction Cette instruction permet de lire la date système. Reportez-vous à l'instruction SETTIME si vous désirez modifier la date système.

Exemple :

Affichage direct de la date système :

```
PRINT DATE$
29.09.1989
```

Affichage de la date système à travers une variable chaîne :

```
c$ = DATE$
? c$
29.09.1989
```

DEC Décrémentation

Format DEC <vn>
où <vn> est une variable numérique (entière ou réelle).

Fonction Cette instruction décrémente de un (c'est-à-dire diminue de un) la valeur de la variable spécifiée. Elle est bien plus rapidement exécutée que la soustraction équivalente.

Exemple :

Démonstration de la vitesse d'exécution de l'instruction DEC;

```
i& = 30000
PRINT TIME$
REPEAT
  DEC i&
UNTIL i& = 0
PRINT TIME$
PRINT
i& = 30000
PRINT TIME$
REPEAT
  i& = i& - 1
UNTIL i& = 0
PRINT TIME$
```

Exécutez le programme. Supposons que l'heure système soit égale à 11:29:10 au moment de l'exécution. Les données suivantes sont affichées sur l'écran.

```
  11:29:10
  11:29.14
  11:29:14
  11:29:22
```

Le temps d'exécution nécessaire dans le premier cas est de 4 secondes. Dans le second cas, il est de 8 secondes !

DEFAULT Voir SELECT

DEFBIT Variables booléennes non marquées par défaut

Format

DEFBIT <cd>
où <cd> est une chaîne de définition.

Fonction

Cette instruction définit par défaut que les variables qui commencent par la ou les lettre(s) spécifiée(s) est (sont) de type booléen. Le signe de déclaration des variables entières codées sur un octet est "**!**".

Exemple :

DEFBIT "a"
Les variables non marquées commençant par a sont de type booléen par défaut.
DEFBIT "a,v"
Les variables non marquées commençant par a ou par v sont de type booléen par défaut.
DEFBIT "c-f,x"
Les variables non marquées commençant par c, d, e, f ou x sont de type booléen par défaut.

DEFBYT Variables entières sur un octet non marquées par défaut

Format

DEFBYT <cd>
où <cd> est une chaîne de définition.

Fonction

Cette instruction définit par défaut que les variables qui commencent par la ou les lettre(s) spécifiée(s) est (sont) de type entier codé sur un octet. Le signe de déclaration de telles variables est "**|**".

Exemple :

DEFBYT "a"
Les variables non marquées commençant par a sont de type entier codé sur un octet par défaut.
DEFBYT "a,v"
Les variables non marquées commençant par a ou par v sont de type entier codé sur un octet par défaut.
DEFBYT "c-f,x"
Les variables non marquées commençant par c, d, e, f ou x sont de type entier codé sur un octet par défaut.



DEFFILL — Motif de remplissage

Format

DEFFILL <c>, <s>, <m>
où <c> est la couleur du motif de remplissage,
<s> est le style de remplissage :
0 : couleur du fond,
1 : couleur <c>
2 : pointillé de couleur <c>
3 : hachures de couleur <c>
4 : symbole ATARI ou motif défini par l'utilisateur
<m> est le motif de remplissage.
Les diverses combinaisons possibles des paramètres <s> et <m> sont résumés dans l'exemple.

ou DEFFILL <c>, <mb>
où <c> est la couleur du motif de remplissage,
et <mb> est une suite de 32 octets qui permettent de définir un motif de remplissage (voir exemple).

Fonction

Cette instruction permet de choisir le motif de remplissage pour les instructions graphiques de remplissage (PBOX, PCIRCLE, PELLIPSE, POLYFILL et FILL).

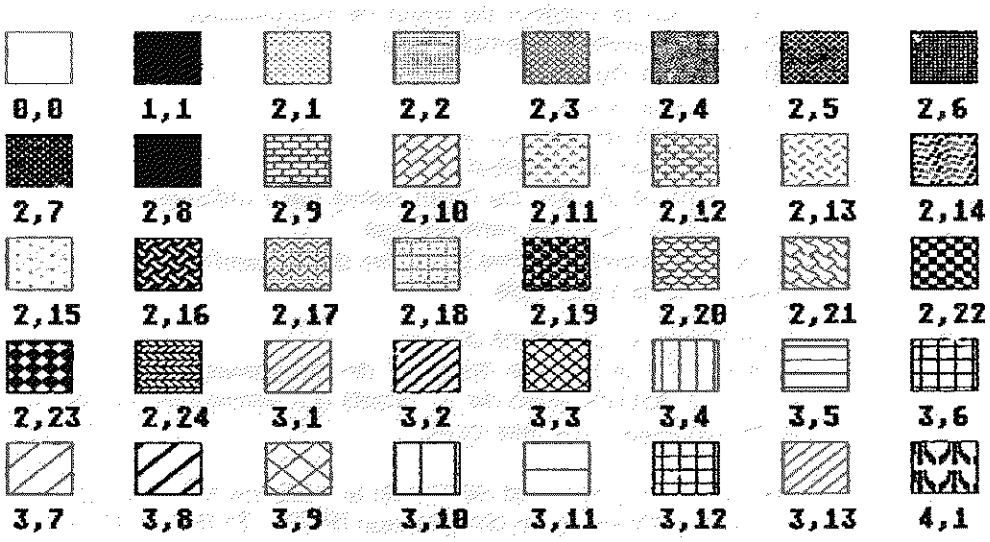
Exemple :

Impression des possibilités de l'instruction SETFILL :

```
FOR i=1 TO 5
  FOR j=1 TO 8
    READ a,b
    DEFFILL 1,a,b
    PBOX (j-1)*60, (i-1)*60, j*60-30, i*60-30
    s$=STR$(a)+","+STR$(b)
    TEXT (j-1)*60+3, i*60-10, s$
  NEXT j
NEXT i
HARDCOPY
DATA 0,0,1,1,2,1,2,2,2,3,2,4,2,5,2,6,2,7,2,8,2,9,2,10
DATA 2,11,2,12,2,13,2,14,2,15,2,16,2,17,2,18,2,19,2,20
DATA 2,21,2,22,2,23,2,24,3,1,3,2,3,3,3,4,3,5,3,6,3,7
DATA 3,8,3,9,3,10,3,11,3,12,3,13,4,1
```

L'expression de ce programme réalise l'impression suivante :

Exemple : (suite)



Exemple de définition d'un motif de remplissage :

```
FOR i=1 TO 16
  READ a%
  a$=a$+MKI$(a%)
NEXT i
DEFFILL 1,a$
PCIRCLE 120,120,100
HARDCOPY
DATA -1,-1,0,-1,0,0,0,0,-1,-1,0,-1,0,0,0,0
DATA -1,-1,0,-1,0,0,0,0,-1,-1,0,-1,0,0,0,0
```

DEFFLT Variables flottantes sur 8 octets non marquées par défaut

Format

DEFFLT <cd>
où <cd> est une chaîne de définition.

Fonction

Cette instruction définit par défaut que les variables qui commencent par la ou les lettre(s) spécifiée(s) est (sont) de type flottant codé sur 8 octets. Le signe de déclaration de telles variables est "#".



Exemple :

DEFFLT "a"
Les variables non marquées commençant par a sont de type flottant codé sur 8 octets par défaut.
DEFFLT "a,v"
Les variables non marquées commençant par a ou par v sont de type flottant codé sur 8 octets par défaut.
DEFFLT "c-f,x"
Les variables non marquées commençant par c, d, e, f ou x sont de type flottant codé sur 8 octets par défaut.

DEFFN Définition de fonctions utilisateur

Format

DEFFN <Nom>[(<v1>[,...[<vN>]...])] = <d>
où <Nom> est le nom de la fonction,
les <vI> sont les éventuelles variables attachées à la fonction,
et <d> est le calcul affecté à la fonction.

Fonction

Cette instruction permet de définir une fonction mathématique. Cette définition est exécutée une seule fois, mais elle peut être utilisée dans tout le programme grâce à l'instruction FN.

Exemple :

Recherche des minima et maxima d'une fonction mathématique :

```
REM ------------------------
REM Recherche des minima et maxima
REM d'une fonction mathématique
REM sur un intervalle donné
REM ------------------------
CLS
PRINT "Entrez l'intervalle d'étude:"
INPUT "Minimum : ";mi
INPUT "Maximum : ";ma
DEFFN a(x)=SIN(x)
'
u=-1.0E+15
v=1.0E+15
FOR x=mi TO ma STEP (ma-mi)/100
  b=FN a(x)
  IF b>u
    u=b
  ENDIF
```

Exemple : (suite)

```
  IF b<v
    v=b
  ENDIF
NEXT x
'
PRINT
PRINT "Le maximum est : ";u
PRINT "Le minimum est : ";v
END
```

DEFINT Variables entières sur 4 octets non marquées par défaut

Format

DEFINT <cd>
où <cd> est une chaîne de définition.

Fonction

Cette instruction définit par défaut que les variables qui commencent par la ou les lettre(s) spécifiée(s) est (sont) de type entier codé sur 4 octets. Le signe de déclaration de telles variables est "**%**".

Exemple :

DEFINT "a"
Les variables non marquées commençant par a sont de type entier codé sur 4 octets par défaut.
DEFINT "a,v"
Les variables non marquées commençant par a ou par v sont de type entier codé sur 4 octets par défaut.
DEFINT "c-f,x"
Les variables non marquées commençant par c, d, e, f ou x sont de type entier codé sur 4 octets par défaut.

DEFLINE Apparence des lignes

Format

DEFLINE <s>, <e>, <d>, <f>
où <s> est le style de tracé :
0 : couleur du fond
1 : ligne pleine
2 : tirets
3 : pointillés espacés



4 : points-tirets
5 : pointillés très espacés
6 : trait point point

Il est également possible de définir le style de tracé par un entier 16 bits dans lequel chaque bit à un correspond à un point allumé et chaque bit à zéro à un point éteint.

<e> est l'épaisseur des traits en pixels,

<d> et <f> sont les symboles affichés en début et en fin de ligne :

0 : coin
1 : flèche
2 : arrondi.

Fonction

Cette instruction permet de définir l'apparence des lignes tracées par les instructions graphiques LINE, BOX, RBOX, CIRCLE, ELLIPSE et POLYLINE.

Exemple :

Dans cet exemple, nous illustrons les possibilités des deux premiers paramètres de l'instruction :

```
FOR i=1 to 6
  FOR j=1 TO 9 STEP 2
    DEFLINE i,j,0
    LINE (j-1)*40, i*20, j*40, i*20
  NEXT j
NEXT i
```

DEFLIST Format des listings

Format

DEFLIST <a>
où <a> vaut 0, 1, 2 ou 3.

Fonction

Cette instruction définit le mode d'affichage des instructions et des variables dans le programme, comme le montre le tableau ci-dessous :

a	Instruction	Variable
0	INPUT	var
1	Input	Var
2	INPUT	var suivi d'un suffixe de type
3	Input	Var suivi d'un suffixe de type

DEFMARK Type de marquage de POLYMARK

Format

DEFMARK <c>, <t>, <ta>
où <c> est la couleur des marques,
<t> le type des marques :
1 :
2 : +
3 : *
4 : rectangle
5 : croix
6 : losange
et <ta> la taille des marques.

Fonction

Cette instruction permet de définir le type des marques utilisées par l'instruction POLYMARK.

Exemple :

Cet exemple marque les coordonnées 50,50 et 100,100 par deux losanges :

```
DIM abs%(1), ord%(1)
abs%(0)=50
ord%(0)=50
abs%(1)=100
ord%(1)=100
DEFMARK 1,6,2
POLYMARK 2,abs%( ),ord%( )
```

DEFMOUSE Définition du curseur souris

Format

DEFMOUSE <s>
où <s> définit le type du curseur souris :
0 : flèche
1 : double parenthèse
2 : abeille
3 : main fermée avec index déplié
4 : main ouverte
5 : réticule mince
6 : réticule épais
7 : réticule encadré
ou DEFMOUSE <m>

où <m> est un motif qui permet de définir son propre curseur.
Le motif doit avoir la structure suivante :
MKI$(abscisse),
MKI$(ordonnée),



MKI$({1, −1}) pour normal ou XOR,
MKI$(couleur masque),
MKI$(couleur curseur),
MKI$(masque),
MKI$(curseur).
Reportez-vous à l'exemple pour prendre connaissance de l'utilisation pratique de cette instruction.

Fonction

Cette instruction permet de définir le curseur de la souris. Le curseur peut être choisi parmi 8 variantes prédéfinies ou entièrement défini par l'utilisateur.

Exemple :

```
FOR i=1 TO 5
  READ a
  s$=s$+MKI$(a)
NEXT i
FOR i=1 TO 16 !Masque
  s$=s$+MKI$(63535)
NEXT i
FOR i=1 TO 16 !Curseur
  s$=s$+MKI$(&HAA)
NEXT i
DEFMOUSE s$
REPEAT !Attente de l'appui sur le bouton gauche
UNTIL MOUSEK=1
DATA 0,0,1,0,1
```

DEFNUM Précision des nombres réels

Format

DEFNUM <n>
où <n> correspond au nombre de chiffres avant et après le point décimal.

Fonction

Cette instruction fixe le nombre de chiffres significatifs des nombres réels.

Exemple :

```
a=1/6
PRINT a
DEFNUM 6
PRINT a
DEFNUM 3
PRINT a
```

Exemple : (suite)

L'exécution de ce programme provoque l'affichage des données suivantes :
0.1666666666667
0.166667
0.167

DEFSTR Variables chaînes non marquées par défaut

Format

DEFSTR <cd>
où <cd> est une chaîne de définition.

Fonction

Cette instruction définit par défaut que les variables qui commencent par la ou les lettre(s) spécifiée(s) est (sont) de type chaîne de caractères. Le signe de déclaration des variables chaînes est "**$**".

Exemple :

DEFSTR "a"
Les variables non marquées commençant par a sont de type chaîne par défaut.
DEFSTR "a,v"
Les variables non marquées commençant par a ou v sont de type chaîne par défaut.
DEFSTR "c-f,x"
Les variables non marquées commençant par c, e, d, f ou x sont de type chaîne par défaut.

DEFTEXT Paramétrage de l'instruction TEXT

Format

DEFTEXT [<c>], [<a>], [<an>], [<h>], [<f>]
où <c> est la couleur d'affichage
<a> est l'attribut d'affichage. Cet attribut est constitué d'un ou de plusieurs des éléments suivants :

0 : normal
1 : gras
2 : brillant
4 : italique
8 : souligné
16 : encadré

Par exemple <a> = 17 pour initialiser un attribut gras encadré.
<an> est l'angle d'affichage du texte en dixième de degrés.



Les valeurs possibles sont les suivantes :
0, 900, 1800 et 2700
<h> est la hauteur d'écriture,
<f> est le numéro du jeu de caractères utilisé.

Fonction Cette instruction fixe les paramètres des textes affichés en environnement graphique avec l'instruction TEXT.

Exemple :

```
DEFTEXT 1, 0, 900, 13
TETX 10,370, "Exemple d'écriture le long d'un axe vertical"
```

DEFWRD Variables entières sur 2 octets non marquées par défaut

Format DEFWRD <cd>
où <cd> est une chaîne de définition.

Fonction Cette instruction définit par défaut que les variables qui commencent par la ou les lettre(s) spécifiée(s) est (sont) de type entier codé sur deux octets. Le signe de déclaration de telles variables est "&".

Exemple :

```
DEFWRD "a"
```
Les variables non marquées commençant par a sont de type entier codé sur deux octets par défaut.
```
DEFWRD "a,v"
```
Les variables non marquées commençant par a ou par v sont de type entier codé sur deux octets par défaut.
```
DEFWRD "c-f,x"
```
Les variables non marquées commençant par c, d, e, f ou x sont de type entier codé sur deux octets par défaut.

DEG Conversion en degrés

Format <ad> = DEG(<ar>)
où <ar> est un angle en radians et <ad> l'angle en degrés correspondant.

Fonction Cette instruction convertit l'angle exprimé en radians qui lui est passé en un angle exprimé en degrés.

Exemple :

```
PRINT DEG(PI)
180
```

DELAY Suspension de l'exécution du programme

Format

DELAY <n>
où <n> est un délai d'attente en secondes.

Fonction

Cette instruction suspend l'exécution du programme pendant un délai paramétrable. Reportez-vous également à l'instruction PAUSE qui est complémentaire.

Exemple :

```
DELAY 10 !Attente pendant 10 secondes
DELAY 3  !Attente pendant  3 secondes
```

DELETE Effacement d'un élément de tableau

Format

DELETE <t(<i>)>
où <i> est la position dans le tableau où l'élément doit être effacé, et <t> est le tableau dans lequel doit se faire l'effacement.

Fonction

Cette instruction efface un élément dans un tableau et déplace les éléments suivants d'une position vers le début du tableau.

Exemple :

```
DIM t(4)
t(1)=5
t(2)=12
t(3)=6
t(4)=1
DELETE t(2)
FOR i=1 to 3
  PRINT t(i)
NEXT i
```

Exemple : (suite)

L'exécution de ce programme provoque l'affichage suivant. Le second élément du tableau a été effacé et les éléments 3 et 4 ont été décalés vers le début du tableau :

```
 5
 6
 1
```

DFREE Place libre sur une unité de disque

Format

<p> = DFREE(<n>)
où <n> est compris entre 0 et 16. 0 correspond au lecteur par défaut, 1 au lecteur A:, 2 au lecteur B:, ... 16 au lecteur P:.
<p> est la place mémoire libre sur le lecteur spécifié.

Fonction

Cette instruction renvoie la place libre en octets sur le lecteur de disque spécifié.

Exemple :

```
PRINT DFREE(3) !Place libre sur lecteur C:
3350000      → 3.35 MOctets libres
PRINT DFREE(1)
227000       → 227 KOctets libres
```

DIM Dimensionnement de tableaux

Format

DIM <t1>(<d1>,[...,<dN>])[,<t2>(<d1>,[...,<dN>]),...
[<tp>(<d1>,[...,<dN>])]...]
où les <ti> sont des noms de tableaux numériques ou alphanumériques, et les <dI> sont les dimensions des tableaux <t1>.

Fonction

Une instruction de ce type doit obligatoirement être exécutée avant de pouvoir utiliser un tableau. Son but est la réservation de la place nécessaire en mémoire.
Reportez-vous également à l'instruction OPTION BASE.

Exemple :

```
DIM t(100)
DIM a$(10,3),ze(4)
```

DIM? Nombre d'éléments d'un tableau

Format

<d>=DIM?(<t>)
où <t> est un tableau déjà dimensionné,
et <d> est la dimension du tableau <t>.

Fonction

Cette fonction renvoie la dimension du tableau dont le nom est spécifié dans l'argument. Reportez-vous également à l'instruction OPTION BASE.

Exemple :

```
DIM t(100)
PRINT DIM?(t())
OPTION BASE 1
PRINT DIM?(t())
101
101
```

Un tableau dimensionné à 100 comporte toujours 101 éléments. L'instruction OPTION BASE n'agit pas sur le résultat de DIM?

DIR Répertoire d'un lecteur de disque

Format

DIR <r> [TO <f>]
où <r> est le nom du disque et éventuellement le chemin dans le disque pour lequel le répertoire doit être établi,
et <f> est un nom de fichier dans lequel le résultat texte de l'instruction DIR peut être stocké.

Fonction

Cette instruction affiche ou sauvegarde dans un fichier le contenu du répertoire spécifié. Reportez-vous également à l'instruction FILES qui est complémentaire.

Exemple :

```
DIR
DIR "a:\super3\*.*"
```



DIR$ Chemin courant sur une unité de disque

Format

{<c>=,PRINT} DIR$(<n>)
où <c> est une variable chaîne,
et <n> identifie un lecteur de disque. <n> doit être compris entre 0 et 16. 0 correspond au lecteur par défaut, 1 au lecteur A:, 2 au lecteur B:, ... 16 au lecteur P:.

Fonction

Cette fonction renvoie le chemin courant sur le lecteur spécifié au moment de l'appel.

Exemple :

```
PRINT DIR$(3)     → chemin courant sur le lecteur C:
c$=DIR$ (0)       → chemin courant sur le lecteur courant
```

DIV Division

Format

DIV(<x>,<y>)
où <x> est une variable numérique,
et <y> est une variable ou quantité numérique.

Fonction

Cette instruction divise le premier argument par le second et stocke le résultat dans le premier argument qui doit être une variable numérique.

Exemple :

```
x=5
y=2
DIV x,y
PRINT x
```

L'exécution de ce programme place la valeur 2.5 dans la variable x et l'affiche sur l'écran. Comme le montre le petit programme ci-dessous, l'exécution de l'instruction DIV est plus rapide que celle de la division classique (14 secondes contre 16 pour effectuer 100 000 divisions)

```
PRINT TIME$
FOR i=1 TO 100000
  DIV x,y
NEXT i
PRINT TIME$
FOR i=100000
  x=x/y
NEXT i
PRINT TIME$.
```

DO Boucle sans fin

Format

DO
<une ou plusieurs instructions>
LOOP.

Fonction

Cette instruction marque le début d'une boucle sans fin. Pour mettre fin prématurément à la boucle, utilisez une instruction EXIT IF ou GOTO.

Exemple :

```
DO
  PRINT "a";
  IF MOUSEK=2
    GOTO fin
  ENDIF
LOOP
fin:
END
```

Ce programme affiche des "a" tant que le bouton droit de la souris n'est pas pressé.

DO UNTIL Boucle DO conditionnelle

Format

DO
<une ou plusieurs instructions>
UNTIL <condition>
où <condition> est une condition logique.

Fonction

Cette instruction marque le début d'une boucle DO qui se poursuit jusqu'à ce qu'une condition logique soit vérifiée.

Remarque :

Contrairement à l'instruction REPEAT, la boucle est au minimum exécutée une fois.

Exemple :

```
DO
  PRINT "a";
LOOP UNTIL MOUSEK=2
```

Ce programme est équivalent au précédent. Il s'exécute cependant nettement plus vite.



DO WHILE Boucle DO conditionnelle

Format

DO
<une ou plusieurs instructions>
WHILE <condition>
où <condition> est une condition logique.

Fonction

Cette instruction marque le début d'une boucle DO qui se poursuit tant qu'une condition logique est vérifiée.

Remarque :

Contrairement à l'instruction REPEAT, la boucle est au minimum exécutée une fois.

Exemple :

```
DO
  PRINT "a";
LOOP WHILE MOUSEK<>2
```

Ce programme est équivalent au précédent.

DOUBLE Lecture/écriture d'une variable double précision

Format

- Premier format :

<x>=DOUBLE{<y>}
où <x> est la variable réelle double précision but,
et <y> est l'adresse de la variable <x>.

- Second format :

DOUBLE{<y>}=<x>
où <x> est la quantité réelle double précision source,
et <y> est l'adresse de stockage de <x>.

Fonction

Cette instruction permet de lire (première forme) ou d'écrire (seconde forme) une variable réelle double précision (codée sur 8 octets).

Exemple :

```
a$=""                    ""
DOUBLE{V:a$}=1234.56789012 !Sauvegarde
c=DOUBLE{V:a$} !Lecture
```

DOWNTO Voir instruction FOR

DPEEK Double PEEK

Format

<x> = DPEEK(<y>)
où <y> est une adresse en mémoire,
et <x> contient le mot (2 octets) qui se trouve en <x>.

Fonction

Cette instruction renvoie le mot dont l'adresse est spécifiée. L'adresse doit être paire, sans quoi, une erreur est générée.

Exemple :

```
POKE 8000,&H12
POKE 8001,&H34
PRINT HEX$(DPEEK(8000))
```

L'instruction PRINT ci-dessus affiche 1234, c'est-à-dire le mot qui se trouve en 8000.

DPOKE Double POKE

Format

DPOKE <x>,<y>
où <x> est une adresse en mémoire,
et <y> contient le mot (2 octets) à stocker en <x>.

Fonction

Cette instruction permet de stocker un mot (deux octets) en mémoire. L'adresse de stockage doit être paire, sans quoi, une erreur est générée.

Exemple :

```
DPOKE 8000,&HABCD
PRINT HEX$(DPEEK(8000))
```

L'instruction PRINT ci-dessus affiche ABCD, c'est-à-dire le mot qui a été stocké en 8000 par l'instruction DPOKE précédente.

DRAW Tracé d'un point ou d'une ligne

Format

- Première forme :

DRAW [TO] <x1>,<y1>
où <x1> est l'abscisse du point terminal,
et <y1> l'ordonnée du point terminal.



- Seconde forme :

DRAW [<x1>,<y1>] [TO <x2>,<y2>] [TO<x3>,<y3>]...
[TO <xN>,<yN>]
où les <xi> et les <yi> sont les abscisses et ordonnées des points à relier.

Fonction

Cette instruction permet, dans sa première forme, de relier le point courant au point dont les coordonnées sont spécifiées. Dans sa seconde forme, elle permet de relier deux ou plusieurs points par des segments de droite.

Exemple :

```
DRAW 100,100 TO 100,200
DRAW TO 200,200
```

DUMP Valeurs des variables du programme ou liste des étiquettes

Format

DUMP [<a> [TO <b>]]
où <a> spécifie sur quoi porte le DUMP:
<a> non spécifié : liste des valeurs de toutes les variables
<a> = lettre : liste des valeurs des variables qui commencent par cette lettre
<a> = ":" : liste des étiquettes du programme,
<a> = "@" : liste des noms de procédures et fonctions du programme
et <b> est un nom de fichier dans lequel stocker le résultat.

Fonction

Cette instruction affiche en fonction du premier paramètre spécifié :
— le contenu des variables du programme (toutes ou seulement celles spécifiées),
— le nom des étiquettes du programme ;
— le nom des procédures et fonctions du programme.

Exemple :

```
DUMP ":" TO "STATUS"
DUMP "@"
DUMP
```

EDIT Retour à l'éditeur

Format EDIT

Fonction Cette instruction met fin au programme et redonne le contrôle à l'éditeur.

ELSE
ELSEIF Voir instruction IF

END Fin du programme

Format END

Fonction Cette instruction met fin au programme et fait apparaître une boîte d'alerte sur l'écran qui contient le message suivant : ''Fin du programme''. L'utilisateur doit alors cliquer sur le seul bouton de la boîte (Return), ce qui provoque le retour à l'éditeur.

ENDFUNC Voir instruction FUNCTION

ENDIF Voir instruction IF

ENDSELECT Voir instruction SELECT

EOF Test de fin de fichier

Format <Instruction de test> = EOF(#<M>)
où <Instruction de test> est une des instructions suivantes : IF, WHILE, DO ou REPEAT
et <M> est un numéro de fichier affecté lors de son ouverture avec l'instruction OPEN.

Fonction Cette fonction renvoie :
— la valeur −1 si la fin du fichier spécifié a été atteinte,
— la valeur 0 dans le cas contraire.



Exemple :

Ouverture et affichage des données alphanumériques contenues dans le fichier "monfich" :

```
OPEN "I",#1,"monfich"
DO WHILE NOT EOF (#1)
  INPUT #1,a$
  PRINT a$
LOOP
CLOSE #1
```

EQV Equivalence logique

Format

<x> = <y> EQV <z>
où <y> et <z> sont des données numériques,
et <x> est le résultat de l'équivalence logique de ces deux données.

Fonction

Cette fonction renvoie l'équivalence logique des deux arguments numériques qui lui sont passés en argument. Les calculs sont faits au niveau du bit et respectent la table de vérité ci-dessous :

y \ z	0	1
0	1	0
1	0	1

Exemple :

```
PRINT -1 EQV 0
0
PRINT - EQV -1
-1
```

ERASE Effacement d'un ou de plusieurs tableaux

Format

ERASE <t1()>[,<t2()>,...[<tN()>]...]
où les <tI> sont les noms des tableaux à effacer.

Fonction

Cette instruction libère la place occupée par un ou plusieurs tableaux en les effaçant de la mémoire.

Exemple :

```
ERASE t$(),x()
```

ERR Numéro de la dernière erreur

Format
<x>=ERR
où <x> est une variable numérique.

Fonction Cette instruction permet de connaître le numéro de la dernière erreur.

Exemple :

```
ON ERROR GOSUB ter
ERROR 3
'
PROCEDURE ter
  PRINT "Erreur ";ERR;" : ";ERR$(ERR)
RETURN
```

ERR$ Dernier message d'erreur

Format
<c>=ERR$(<e>)
où <e> est un numéro d'erreur,
et <c> une variable chaîne.

Fonction Cette instruction renvoie le texte affecté au numéro d'erreur qui lui est passé en argument.

Exemple :

Voir instruction ERR.



ERROR Génération logicielle d'une erreur

Format

ERROR <n>
où <n> est le numéro de l'erreur à déclencher.

Fonction

Cette instruction déclenche l'erreur spécifiée. Elle est surtout utilisée lors de la mise au point de routines de traitement d'erreurs.

Reportez-vous à l'annexe 1 pour prendre connaissance de la liste des erreurs.

Exemple :

```
ON ERROR GOSUB erreur
ERROR 10
END
'
PROCEDURE erreur
  PRINT "L'erreur ";ERR;" a été interceptée."
  RESUME NEXT
RETURN
```

EVEN Test de parité

Format

<x> = EVEN(<y>)
où <y> est une expression numérique,
et <x> une variable booléenne.

Fonction

Cette fonction renvoie la valeur :
- TRUE (−1) si l'expression numérique passée est paire,
- FALSE (0) si l'expression numérique passée est impaire.

Exemple :

```
PRINT EVEN(-1), EVENT(12),EVEN(3)
```

Cette instruction PRINT provoque l'affichage des trois nombres suivants :

```
 0    -1    0
```

EVERY — Activation d'une procédure à intervalle régulier

Format

EVERY <délai> GOSUB <proc>
où <délai> est le délai d'activation de <proc> est 1/200 seconde, et <proc> la procédure à activer.

Fonction

Cette instruction active la procédure <proc> tous les <délai>*1/200 seconde. La précision est de 1/50 seconde.

Exemple :

```
EVERY 400 GOSUB heure
CLS
REPEAT
  t$=INKEY$
UNTIL t$<>" "
'
PROCEDURE HEURE
  LOCATE 1,1
  PRINT TIME$
RETURN
```

Ces quelques instructions affichent l'heure système en haut et à gauche de l'écran. Notez l'activation de la procédure heure toutes les deux secondes. En effet, la précision de l'horloge système étant de deux secondes, il est inutile d'activer la procédure heure plus souvent.

EVERY CONT — Rétablissement de l'appel à la procédure

Format

EVERY CONT.

Fonction

Cette instruction réactive l'appel à la procédure spécifiée dans une instruction EVERY précédemment exécutée. Elle n'a de sens que si une instruction EVERY STOP a été exécutée.

EVERY STOP — Désactivation de l'appel à la procédure

Format

EVERY STOP.

Fonction

Cette instruction désactive l'appel à la procédure spécifiée dans une instruction EVERY précédemment exécutée. Son effet peut être annulé en utilisant l'instruction EVERY CONT.

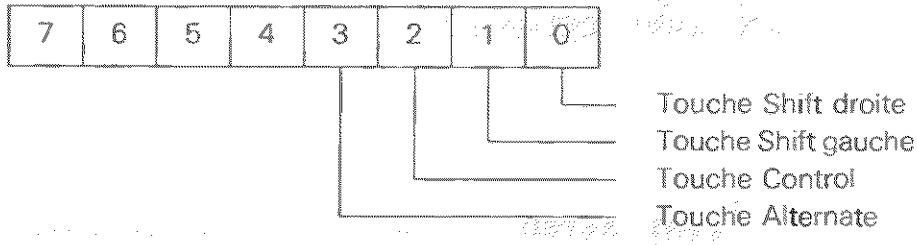


EVNT_BUTTON — Attente d'un événement sur les boutons de la souris

Format

<x> = <EVENT_BUTTON(<c>, <m>, <e>[, <sx>, <sy>, <b>, <cs>])

où <x> est le nombre d'activation des boutons,
<c> est le nombre de clics nécessaire,
<m> est le masque demandé (bit 0 pour le bouton de gauche, bit 1 pour le bouton de droite),
<e> est l'état demandé pour déclencher l'événement (bit 0 pour le bouton de gauche, bit 1 pour le bouton de droite),
<sx> est l'abscisse du curseur souris lors du déclenchement de l'événement,
<sy> est l'ordonnée du curseur souris lors du déclenchement de l'événement,
<b> est l'état des boutons de la souris (bit 0 pour le bouton de gauche, bit 1 pour le bouton de droite),
<cs> est l'état des touches spéciales du clavier lors du déclenchement de l'événement :



Fonction

Cette instruction attend un événement particulier sur les boutons de la souris et rend le contrôle au programme lorsqu'il survient.

Exemple :

```
REPEAT
  x% = EVNT_BUTTON(2,1,1,sx%,sy%,b%,cs%)
  IF x% = 1
    PRINT "Appui simple"
  ELSE
    PRINT "Appui double"
  ENDIF
UNTIL cs% = 8
PRINT "Souris en ";sx%;",";sy%
```

Affichage du message "Appui simple" à chaque simple clic sur le bouton de gauche de la souris et affichage du message "Appui double" à chaque double clic sur le bouton gauche de la souris. La boucle REPEAT UNTIL prend fin lorsque le bouton de gauche est pressé alors que la touche Alternate du clavier est enfoncée. Les coordonnées du curseur souris sont alors affichées.

EVNT_DCLICK Initialisation de la vitesse des doubles clics

Format

<x> = EVENT_DCLICK(<v>,<le>)
où <x> est l'ancienne vitesse (0 à 4),
<v> est la nouvelle vitesse (0 à 4),
et <le> détermine l'utilisation de EVENT_DCLICK :
<le> = 0 pour une lecture (<v> n'a alors aucune signification),
<le> = 1 pour une écriture.

Fonction

Cette instruction lit ou initialise (selon la valeur de <le>) la vitesse des doubles clics de la souris.

Exemple :

```
Lecture de l'ancienne vitesse de double click
    PRINT EVNT_DCLICK (dum%,0)
    0
Initialisation de la vitesse de double click
    x% = EVNT_DCLICK(2,1)
```

EVNT_KEYBD Attente d'un événement sur le clavier

Format

<x> = EVNT_KEYBD()
où <x> est le code clavier retourné par la fonction.

Fonction

Cette fonction se met en attente d'une frappe clavier et renvoie :

— le code de la touche tapée sur l'octet de poids faible,

— le code clavier sur l'octet de poids fort.

Exemple :

```
x% = EVNT_KEYBD()
PRINT "Code touche";HEX$(INT(x%/256))
PRINT "Code clavier";HEX$(x% - 256*INT(x%/256))
```



EVNT_MESSAG Attente d'un message dans le buffer d'événement

Format

<x>=EVNT_MESSAG(<ad>)
où <x> est toujours égal à 1,
et <ad> est l'adresse du buffer de lecture (16 octets).

Fonction

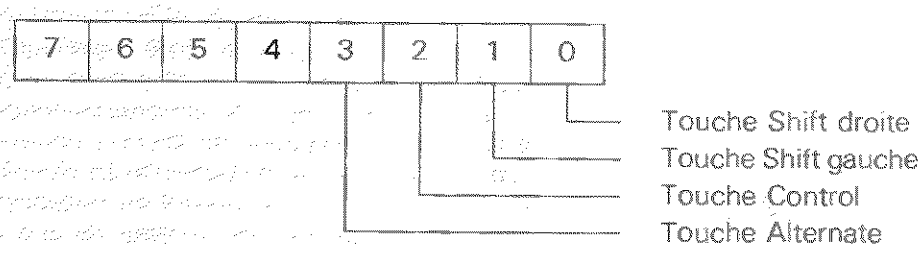
Cette instruction attend l'arrivée d'un message dans le buffer d'événement. <ad> est l'adresse de ce buffer. Si <ad>=0, le buffer MENU(1) à MENU(8) est utilisé pour recevoir le message.

EVNT_MOUSE Attend que le curseur de la souris sorte ou pénètre dans une zone déterminée de l'écran

Format

<x>=EVNT_MOUSE(<f>, <sx>, <sy>, <sl>, <sh>, <cx>, <cy>, <b>, <cs>)
où <x> est toujours égal à un,
<f> vaut :
- 0 pour attendre l'entrée dans la zone,
- 1 pour attendre la sortie de la zone.

<sx> est l'abscisse du côté supérieur gauche du rectangle,
<sy> est l'ordonnée du côté supérieur gauche du rectangle,
<sl> est la largeur du rectangle,
<sh> est la hauteur du rectangle,
<cx> est l'abscisse du curseur de la souris,
<cy> est l'ordonnée du curseur de la souris,
<b> est l'état des boutons de la souris (bit 0 pour le bouton de gauche, bit 1 pour le bouton de droite),
<cs> est l'état des touches spéciales du clavier lors du déclenchement de l'événement :



Fonction

Cette instruction attend l'entrée ou la sortie du curseur souris dans une zone rectangulaire de l'écran.

Exemple :

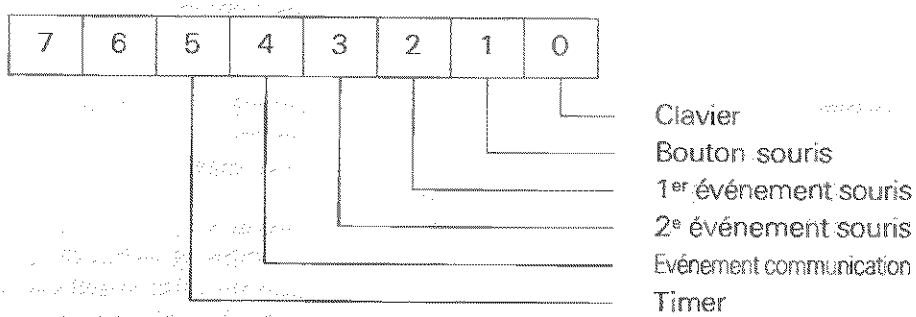
```
BOX 100, 100, 100, 100
x%=EVNT_MOUSE (0, 100, 100, 100, 100, cx%, cy%, b%, cs%)
PRINT "Souris en ";cx%;",";cy%
```

Tracé de la zone rectangulaire de test, attente de la pénétration du curseur souris dans une zone rectangulaire et affichage des coordonnées du curseur souris lors de la pénétration dans la zone.

EVNT_MULTI Attente de divers événements

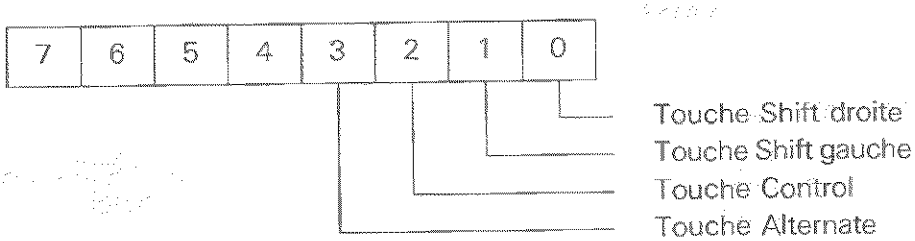
Format

<x>=EVENT_MULTI(<f>,<c>,<m>,<e>,<e1x>,<e1y>,<e1l>,<e1h>,<e2x>,<e2y>,<e2l>,<e2h>,<buf>,<cpt>,<sx>,<sy>,<b>,<cs>,<casc>,<nc>)
où <x> est le code de l'événement qui s'est produit,
<f> spécifie l'événement à attendre selon le codage suivant :



<m> est le masque demandé (bit 0 pour le bouton de gauche, bit 1 pour le bouton de droite),
<e> est l'état demandé pour déclencher l'événement (bit 0 pour le bouton de gauche, bit 1 pour le bouton de droite),
<e1x> est l'abscisse du côté supérieur gauche du premier rectangle,
<e1y> est l'ordonnée du côté supérieur gauche du premier rectangle,
<e1l> est la largeur du premier rectangle,
<e1h> est la hauteur du premier rectangle,
<e2x> est l'abscisse du côté supérieur gauche du second rectangle,
<e2y> est l'ordonnée du côté supérieur gauche du second rectangle,
<e2l> est la largeur du second rectangle,
<e2h> est la hauteur du second rectangle,
<buf> est l'adresse du buffer de 16 octets destiné à recevoir le message,
<cpt> est le délai d'attente en millisecondes,
<sx> est l'abscisse du curseur de la souris,
<sy> est l'ordonnée du curseur de la souris,
<b> est l'état des boutons de la souris (bit 0 pour le bouton de gauche, bit 1 pour le bouton de droite),
<cs> est l'état des touches spéciales du clavier lors du déclenchement de l'événement :

▶▶▶▶▶



<casc> est le code ASCII de la touche du clavier actionnée
<nc> est le nombre d'activation des boutons de la souris.

Fonction Cette instruction attend l'arrivée d'un ou de deux événements en fonction des paramètres qui lui sont passé. Elle est relativement complexe à utiliser, et nous lui préférerons les autres instructions EVNT.

EVNT_TIMER Attente d'un délai paramétrable

Format
<x>=EVNT_TIMER(<délai>)
où <x> vaut toujours 1,
et <délai> est le délai d'attente en millisecondes.

Fonction Cette instruction attend pendant la durée spécifiée dans l'argument qui lui est passé.

Exemple :

```
x%=EVNT_TIMER(1000)      'Attente d'une seconde
```

EXEC Exécution d'un programme sur disque et retour au programme courant

Format
EXEC <ma>, <nom>, <li>, <env>
ou <x>=EXEC (<ma>, <nom>, <li>, <env>)

où <x> est le code renvoyé par le programme,
<ma> est le mode d'appel du programme :
- <ma>=0 pour charger et lancer le programme,
- <ma>=3 pour charger le programme,

<nom> est le nom du programme à exécuter,
 est la ligne d'instructions de la basepage,
<env> est la chaîne d'environnement.

Fonction Cette instruction permet d'exécuter des overlays tout en restant dans le programme GFA Basic courant.

EXIST Test de l'existence d'un fichier

Format

<x> = EXIST (<nom>)
où <x> est une variable booléenne qui contient :
- 0 si le fichier <nom> n'existe pas,
- 1 si le fichier <nom> existe.

Fonction

Cette fonction détermine l'existence ou la non existence d'un fichier disque.

Exemple :

Saisie d'un nom de fichier jusqu'à ce qu'il soit trouvé sur la disquette courante :

```
REPEAT
  INPUT "Entrez le nom du fichier à examiner : ";n$
UNTIL EXIST(n$)
```

EXIT IF Annulation conditionnelle d'une boucle

Format

EXIT IF <condition>
où <condition> est une condition booléenne d'annulation de la boucle.

Fonction

Cette instruction annule la boucle dans laquelle elle se trouve si la condition booléenne spécifiée est vérifiée.

Exemple :

Attente de l'appui sur le bouton droit de la souris :

```
PRINT "Appuyez sur le bouton droit de la souris"
DO
  EXIT IF MOUSEK = 2
LOOP
PRINT "Le bouton droit de la souris a été actionné."
```

EXP Exponentielle

Format

<x> = EXP(<y>)
où <y> est l'exposant,
et <x> est l'exponentielle de <y>.



Fonction Cette fonction renvoie l'exponentielle du nombre passé en entrée, c'est-à-dire l'élévation à la puissance de l'argument du nombre d'Euler (e = 2.718281828459).

Exemple :

```
PRINT EXP (1)
2.718281828459
```

FALSE Valeur booléenne FAUX

Format <x> = FALSE
où <x> est une variable booléenne.

Fonction Cette fonction renvoie la valeur booléenne FAUX, c'est-à-dire zéro.

Exemple :

```
PRINT FALSE
0
```

FATAL Adresse de l'erreur non identifiée

Format <x> = FATAL
où <x> est une variable booléenne.

Fonction Cette fonction renvoie la valeur booléenne TRUE si l'erreur qui s'est produite n'est pas identifiable par l'interpréteur GFA. Elle renvoie la valeur FALSE dans le cas contraire.

Exemple :

```
ON ERROR GOSUB erreur
INPUT "Entrez un numéro d'erreur : ";e%
ERROR e%
END
'
PROCEDURE erreur
  IF FATAL
    PRINT "L'erreur ";ERR;" n'est pas identifiée."
  ELSE
    PRINT "L'erreur ";ERR;" n'est pas fatale."
  ENDIF
RESUME NEXT
RETURN
```

FGETDTA Adresse de la DTA

Format <x>=FGETDTA( ).

Fonction Cette fonction renvoie l'adresse de l'aire de transfert disque DTA. L'adresse de base de la DTA est BASEPAGE+128.

Exemple :

```
PRINT FGETDTA( )
82954
```

FIELD AS Définition des champs d'un fichier à accès direct

Format FIELD #<n>, <l1> AS <ch1>[, <l2> AS <ch2>[...]...]
où <n> est le numéro du canal (entre 0 et 99),
et les <li> sont les longueurs des champs <chi>.

Fonction Cette instruction définit la structure des enregistrements d'un fichier à accès direct.

Exemple :

```
FIELD #1,4 AS a$, 2 AS b$, 6 AS c$
```



FIELD AT Définition des champs numériques d'un fichier à accès direct

Format

FIELD #<n>, <l1> AT (<ch1>) [, <l2> AT (<ch2>) [...]...]
où <n> est le numéro du canal (entre 0 et 99),
et les <li> sont les longueurs des champs <chi>.

Fonction

Cette instruction définit les champs numériques d'un fichier à accès direct. Grâce à elle, des champs numériques peuvent être stockés dans un fichier à accès direct sans qu'il soit nécessaire d'en faire la conversion chaîne.

Exemple :

```
FIELD #1,2 AT (*a&),8 AT (*b#)
```

FILES Sortie d'un répertoire

Format

FILES <chemin> [TO <nom>]
où <chemin> est le chemin d'accès au répertoire,
et <nom> est le nom du périphérique de sortie.

Fonction

Cette instruction affiche, imprime ou sauvegarde dans un fichier le contenu du répertoire spécifié.

Exemple :

Répertoire du disque a :

```
FILES "a:"
```

```
I1.PC2          6564  02:39:54  24.04.1986
I2.PC2         11780  02:40:00  24.04.1986
I3.PC2         10222  02:40:08  24.04.1986
I4.PC2          8346  02:40:14  24.04.1986
LECT1.DOC       4128  02:40:20  24.04.1986
OB1.DOC        56696  02:40:26  24.04.1986
OB2.DOC        21799  08:17:44  24.04.1986
OB3.DOC        16320  00:02:42  24.04.1986
OBA1.DOC        7527  00:02:50  24.04.1986
OBA3.DOC        9865  00:02:56  24.04.1986
OBSOM.DOC       1064  04:41:00  24.04.1986
C16.DOC        14873  11:14:20  24.04.1986
OBPLUS1.DOC    15639  07:02:50  24.04.1986
P1.DOC          1308  07:03:00  24.04.1986
WOB1.BAS        1745  00:09:44  24.04.1986
Z1.DOC         43155  03:29:26  24.04.1986
C23.DOC         4898  03:29:46  24.04.1986
D.GFA            366  10:49:52  16.11.1989
```

FILESELECT Affichage d'un sélecteur d'objet

Format

FILESELECT <ch>, <def>, <n>
où <ch> est le chemin d'accès des objets à afficher,
<def> est la sélection par défaut dans le sélecteur,
et <n> est une variable qui contient le choix de l'utilisateur lorsque celui-ci appuie sur le bouton confirmer.

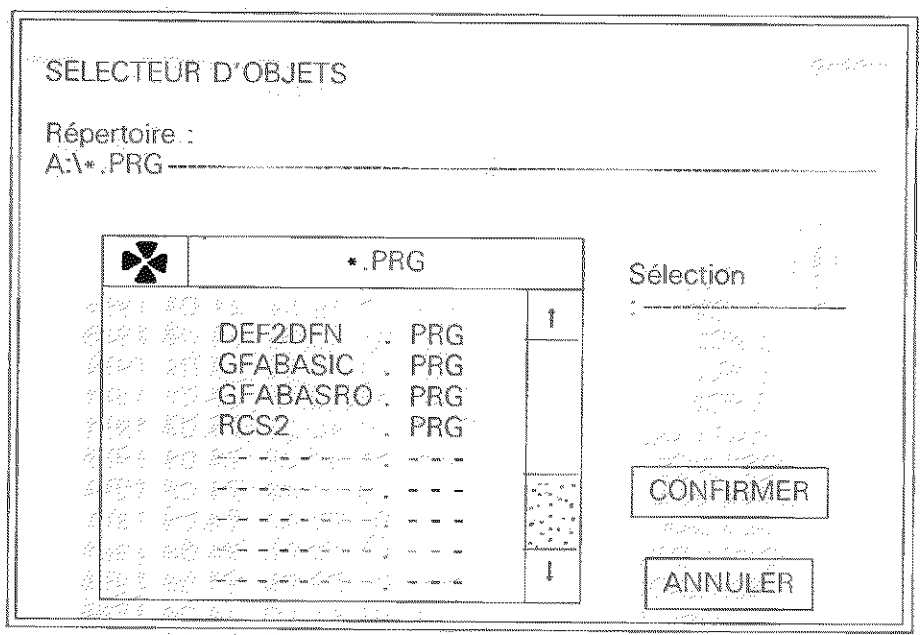
Fonction

Cette instruction affiche un sélecteur d'objets (entendez par là une liste de fichiers répondant à un certain critère) et permet de choisir un objet parmi la liste. L'objet choisi est renvoyé par une variable.

Exemple :

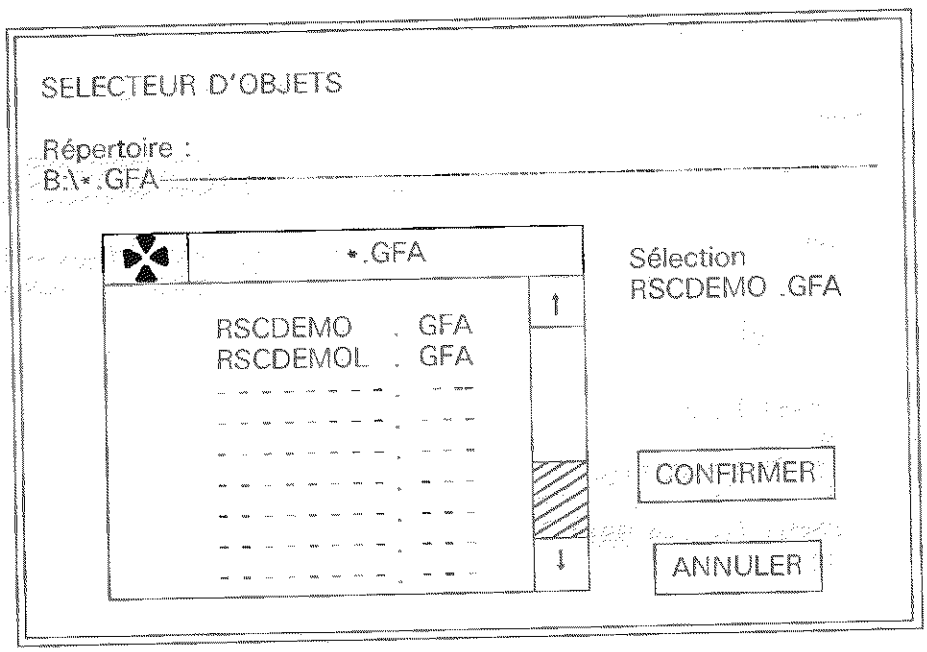
Affichage de tous les fichiers du disque A: d'extension .PRG, aucune présélection et saisie du choix de l'utilisateur dans la variable sel$:

```
FILESELECT "a:\*.prg","","",sel$
```



Affichage de tous les fichiers du disque B: d'extension .GFA, présélection sur le fichier "RSCDEMO.GFA" et saisie du choix de l'utilisateur dans la variable sel$:

```
FILESELECT "b:\*.gfa ","rscdemo.gfa",sel$
```



FILL Remplissage d'une surface

Format

FILL <x>, <y>[, <c>]
où <x> est l'abscisse d'un point interne à 1 surface à remplir,
<y> est l'ordonnée d'un point interne à la surface à remplir,
et <c> est la couleur d'arrêt de remplissage.

Fonction

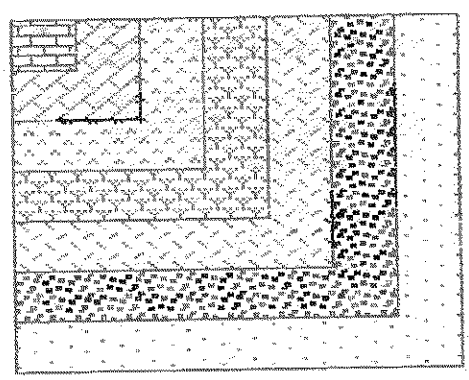
Cette instruction remplit une surface délimitée par un contour fermé en arrêtant le remplissage sur la couleur spécifiée ou sur toute couleur différente du point de départ si aucune couleur n'est spécifiée.

Le motif de remplissage doit être déterminé au préalable à l'aide de l'instruction DEFFILL.

Exemple :

Tracé de carrés ayant un côté en commun, de taille de plus en plus grande, et remplissage de ces carrés avec l'instruction FILL :

```
FOR i%=1 TO 15
  DEFFILL 1, 2, i%
  BOX 90, 90, 90+i%*20, 90+i%*20
  FILL 85+i%*20,100
NEXT i%
```



FIX Extraction de la partie entière

Format

<x>=FIX(<y>)
où <y> est une expression numérique,
et <x> est la partie entière de l'expression numérique <y>.

Fonction

Cette fonction renvoie la partie entière de l'expression numérique qui lui est passée. Elle est équivalente à l'instruction TRUNC.

Exemple :

```
PRINT FIX(PI)
3
PRINT FIX (PI*15)
47
PRINT FIX (34.998)
34
```

FLOAT Lecture/écriture d'une variable en virgule flottante

Format

<x>=FLOAT{<ad>}
ou FLOAT {<ad>}=<y>

ou <ad> est une adresse mémoire,
<x> est la valeur flottante lue en <ad>,
et <y> est la valeur flottante à stocker en <ad>.

Fonction

Cette instruction permet de lire ou d'écrire des nombres flottants codés sur 8 octets en format GFA Basic 3.0.

Exemple :

Stockage d'une valeur flottante en mémoire :

```
FLOAT {&H1234}=123.456789
```

Lecture d'une valeur flottante en mémoire :

```
PRINT FLOAT{&H1234}
123.456789
```

FN Appel d'une fonction utilisateur

Format

FN <fonction>[(<p1>,...<pn>)]
où <fonction> est un nom de fonction utilisateur,
et les <pi> sont les paramètres passés à la fonction.

Fonction

Cette instruction permet d'activer une fonction utilisateur précédemment définie à l'aide de l'instruction DEFFN.

Exemple :

Reportez-vous à l'instruction DEFFN.

FOR Boucle itérative

Format

FOR <index> = <min> TO <max> [STEP <pas>]
... Instruction (s) ...
NEXT <index>

où <index> est une variable qui est incrémentée de un (ou de <pas> si précisé) à chaque parcours de la boucle,
<min> est la valeur minimale de <index>,
<max> est la valeur maximale de <index>,
et <pas> est le pas de progression de <index>.

Fonction

Cette instruction permet d'exécuter un nombre de fois connu à l'avance une ou plusieurs instruction(s).

Exemple :

Affichage des dix premiers entiers élevés au carré :

```
FOR i% = 1 to 10
    PRINT i%;"^2 =";i%*i%,
NEXT i%
```

```
1^2 = 1      2^2 = 4      3^2 = 9      4^2 = 16     5^2 = 25
6^2 = 36     7^2 = 49     8^2 = 64     9^2 = 81     10^2 = 100
```

FORM INPUT Saisie de chaînes formattées

Format

FORM INPUT <l>,<c>
ou FORM INPUT <l> AS <c>
où <l> est la longueur maximale de la chaîne saisie,
et <c> est une variable chaîne dans laquelle la valeur saisie sera stockée.

Fonction

Cette instruction est utilisée pour saisir une chaîne au clavier dont la longueur maximale est fixée à l'avance. La valeur saisie est stockée dans une variable chaîne. La seconde forme de l'instruction affiche la valeur actuelle de la variable chaîne, permettant ainsi de l'éditer.

Exemple :

```
PRINT "Entrez 10 caractères au maximum :" ;
FORM INPUT 10,a$
PRINT "Corrigez si nécessaire la chaîne
entrée : " ;
FORM INPUT 10 AS a$
```

FORM_ALERT Affichage d'une boîte d'alerte

Format

FORM ALERT (<bouton>,<message>)
où <bouton> est le bouton par défaut :
0 = aucun,
1 = premier bouton,
2 = second bouton,
3 = troisième bouton,
et <message> est une chaîne dont le format est le suivant :
[<i>] [<t1> <t2> <t3> <t4> <t5>] [<b1> <b2> <b3>]
où <i> représente l'icône employé, selon les conventions suivantes :
0 : Aucun icône,
1 : Point d'exclamation,
2 : Point d'interrogation,
3 : Panneau stop,
les <ti> sont des lignes de texte (<t2> à <t5> sont optionnelles),
les <bi> sont des boutons (<t2> et <t3> sont optionnels)

Fonction

Cette instruction affiche une boîte d'alerte et renvoie le bouton poussé par l'utilisateur.

Exemple :

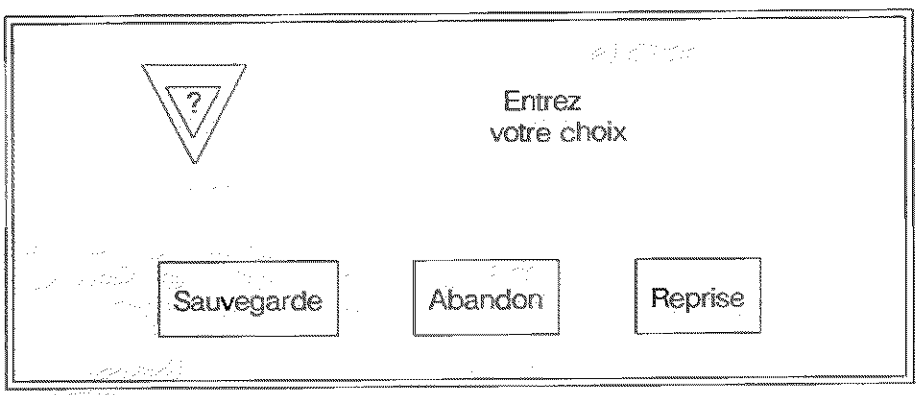
Affichage d'une boîte d'alerte en point d'interrogation spécifiant "Entrez votre choix" et proposant les trois possibilités suivantes :

— Sauvegarde,

— Abandon,

— Reprise.

```
a$="[2] [Entrez votre choix] [Sauvegarde Abandon Reprise] "
x%=FORM_ALERT (1,a$)
PRINT x%
```



FRAC Partie décimale d'un réel

Format

<résultat> = FRAC(<nombre>)
où <nombre> est un nombre réel,
et <résultat> est la partie décimale de <nombre>.

Fonction

Cette fonction extrait la partie décimale d'un nombre réel.

Exemple :

```
a=12.132
b=-65.004
PRINT FRAC(a)
PRINT FRAC(b)

0.132
-0.004
```

FRE Place mémoire libre

Format

<place> = FRE(0)
où <place> est la place libre.

Fonction

Cette fonction renvoie la place restante en mémoire vive, en octets.

Exemple :

```
PRINT FRE(0)
1737260 ... soit 1,7 MOctets (sur un Méga ST2)
```

FSETDTA Initialisation de la DTA

Format

~FSETDTA(<adresse>)
où <adresse> est l'adresse de la DTA.

Fonction

Cette instruction fixe l'adresse de la DTA (Disk Transfert Area) qui par défaut se trouve en BASEPAGE+128. Cette zone contient des données en rapport avec le dernier fichier accédé :

Adresse	Donnée
DTA+21	Attribut
DTA+22 à DTA+23	Heure
DTA+24 à DTA+25	Date
DTA+26 à DTA+29	Longueur du fichier
DTA+30 à DTA+43	Nom du fichier

Exemple :

```
~FSETDTA(BASEPAGE+128)
```

FSFIRST Premier fichier correspondant au modèle

Format

<rés> = FSFIRST(<modèle>,<attrib>)
où <modèle> une chaîne qui contient le modèle du fichier recherché, par exemple "*.DOC" ou encore "a??.*",
<rés> vaut false si un fichier correspond au modèle et true dans le cas contraire.
et <attrib> est l'attribut du fichier recherché, selon le codage suivant :



bit	Signification
0	Fichier protégé en écriture
1	Fichier caché
2	Fichier système
3	Label de disque
4	Dossier
5	Fichier archive

Fonction Cette fonction recherche le fichier dont le modèle et l'attribut sont spécifiés. Si la recherche aboutit les données relatives au fichier sont placées dans la DTA.

Exemple :

Affichage des fichiers correspondant au modèle entré :

```
~FSETDTA(BASEPAGE+128)
INPUT "Entrez le modèle recherché : ";m$
bis%=FSFIRST(m$,-1)
REPEAT
    PRINT CHAR{BASEPAGE+158}
    bis%=FSNEXT()
UNTIL bis%
```

FSNEXT — Autres fichiers correspondant au modèle

Format <rés> = FSNEXT()
où <rés> vaut false si un fichier correspond au modèle et true dans le cas contraire.

Fonction Cette fonction recherche les éventuels autres fichiers correspondant au modèle défini dans une instruction FSFIRST précédente. Si la recherche aboutit les données relatives au fichier sont placées dans la DTA.

Exemple :

Voir FSFIRST

FULLW — Taille maximale pour une fenêtre

Format FULLW<No>
où <No> est le numéro d'une fenêtre ouverte avec OPENW.

Fonction Cette instruction donne la taille maximale à la fenêtre dont le numéro est spécifié.

Exemple :

```
FULLW 3 → Taille maximale à la fenêtre 3
```

FUNCTION Définition d'une fonction

Format

```
FUNCTION<nom>[(<p1>,...,<pN>)]
  <bloc d'instructions>
  RETURN <valeur>
ENDFUNC
```

où <nom> est le nom de la fonction,
les <pi> sont les paramètres transmis à la fonction,
et <valeur> est la valeur retournée par la fonction.

Fonction Le mot clé FUNCTION permet de définir des fonctions. Une fonction est l'équivalent d'une procédure, mais contrairement aux procédures, elle renvoie une valeur.

Exemple :

Calcul d'un sinus hyperbolique :

```
INPUT "Entrez l'argument : ";c
PRINT "sinh(c) = "@sinh(c)
END
'
FUNCTION sinh(c)
  RETURN (exp(c)-exp(-c))/2
ENDFUNC
```

GDOS? Le GDOS est-il chargé ?

Format

<rés> = GDOS?
Après l'appel, <rés> vaut TRUE si le GDOS est en mémoire et FALSE dans le cas contraire.

Fonction Cette fonction teste si le GDOS (Graphic Device Operating System) est présent en mémoire. Elle renvoie TRUE si le GDOS est présent et FALSE dans le cas contraire.



GEMDOS Accès aux fonctions du GEMDOS

Format

<rés> = GEMDOS(<numéro>[,<p1>,...,<pN>])
où <rés> est le résultat de la fonction activée,
<numéro> est le numéro de la fonction GEMDOS à activer, et les <pi> sont les paramètres à passer à la fonction.

Fonction

Le mot clé GEMDOS donne accès aux fonctions du GEMDOS. Un ou plusieurs paramètres peuvent être passés à la fonction, et la fonction peut dans certains cas renvoyer une valeur.

Exemple :

```
~GEMDOS(2,65) → Affichage d'un "A" sur l'écran
PRINT GEMDOS(17) → Test de l'imprimante
                   (TRUE si l'imprimante est prête, FALSE sinon)
```

GET Lecture d'une portion d'écran

Format

GET<x1>, <y1>, <x2>, <y2>, <var>
où <x1> et <y1> sont les coordonnées du coin supérieur gauche du rectangle à mémoriser,
<x2> et <y2> sont les coordonnées du coin inférieur droit du rectangle à mémoriser,
et <var> est une variable chaîne dans laquelle le rectangle est mémorisé.

Fonction

Cette fonction mémorise une zone d'écran rectangulaire dans une variable chaîne. La zone peut être réaffichée à l'aide de l'instruction PUT.

Exemple :

```
GET 100,100,120,160,a$
```

GET # Lecture d'un enregistrement dans un fichier à accès direct

Format

GET # <Canal>[,<enreg>]
où <canal> est le numéro du fichier à accès direct à accéder,
et <enreg>, lorsqu'il est précisé est le numéro de l'enregistrement à lire.
Lorsqu'il n'est pas précisé, l'enregistrement lu est l'enregistrement courant.
L'instruction RECORD permet de modifier l'enregistrement courant.

Fonction Cette instruction permet de lire un enregistrement dans un fichier à accès direct.

Exemple :

```
GET #1          — Lecture de l'enregistrement courant dans le fichier #1
GET #1,123      — Lecture de l'enregistrement 123 dans le fichier #1
```

GOSUB Exécution d'une procédure

Format GOSUB<proc>[<v1>,...,<vN>]
où <proc> est le nom de la procédure à exécuter,
et les <vI> sont les éventuelles variables à passer à la procédure.

Fonction Cette instruction exécute la procédure dont le nom est spécifié.

Remarque Il est également possible d'exécuter une procédure à l'aide du signe arrobas (@).

Exemple :

```
@ carre(10)
ou
GOSUB carre(10) —Exécution de la procédure carre en lui transmettant la valeur 10
```

GOTO Débranchement à l'étiquette spécifiée

Format GOTO<Etiq>
où <Etiq> est une étiquette.

Fonction Cette instruction débranche le programme à l'étiquette spécifiée. Elle devra autant que possible être évitée, car elle alourdit et complique inutilement le programme... On lui préférera une structure du type REPEAT UNTIL ou DO WHILE.

Exemple :

```
GOTO ici
...
ici:
```



GRAPHMODE Définition du mode d'affichage graphique

Format

GRAPHMODE<n>
où <n> vaut
1 : remplacement
2 : transparence
3 : OU exclusif
4 : OU exclusif transparent

Fonction

Cette instruction définit le mode d'affichage des graphiques. Elle entre en vigueur dès que deux éléments graphiques se chevauchent.

Exemple :

```
DEFFILL 2,3,8
GRAPHMODE 1
PBOX 10,10,100,100
'

DEFFILL 3,3,7
GRAPHMODE 2
PBOX 90,20,200,110

DEFFILL 2,3,6
GRAPHMODE 3
PBOX 190,30,300,120

DEFFILL 3,3,5
GRAPHMODE 4
PBOX 290,40,400,130
```

HARDCOPY Copie de l'écran sur imprimante

Format

HARDCOPY

Fonction

Cette instruction copie le contenu de l'écran sur imprimante.

HEX$ Conversion en hexadécimal

Format

<rés> = HEX$(<nombre>[,<précision>])
où <rés> est la conversion hexadécimale de <nombre>,
et <précision> est le nombre de chiffres de la conversion.

Fonction

Cette fonction convertit une expression entière en un nombre hexadécimal. La taille du nombre hexadécimal peut être précisée à concurrence de 8 digits.

Exemple :

```
x%=1000
PRINT HEX$(x%)          → 3E8
PRINT HEX$(32000,8)     → 00007D00
```

HIDEM Désactivation du curseur de la souris

Format HIDEM

Fonction Cette instruction dévalide l'affichage du curseur de la souris. Pour éviter tout problème lié au déplacement de la souris pendant un affichage graphique, nous vous conseillons d'interdire l'affichage de la souris pendant tout affichage graphique avec l'instruction HIDEM et de l'autoriser lorsque l'affichage est terminé avec l'instruction SHOWM.

Exemple :

```
HIDEM
' une ou plusieurs instructions graphiques
SHOWM
```

HIMEM Première adresse non utilisée par le GFA Basic

Format <rés>=HIMEM
où <rés> est la première adresse non utilisée par le GFA.

Fonction Cette fonction renvoie l'adresse du premier octet non utilisé par le GFA. Cette adresse est par défaut égale à l'adresse de la mémoire d'écran - 16384 octets.

Exemple :

```
PRINT HIMEM
999414 → sur un 1040 STF
```

HLINE Tracé de lignes horizontales

Format HLINE <X1>, <X2>, <Y>, <couleur>, <mode>, <motif>, <nombre>
où <X1> et <X2> sont les abscisses de début et de fin de la ligne,

<Y> est l'ordonnée de la ligne,

<couleur> est la couleur de tracé (0 à 15 en basse résolution, 0 à 3 en moyenne résolution, 0 ou 1 en haute résolution),



<mode> est le mode graphique :
0 : remplacement
1 : transparent
2 : inversion
3 : inversion transparente

<motif> est une variable qui contient un ou plusieurs motifs de 16 bits. Chaque bit à 1 est un pixel sur l'écran et <nombre> est le nombre de motifs contenus dans la variable de définition du motif. Ce paramètre est calculé avec la formule suivante : (2^motif – 1) – 1.

Fonction

Cette instruction permet de tracer des lignes horizontales tout en définissant le mode d'affichage, la couleur de tracé et le ou les motifs utilisés pour le tracé.

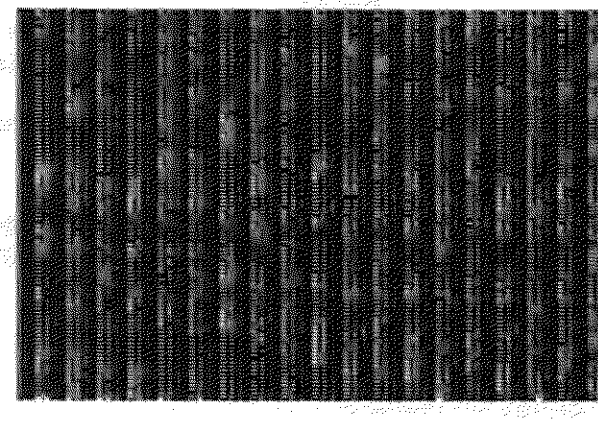
Exemple :

En haute résolution :

```
motif% = &X11110111001100011111111111111111
adresse% = V:motif%
FOR i% = 0 TO 400
  HLINE 0,i%,639,1,0,adresse%,1
NEXT i%
```

La variable motif% contient deux motifs de 16 bits. Le dernier paramètre de l'instruction HLINE est donc égal à (2^1) – 1 = 1.
HLINE utilise alternativement les deux motifs de lignes définis dans motif%.

Résultat :



HTAB Positionnement horizontal du curseur

Format

HTAB <colonne>
où <colonne> est l'abscisse texte du curseur.

Fonction

Cette instruction initialise l'ordonnée du curseur texte sur l'écran.

Exemple :

```
FOR i=1 TO 6
  HTAB(i)
  PRINT "*"
NEXT i
```

Ce programme affiche des astérisques décalés d'une position supplémentaire à chaque ligne :

```
*
  *
    *
      *
        *
          *
```

IF THEN ELSE Exécution conditionnelle d'instructions

Format

IF <condition> [THEN]
  <bloc 1>
[ELSE
  <bloc 2>]
ENDIF
où <condition> est une condition logique,
<bloc 1> est un bloc d'instructions qui est exécuté si <condition> est vraie,
et <bloc 2> est un bloc d'instructions qui est exécuté si <condition> est fausse.

Fonction

Cette instruction permet d'exécuter un bloc d'instructions ou un autre en fonction d'une condition logique.

Exemple :

```
INPUT "Entrez un nombre : ";a
IF a<0 THEN
  PRINT "Ce nombre est négatif"
ELSE
  PRINT "Ce nombre est positif ou nul"
ENDIF
```



IMP Implication logique

Format

<rés> = <par 1> IMP <par 2>
où <par 1> et <par 2> représentent deux paramètres entiers, et <rés> est le résultat de l'implication logique entre <par 1> et <par 2>.

Fonction

L'opérateur IMP calcule l'implication logique entre deux paramètres entiers. Au niveau du bit, l'implication est calculée selon la table de vérité suivante :

IMP	0	1
0	1	1
1	0	1

Le calcul au niveau des deux paramètres entiers est le résultat de la table de vérité ci-dessus appliquée à chaque bit de même rang des deux paramètres.

INC Incrémentation d'une variable numérique

Format

INC <var>
où <var> est la variable à incrémenter.

Fonction

Cette instruction incrémente d'une unité une variable numérique. Son exécution est bien plus rapide qu'une classique addition (voir exemple).

Exemple :

Cet exemple compare le temps d'exécution d'une boucle contenant 100000 additions unaires et d'une autre boucle contenant 100000 INC.

Version « addition » :

```
z=TIMER
f=0
FOR i=1 TO 100000
  f=f+1
NEXT i
PRINT TIMER-z
```

Temps d'exécution : 3625

Version INC :

```
z=TIMER
f=0
FOR i=1 TO 100000
  INC f
NEXT i
PRINT TIMER-z
```

Temps d'exécution : 1874

INFOW Affichage d'une chaîne dans la ligne d'information d'une fenêtre d'écran

Format

INFOW [#]<fenêtre>,<chaîne>
où <fenêtre> est le numéro d'une fenêtre ouverte,
et <chaîne> est la chaîne à afficher.

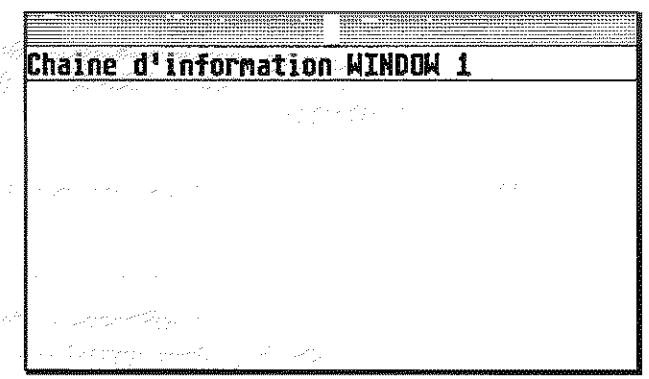
Fonction

Cette instruction affiche une chaîne alphanumérique dans la ligne d'information d'une fenêtre ouverte.

Exemple :

```
OPENW 1
INFOW 1, "Chaîne d'information WINDOW 1"
```

Résultat :



INKEY$ Saisie de caractères « au vol »

Format

<ch> = INKEY$
où <ch> est une variable chaîne.

Fonction

Cette fonction teste le clavier et renvoie le cas échéant le caractère tapé. Si aucun caractère n'est tapé, elle renvoie une chaîne vide. Son exécution n'interrompt pas le fonctionnement du programme.

Exemple :

```
REPEAT
  a$=INKEY$
  PRINT "a";
UNTIL UPPER$(a$)="Q"
```

Ce programme affiche une suite de lettres "a" jusqu'à ce que la touche "Q" du clavier soit pressée.



INLINE Intégration d'un programme assembleur

Format
INLINE <adresse>,<valeur>
où <adresse> est l'adresse d'implantation en mémoire de l'entier <valeur>.

Fonction
Cette instruction permet d'intégrer un programme en langage machine dans un programme GFA Basic.

INP Lecture d'un octet

Format
INP(<port>)
ou INP(# <canal>)
port représente le port d'entrée/sortie à accéder :
0 = imprimante
1 = RS 232
2 = clavier
3 = interface MIDI
4 = processeur clavier
5 = moniteur
et <canal> l'identificateur du fichier séquentiel à lire.

Fonction
Cette fonction lit un octet sur un périphérique d'entrée/sortie ou dans un fichier séquentiel ouvert en lecture.

Exemple :

```
PRINT "Appuyez sur une touche s.v.p."
d=INP(2)

' Lecture de 100 octets sur l'interface série RS232
nb=1
REPEAT
  t(nb)=INP(1)
  nb=nb+1
UNTIL nb=101
```

INPAUX$ Lecture rapide sur la RS 232

Format
<rés> = INPAUX$
où <rés> est une variable chaîne.

Fonction
Cette instruction permet de lire rapidement les données renvoyées par l'interface série.

Exemple :

```
REPEAT
  ch$=INKEY$
  PRINT INPAUX$
UNTIL ch$<>""
```

INPMID$ Lecture rapide sur le port MIDI

Format

<rés> = INPMID$
où <rés> est une variable chaîne.

Fonction

Cette instruction permet de lire rapidement les données renvoyées par l'interface musicale MIDI.

Exemple :

```
t$ = ""
REPEAT
  t$ = t$ + INPAUX$
UNTIL len(t$) = 100
```

INPUT Lecture de données

Format

Deux formats possibles :

1) INPUT [" <texte> " ,;]<var1>,...,<varN>
où <texte> est une chaîne facultative affichée à la position courante du curseur, et les <varI> sont des variables de tous types dans lesquelles seront stockées les données entrées au clavier.

2) INPUT # <N>,<var1>,...,<varN>
où <N> est l'identificateur d'un fichier séquentiel ouvert en lecture, et les <varI> sont les variables dans lesquelles seront stockées les données lues dans le fichier.

Fonction

Cette instruction lit une ou plusieurs données au clavier ou dans un fichier séquentiel ouvert en lecture et stocke ces données dans des variables.

Dans le cas de la lecture au clavier, un texte peut être affiché sur l'écran pour préciser la nature des données à saisir. Ce texte est séparé de la première variable par une virgule ou par un point virgule. Le séparateur virgule positionne le point de saisie à la suite du texte. Le séparateur point virgule affiche un point d'interrogation et un espace avant le point de saisie.

Exemple :

```
INPUT "Entrez deux nombres : ",a,b

DIM t(100)
OPEN "i",#1,"DONNEES.DAT"
FOR i=1 TO 100
  INPUT #1,t(i)
NEXT i
CLOSE #1
```



INPUT$ Lecture d'une chaîne de longueur définie

Format

<chaîne> = INPUT$(<len>[,#<N>]
où <chaîne> est la variable dans laquelle doit être stockée la chaîne lue,
<len> est la longueur de la chaîne à lire,
<N> est l'identificateur du fichier dans lequel doit se faire la lecture.

Fonction

Cette fonction lit une chaîne dont la longueur est spécifiée dans l'argument. La lecture peut se faire au clavier ou dans un fichier séquentiel.

Exemple :

```
PRINT "Entrez un mot de 5 lettres : ";
a$ = INPUT$(5)
INPUT$(100,#1)
```

INSERT Insertion d'un élément dans un tableau

Format

INSERT <tableau>(<élément>) = <valeur>
où <tableau> est le tableau dans lequel doit se faire l'insertion,
<élément> est l'élément du tableau à insérer.
et <valeur> est la valeur de l'élément à insérer.

Fonction

Cette instruction insère une donnée dans un tableau.

Exemple :

```
DIM t(6)
FOR i = 1 TO 5
  t(i) = i
NEXT i
INSERT t(3) = 12
FOR i = 1 TO 6
  PRINT t(i);" ";
NEXT i
```

Résultat :
1 2 12 3 4 5

INSTR Recherche d'une sous-chaîne dans une chaîne

Format

<p> = INSTR(<source>,<ch>[,<pos>])
où <p> est la position de <ch> dans <source>, ou 0 si <ch> n'a pas été trouvé dans <source>.
Si <pos> est précisé, la recherche commence à partir du caractère de position <p>.

Fonction

Cette fonction retourne la position d'une sous-chaîne dans une chaîne.

KEYLOOK Lecture du buffer clavier sans écrasement

Format KEYLOOK <n>

où <n> est un mot long qui contient des informations sur la touche pressée.

Fonction Cette instruction renvoie des informations sur le contenu du buffer clavier sous la forme d'un mot long codé comme suit :

Octet 1 : code ASCII
Octet 2 : 0
Octet 3 : code clavier
Octet 4 : état des touches de commutation

Le buffer clavier n'est pas vidé par la lecture.

Exemple :

Ce petit exemple vous donnera une idée plus précise sur l'instruction KEYLOOK.

```
REPEAT
  KEYLOOK n%
  IF n%<>0 THEN
    CLS
    PRINT "Code renvoyé : ";n%
    s$=HEX$(n%)
    WHILE LEN(s$)<>8
      s$="0"+s$
    WEND
    PRINT "Code ASCII : ";VAL("$"+MID$(s$,7,2))
    PRINT "Code clavier : ";VAL("$"+MID$(s$,3,2))
    PRINT "Touches de commutation : ";VAL("$"+MID$(s$,1,2))
  ENDIF
  IF n%<>0 THEN
    KEYGET nocare%
  ENDIF
UNTIL FALSE
```

Lorsqu'une touche du clavier est pressée, les codes qui la caractérisent sont lus et placés dans la variable n%. Ils sont ensuite décodés et affichés sur l'écran. L'instruction KEYGET vide le buffer clavier et permet de lire une autre touche.

KEYPAD Fonctionnement du clavier

Format KEYPAD <n>

où <n>détermine l'activité du clavier :

Bit	0	1
1	Bloc numérique normal	Bloc numérique=touches flèches



Fonction Cette fonction permet d'accéder en lecture ou en écriture au tableau de sortie entière VDI (reportez-vous à la Partie 5 chapitre 2 pour avoir plus de détails).

KEYDEF

Redéfinition d'une touche de fonction

Format KEYDEF <touche>,<chaîne>

où <touche> est une valeur entière comprise entre 1 et 20.
Les valeurs 1 à 10 correspondent aux touches <F1 > à <F10>,
Les valeurs 11 à 20 correspondent aux touches Shift + < F1 > à Shift + <F10 >,
et <chaîne> est la chaîne à affecter à la touche de fonction.

Fonction Cette instruction affecte une chaîne de caractères à une touche de fonction.

Exemple :

Après l'exécution de cette instruction :

```
KEYDEF 1,"Au secours".
```

tout appui sur la touche F1 produira l'affichage de la chaîne "Au secours".

KEYGET

Attente de l'appui sur une touche

Format KEYGET <touche>

où <touche> est un mot long qui contient des informations sur la touche pressée.

Fonction Cette fonction attend qu'une touche du clavier soit pressée. Elle renvoie des informations sur cette touche sous la forme d'un mot long codé comme suit :

Octet 1 : code ASCII
Octet 2 : 0
Octet 3 : code clavier
Octet 4 : état des touches de commutation

Exemple :

Ce petit exemple vous donnera une idée plus précise sur l'instruction KEYGET.

```
REPEAT
  KEYGET n%
  s$ = HEX$(n%)
  WHILE LEN(s$) < > 8
    s$ = "0" + s$
  WEND
  PRINT "Code ASCII :";VAL("$" + MID$(s$,7,2))
  PRINT "Code clavier: ";VAL("$" + MID$(s$,3,2))
  PRINT "TQuche de commutation: ";VAL("$"+MID$(s$,1,2))
UNTIL n%=65563
```

(suite)

Dans cet exemple, les touches frappées au clavier sont acquises avec la fonction KEYGET. Leur code ASCII est extrait à l'aide d'une instruction RIGHT$. S'il est compris entre 97 et 122, il s'agit d'une lettre minuscule. On lui soustrait 32 de manière à le convertir en une lettre majuscule. Le caractère est placé dans le buffer clavier avec une instruction KEYPRESS. En fin de saisie (appui sur la touche ESC), les caractères tapés sont lus dans une instruction LINE INPUT et affichés sur l'écran.

KEYTEST Lecture du clavier

Format KEYTEST <n>

où <n> est un mot long qui contient des informations sur la touche pressée (voir KEYLOOK).

Fonction Cette instruction teste l'appui sur une touche du clavier et renvoie les informations afférentes : code ASCII, code clavier, état des touches de commutation (voir KEYLOOK pour le codage).

Exemple :

```
REPEAT
  KEYTEST n%
  IF n%<>0 THEN
    PRINT "code clavier : $";HEX$(n%)
    a$=RIGHT$(HEX$(n%),2)
  ENDIF
UNTIL VAL("$"+a$)=27
```

Cet exemple utilise l'instruction KEYTEST dans une boucle de type REPEAT UNTIL qui prend fin lorsque la touche ESC est pressée. Suite à l'appui sur une touche, la valeur renvoyée est convertie en hexadécimal et affichée sur l'écran.

KILL Effacement d'un fichier disque

Format KILL <nom$>

où <nom$> est une chaîne de caractères qui indique le chemin et le nom du fichier disque à effacer.

Fonction Cette instruction efface le fichier dont le chemin et le nom sont spécifiés en paramètre.

Exemple :

```
KILL "a:\lous.tik"
```

efface le fichier lous.tik du disque A:.



2	Gestion curseur par Control+Flèche
3	Saisie de caractères ASCII avec touche ALT
4	Affectation possible pour F1 à F10
5	Affectation possible pour Shift+F1 à Shift+F10

Fonction Cette fonction définit le fonctionnement du clavier (pavé numérique, gestion du curseur, saisie de caractères ASCII, affectation de chaînes de caractères aux touches de fonction).

Exemple :

```
KEYPAD 8
```

Après l'exécution de cette instruction, tout appui sur Alt+3 chiffres du pavé numérique donnera lieu à l'affichage du caractère correspondant.

KEYPRESS Simulation d'une action au clavier

Format KEYPRESS <n>

où <n> est le code ASCII de la touche.

Fonction KEYPRESS simule la frappe d'une touche au clavier. Cette touche peut être lue avec une des formes de l'instruction INPUT.

Exemple :

```
KEYDEF 1,"Salut"+CHR$(13)
KEYPRESS &H83B0000
PAUSE 1
LINE INPUT a$
PRINT a$
REPEAT
  KEYGET n%
  a$=RIGHT$(HEX$(n%),2)
  a=VAL("$"+a$)
  IF (a<=122) AND (a>=97) THEN
    a=a-32
  ENDIF
  KEYPRESS a
UNTIL a=27
KEYPRESS 13
LINE INPUT z$
PRINT z$
```

Fonction Cette instruction affecte le second opérateur au premier. Elle est de moins en moins utilisée. Les instructions suivantes sont, en effet, équivalentes :

LET a=b

a=b

LINE — Tracé d'un segment de droite

Format LINE <a>,<b>,<c>,<d>

où <a> est l'abscisse de départ,
<b> est l'ordonnée de départ,
<c> est l'abscisse d'arrivée,
<d> est l'ordonnée d'arrivée.

Fonction Cette instruction trace un segment de droite entre les coordonnées qui lui sont passées.

Exemple :

Ce petit programme trace des segments de droite aléatoires. Le tracé prend fin lorsqu'une touche du clavier est pressée.

```
REPEAT
  KEYTEST n%
  a=RANDOM(640)
  b=RANDOM(400)
  c=RANDOM(640)
  d=RANDOM(400)
  LINE a,b,c,d
UNTIL n%<>0
```

LINE INPUT — Lecture d'une chaîne au clavier ou dans un fichier

Format LINE INPUT ["texte",]<v1$>[,...,<vn$>]

où "texte" est un texte facultatif précédant l'entrée des données,

et les <vi$> sont la ou les chaînes à lire.

Fonction Cette instruction lit une chaîne au clavier qui peut comporter une ou plusieurs virgules. La saisie prend fin lorsque la touche Return est pressée.



LEFT$ Extraction d'une sous-chaîne

Format

<res$>=LEFT$(<nom$>,<longueur>)

où <nom$> est le nom de la chaîne source,

<longueur> est la longueur de l'extraction,

<res$> est la chaîne extraite.

Fonction

Cette fonction extrait une sous-chaîne d'une chaîne. La sous-chaîne consiste en <longueur> caractères à partir de la gauche de la chaîne source.

Exemple :

```
a$ = "bibabobu"
PRINT LEFT$(a$,2)
PRINT LEFT$(a$,4)
```

Résultat :

```
bi
biba
```

LEN Longueur d'une chaîne

Format

<l>=LEN(<ch$>)

où <ch$> est la chaîne dont on désire connaître la longueur,

et <l> est la longueur en caractères de la chaîne <ch$>.

Fonction

Cette fonction renvoie la longueur de la chaîne de caractères qui lui est passée.

Exemple :

```
a$ = "test de longueur"
PRINT LEN(a$)
```

Résultat :

```
16
```

LET Affectation à une variable

Format

LET <a>=<b>

où <b> est une valeur ou une variable à affecter à <a>.

LOAD Chargement d'un programme

Format LOAD <"nom$">

où <"nom$"> est le nom du fichier à charger.

Fonction Cette fonction charge un programme en mémoire.

Exemple :

Chargement du fichier ESSAI.BAS qui se trouve sur le disque B: dans le répertoire SYSTEM :

```
LOAD "B:\SYSTEM\ESSAI.BAS"
```

LOC Position dans un fichier

Format LOC(#n)

où n est le numéro de canal sous lequel a été ouvert le fichier.

Fonction Cette instruction indique la position du pointeur de données dans le fichier.

Exemple :

Ouverture du fichier LISTE.LST. Affichage des lignes qui le composent précédées d'un numéro de ligne.

```
OPEN "I",#1,"liste.lst"
REPEAT
  INPUT #1,a$
  PRINT LOC(#1),a$
UNTIL EOF(#1)
CLOSE #1
```

LOCAL Définition de variables locales

Format LOCAR <v1>[,...,<vn>]

où les <vi> sont les variables locales à la procédure.

Fonction Cette instruction définit une ou plusieurs variables locales à la procédure courante. Des variables de même nom pourront être utilisées dans le reste du programme sans intéragir avec celles de la procédure.

Exemple :

```
PROCEDURE testlocal
  LOCAL a,b
  a=1
  b=2
RETURN
```



Exemple :

```
LINE INPUT a$
PRINT a$
```

Entrez la chaîne de votre choix. Quel qu'en soit son contenu, il est réaffiché après l'appui sur la touche Return.

LINE-A Appel à une routine graphique

Fonction

Les routines LINE-A sortent du cadre de cet exposé. Vous trouverez tous les détails nécessaires à leur exploitation dans d'autres parties de l'ouvrage.

LIST Listage d'un programme

Format

LIST ["ch$"]

où <ch$> contient le chemin et le nom du fichier à sauvegarder.

Fonction

Lorsque aucune variable n'est précisée, cette instruction liste le contenu du programme sur l'écran. Lorsqu'une variable chaîne est précisée, le programme est sauvegardé sous format ASCII (extension .LST) sur le disque spécifié.

Exemple :

```
LIST "a:\monprog.lst"
```

sauvegarde le programme en mémoire sur le disque A: au format ASCII sous le nom MONPROG.LST.

LLIST Impression du listing d'un programme

Format

LLIST

Fonction

Cette instruction imprime le listing du programme courant.

Le nombre de colonnes, et le nombre de lignes par page peuvent être spécifiés à l'aide des instructions suivantes :

.ll <col>

.pl <lig>

où <col> est le nombre de colonnes,

et <lig> le nombre de lignes par page.

LOG10 Logarithme décimal d'une expression

Format

<res>=LOG10(<val>)

où <val> est une valeur ou variable numérique,

et <res> est le logarithme décimal de <val>.

Fonction

Cette fonction renvoie le logarithme décimal de l'argument numérique qui lui est passé.

Exemple :

```
PRINT LOG10(32.615)
```

LONG Lecture/Ecriture d'un entier long

Format

<x>=LONG{<ad>}

ou LONG{<ad>}=<y>

où <ad> est une adresse mémoire,

<x> est l'entier long lu en <ad>,

et <y> est l'entier long à stocker en <ad>.

Fonction

Cette instruction permet de lire ou de stocker un entier long en mémoire.

Exemple :

Stockage d'un entier long en mémoire :

```
LONG{&H1234}=1234567890
```

Lecture d'un entier long en mémoire :

```
PRINT LONG{&h1234}
```

LOOP Instructions de répétition

Format

Boucle infinie

```
DO
LOOP
```

Répétition tant qu'une condition est réalisée

```
DO WHILE condition
LOOP
```

ou

```
LOOP WHILE condition
WEND
```



LOCATE Positionnement du curseur

Format LOCATE <ligne>,<colonne>

où <ligne> est la ligne où doit être positionné le curseur,

et <colonne> la colonne où doit être positionné le curseur.

Fonction Cette fonction positionne le curseur sur l'écran.

Exemple :

```
LOCATE 10,4
PRINT "Bonjour"
```

Le mot **"Bonjour"** est affiché sur la ligne 10, et à partir de la colonne 4.

LOF Longueur d'un fichier

Format <l>=LOF(#n)

où n est le numéro du canal affecté au fichier lors de son ouverture,

et <l> la longueur du fichier renvoyée par la fonction.

Fonction Cette fonction renvoie la longueur du fichier courant.

Exemple :

```
OPEN "I",#1,"fichier.txt"
PRINT "La longueur de ce fichier est ";LOF(#1)
CLOSE #1
```

LOG Logarithme népérien d'une expression

Format <res>=LOG(<val>)

où <val> est une valeur ou variable numérique,

et <res> est le logarithme népérien de <val>.

Fonction Cette fonction renvoie le logarithme népérien de l'argument numérique qui lui est passé.

Exemple :

```
PRINT LOG(1)
```

Exemple :

```
Voir LPEEK
```

LPOS Position horizontale de la tête d'impression

Format <p>=LPOS(x)

où <p> est la position horizontale de la tête d'impression.

Fonction Cette fonction renvoie la position horizontale de la tête d'impression d'une imprimante matricielle.

Exemple :

```
PRINT LPOS(x)
```

LPRINT Impression de données

Format LPRINT <exp1>[,...,<expN>]

où les <expI> sont les expressions à imprimer.

Fonction Cette instruction évalue et imprime une ou plusieurs expression.

Exemple :

```
LPRINT a*SIN(b)
```

LSET Insertion d'une chaîne à gauche

Format LSET <a$>=<b$>

où <b$> est une chaîne de caractères placée à gauche de <a$>.

Fonction Cette instruction concatène les chaînes a$ et b$ en plaçant a$ à gauche de b$.

Exemple :

```
a$=">>>>"
FOR i=1 TO 100
  LSET a$=STR$(i)
  PRINT a$;
NEXT i
```



Répétition jusqu'à ce qu'une condition soit réalisée

DO UNTIL condition
LOOP

ou

LOOP UNTIL condition
WEND

Fonction Ces structures répètent un bloc d'instructions indéfiniment, jusqu'à ce que ou tant qu'une condition soit réalisée.

Exemple :

Lecture de données au clavier jusqu'à ce que la valeur entrée soit égale à "q" ou à "Q" :

```
REPEAT
  LINE INPUT a$
UNTIL (a$ = "q") OR (a$ = "Q")
```

LPEEK Lecture mémoire sur 4 octets

Format <x>=LPEEK(<y>)

où <y> est une adresse en mémoire,

et <x> contient après exécution les quatre octets qui se trouvent en <y>.

Fonction Cette fonction renvoie les quatre octets situés à partir de la mémoire spécifiée.

Exemple :

```
LPOKE 8000,&H12345678
PRINT LPEEK(8000)
```

LPOKE Ecriture mémoire sur 4 octets

Format LPOKE <x>,<y>

où <x> est une adresse en mémoire,

et <y> contient la valeur sur 4 octets à stocker en <x>.

Fonction Cette fonction stocke les quatre octets du second paramètre à l'adresse contenue dans le premier paramètre.

et <res> est le résultat de la libération :

0 si tout s'est bien passé,

un numéro d'erreur négatif dans le cas contraire.

Fonction Cette fonction libère la place précédemment réservée avec une instruction MALLOC.

MID$ Extraction d'une sous-chaîne

Format <res$>=MID$(<so$>,<deb>,<len>)

où <so$> est la chaîne source,
<deb> est la position du premier caractère à extraire,
<len> est la longueur de la sous-chaîne à extraire,
<res$> est la sous-chaîne résultat.

Fonction Cette fonction extrait une sous-chaîne d'une chaîne.

Exemple :

```
a$="123456789"
PRINT MID$(a$,3,4)
```

Résultat :

```
3456
```

MIN Plus petite des expressions passées

Format <res>=MIN(<x1>[,...,<xN>)

où les <xI> sont des valeurs ou expressions numériques,

et <res> est la pus petite des valeurs passées.

Fonction Cette fonction renvoie la plus petite des valeurs qui lui sont passées.

Exemple :

```
PRINT MIN(1,432,-12,SIN(3))
```

MKD Conversion Flottant/Chaîne sur 8 octets

Format <x$>=MKD$(<y>)

où <y> est un nombre flottant,

et <x$> une chaîne de 8 octets qui est la conversion en caractères de <y>.



MALLOC Allocation de mémoire

Format <res>=MALLOC(<mem>)

où <mem> est la taille mémoire à allouer en octets. Si <mem> vaut -1, MALLOC renvoie la plus grande taille mémoire contiguë disponible.

Fonction Cette fonction alloue un espace mémoire. Elle renvoie :

— la taille maximale disponible si son argument vaut -1 ;

— l'adresse de départ de la zone réservée dans le cas contraire.

Exemple :

Affichage de la mémoire disponible, puis réservation de 32000 octets :

```
PRINT MALLOC(-1)
ad%=MALLOC(32000)
PRINT ad%
```

MAX Plus grande des expressions passées

Format <res>=MAX(<x1>[,...,<xN>)

où les <xI> sont des valeurs ou expressions numériques,

et <res> est la pus grande des valeurs passées.

Fonction Cette fonction renvoie la plus grande des valeurs qui lui sont passées.

Exemple :

```
PRINT MAX(1,432,-12,SIN(3))
```

MENU Gestion de menus utilisateurs

Fonction Les instructions de type MENU (ON MENU, MENU, ON MENU BUTTON, ON MENU KEY, ON MENU IBOX ON MENU OBOX et ON MENU MESSAGE) sortent du cadre de cet exposé. Ces instructions permettent de créer et de gérer des menus utilisateur.

MFREE Libération de la place réservée avec MALLOC

Format <res>=MFREE(<ad>)

où <ad> est l'adresse du bloc mémoire à libérer.

MKI$ Conversion d'un entier 16 bits en une chaîne sur 2 octets

Format <x$>=MKI$(<y>)

où <y> est un nombre entier sur 16 bits,

et <x$> une chaîne de 2 octets qui est la conversion en caractères de <y>.

Fonction Cette fonction convertit un nombre entier codé sur 16 bits en une chaîne sur 2 caractères.

Exemple :

```
s$=MKI$(32767)
PRINT CVI(s$)
```

Résultat :

32767

MKL Conversion d'un entier 32 bits en une chaîne sur 4 octets

Format <x$>=MKL$(<y>)

où <y> est un nombre entier 32 bits (compris entre -2147483648 et 2147483647),

et <x$> une chaîne de 4 octets qui est la conversion en caractères de <y>.

Fonction Cette fonction convertit un nombre entier codé sur 32 bits en une chaîne sur 4 caractères.

Exemple :

```
s$=MKL$(2147483647)
PRINT CVL(s$)
```

Résultat :

2147483647

MKS Conversion Flottant/Chaîne sur 4 octets

Format <x$>=MKS$(<y>)

où <y> est un nombre flottant,

et <x$> une chaîne de 4 octets qui est la conversion en caractères de <y>.



Fonction Cette fonction convertit un nombre flottant en une chaîne sur 8 caractères.

Exemple :

```
s$=MKD$(12.34567890123)
PRINT CVD(s$)
```

Résultat :

12.34567890123

MKDIR Création d'un nouveau dossier

Format MKDIR <n$>

où <n$> est le nom du dossier à créer.

Fonction Cette instruction crée un nouveau dossier dans le disque spécifié.

Exemple :

```
MKDIR "B:\PROGMAN"
```

MKF Conversion Flottant/Chaîne sur 6 octets

Format <x$>=MKF$(<y>)

où <y> est un nombre flottant,

et <x$> une chaîne de 6 octets qui est la conversion en caractères de <y>.

Fonction Cette fonction convertit un nombre flottant en une chaîne sur 6 caractères.

Exemple :

```
s$=MKF$(12.345678901)
PRINT CVF(s$)
```

Résultat :

12.34567890095

(suite)

```
MODE 2
PRINT "Nous sommes le ",DATE$
PRINT "La surface de la terre est environ de ";
PRINT USING "##.###,## km^2",40076.5
MODE 3
PRINT "Nous sommes le ",DATE$
PRINT "La surface de la terre est environ de ";
PRINT USING "##.###,## km^2",40076.5
MODE 4
PRINT "Nous sommes le ",DATE$
PRINT "La surface de la terre est environ de ";
PRINT USING "##,###.## km^2",40076.5
```

Résultat :

```
Nous sommes le 04/22/1992
La surface de la terre est environ de 40,076.50 km^2
Nous sommes le 22.04.1992
La surface de la terre est environ de 40.076,50 km^2
Nous sommes le 04/22/1992
La surface de la terre est environ de 40.076,50 km^2
Nous sommes le 22.04.1992
La surface de la terre est environ de 40,076.50 km^2
```

MONITOR Appel de routines Assembleur

Cette instruction dépasse le cadre de cet exposé.

MOUSE Coordonnées souris et état des boutons

Format MOUSE <x>,<y>,<b>

Après l'appel, <x> et <y> contiennent les coordonnées de la souris, et <b> l'état des boutons.

Fonction Cette fonction renvoie la position et l'état des boutons de la souris.

MOUSEK Etat des boutons de la souris

Format <res>=MOUSEK

Après l'appel, <res> contient l'état des boutons de la souris.

Fonction Cette fonction renvoie l'état des boutons de la souris.



Fonction Cette fonction convertit un nombre flottant en une chaîne sur 4 caractères.

Exemple :

```
s$ = MKS$(12.3456789)
PRINT CVS(s$)
```

Résultat :

12.34567832947

MOD Opérateur MODulo

Format <res>=<x> MOD <y>

ou <x> et <y> sont des nombres entiers,

et <res> est le reste de la division de <x> par <y>.

Fonction Cet opérateur calcule le reste de la division entière des deux arguments qui lui sont passés.

Exemple :

```
PRINT 17 MOD 3          2
```

MODE Mode d'affichage des flottants et de la date

Format MODE <n>

où <n> vaut 0, 1, 2 ou 3 :

<n>	Dates	Nombres flottants
0	15.08.1992	#,###.##
1	08/15/1992	#.###,##
2	15.08.1992	#.###,##
3	08/15/1992	#,###.##

Fonction Cette fonction définit le format d'affichage des nombres flottants et des dates.

Exemple :

```
MODE 1
PRINT "Nous sommes le ",DATE$
PRINT "La surface de la terre est environ de ";
PRINT USING "##,###.## km^2",40076.5
```

NAME Modification du nom d'un fichier

Format NAME <"anc$"> AS <"nouv$">

où <"anc$"> est l'ancien nom du fichier,

et <"nouv$"> le nouveau nom du fichier.

Fonction Cette fonction renomme le fichier dont le nom est passé en premier argument selon le nom passé en second argument.

Exemple :

```
NAME "MONPROG.BAS" AS "MONPROG1.BAS"
```

NEW Effacement du programme en mémoire

Format NEW ou Shift+F4

Fonction Après confirmation, cette instruction efface le programme en mémoire.

NOT Négation logique

Format <res>=NOT <val>

où <val> est une donnée numérique,

et <res> est la négation logique de <val>.

Fonction Cette fonction renvoie l'inversion logique de l'argument qui lui est passé.

Exemple :

```
PRINT NOT 123
Résultat :
-124
```

OCT$ Conversion d'un nombre entier en octal

Format OCT$(<nombre>,[<taille>])

où <nombre> est le nombre à convertir,

et <taille> le nombre de chiffres de la conversion.

Fonction Cette instruction convertit le nombre entier qui lui est passé en son équivalent octal. Le nombre de chiffres de la conversion peut éventuellement être précisé comme second paramètre.



MOUSEX Abscisse de la souris

Format <res>=MOUSEX

Après l'appel, <res> contient l'abscisse de la souris.

Fonction Cette fonction renvoie l'abscisse de la souris.

MOUSEY Ordonnée de la souris

Format <res>=MOUSEY

Après l'appel, <res> contient l'ordonnée de la souris.

Fonction Cette fonction renvoie l'ordonnée de la souris.

MSHRINK Restriction d'une zone mémoire réservée avec MALLOC

Format MSHRINK(<ad>,<taille>)

où <ad> est l'adresse du bloc mémoire précédemment réservé,

<taille> est la nouvelle taille du bloc,

et <er> est un code d'erreur (0 si tout s'est bien passé, un nombre négatif sinon).

Fonction Cette fonction restreint la taille d'un bloc de mémoire précédemment réservé avec l'instruction MALLOC.

Exemple :

```
ad%=MALLOC(32000)
<er>=MSHRINK(ad%,20000)
```

MUL Multiplication de deux expressions

Format MUL <x>,<y>

où <x> est une variable numérique,

et <y> une variable ou une expression numérique.

Fonction Cette fonction multiplie ses deux arguments et place le résultat dans le premier.

Exemple :

```
a%=3
MUL a%,4
```

(suite)

```
'
'
PROCEDURE arret
  PRINT "Dans le programme d'interruption"
RETURN
```

Attention! Ce programme possède une boucle infinie dont il est impossible de sortir par les moyens conventionnels (Control + Shift + Alternate), puisque cette séquence est traitée dans la procédure arret qui interdit l'interruption du programme.

ON ERROR Action lorsqu'une erreur se produit

Format

ON ERROR GOSUB <proc>

où <proc> est le nom de la procédure à activer en cas d'erreur.

Fonction

Cette instruction provoque le débranchement du programme à la procédure dont le nom est indiqué lorsqu'une erreur survient dans le programme.

ON MENU Action lors de la sélection dans un menu

Format

ON MENU

ON MENU GOSUB <proc>

où <proc> est le nom de la procédure à activer en cas de sélection dans un menu.

Fonction

Cette instruction provoque le débranchement du programme à la procédure de traitement dont le nom est indiqué lorsqu'une sélection de menu est activée.

Exemple :

```
DIM t$(22)
FOR i=0 TO 21
  READ t$(i)
NEXT i
DATA Bureau,A propos des menus,"----------",1,2,3,4,5,6,""
DATA Menu 1,Option 1,Option 2,Option 3,""
DATA Menu 2,Option 1,Option 2,------',Fin,"","",""
fin=FALSE
MENU t$()
ON MENU GOSUB option
REPEAT
  ON MENU
```



Exemple :

```
PRINT OCT$(1234)
PRINT OCT$(1234,7)
```

Résultat :

```
2322
0002322
```

ODD Test d'imparité

Format ODD(<Nombre>)

où <Nombre> est un nombre entier dont on désire tester l'imparité.

Fonction Cette fonction renvoie -1 si le nombre qui lui est passé est impair et 0 dans le cas contraire.

Exemple :

```
PRINT ODD(1)
PRINT ODD(2)
```

Résultat :

```
-1
0
```

ON BREAK Action lors de l'appui sur Control+Shift+Alternate

Format ON BREAK CONT

ou ON BREAK GOSUB <proc>

où <proc> est le nom de la procédure à activer en cas d'interruption.

Fonction L'instruction ON BREAK CONT annule l'effet des touches Control+Shift+Alternate.

L'instruction ON BREAK GOSUB provoque le débranchement du programme à la procédure dont le nom est indiqué lorsque les touches Control, Shift et Alternate sont pressées simultanément.

Exemple :

```
ON BREAK GOSUB arret
PRINT "En attente d'interruption."
PRINT "Appuyez sur Control+Shift+Alternate"
REPEAT
UNTIL false
```

— affichage d'un message ;

— affichage de l'option sélectionnée ;

— initialisation de la variable fin à TRUE.

OPEN Ouverture d'un fichier ou d'un périphérique

Format

OPEN <mode$>,<#N>,<nom$>

où <mode$> représente le canal ou le périphérique à ouvrir :

A	Ajout de données à la suite d'un fichier existant
I	Ouverture d'un fichier en lecture
O	Ouverture d'un fichier en écriture
R	Ouverture d'un fichier en lecture/écriture
U	Ouverture d'un fichier existant en lecture/écriture

<#N> est un numéro de canal quelconque compris entre 0 et 99,

<nom$> est le nom du fichier ou du périphérique. Les périphériques accessibles sont :

AUX:	RS 232
CON:	Clavier
IKB:	Processeur clavier
LST:	Imprimante
MID:	Interface MIDI
PRN:	Imprimante
VID:	Moniteur

Fonction

Cette instruction ouvre un fichier ou un périphérique en lecture, écriture ou lecture/écriture. Elle doit être exécutée avant tout accès au fichier ou au périphérique.

Exemple :

```
OPEN "I",#1,"essai.asc"
OPEN "O",#5,"LST:"
```

OPENW Ouverture d'une fenêtre

Format

OPENW <#N>,<x>,<y>,<w>,<h>,<attr>

où <#N> est le numéro de la fenêtre,

<x> et <y> sont les coordonnées du coin supérieur gauche de la fenêtre,



```
(suite)

UNTIL fin
'
'
PROCEDURE option
  MENU OFF
  m%=MENU(0)
  LOCATE 10,10
  IF m%=1 THEN
    PRINT "Sélectionnez Menu 2/Fin pour sortir"
  ELSE
    PRINT t$(m%);SPACE$(30)
    IF m%=19 THEN
      fin=TRUE
    ENDIF
  ENDIF
RETURN
```

La première option du menu comporte neuf entrées :

Entrée	Fonction
t$(0)	Titre de la première option
t$(1)	Nom de le première sous-option
t$(2)	Autant de signes "-" que le nombre de lettres de la première option
t$(3) à t$(8)	Emplacements réservés aux accessoires
t$(9)	Chaîne vide : fin de première option

Les autres options sont toutes construites sur le même modèle :

titre
sous-option 1
...
sous-option N
chaîne vide : fin de nième option

Le menu est activé par l'instruction MENU t$() où t$ est le tableau qui contient les options et sous-options.

Le nom de la procédure de traitement est défini dans l'instruction ON MENU GOSUB.

La boucle REPEAT UNTIL prend fin lorsque la variable fin a pour valeur true, c'est-à-dire lorsque la dix-neuvième entrée est sélectionnée.

La procédure de traitement effectue diverses actions en fonction de l'option sélectionnée :

OPTION BASE Utilisation de l'élément 0 dans les tableaux

Format OPTION BASE 0
ou OPTION BASE 1

Fonction OPTION BASE 0 permet d'utiliser l'élément de rang 0 dans les tableaux, alors que OPTION BASE 1 exclut cet élément.

OR OU logique

Format <x> OR <y>
où <x> et <y> sont deux expressions logiques.

Fonction Cette fonction renvoie la combinaison logique de ses deux opérateurs.

Exemple :

```
IF (x=3) OR (x*2>17) THEN
  c=a OR b
ENDIF
```

OUT Ecriture d'un octet dans un fichier

Format OUT <#N>,<a>[,...,<u>]
où <#N> est le numéro du canal lié au fichier,
et <a> à <u> sont les octets à écrire dans le fichier.

Fonction Cette instruction écrit un ou plusieurs octets dans un fichier ouvert en écriture.

Exemple :

```
OPEN "O",#1,"essai.dat"
OUT #1,1,2,3
CLOSE #1
```

PAUSE Suspension de l'exécution du programme

Format PAUSE <délai>
où <délai> est un délai exprimé en 1/50 secondes.

Fonction Cette fonction suspend l'exécution du programme pendant un délai paramétrable.

Exemple :

```
PAUSE 50        --> Arrêt pendant 1 seconde
```



<w> et <h> sont la largeur et la hauteur en pixels de la fenêtre,

<attr> est un attribut composé d'un ou de plusieurs des éléments suivants :

&H0001	Barre de titre
&H0002	Case de fermeture
&H0004	Case d'occupation maximale
&H0008	Barre de déplacement
&H0010	Ligne d'informations
&H0020	Case de modification de taille
&H0040	Flèche vers le haut
&H0080	Flèche vers le bas
&H0100	Ascenceur vertical
&H0200	Flèche vers la gauche
&H0400	Flèche vers la droite
&H0800	Ascenceur horizontal

Fonction

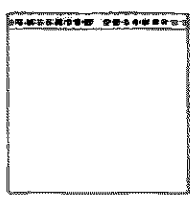
Cette instruction ouvre une fenêtre sur l'écran. Elle doit être exécutée avant de pouvoir utiliser toute instruction d'accès a la fenêtre (FULLW, CLEARW, TITLEW, TOPW, etc.).

Exemple :

Du plus simple :

```
OPENW 1,50,50,200,200,1
```

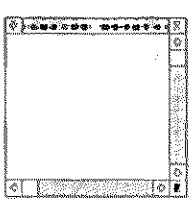
Résultat :



Au plus complet :

```
OPENW 1,50,50,200,200,4095
```

Résultat :



PELLIPSE Affichage d'une ellipse pleine

Format PELLIPSE <x>,<y>,<rx>,<ry>[,<arc1>,<arc2>]

où <x> et <y> sont les coordonnées du centre de l'ellipse,

<rx> et <ry> sont les rayons horizontal et vertical en pixels,

<arc1> et <arc2> sont les angles de départ et de fin de l'arc d'ellipse à tracer. Ces deux paramètres sont facultatifs.

Fonction Cette instruction affiche une ellipse pleine ou un arc d'ellipse plein sur l'écran. La couleur de remplissage doit avoir été précédemment définie avec une instruction DEFFILL.

Exemple :

```
DEFFILL 1,4,2
PELLIPSE 100,100,50,30
```

PI Nombre transcendental PI

Format PI

Fonction Cette fonction renvoie le nombre transcendental PI.

Exemple :

```
PRINT PI
Résultat :
3.14159265359
```

PLOT Affichage d'un point graphique

Format PLOT <x>,<y>

où <x> et <y> sont les coordonnées du point à afficher.

Fonction Cette instruction affiche un point graphique (pixel) sur l'écran. La couleur du point doit être choisie au préalable avec l'instruction COLOR.

Exemple :

```
Tracé d'une parabole
FOR i=-20 TO 20 STEP 0.1
  PLOT 200+i*10,400-i^2
NEXT i
```



PBOX Affichage d'un rectangle plein

Format PBOX <x1>,<y1>,<x2>,<y2>

où <x1> et <y1> sont les coordonnées du coin supérieur gauche du rectangle,

et <x2> et <y2> les coordonnées du coin inférieur droit du rectangle.

Fonction Cette instruction affiche un rectangle plein sur l'écran. La couleur de remplissage doit avoir été précédemment définie avec une instruction DEFFILL.

Exemple :

```
DEFFILL 1,2,3
PBOX 100,100,200,200
```

PCIRCLE Affichage d'un cercle plein

Format PCIRCLE <x>,<y>,<r>[,<arc1>,<arc2>]

où <x> et <y> sont les coordonnées du centre du cercle,

<r> est le rayon du cercle en pixels,

<arc1> et <arc2> sont les angles de départ et de fin de l'arc de cercle à tracer. Ces deux paramètres sont facultatifs.

Fonction Cette instruction affiche un cercle plein ou un arc de cercle plein sur l'écran. La couleur de remplissage doit avoir été précédemment définie avec une instruction DEFFILL.

Exemple :

```
DEFFILL 1,2,4
PCIRCLE 100,100,50
```

PEEK Lecture d'un octet en mémoire

Format <res>=PEEK(<adr>)

où <adr> est une adresse en mémoire,

et <res> est l'octet situé à l'adresse <adr>.

Fonction Cette fonction renvoie l'octet situé à l'adresse mémoire passée en argument.

Exemple :

```
PRINT PEEK(1043)
```

Exemple :

```
DIM x(4),y(4)
FOR i=0 TO 4
  READ x(i),y(i)
NEXT i
DATA 100,0,200,100,100,200,0,100,100,0
POLYLINE 5,x(),y() OFFSET 100,100
DEFFILL 1,2,3
POLYFILL 5,x(),y() OFFSET 100,100
```

POLYMARK Marquage des extrémités d'un polygone

Format POLYMARK <n>,<x()>,<y()> [OFFSET <ox>,<oy>]

où <n> est le nombre de côtés du polygone,

<x()> et <y()> sont des tableaux qui contiennent les coordonnées des côtés du polygone,

<ox> et <oy> lorsqu'ils sont spécifiés sont ajoutés aux coordonnées spécifiées dans les tableaux <x()> et <y()>.

Fonction Cette instruction place un point à chacune des extrémités du polygone définies dans les tableaux x() et y().

Exemple :

```
Voir POLYLINE
```

POS Abscisse du curseur

Format <res>=POS(0)

où <res> est la position du curseur depuis le dernier retour chariot.

Fonction Cette fonction renvoie le nombre de caractères affichés depuis le dernier retour chariot, c'est-à-dire l'abscisse du curseur si la longueur du dernier affichage ne dépassait pas 80 caractères.

Exemple :

```
PRINT POS(0)
```



POKE Stockage d'un octet en mémoire

Format POKE <adr>,<val>

où <adr> est l'adresse de stockage,

et <val> l'octet à stocker en <adr>.

Fonction Cette instruction stocke un octet en mémoire.

Exemple :

POKE 32000,45 --> Stockage de la valeur 45 à l'adresse 32000

POLYFILL Remplissage d'un polygone

Format POLYFILL <n>,<x()>,<y()> [OFFSET <ox>,<oy>]

où <n> est le nombre de côtés du polygone,

<x()> et <y()> sont des tableaux qui contiennent les coordonnées des côtés du polygone,

<ox> et <oy> lorsqu'ils sont spécifiés sont ajoutés aux coordonnées spécifiées dans les tableaux <x()> et <y()>.

Fonction Cette fonction remplit le polygone défini par ses arguments en utilisant la couleur de remplissage définie dans DEFFILL.

Exemple :

Voir POLYLINE

POLYLINE Affichage d'un polygone

Format POLYLINE <n>,<x()>,<y()> [OFFSET <ox>,<oy>]

où <n> est le nombre de côtés du polygone,

<x()> et <y()> sont des tableaux qui contiennent les coordonnées des côtés du polygone,

<ox> et <oy> lorsqu'ils sont spécifiés sont ajoutés aux coordonnées spécifiées dans les tableaux <x()> et <y()>.

Fonction Cette fonction affiche un polygone sur l'écran. Les coordonnées des coins du polygone sont définies dans les tableaux x() et y(). Un offset peut être ajouté à chacune des coordonnées de ces tableaux lorsque les paramètres ox et oy sont définis.

PRINT AT Positionnement puis affichage alphanumérique

Format PRINT AT (<col>,<lig>);<exp>

où <col> et <lig> sont les coordonnées du début de l'affichage,

et <exp> est l'expression alphanumérique à afficher.

Fonction Cette instruction positionne le curseur aux coordonnées spécifiées dans les deux premiers paramètres, puis réalise un affichage alphanumérique.

Exemple :

```
PRINT AT (10,4);"Bonjour"
```

PRINT USING Affichage alphanumérique formaté

Format PRINT USING <format>,<exp>

où <format> est le format d'affichage,

et <exp> l'expression à afficher.

Fonction Cette instruction formatte puis affiche des données alphanumériques.

La chaîne de formattage peut être composée d'un ou de plusieurs des éléments suivants :

#	Chiffre
.	Séparation entier/décimal
,	Séparation des milliers
-	Affichage du signe - pour les nombres négatifs
+	Affichage du signe + pour les nombres positifs
*	Remplacement des premiers 0 du nombre par des *
$	Affichage d'un $ devant le nombre
^	Format exponentiel
!	Premier caractère de la chaîne seulement
\N\	N+2 premiers caractères de la chaîne seulement

Exemple :

```
PRINT USING "$**###.##",1.12345
Résultat :
****$1.12
```



PRBOX Affichage d'un rectangle plein à coins arrondis

Format PRBOX <x1>,<y1>,<x2>,<y2>

où <x1> et <y1> sont les coordonnées du coin supérieur gauche du rectangle,

et <x2> et <y2> les coordonnées du coin inférieur droit du rectangle.

Fonction Cette instruction affiche un rectangle plein à coins arrondis sur l'écran. La couleur de remplissage doit avoir été précédemment définie avec une instruction DEFFILL.

Exemple :

```
DEFFILL 1,2,3
PRBOX 100,100,200,200
```

PRED Entier précédent

Format <res>=PRED(<nombre>)

où <nombre> est un nombre entier,

et <res> le prédécesseur de <nombre>.

Fonction Cette fonction renvoie le prédécesseur du nombre entier passé en argument.

Exemple :

```
a% = 1234
b% = PRED(a%)
PRINT b%
```

Résultat :

```
1233
```

PRINT Affichage alphanumérique sur l'écran

Format PRINT [<exp>]

où <exp> est l'expression alphanumérique à afficher.

Fonction Cette instruction réalise un affichage texte sur l'écran, à partir de la position occupée par le curseur.

Exemple :

```
PRINT              --> Affichage d'une ligne blanche
PRINT "Bonjour"    --> Affichage du texte "Bonjour"
PRINT 2*3          --> Affichage du résultat d'un calcul
```



PROCEDURE Définition d'une procédure

Format PROCEDURE <nom>[<par1>,...,<parN>]

où <nom> est le nom de la procédure,

et les <parI> sont des paramètres optionnels passés à la procédure.

Fonction Cette instruction permet de définir une procédure et les éventuels paramètres qui lui sont passés.

Exemple :

Définition d'une procédure demandant deux paramètres :

```
PROCEDURE essai(x,y)
  ...
RETURN
```

PSAVE Sauvegarde "anti list"

Format PSAVE <nom>

où <nom> est le nom de la sauvegarde.

Fonction Cette instruction sauvegarde le code source d'un programme dans un fichier dont le contenu ne pourra pas être listé. Cela permet de rendre confidentielles les instructions qui le composent.

Exemple :

```
PSAVE "essai"
```

PSET Affichage d'un point graphique coloré

Format PSET <x>,<y>,<c>

où <x> et <y> sont les coordonnées du point à afficher,

et <c> sa couleur.

Fonction Cette instruction affiche un point graphique (pixel) sur l'écran. La couleur du point est précisée dans le troisième paramètre.

Exemple :

Tracé d'une parabole en utilisant la couleur 2 :

```
FOR i=-20 TO 20 STEP 0.1
  PLOT 200+i*10,400-i^2,2
NEXT i
```

4/2.3

Le langage STOS par la pratique

Le STOS est un langage Basic assez étonnant, puisqu'il permet à des programmeurs quasiment débutants de créer leurs propres jeux (arcades, réflexion, pédagogiques, etc.).

Ce langage peut être utilisé pour créer des programmes autres que des jeux (comme nous le verrons par la suite), mais son domaine de prédilection est indéniablement ludique.

Plutôt que de décrire les instructions du langage, nous allons étudier plusieurs programmes qui mettent en œuvre :

— les accessoires (éditeurs de sprites, de musique, de caractères, d'icônes et compacteur d'écran) ;

— les instructions du langage ;

— des astuces de programmation.

Notre premier programme (DANGER BACTERIES) utilise :

— l'éditeur de musique pour créer puis jouer une musique de fond pendant que le jeu se déroule ;

— l'éditeur de sprites pour créer puis animer plusieurs sprites sur l'écran ;

— diverses instructions graphiques pour créer le décor du jeu ;

— les instructions relatives au joystick ;

— divers effets spéciaux d'écran.

4/2.3.1

L'éditeur du STOS - Généralités

Pour accéder au STOS, vous pouvez procéder de deux manières :

a) insérer la disquette LANGAGE dans le lecteur A: et booter l'ordinateur ;

b) double-cliquer sur l'icône BASIC205.PRG.

Quelle que soit la manière utilisée, le STOS se charge en mémoire, et vous vous trouvez dans l'éditeur après quelques secondes.

Les commandes affectées aux dix premières touches de fonction apparaissent dans une fenêtre en haut de l'écran. Pour afficher les commandes des fonctions <F11> à <F20>, il suffit d'appuyer sur une des deux touches <Shift>.

Nous retiendrons dans un premier temps :

— la touche <F2> qui permet de lister le programme ou une partie du programme en mémoire ;

— les touches <F4> et <F5> qui permettent respectivement de charger en mémoire un programme qui se trouve sur disquette et de sauvegarder sur disquette le programme qui se trouve en mémoire à l'aide d'un sélecteur de fichiers ;

— la touche <F6> qui permet d'exécuter le programme en mémoire ;

— la touche <F1> qui contient la dernière commande entrée.

Remarque :

Le signe en fin de commande (par exemple f6: run') représente la touche <Return>. Lorsqu'il est présent, il n'est pas nécessaire d'appuyer sur <Return> pour exécuter la commande correspondante.

COMMENT UTILISER LE CLAVIER

L'éditeur du STOS est de type LIGNE. C'est-à-dire que les instructions qui composent un programme sont mémorisées dans des lignes qui commencent obligatoirement par un nombre 10, 1000, etc. Par exemple :

```
10 print "Bonjour"
```

Les lignes du programme courant peuvent être listées à l'aide d'une commande LIST.

Lorsqu'une instruction n'est pas précédée d'un numéro de ligne, elle est exécutée immédiatement (on parle alors d'instruction exécutée en mode direct), et il n'est pas possible de la retrouver avec une commande LIST.

Les caractères tapés au clavier sont affichés sur l'écran à la position courante du curseur. Cette position peut être modifiée à l'aide des touches flèches et de la souris.

Par défaut, les caractères entrés au clavier se superposent aux éventuels caractères qui se trouvent sous le curseur. La touche <Insert> peut être utilisée pour passer en mode INSERTION. Dans ce cas, les caractères entrés s'inscrivent entre la position courante du curseur et le caractère à gauche du curseur.

Les touches <Backspace> et <Delete> permettent d'effacer (respectivement) le caractère à gauche du curseur ou sous le curseur.

La touche <Home> positionne le curseur en haut et à gauche de la fenêtre texte.

La touche <Clr> efface l'écran et positionne le curseur en haut et à gauche de la fenêtre texte.

Enfin, l'appui deux fois de suite sur la touche <Undo> réinitialise l'éditeur, efface l'écran et positionne le curseur en haut et à gauche de la fenêtre texte.

LES ACCESSOIRES

Outre le langage Basic STOS, il est également possible d'utiliser des accessoires, sortes de programmes résidants qui se chargent en parallèle du Basic et qui sont accessibles à l'aide de la touche <Help> du clavier.

Les accessoires sont des programmes écrits en Basic STOS. Pour charger un accessoire en mémoire, insérez la disquette contenant les accessoires décompactés dans le lecteur A:, tapez ACCLOAD suivi (entre guillemets) du nom de l'accessoire à charger, et appuyez sur <Return>.

Par exemple, pour charger les accessoires MUSIC et SPRITE, vous taperez, en mode direct :

```
accload "music"
accload "sprite"
```

Lorsqu'un accessoire est chargé en mémoire, vous pouvez l'exécuter en appuyant sur la touche <Help>, et en le sélectionnant dans la liste des accessoires chargés.

Par exemple, lorsque les accessoires MUSIC et SPRITE ont été chargés à l'aide d'instructions ACCLOAD, l'appui sur les touches <Help> puis <F1> exécutent l'accessoire MUSIC.

4/2.3.2

Utilisation de l'éditeur de musique (accessoire MUSIC)

Chargez l'accessoire MUSIC en procédant comme il a été dit plus haut (instruction **accload**) et activez l'éditeur de musique en appuyant sur les touches **<Help>** puis **<F1>**. L'écran de l'éditeur de musique apparaît au bout de quelques secondes.

Cet écran peut paraître complexe, mais rassurez-vous, il n'en est rien...

La partie haute de l'écran rassemble les options principales d'un menu accessible à la souris.

• Le menu **STOS** permet :

— de retourner au Basic en oubliant les données musicales entrées (Quitter) ;

— de retourner au Basic en mémorisant les données musicales entrées dans une banque mémoire (Quitter/Récupérer).

• Le menu **BANQUE** renferme toutes les options permettant de charger et de sauvegarder des données musicales. Pour commencer, nous nous intéresserons surtout aux options :

— charger une banque : lecture et mémorisation d'une banque mémoire (c'est-à-dire d'un ou de plusieurs morceaux précédemment saisis à l'aide de l'éditeur de musique) ;

— sauver la banque : sauvegarde sur disquette de la banque qui se trouve en mémoire ;

— sauver comme ... : sauvegarde sur disquette sous un nom différent de la banque qui se trouve en mémoire.

• Le menu **MUSIQUE** permet de manipuler les données musicales qui se trouvent en mémoire. Nous retiendrons essentiellement les options suivantes :

— **NOUVELLE** musique : définition du nom d'un nouveau morceau ;

— **PRENDRE** musique : extraction d'un morceau de la banque musicale en mémoire ;

— **STOCKER** musique : stockage des données musicales courantes dans la banque musicale courante ;

— **STOCKER et JOUER** : mémorisation d'un des morceaux en mémoire dans une zone spéciale destinée à son interprétation et activation du morceau ;

— **IMPRIMER** musique : impression des notes qui se trouvent en mémoire.

• Le menu **BLOC** permet de créer des blocs de notes. Ce menu est intéressant lorsqu'un groupe de notes se répète plusieurs fois dans un même morceau. L'option

Copier bloc

permet alors de copier une ou plusieurs fois le bloc de notes délimité par les options

Début bloc

et

Fin bloc

• Enfin, le menu **OUTILS** permet de choisir une enveloppe ou un trémolo.

A titre d'exercice, nous vous proposons d'entrer les données suivantes dans la voix 1. Ces données représentent les premières notes du dessin animé « La Panthère rose » :

```
 1 : ENVEL 8
 2 : VOLUME 15
 3 : DO#     1 DC
 4 : RE      1 NR.
 5 : MI      1 DC
 6 : FA      1 NR
 7 : DO#     1 DC
 8 : RE      1 SC.
 9 : MI      1 DC
10 : FA      1 SC.
11 : LA#     1 DC
12 : LA      1 SC.
13 : RE      1 DC
14 : FA      1 SC.
15 : LA      1 DC
16 : SOL#    1 NR.
17 : SOL     1 DC
18 : FA      1 DC
19 : RE      1 DC
20 : DO      1 DC
21 : RE      1 BL
```

Remarques :

• Dans l'éditeur de musique,

— les notes sont entrées en clair (DO, SOL, ...) ;

— les octaves sont codées entre 0 et 7 (0 est très grave, 7 est très aigu) ;

— les durées sont codées comme suit :

Codage	Note
DC	Double croche (1/4 temps)
SC	Croche (1/2 temps)
NR	Noire (1 temps)
BL	Blanche (2 temps)
RD	Ronde (4 temps)



Toutes les notes peuvent être pointées pour augmenter leur durée de 50 %. Par exemple, une blanche pointée (BL.) aura une durée de trois temps. Le tableau ci-dessous donne les diverses durées possibles et le codage correspondant.

Codage	Note
DC.	Double croche pointée (3/8 temps)
SC.	Croche pointée (3/4 temps)
NR.	Noire pointée (1 temps 1/2)
BL.	Blanche pointée (3 temps)
RD.	Ronde pointée (6 temps)

Toutes les données musicales étant entrées, vous êtes certainement impatient d'entendre le résultat. Sélectionnez l'option **"STOCKER et JOUER"** du menu **MUSIQUE**. Une fenêtre comportant 31 entrées est affichée. Cliquez sur l'entrée 1. Un message vous demande de patienter quelques instants, puis l'écran relatif à l'exécution du morceau apparaît.

Sur cet écran :

— le **TEMPO**, c'est-à-dire la vitesse d'exécution du morceau peut être modifiée à l'aide des touches :

+ pour augmenter la vitesse d'exécution,
– pour diminuer la vitesse d'exécution.

— la **TRANSPOSITION**, c'est-à-dire la fréquence des notes jouées peut être modifiée à l'aide des touches :

/ pour jouer plus grave,
* pour jouer plus aigu.

— la touche **S** permet de réentendre le morceau (lorsque celui-ci ne boucle pas indéfiniment).

— la touche **Escape** permet de retourner à l'éditeur de musique.

— enfin, la touche **Espace** permet d'exécuter le morceau note par note.

Faites des essais de réglage de TEMPO et de TRANSPOSITION. Pour notre part, nous avons jugé satisfaisant le **TEMPO 34** et la **TRANSPOSITION 0**.

Nous allons compléter ce morceau par un mouvement d'horloge répétitif dont le but est de symboliser le temps qui passe. Ce morceau sera créé à l'aide de bruits. Entrez les données musicales suivantes à la suite du morceau :

```
22 : ENVEL 4
23 : BRUIT 15
24 : REPETER 0,25
25 : MI 3 NR
26 : MI 2 NR
```

Remarques :

• L'instruction **BRUIT** (ligne 23) permet d'accéder au générateur de bruit pour générer des souffles.

• L'instruction **REPETER** (ligne 24) repète indéfiniment les notes situées à partir de la ligne 25 (ici MI 3 NR et MI 2 NR).

Procédez comme précédemment pour stocker et jouer le morceau (option **STOCKER et JOUER** du menu MUSIQUE).

Sauvegardez le morceau complet sur disquette en sélectionnant l'option **"Sauver la banque"** du menu BANQUE.

Repassez au STOS en sélectionnant l'option **"Quitter/récupérer"** du menu STOS, et en appuyant sur le bouton **CONFIRMER** avec la souris.

Appuyez sur la touche de fonction **<F3>** ou tapez **listbank** en mode direct.

Le STOS affiche une information du type :

Banques mémoire réservées :
3 music S:$1C7900 E:1C7C00 L:$000300

Les données musicales sont bien en mémoire !

Pour vous en persuader, tapez en mode direct :

music 1

Le message **OK** vous signale que l'instruction a été exécutée. Le morceau saisi dans l'éditeur de musique s'exécute ... alors que vous avez accès au Basic. Cela vous laisse entrevoir une des fantastiques possibilités du STOS : l'exécution multitâche.

Remarques :

• Si une banque musicale est composée de plusieurs morceaux, l'instruction **music N** permet d'activer le morceau N.

• Une musique en cours d'exécution peut être suspendue à l'aide d'une instruction **music freeze** et interrompue à l'aide d'une instruction **music off**.

• Une musique interrompue par une instruction **music freeze** peut reprendre à l'endroit où elle a été interrompue à l'aide d'une instruction **music on**.



• La vitesse d'exécution d'un morceau peut être modifiée pendant ou avant son exécution à l'aide d'une instruction tempo en précisant la nouvelle vitesse : **entre 1** (très lent) et **100** (très rapide).

Exemple : **tempo 50**

• La fréquence des notes émises peut être modifiée pendant ou avant l'exécution d'un morceau à l'aide d'une instruction **transpose** en précisant la fréquence à ajouter (−90 à 90).

Exemple : **transpose −10**

• Encore quelques détails

Les mélomanes auront sans doute remarqué que, jusqu'à présent, nous avons passé sous silence les divers temps morts (pauses, silence, etc.) que l'on rencontre fréquemment dans les morceaux. En STOS, il est très simple de marquer un temps d'arrêt. Il suffit d'utiliser l'instruction PA suivie de l'octave (qui n'a aucune signification ici) et de la durée de l'arrêt.

Exemples : **PA 0 SC → arrêt : 1/2 temps**
PA 0 BL → arrêt : 2 temps

Lorsque vous aurez une certaine habitude du fonctionnement de l'éditeur de musique, vous apprécierez que ses concepteurs aient prévu des facilités de déplacements rapides, d'insertion et de suppression dans les voix :

<F1> : page précédente ;

<F2> : page suivante ;

<F3> : première note de la voix courante ;

<F4> : dernière note de la voix courante ;

<Insert> : insertion d'une note à la position du curseur ;

<Delete> : effacement de la note courante.

© Copyright

Atari 520 ST, 1040 ST, MEGA ST2, MEGA ST4 : Atari Corp.
Basic GFA : GFA
Basic ST : Atari Corp.
Calcomat 2 : Micro Application
Cyber studio : Upgrade
Datamat : Micro Application
Degas Elite : Electronics Arts
GEMDOS : Digital Research Inc.
IBM : IBM Corporation
K-RAM : Kuma Software
K-Ressource : Kuma
GST Holdings Limited
Timeworks Publisher
Lattice C : Lattice
LDW Power : Upgrade
Le Rédacteur : Libération
Lotus 1.2.3 : Lotus
MAD MAC : Atari Corp.
Motorola : Motorola
Mutil : Michtron
Turbo C : Borland International Inc.
Spectrum 512 : Upgrade
Wercs : Hisoft
ZZ lazy paint : Human Technologies

Tous droits de traduction, d'adaptation et de reproduction de la version française par tous procédés réservés pour tous pays

La loi du 11 mars 1957 n'autorisant, aux termes des alinéas 2 et 3 de l'article 41, d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale, ou partielle, faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause, est illicite » (alinéa 1er de l'article 40).

Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles 425 et suivants du Code pénal.

7e Complément

© 1990 Editions WEKA, 82, rue Curial, 75935 Paris cedex 19
Tél. : (1) 40 37 01 00 - Télex 210 504

Editions WEKA - Paris - Kissing - Zurich - Milan - Amsterdam - Vienne - Londres - New York

En Suisse : WEKA Verlag, Hermetschloostr. 77, 8010 Zurich

Editeur : Brigitte Morvant
Secrétariat d'édition : Brigitte Chevallier
Fabrication : Tina Lê Xuân

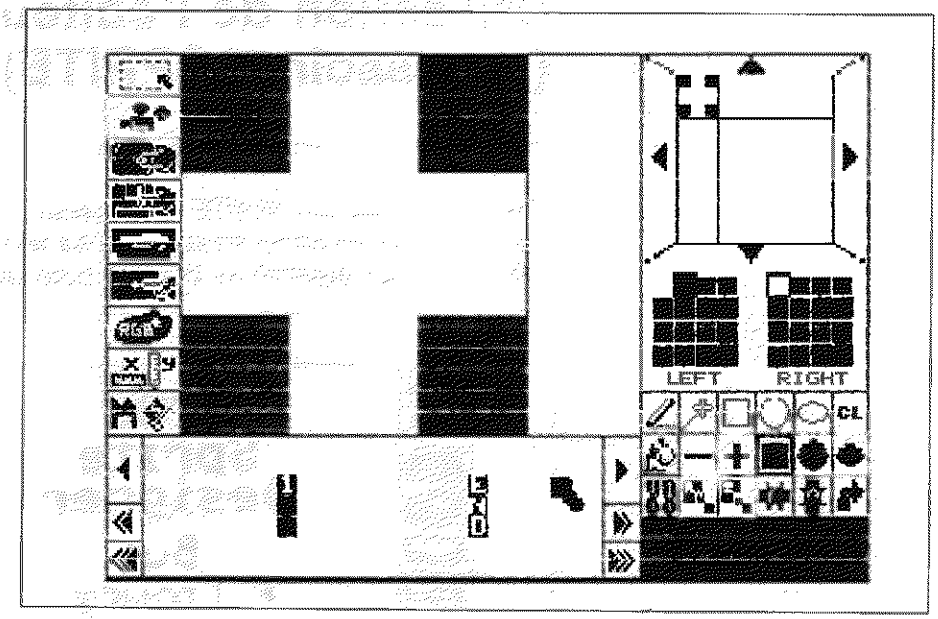
Composition : Compo-Méca - 64990 Mouguerre
Corlet imprimeur s.a. - 14110 Condé-sur-Noireau
Reproduction interdite - Tous droits réservés
Imprimé en France, 1990

Dépôt légal (ouvrage de base) : 2e trimestre 1989

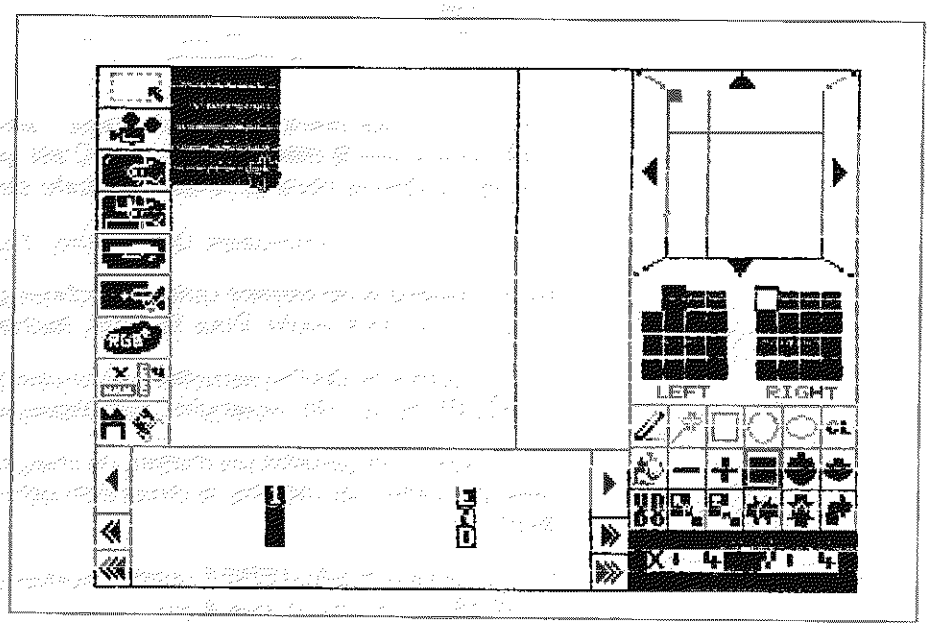
ISBN 2-7337-0048-0

DÉFINITION DU PREMIER SPRITE

Le premier sprite est une croix rouge dans un carré blanc :



Ce sprite a une dimension de 16 points sur 16. Avant de le définir, modifiez la dimension du sprite (partie centrale de l'écran) en cliquant sur l'icône **XY** qui se trouve dans la partie gauche de l'écran, puis dans la fenêtre située en haut et à droite de l'écran jusqu'à ce que la dimension du sprite soit **X = 16, Y = 16.** Appuyez sur le bouton droit de la souris pour revenir à l'écran précédent.

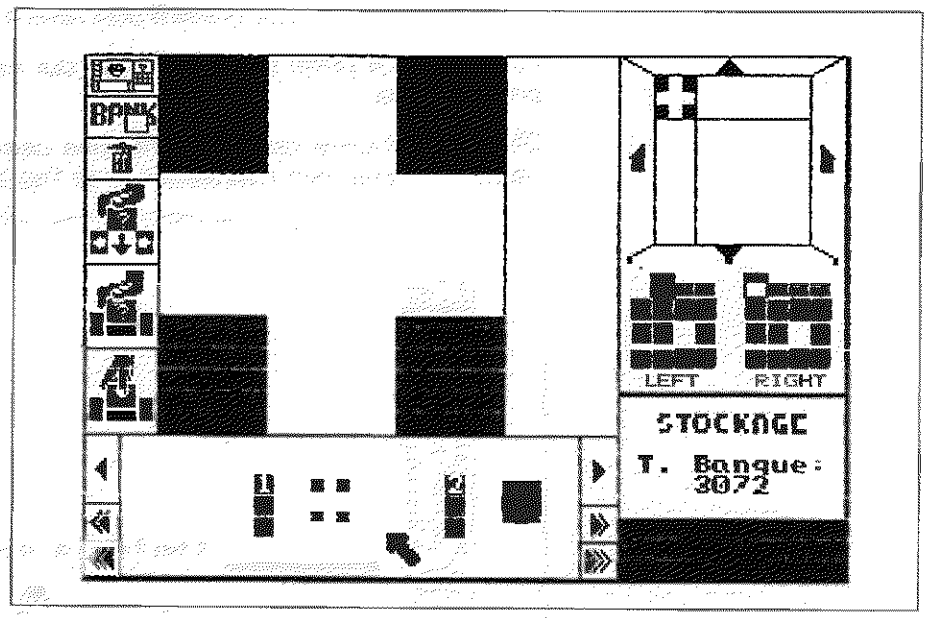


Passons à la définition du sprite.

1) Choisissez la couleur blanche en cliquant sur la seconde couleur du premier bloc de couleurs (LEFT).

2) Cliquez sur l'icône qui représente un carré plein dans la partie droite de l'écran. Le curseur de la souris change d'aspect. Positionnez-le dans le coin supérieur gauche de la partie réservée à la définition du sprite. Maintenez le bouton gauche de la souris enfoncé tout en déplaçant la souris vers le bas et la droite jusqu'à ce que les coordonnées de la souris soient égales à X: 4 Y: 4

Si vous déplacez la souris, le carré blanc suit ces mouvements. Positionnez-le en haut et à gauche de la zone de définition et cliquez. Le carré blanc reste affiché. Déplacez la souris pour dupliquer le carré blanc trois fois :



La partie la plus difficile a été faite. Pour remplir la partie centrale du sprite en rouge, il suffit :

1) de sélectionner la couleur rouge en cliquant sur la 11e couleur du bloc LEFT ;

2) de cliquer sur l'icône qui représente un pot de peinture dans la partie droite de l'écran (le curseur de la souris change d'aspect) ;

3) de cliquer dans la partie centrale du sprite.

Le sprite étant maintenant défini, stockez-le en mémoire en cliquant :

— sur le dernier icône de la partie gauche de l'écran ;

— sur le quatrième icône de la nouvelle série ;

— sur le bouton droit de la souris pour revenir à la série d'icônes précédente.

La seconde image du premier sprite est très proche de la première. La croix rouge est bleue. Pour construire cette image, il vous suffit :

— de sélectionner la couleur bleue en cliquant sur la 14e couleur du bloc LEFT ;

— de cliquer sur l'icône qui représente un pot de peinture dans la partie droite de l'écran (le curseur de la souris change d'aspect) ;

— de cliquer dans la partie centrale du sprite.

Stockez la sprite en mémoire en cliquant (comme précédemment) :

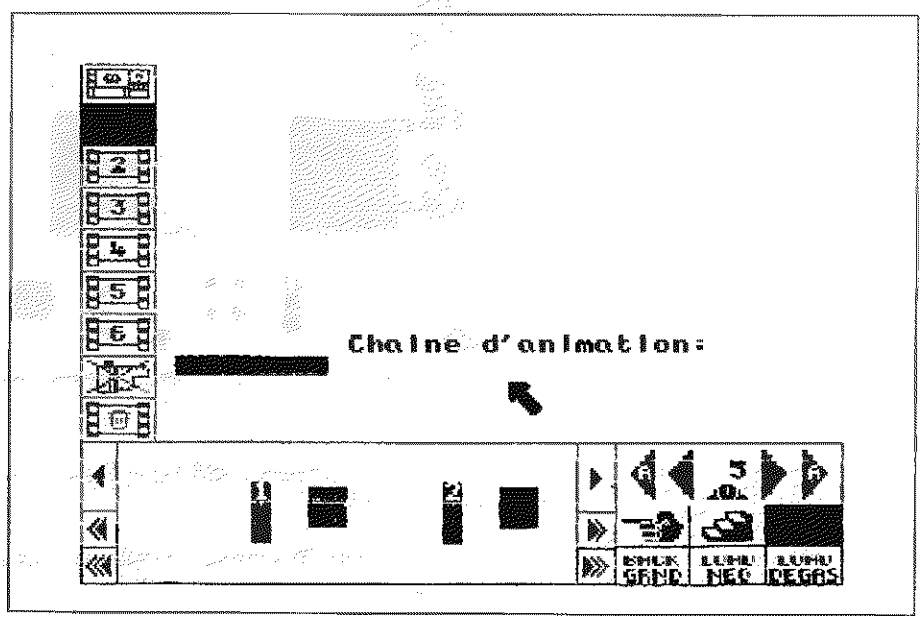
— sur le dernier icône de la partie gauche de l'écran ;

— sur le quatrième icône de la nouvelle série ;

— sur le bouton droit de la souris pour revenir à la série d'icônes précédente.

Pour faire une petite pause après ce dur exercice, nous allons animer le premier sprite.

Cliquez sur l'icône qui représente une caméra (en haut et à gauche de l'écran). L'écran doit maintenant avoir l'apparence suivante :



Cliquez sur le premier dessin (en bas de l'écran). Au besoin amenez le premier dessin dans la fenêtre en cliquant sur la triple flèche vers la gauche. Un couplet de deux nombres apparaît sur l'écran, en-dessous de la chaîne d'animation :

(1, 5)

Le premier nombre représente le numéro du sprite, et le second la durée d'affichage de ce sprite.

Cliquez sur le second sprite. La chaîne d'animation devient :

(1, 5) (2 , 5)

et, oh merveille, le sprite est animé...

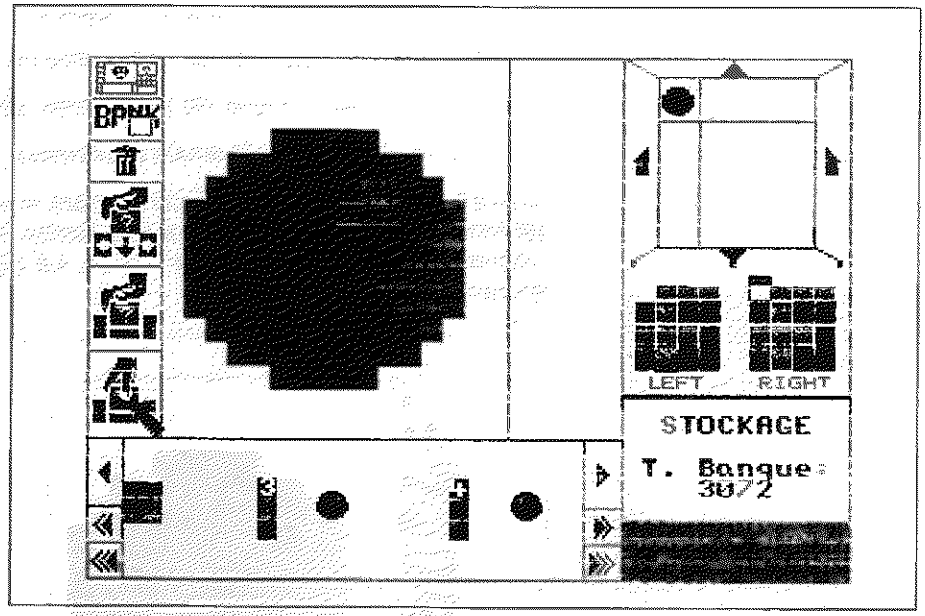
Vous pouvez modifier la vitesse du clignotement en cliquant sur les icônes A dans la partie droite de l'écran.

Pour retourner à l'écran principal de l'éditeur de sprites, il suffit d'appuyer sur le bouton droit de la souris.

DÉFINITION DU SECOND SPRITE

Le second sprite utilisé dans le jeu DANGER BACTERIES représente un pac-man qui se déplace en ouvrant et en fermant la bouche et les yeux (2 images par direction). Le nombre de directions possibles étant égal à quatre, il faudra définir 8 images. Mais rassurez-vous, lorsque vous aurez défini la première image, le travail sera presque terminé.

Comme vous le voyez sur cette copie d'écran :



Le pacman a une forme circulaire. Il est défini à l'aide de trois couleurs :

— œil bleu (14e couleur) ;

— bouche marron (6e couleur) ;

— tête jaune (5e couleur).

Pour définir la tête, procédez comme suit :

1) cliquez sur la 5e couleur dans le bloc LEFT ;

2) cliquez sur l'icône cercle plein dans la partie droite de l'écran ;

3) appuyez sur le bouton gauche de la souris et déplacez-la jusqu'à ce que le cercle ait la dimension requise ;

4) si sa dimension ne vous convient pas, il est toujours possible d'appuyer sur le bouton droit avant de le placer. Placez le cercle sur l'écran en cliquant.

Pour définir la bouche, procédez comme suit :

1) cliquez sur la 6e couleur du bloc LEFT ;

2) cliquez sur l'icône qui représente un crayon dans la partie droite de l'écran ;

3) cliquez autant de fois que nécessaire pour obtenir le dessin de la bouche tel qu'il figure dans la copie d'écran précédente.

Enfin, pour définir l'œil :

1) cliquez sur la 14e couleur dans le bloc LEFT ;

2) cliquez autant de fois que nécessaire pour obtenir le dessin de l'œil tel qu'il figure dans la copie d'écran précédente.

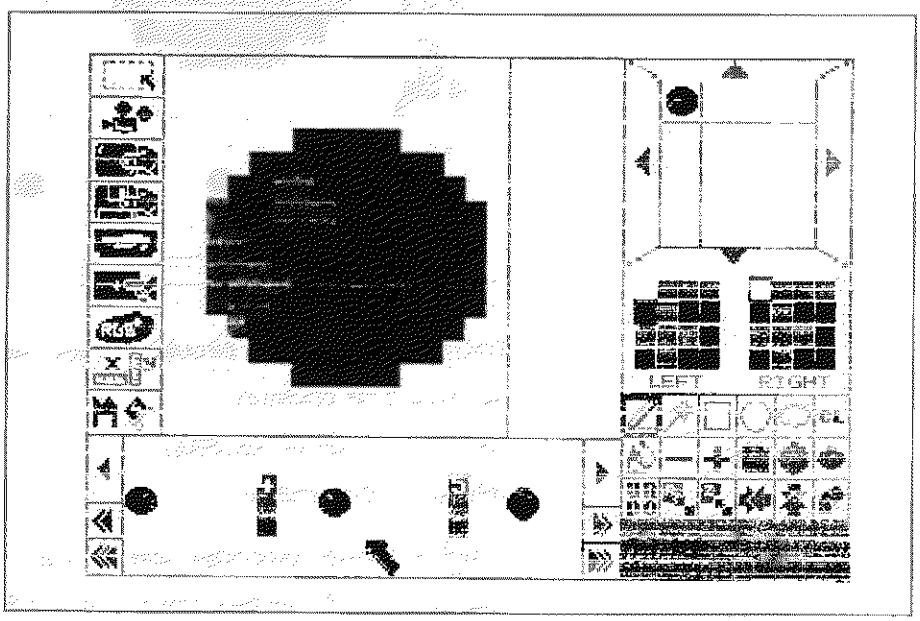
Stockez le sprite en mémoire en cliquant :

— sur le dernier icône de la partie gauche de l'écran ;

— sur le quatrième icône de la nouvelle série ;

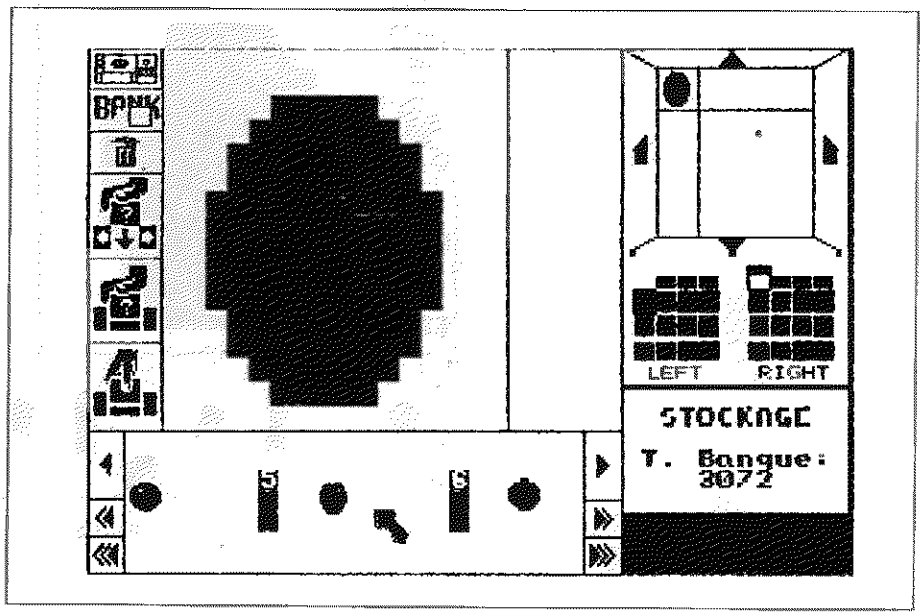
— sur le bouton droit de la souris pour revenir à la série d'icônes précédente.

Arrivé à ce point, votre travail concernant le second sprite est pratiquement terminé. Cliquez sur l'icône qui représente deux flèches horizontales tête-bêche dans la partie droite de l'écran. Le dessin affiché est maintenant le suivant :



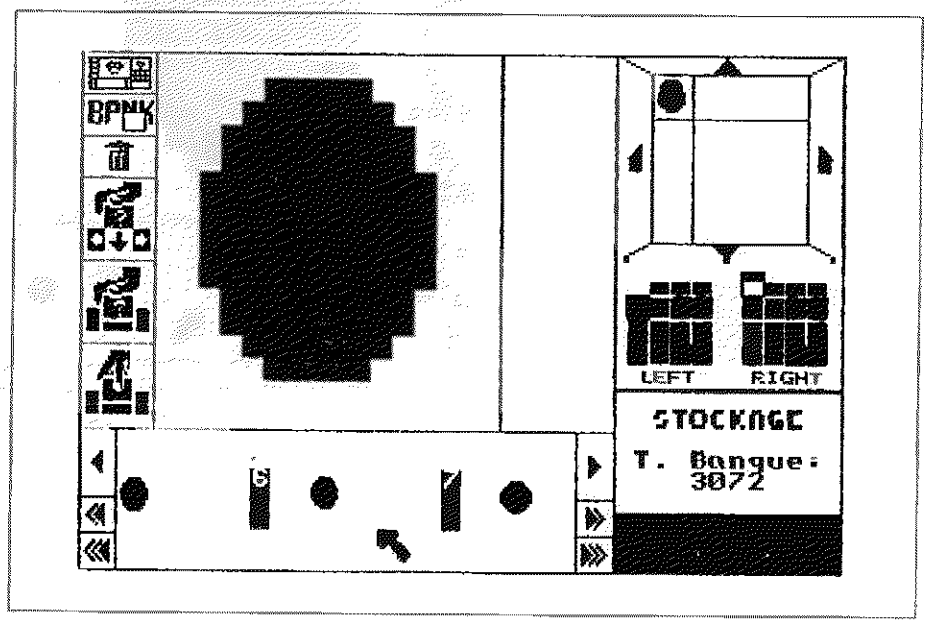
Stockez le sprite en mémoire selon le procédé habituel.

Cliquez trois fois de suite sur l'icône qui représente une flèche verticale qui bifurque vers la droite dans la partie droite de l'écran. Le dessin affiché est maintenant le suivant :

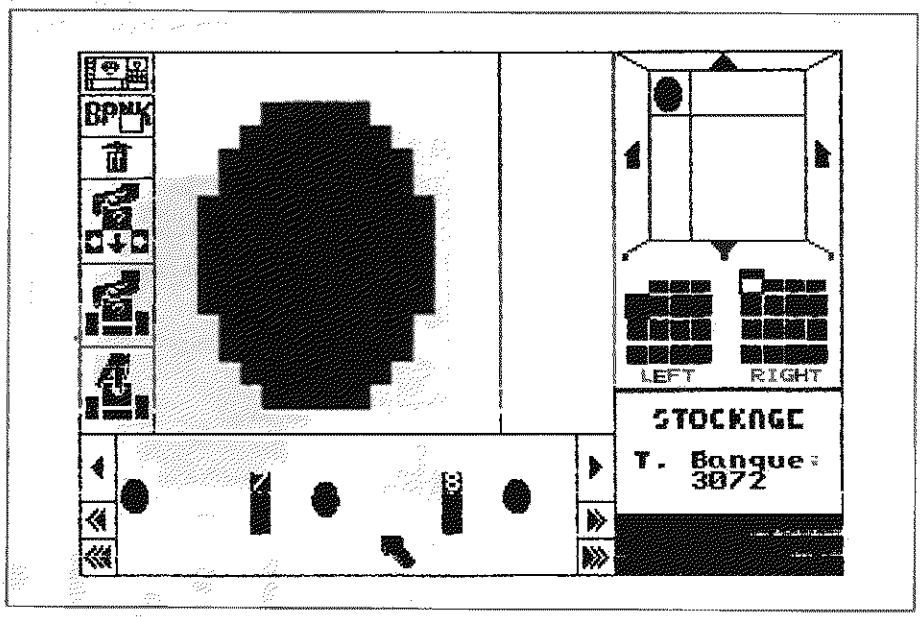


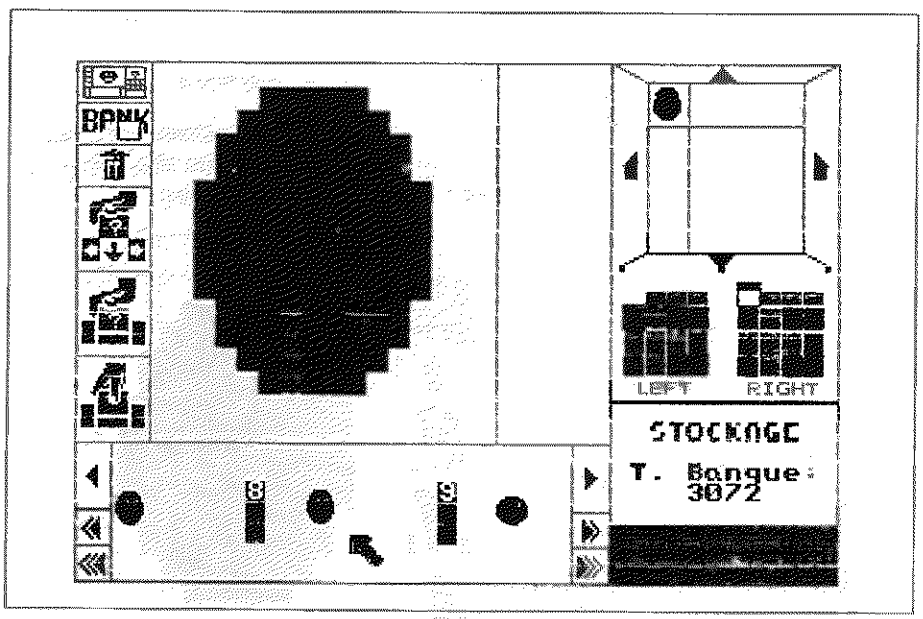
Stockez le sprite en mémoire.

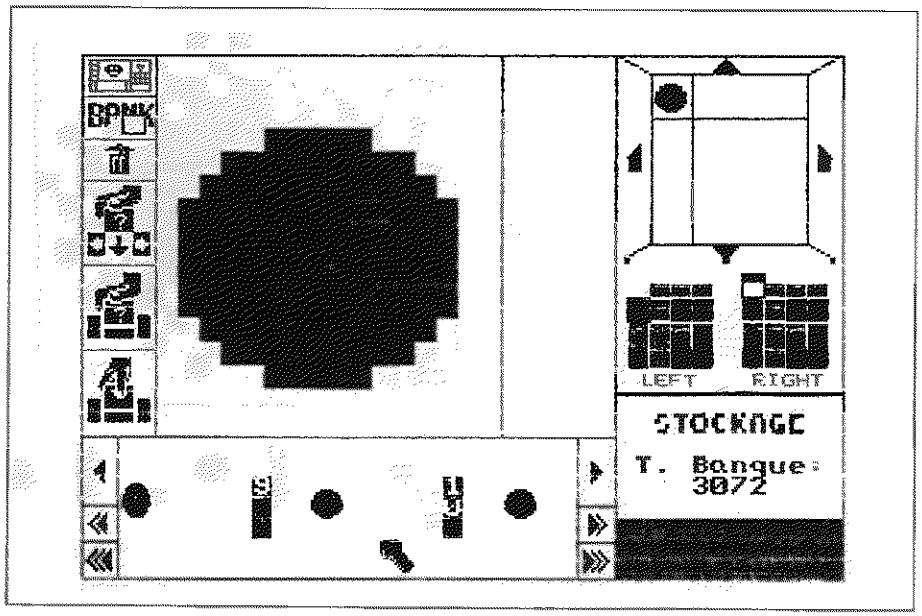
Définissez le symétrique du dessin précédent en cliquant sur l'icône qui représente deux flèches verticales tête-bêche dans la partie droite de l'écran et stockez le résultat en mémoire :

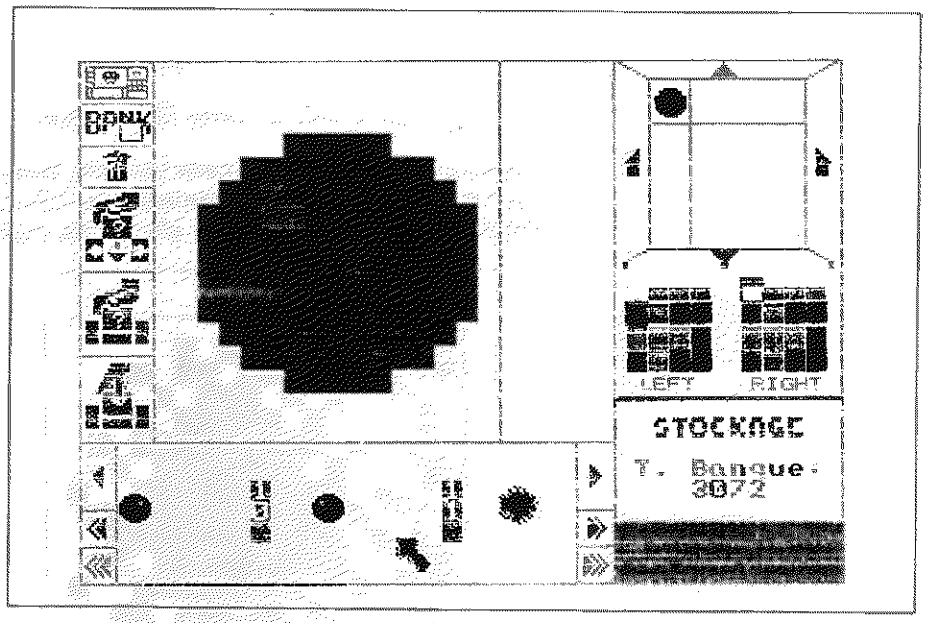


Utilisez les trois icônes dont nous venons de parler pour définir les autres formes du sprite :



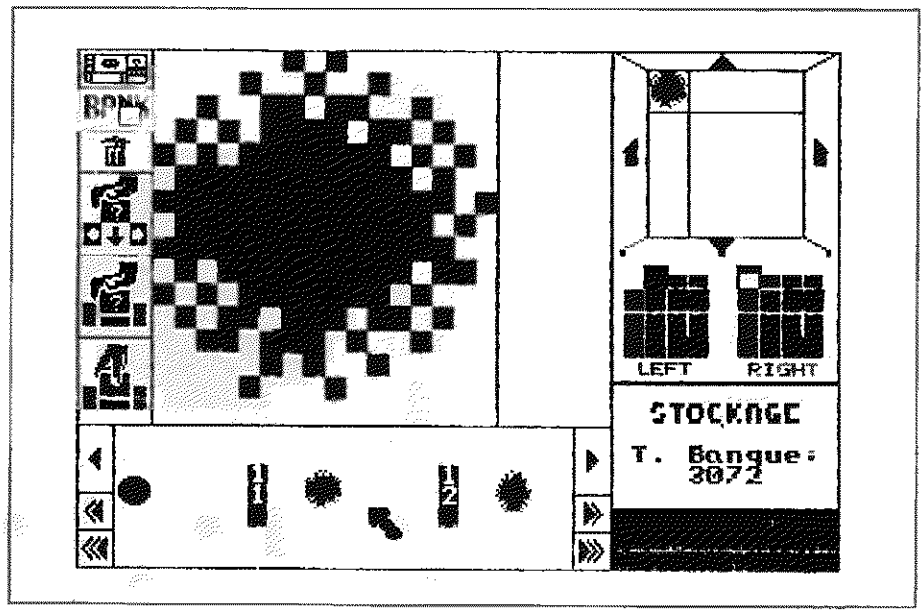






DÉFINITION DU TROISIÈME SPRITE

Le troisième sprite utilisé dans le jeu DANGER BACTERIES représente une bactérie qui se déplace en agitant ses tentacules (4 images). Comme vous le voyez sur cette copie d'écran :

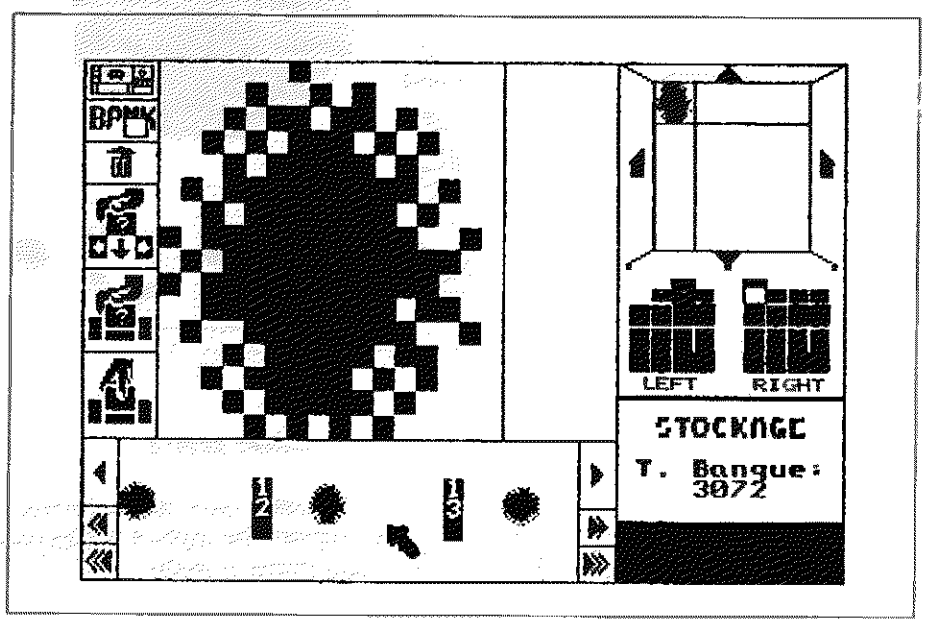


La bactérie a une forme vaguement circulaire. Elle est définie à l'aide de deux couleurs :

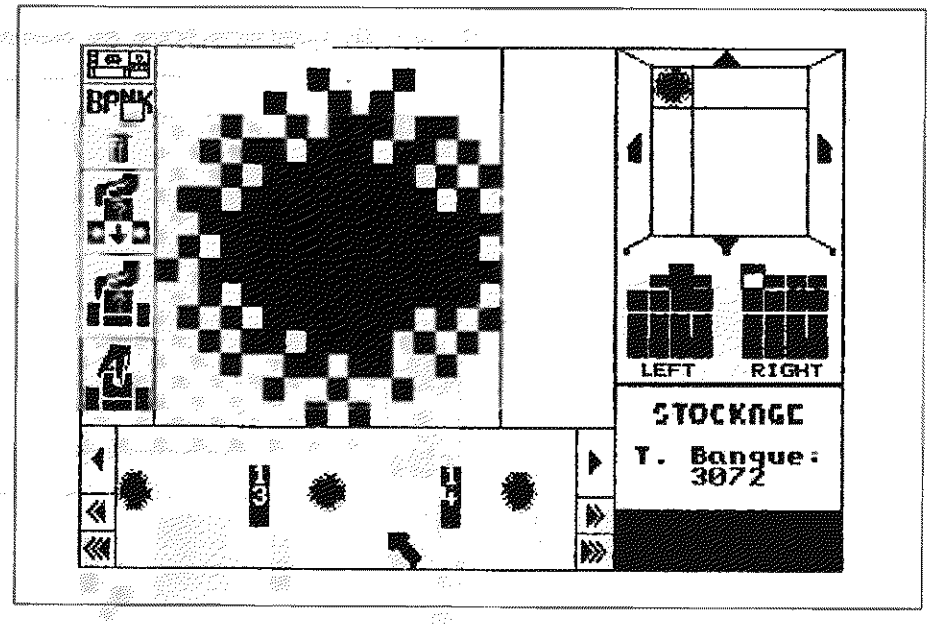
— partie centrale rose (15e couleur) ;

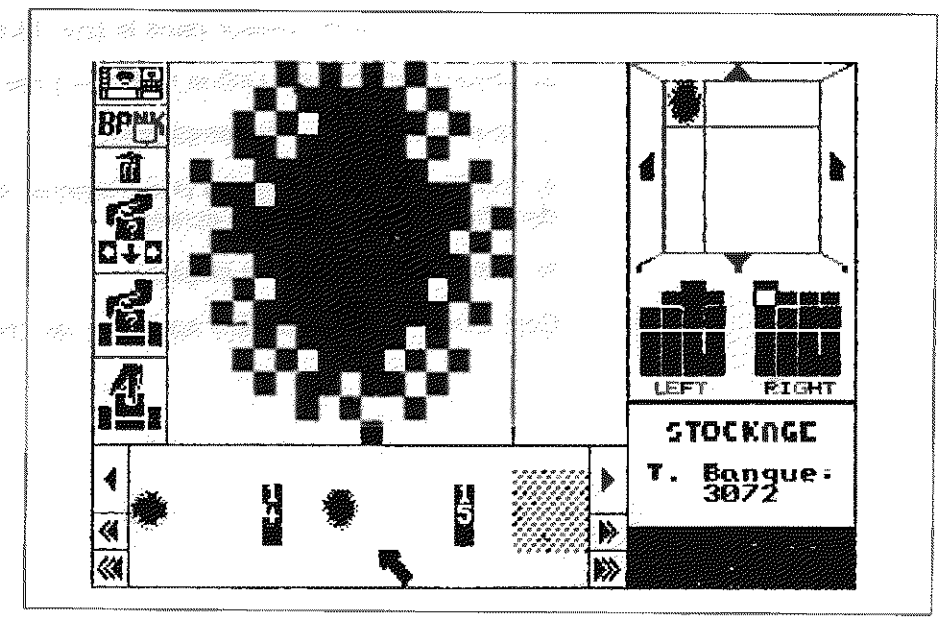
— partie périphérique verte (3e couleur).

Lorsque vous aurez défini cette bactérie, faites-la tourner en cliquant sur l'icône qui représente une flèche verticale qui bifurque vers la droite dans la partie droite de l'écran. Le dessin affiché est maintenant le suivant :



Recommencez deux fois cette manipulation pour obtenir les sprites suivants :

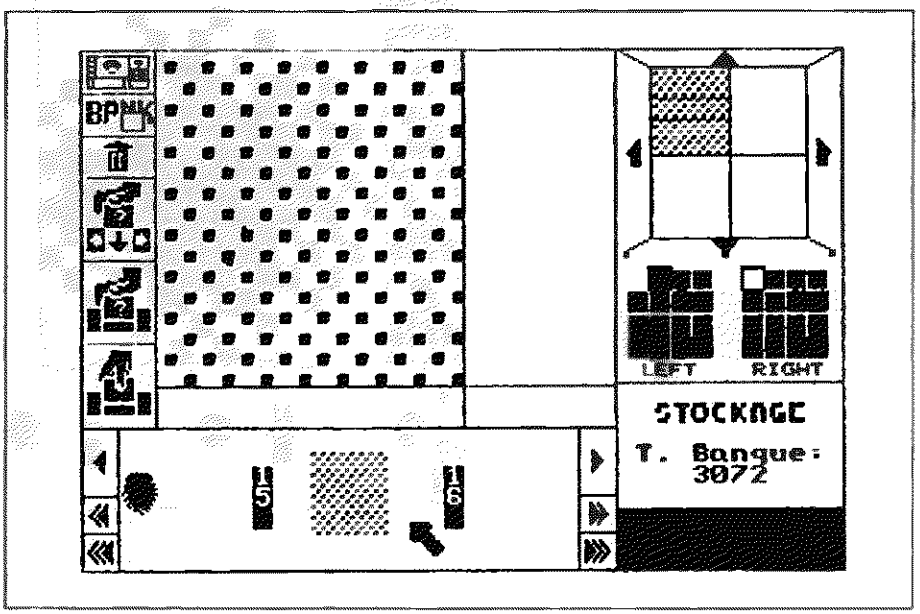




Encore un peu de courage. Nous allons définir le dernier sprite du jeu qui sera utilisé par le programme pour constituer le décor.

DÉFINITION DU QUATRIÈME SPRITE

La définition du quatrième sprite est élémentaire.



1) cliquez sur la 8e couleur dans le bloc LEFT ;

2) cliquez sept fois sur l'icône moins (–) dans la partie droite de l'écran ;

3) cliquez sur l'icône de remplissage ;

4) modifiez la taille du sprite pour l'amener à 32 points sur 32 (icône XY dans la partie gauche de l'écran) ;

5) remplissez le sprite en cliquant.

Ceci termine notre première approche de l'éditeur de sprites.

4/2.3.4

Le programme DANGER BACTERIES

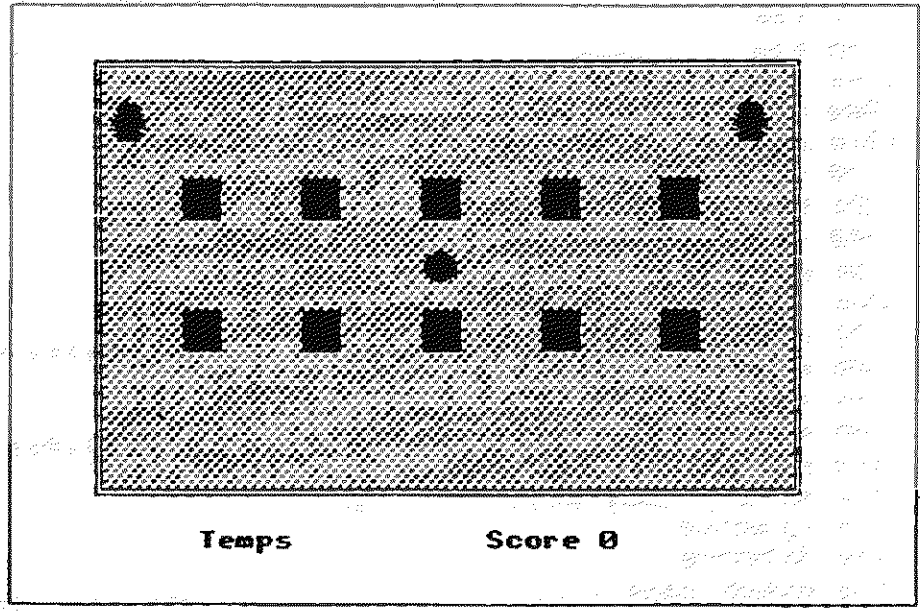
Un glouton dirigé au joystick essaye de manger tout ce qui se trouve sur son passage. Tout, sauf ... deux bactéries qui sont à sa poursuite. Vingt secondes seulement sont données au glouton pour accomplir sa triste besogne. Trois cas peuvent se présenter :

— tous les objets sont mangés en moins de 20 secondes : la partie est gagnée ;

— le temps imparti est écoulé et tous les objets ne sont pas mangés : la partie est perdue ;

— une bactérie touche le glouton : la partie est perdue.

Voici une copie de l'écran de jeu :



COMMENT UTILISER LE PROGRAMME

Entrez le listing suivant :

```
1000 rem ==========================
1010 rem Danger bactéries
1020 rem ==========================
1030 rem
1040 rem ------------------------------
1050 rem Initialisations diverses
1060 rem ------------------------------
1070 rem
1080 clear
1090 dim X(10),Y(10)
1100 key off
1110 mode 0
1120 hide
1130 curs off
1140 click off
1150 SC=0 : rem Score au début de la partie
1160 music 1 : rem activation de la musique de fond
1170 rem
1180 rem ----------------
1190 rem Tracé du décor
1200 rem ----------------
1210 rem
1220 box 15,7 to 305,168
1230 box 13,5 to 307,170
1240 sprite 1,16,8,15
1250 wait 2
1260 for I=1 to 4
1270 screen copy logic,16,8,48,40 to logic,16,32*I+8
1280 next I
1290 for I=1 to 8
1300 screen copy logic,16,8,48,168 to logic,32*I+16,8
1310 next I
1320 screen copy physic to back
1330 timer=0
1340 TEMPS=0
1350 gosub 2350 : rem Initialisation des paramètres de la
partie
1360 locate 7,23
1370 print "Temps           Score";SC
1380 NAR=10 : rem Nombre d'objets à ramasser
1390 rem
1400 rem ------------------------------
1410 rem Affichage des divers sprites
1420 rem ------------------------------
1430 rem
1440 for I=1 to 10
1450 sprite I,X(I),Y(I),1
```



(suite)

```
1460 anim I,"(1,10)(2,10)L"
1470 next I
1480 sprite 11,150,75,3
1490 sprite 12,20,20,11
1500 anim 12,"(11,5)(12,5)(13,5)(14,5)L"
1510 sprite 13,280,20,11
1520 anim 13,"(11,5)(12,5)(13,5)(14,5)L"
1530 anim on
1540 rem


1550 rem ------------------------------
1560 rem Boucle principale du jeu
1570 rem ------------------------------
1580 rem
1590 repeat
1600 rem
1610 rem ----------------------
1620 rem Affichage du temps
1630 rem ----------------------
1640 rem
1650 ZZ=int(timer/50) : if TT<>ZZ then locate 13,23 : print
21-ZZ;" " : TT=ZZ
1660 rem
1670 rem ------------------------------------
1680 rem Test du joystick et déplacement
1690 rem ------------------------------------
1700 rem
1710 if jright and D=0 then move y 11,"(3,0,1)" : move x 11,
"(3,2,1)L" : move on : anim 11,"(3,10)(9,10)L" : anim on : D=1
1720 if jleft and G=0 then move y 11,"(3,0,1)" : move x 11,
"(3,-2,1)L" : move on : anim 11,"(4,10)(10,10)L" : anim on :
G=1
1730 if jup and H=0 then move x 11,"(3,0,1)" : move y 11,
"(3,-2,1)L" : move on : anim 11,"(5,10)(7,10)L" : anim on :
H=1
1740 if jdown and B=0 then move x 11,"(3,0,1)" : move y 11,
"(3,2,1)L" : move on : anim 11,"(6,10)(8,10)L" : anim on : B=1
1750 if jright and x sprite(11)>290 then move freeze (11)
1760 if jleft and x sprite(11)<15 then move freeze (11)
1770 if jup and y sprite(11)<6 then move freeze (11)
1780 if jdown and y sprite(11)>154 then move freeze (11)
1790 if joy=0 then G=0 : D=0 : H=0 : B=0 : move freeze (11) :
 anim freeze (11)
1800 rem
1810 rem ------------------------------------
1820 rem Test de collision avec une croix
1830 rem ------------------------------------
1840 rem
```

(suite)

```
1850 A=collide(11,10,10)
1860 C=0
1870 for I=0 to 10
1880 if btst(I,A)=-1 then C=I
1890 next I
1900 if (C<>0) and (C<11) then sprite C,-100,0,0 : NAR=NAR-1 :
 SC=SC+10 : locate 27,23 : print SC
1910 rem
1920 rem ----------------------------
1930 rem Déplacement des bactéries
1940 rem ----------------------------
1950 rem
1960 X11=x sprite(11) : Y11=y sprite(11)
1970 X12=x sprite(12) : Y12=y sprite(12)
1980 X13=x sprite(13) : Y13=y sprite(13)
1990 if X11>X13 then IX2#=IX2#+0.2
2000 if X11<X13 then IX2#=IX2#-0.2
2010 if Y11<Y13 then IY2#=IY2#-0.2
2020 if Y11>Y13 then IY2#=IY2#+0.2
2030 if X11>X12 then IX#=IX#+0.2
2040 if X11<X12 then IX#=IX#-0.2
2050 if Y11<Y12 then IY#=IY#-0.2
2060 if Y11>Y12 then IY#=IY#+0.2
2070 if IX#=0.4 then IX#=0 : X12=X12+1
2080 if IX#=-0.4 then IX#=0 : X12=X12-1
2090 if IY#=0.4 then IY#=0 : Y12=Y12+1
2100 if IY#=-0.4 then IY#=0 : Y12=Y12-1
2110 sprite 12,X12,Y12
2120 move on 12
2130 if IX2#=0.4 then IX2#=0 : X13=X13+2
2140 if IX2#=-0.4 then IX2#=0 : X13=X13-2
2150 if IY2#=0.4 then IY2#=0 : Y13=Y13+2
2160 if IY2#=-0.4 then IY2#=0 : Y13=Y13-2
2170 sprite 13,X13,Y13
2180 move on 13
2190 until (21-ZZ)=0 or (NAR=0) or btst(12,A)=-1 or
btst(13,A)=-1
2200 rem
2210 rem ----------------------
2220 rem Fin de la partie
2230 rem ----------------------
2240 rem
2250 if (21-ZZ)=0 or btst(12,A)=-1 or btst(13,A)=-1 then fade
25
2260 if (NAR=0) then fade
5,$777,$777,$777,$777,$777,$777,$777,
$777,$777,$777,$777,$777,$777,$777,$777,$777
2270 locate 1,1
2280 move freeze 11 : move freeze 12
```



(suite)

```
2290 wait 100
2300 default
2310 if NAR=0 then print "Bravo."
2320 if (21-ZZ)=0 then print "Le temps imparti est écoulé."
2330 if btst(12,A)=-1 or btst(13,A)=-1 then print "Une
bactérie
a eu raison de vous."
2340 end
2350 rem
2360 rem ------------------------------------------------
2370 rem Initialisation des paramètres de la partie
2380 rem ------------------------------------------------
2390 rem
2400 for I=0 to 4
2410 for J=0 to 1
2420 X(J+I*2+1)=50+I*50
2430 Y(J+I*2+1)=50+J*50
2440 next J
2450 next I
2460 return
```

Chargez les deux banques (sprite et musique) en tapant en mode direct :

```
    LOAD "esprit"
et  LOAD "muse"
```

Ces deux commandes supposent que les banques ont pour nom **esprit.mbk** et **muse.mbk**.

Sauvegardez le programme sur disquette. Les deux banques seront également sauvegardées, ce qui évitera d'avoir à les charger à chaque exécution du programme.

Lancez le programme en appuyant sur la touche de fonction <F6> ou en tapant **run** en mode direct.

Le joystick vous permet de diriger le glouton dans les quatre directions (haut, bas, droite et gauche). Dévorez le plus d'objets possibles en évitant les bactéries et en observant le compteur de temps en bas de l'écran. Bonne chance...

LE PROGAMME EN DÉTAIL

La logique du programme apparaît dans l'ordinogramme de la page suivante.

Le programme débute par diverses initialisations.

Les tableaux **X** et **Y** sont dimensionnés. Ces tableaux contiendront les coordonnées des objets qui doivent être dévorés par le glouton :

```
1090 dim X(10), Y(10)
```

L'affichage des touches de fonction en haut de l'écran est dévalidé :

```
1100 key off
```

Le mode d'affichage est initialisé en basse résolution :

```
1110 mode 0
```

Les curseurs (souris et clavier) sont dévalidés en 1130.

Le cliquetis du clavier est inhibé pour éviter toute interaction entre la musique de fond et une éventuelle action au clavier (ligne 1140).

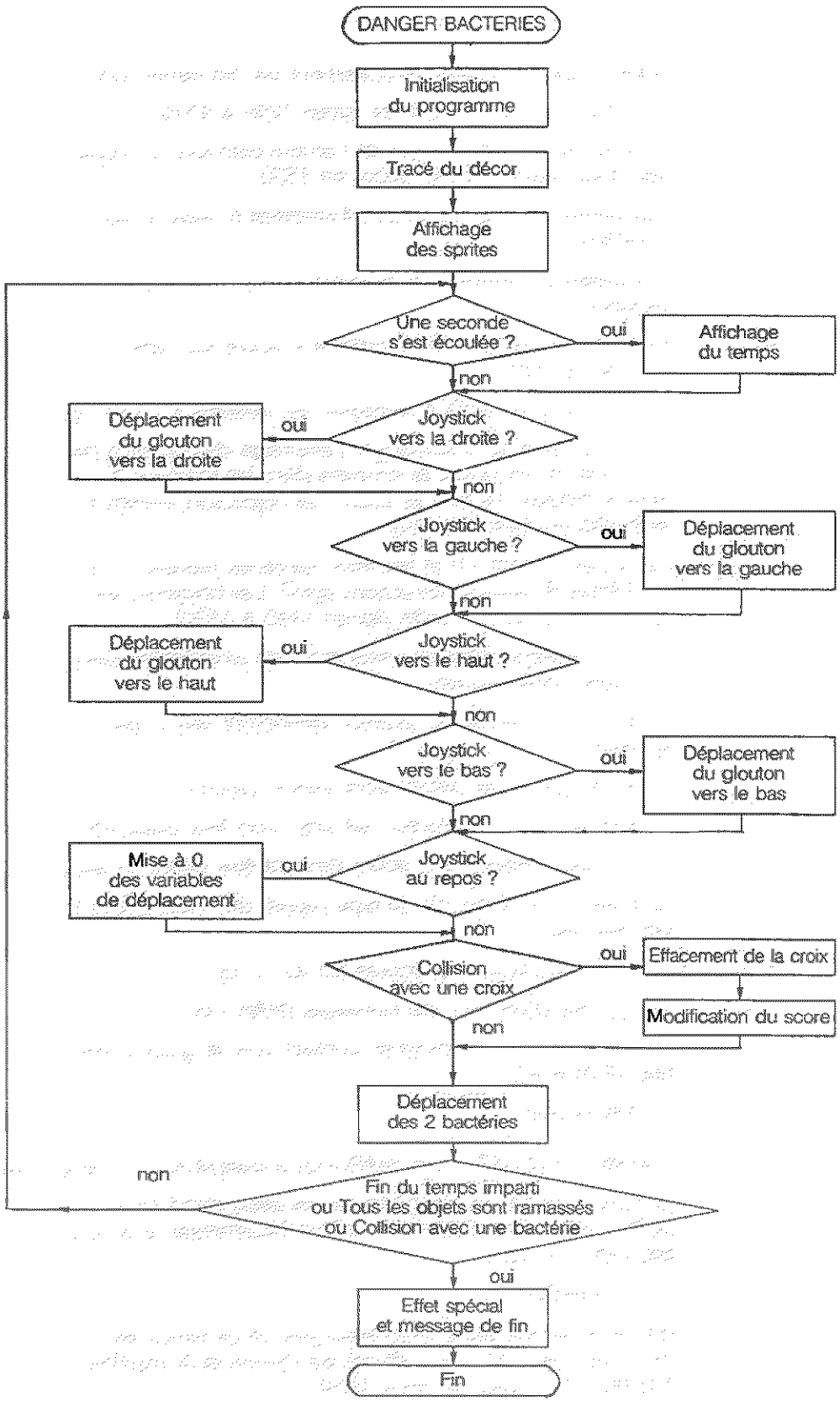
La partie d'initialisation se termine par le déclenchement de la musique de fond, sans modification des paramètres vitesse et transposition (ligne 1160).

Le programme se poursuit par l'affichage de l'arrière-plan (décor du jeu).

L'arrière-plan est délimité par deux cadres tracés à l'aide d'instructions **box** (lignes 1220 et 1230).

Il est créé à partir du sprite 16 qui est dupliqué autant de fois que nécessaire. Le sprite 16 est affiché en haut et à gauche du cadre (ligne 1240).

Une instruction **wait** lui donne le temps d'apparaître sur l'écran (ligne 1250).


DANGER BACTERIES
Initialisation du programme
Tracé du décor
Affichage des sprites
Une seconde s'est écoulée ?
oui
Affichage du temps
non
Déplacement du glouton vers la droite
oui
Joystick vers la droite ?
non
Joystick vers la gauche ?
oui
Déplacement du glouton vers la gauche
non
Déplacement du glouton vers le haut
oui
Joystick vers le haut ?
non
Joystick vers le bas ?
oui
Déplacement du glouton vers le bas
non
Mise à 0 des variables de déplacement
oui
Joystick au repos ?
non
Collision avec une croix
oui
Effacement de la croix
Modification du score
non
Déplacement des 2 bactéries
non
Fin du temps imparti ou Tous les objets sont ramassés ou Collision avec une bactérie
oui
Effet spécial et message de fin
Fin

Il est ensuite dupliqué verticalement par les lignes 1260 à 1280,

puis horizontalement par les lignes 1290 à 1310.

Pour éviter tout effacement de l'arrière-plan lors du déplacement des sprites, il est copié dans le décor en 1320.

Les paramètres de la partie sont initialisés à l'aide du sous-programme situé en 2350.

Cette astuce permettra de compliquer le jeu par la suite en créant plusieurs niveaux.

Le temps avant la fin de la partie et le score sont affichés en bas de l'écran en 1360 et 1370,

et le nombre d'objets à ramasser est initialisé à 10 en ligne 1380.

Le programme se poursuit par l'affichage des objets à dévorer. Les coordonnées de ces objets se trouvent dans les tableaux X et Y initialisés dans le sous-programme situé en 2350. Ces objets sont animés à l'aide d'instruction anim (lignes 1440 à 1470).

Le glouton (sprite 11) et les deux bactéries (sprites 12 et 13) sont affichés sur l'écran à l'aide d'instructions sprite. Les bactéries sont animées à l'aide de deux instructions anim (lignes 1480 à 1530).

La partie suivante est la plus importante du programme. Elle contient la boucle principale dans laquelle :

— les mouvements du joystick donneront lieu à des déplacements du glouton ;

— les bactéries se déplaceront vers le glouton ;

— tout passage du glouton sur une croix fera disparaître la croix ;

— le temps restant et le score devront être gérés en temps réel.

La boucle principale est du type repeat until. Elle prend fin dans les trois cas suivants :

— le temps imparti est écoulé (21-ZZ = 0) ;

— tous les objets ont été ramassés (NAR = 0) ;

— une bactérie est entrée en contact avec le glouton (btst(12,A) = −1 or btst(13,A) = −1) :

```
1590 repeat
...
2190 until (21-ZZ)=0 or (NAR=O) or btst(12,A)=-1 or btst(13,A)=-1
```

Le temps imparti est affiché toutes les secondes à l'aide d'une astuce logicielle. La variable système timer est incrémentée d'un cinquante fois par seconde. Le calcul

```
int(timer/50)
```

donne un temps exprimé en secondes. Si ce temps est différent du temps mémorisé dans TT, il est affiché sur l'écran et la variable TT est initialisée à la nouvelle valeur en ligne 1650.



Le programme se poursuit par le test du joystick. Les variables système **jright, jleft, jup** et **jdown** ont pour valeur −1 lorsque le joystick est déplacé (respectivement) vers la droite, la gauche, le haut et le bas.

Les variables **D, G, H** et **B** sont initialisées à 1 lorsque le joystick est déplacé vers (respectivement) la droite, la gauche, le haut et le bas.

Lorsque le joystick est déplacé vers la droite (**jright**) et lorsqu'il était au repos ou déplacé dans une autre direction (D=0), le glouton doit être déplacé vers la droite :

— l'éventuel mouvement en **Y** est stoppé à l'aide d'une instruction **move y** ;

— le déplacement en **X** vers la droite est activé à l'aide d'une instruction **move x** ;

— le glouton est animé avec une instruction **anim** ;

— la variable **D** est mise à 1 pour mémoriser le déplacement (ligne 1710).

Des blocs d'instructions du même type réalisent le déplacement du glouton vers la gauche, le haut et le bas (lignes 1720 à 1740).

Les instructions suivantes testent si le glouton sort des limites de l'aire de jeu.

Si le joystick est orienté vers la droite alors que l'abscisse du glouton est supérieure à 290, le déplacement est figé en ligne 1750.

Trois instructions du même type figent les déplacements vers la gauche, le haut et le bas lorsque le glouton sort des limites de l'aire de jeu (lignes 1760 à 1780).

Lorsque le joystick est au repos, les variables **G, D, H** et **B** sont mises à 0 pour mémoriser l'inaction, et tout mouvement et animation du glouton est stoppé en 1790.

Le programme se poursuit par un ensemble de tests qui détermine si le glouton est entré en contact avec un des objets à dévorer. L'instruction collide est utilisée. Les paramètres qui lui sont transmis sont :

— le sprite dont on désire tester la collision ;

— le nombre de points (en **X** et en **Y**) autour du point chaud à partir desquels le programme doit considérer qu'il y a collision en 1850.

Cette fonction renvoie une donnée entière dont le bit **N** vaut 1 dans le cas où le sprite testé est entré en collision avec le sprite **N**. Un boucle **for next** teste les 10 collisions possibles (lignes 1860 à 1890).

Lorsqu'une collision a eu lieu, le sprite en question est effacé de l'écran et le score est augmenté de 10 en ligne 1900.

Le programme se poursuit par le déplacement calculé des bactéries.

Les coordonnées des sprites 11 (glouton), 12 (bactérie 1) et 13 (bactérie 2) sont mémorisées dans les variables **X11, Y11, X12, Y12, X13** et **Y13** (lignes 1960 à 1980).

Lorsque l'abscisse de la bactérie 2 est inférieure à l'abscisse du glouton, la variable **IX2#** est incrémenté de 0.2 en ligne 1990.

De même, lorsque l'abscisse de la bactérie 2 est supérieure à l'abscisse du glouton, la variable **IX2#** est décrémentée de 0.2 en ligne 2000.

Deux instructions du même type incrémentent ou décrémentent la variable **IY#** en fonction de l'ordonnée relative du glouton et de la bactérie 2 (lignes 2010 et 2020).

Quatre instructions similaires gèrent le déplacement de la bactérie 1 (lignes 2030 à 2060).

Lorsque la variable **IX#** vaut 0.4, c'est-à-dire lorsqu'elle a été incrémentée deux fois, l'abscisse de la bactérie 1 est incrémentée en ligne 2070.

L'ordonnée de la bactérie 1 est décrémentée lorsque **IX#** vaut − 0.4 en ligne 2080.

L'ordonnée de la bactérie est gérée de la même manière (lignes 2090 et 2100).

Lorsque les coordonnées de la bactérie 2 ont été modifiées, la bactérie est déplacée à l'aide d'une instruction sprite en ligne 2110.

Pour augmenter l'attrait du jeu, la bactérie 2 se déplace deux fois plus vite que la bactérie 1. Cela se traduit au niveau des variables **X13** et **Y13** qui sont incrémentées/décrémentées de 2 pour la bactérie 2 (elles étaient incrémentées/décrémentées de 1 pour la bactérie 1) (lignes 2130 à 2170).

Lorsque la partie est terminée (temps écoulé, plus d'objets ou glouton dévoré par une bactérie) en ligne 2190.

Une effet spécial d'écran est activé, en fonction de l'issue de la partie.

Lorsque la partie est perdue, une instruction fade éteint progressivement l'écran en 2250.

Lorsque la partie est gagnée, une autre instruction fade fait progressivement virer toutes les couleurs affichées au blanc (ligne 2260).

Tous les mouvements de sprites sont arrêtés en 2280,
et un message qui dépend de l'issue de la partie est affiché sur l'écran après le temps nécessaire pour l'exécution de l'instruction fade (lignes 2290 à 2330).

Le programme se termine par le sous-programme d'initialisation des paramètres de la partie.

Les coordonnées des objets à dévorer sont calculées à l'intérieur de deux boucles **fort** imbriquées (lignes 2400 à 2450).

Comme nous l'avons déjà précisé plus haut, cette technique d'initialisation est attrayante car elle permet de créer des niveaux dans le jeu.

4/2.4

Le Basic OMIKRON

PRÉSENTATION GÉNÉRALE

Le Basic OMIKRON est l'un des basics les plus rapides et les plus puissants qui existent sur Atari. Après la présentation générale de ce langage, nous en effectuerons une comparaison avec le GFA qui est le principal autre Basic évolué disponible sur le marché actuellement.

COMPATIBILITÉ

Dès le départ, le point intéressant de ce Basic est sa compatibilité avec le Microsoft Basic qui est, avec le GW Basic, une norme quasi standard sur la plupart des ordinateurs. Cette possibilité permet de reprendre presque instruction par instruction la majorité des programmes écrits en Basic traditionnel. Quelques différences essentielles sont cependant à noter.

LOG

La fonction LN calcule le logarithme népérien (de base e). Dans la majorité des basics, cette fonction s'appelle LOG. En Basic OMIKRON, la fonction LOG permet de calculer le logarithme d'une variable dans n'importe quelle base.

Exemple : A=LOG(B,C)
A sera égal au logarithme de base B et C.
Si B=10 et C=10, A=LOG(10,10) prendra la valeur 1.

Les limites de l'instruction sont les mêmes que pour les autres basics puisqu'elles dépendent uniquement de la définition mathématique de la fonction logarithme, à savoir B et C positifs et différents de zéro.

MKD$ et MKS$

La représentation des nombres est plus précise que dans les basics classiques. L'opération de conversion des réels et des entiers codés en simple ou double précision en chaînes de caractères demandera donc plus de place mémoire. Pour exemple, un réel double précision est codé sur 10 octets, rien à voir avec les basics traditionnels !

Les instructions concernées sont la série des MKS$ et MKD$ ainsi que leurs fonctions inverses CVS et CVD. Il faut surtout faire attention à cette différence lorsqu'on utilise l'instruction FIELD pour les transferts de variable normale vers variable tampon au cours des lectures/écritures sur les fichiers relatifs, mais nous en verrons un exemple plus tard.

DATA

Les valeurs de caractères (autres que des valeurs numériques) doivent être écrites entre guillemets. Sans les guillemets, la variable est interprétée comme une valeur numérique.

Exemple : DATA Weka
affecte à la variable correspondante la valeur comprise dans la variable Weka.
DATA "WEKA"
affecte à la variable correspondante la chaîne de caractères Weka.

Dans le premier cas, la variable devra être de type réel et dans le second cas de type chaîne de caractère (variable ou tableau dans chacun des cas) pour que l'affectation puisse s'effectuer sans voir apparaître le message d'erreur 'CONFUSION DE TYPE'.

RANDOMIZE N'existe pas en OMIKRON.

RND(-X) Donne une valeur réelle comprise entre 0 et −1 et non la valeur initiale du générateur aléatoire. Par contre RND(0) donne la dernière valeur aléatoire générée.

WIDTH N'existe pas en OMIKRON.

ERASE N'existe pas en OMIKRON. On dimensionne ou redimensionne (ce qui est exceptionnel pour un Basic) un tableau avec DIM. Attention tout de même lorsque des tableaux à plusieurs dimensions se mêlent de la partie, le problème est un peu délicat et nous en verrons un exemple ultérieurement.

CHAIN...ALL Cette instruction n'existe pas pour chaîner les variables à transmettre. Il faut toutes les taper. Ne pas confondre avec l'instruction CHAIN de l'OMIKRON qui permet le chaînage des programmes et le passage d'informations d'un programme à l'autre. Pour passer des paramètres d'un programme à l'autre, il faut utiliser l'instruction COMMON comme en Basic Microsoft. A la différence de ce dernier langage, COMMON doit contenir les mêmes noms de variable dans les deux programmes.

PRINT USING Les caractères de contrôle sont très proches de ceux utilisés dans le GFA. Le point d'interrogation (?), standard d'abréviation de PRINT dans presque tous les basics est aussi un raccourci clavier en OMIKRON.

CLOSE En OMIKRON, cette instruction ferme les fichiers spécifiés. Lorsque le numéro de canal d'entrée sortie n'est pas spécifié (voir Basic GFA), CLOSE ferme tous les fichiers ouverts, après avoir sauvegardé toutes les données qui se trouvaient dans le buffer.

LES ÉDITEURS DE L'OMIKRON

L'éditeur standard

Dès le chargement de l'OMIKRON, on se trouve dans le premier éditeur dont un des modes de fonctionnement correspond au mode direct du GFA. Cet éditeur est donc plutôt adapté à la mise au point des programmes et à la réalisation rapide des calculs. En inscrivant des numéros de ligne devant les instructions, on fonctionne en mode programme et sinon (sans numéro de ligne) on fonctionne en mode direct.

- *En mode direct*, on peut faire des tests instruction par instruction, presque en pas à pas. L'interprétation de l'instruction se fait immédiatement après



l'appui de la touche **RETURN** (ne pas oublier de positionner le curseur en fin de ligne).

• *Dans le mode programme*, on doit rentrer les instructions de façon séquentielle et en les numérotant. L'instruction **RUN** exécute le programme. C'est l'option classique des basics.

Les options de listage **(LIST)** et modification du texte à l'écran (insertion et recouvrement par la touche **CTRL-Insert** sont classiques. Les sauvegardes peuvent s'effectuer :

— soit en code **FIT** *(Fast Interpreting Technic)*, code semi-compilé de l'OMIKRON qui lui confère sa rapidité ;

— soit en code **ASCII**, ce qui permet le portage sur d'autres basics. Il faut noter que les opérations de chargement et sauvegarde dans ce cas seront nettement plus lentes, car l'OMIKRON devra pré-compiler le programme instruction par instruction.

En prenant la précaution d'utiliser des numéros de lignes différents, il est possible de merger deux programmes (concatener deux programmes en un seul).

En tapant **'EDIT'** ou en frappant la touche **HELP**, vous pouvez passer au super éditeur de l'OMIKRON, qui combine les fonctions classiques d'un éditeur activable par le menu, le clavier ou la souris, ainsi que certaines fonctions qu'on avait l'habitude de trouver dans un traitement de texte, ce qui en fait un outil peu commun pour la mise au point des programmes.

Le super éditeur

Le menu déroulant du super éditeur présente six titres que nous allons explorer rapidement.

Titre FILE (Fichiers)

Sous ce titre on trouve les options classiques de tous les menus de langages de programmation qui permettent le chargement et la sauvegarde des fichiers. La sauvegarde en ASCII se fait dans les options qui traitent des blocks. Il faut évidemment avoir pensé à sélectionner un block auparavant car l'éditeur ne le signale pas par un **'ping'** retentissant. L'option **LOAD BLOCK** permet de merger le block appelé au programme actuellement en mémoire. Le block chargé reste activé après le chargement, ce qui est bien utile si on a commis une erreur sur l'emplacement de l'insertion. Il suffit de le déplacer avec l'option **INSERT** du titre **BLOCK**. L'option **DIRECTORY** permet l'affichage à l'écran du répertoire du lecteur pris par défaut. L'option **QUIT**, au contraire du Basic GFA, ne provoque pas le retour au bureau, mais rend la main à l'éditeur standard.

Titre FIND (chercher)

Les options de ce titre sont celles qu'on trouve dans un traitement de texte dans la recherche et le traitement simplifié des chaînes de caractères. A la différence de l'option équivalente en GFA, n'importe quelle chaîne peut être trouvée, remplacée, à la demande ou de façon globale dans tout le programme et ce à une vitesse étonnante. On peut aussi lister d'un coup toutes les occurrences d'une chaîne. Ces dernières apparaissent alors en inverse vidéo sur un écran spécial. Il devient facile ensuite de modifier la ligne qui nous intéresse. On peut aussi effectuer un traitement parallèle sur les mots clefs du langage en sélectionnant les options **TOKEN**. Renommer ces der-

niers est faisable avec l'option RENAME TOKEN. FIND ERROR est équivalent à l'option TEST du GFA. Le curseur se positionne le cas échéant juste dernière l'erreur de syntaxe (autre que l'erreur de premier niveau de syntaxe qui, elle, est détectée après la validation de chaque instruction par RETURN). Un 'ping' retentit si votre programme ne comporte aucune erreur.

Titre BLOCK

Sous ce titre on trouve les mêmes possibilités qu'en GFA. La saisie du block peut se faire par le curseur ou la souris.

La notion de block est cependant comparable à celle qu'on trouve dans un traitement de texte et non pas limitée à la ligne instruction du GFA. Il est vrai que l'OMIKRON a une structure d'instruction qui permet de considérer par moments qu'on peut presque mettre plusieurs instructions (certaines uniquement) sur une même ligne en les séparant par deux points (:). Il est donc assez normal que l'éditeur propose un traitement analogue aux éditeurs de programme en C, même si, dans ce langage, il n'y a structurellement aucun empêchement à écrire plusieurs instructions quelconques sur une même ligne.

Titre MODE

Les options de ce titre permettent la sélection des modes insertion ou recouvrement, le partage de l'écran en deux fenêtres, les options de choix de taille des caractères, activation des numéros de ligne et d'affichage des erreurs. Cette configuration peut être sauvegardée avec le programme, ce qui permet par exemple de recharger directement votre programme avec un multi fenêtrage et une taille de caractères présélectionnée.

Titre GO

Cette option relève encore d'un traitement d'édition de texte assez performant. On peut positionner rapidement le curseur en début ou fin de programme ainsi que sur un numéro de ligne quelconque. On peut aussi positionner quatre marqueurs dans le programme et aller au marqueur correspondant. Ces options ne figurent pas dans les éditeurs classiques des autres basics.

Titre RUN

Sous ce titre, on trouve aussi des options uniques qui permettent de faire exécuter le programme en mémoire, de le sauver et l'exécuter, de mettre en route le mode TRACE et lancer le programme en mémoire, de le compiler directement depuis l'interpréteur sans quitter l'OMIKRON, de le sauver et le compiler, d'exécuter un programme basic (.BAS) sur disquette ou d'exécuter un programme (.PRG) sur disquette.

Attention pour ces deux dernières options, le lancement d'un programme déjà compilé (.PRG) se trouvant sur disquette rend la main au super éditeur à condition de respecter les normes précises d'Atari en ce qui concerne le GEM. C'est presque une option de switcher simplifié qui est possible ici. Le lancement d'un programme basic (.BAS) se trouvant sur un disque, revient à charger et lancer un nouveau programme et donc à effacer le programme actuel résidant en mémoire.

LE RESET ET LES ERREURS

Une innovation importante et unique du Basic OMIKRON est le détournement du Reset. Au lieu de rebooter la machine, le Reset retourne dans le super éditeur. Cette possibilité est on ne peut plus utile au cours de la mise au



point d'un programme en cas de plantage de la machine. Votre programme n'est ainsi jamais perdu !

L'OMIKRON est un véritable interpréteur, ce qui le rapproche en beaucoup de points des langages compilés. Il est possible d'exécuter et sauver un programme même s'il comprend des erreurs de syntaxe. A la différence du Basic GFA, on peut sortir d'une ligne qui comporte des erreurs. Le portage d'un autre basic s'en trouve facilité puisque vous pourrez essayer le programme avant d'avoir corrigé les erreurs éventuelles. Bien sûr, le programme s'arrêtera tout de même à la première erreur rencontrée !

LE LANGAGE

Organisation des variables

Comme en Basic GFA, l'OMIKRON dispose des variables booléennes (0 ou −1), octets (0 à 255), entier court (mot de 16 bits), entier long (32 bits). Différence sur les réels car l'OMIKRON propose un codage de réel en simple précision codé sur 6 octets et un codage de réel en double précision codé sur 10 octets. La double précision ne permet pas de coder des nombres réels plus petits ou plus grands qu'en simple précision (la limite est la même dans les deux cas : **−5.11E−4931** pour la valeur minimum et **+5.11E+4931** comme valeur maximum), mais améliore sensiblement le nombre de chiffres significatifs (9 en simple précision et 19 en double précision). Le GFA propose lui un codage unique sur 8 octets pour les nombres réels. Dans ces deux langages toutefois, les formats utilisés pour coder les réels correspondent aux normes IEEE, ce qui permet l'utilisation des coprocesseurs arithmétiques les plus utilisés (68881 de chez Motorola ou 80287 de chez Intel). En GFA, il faudra passer cependant par les instructions **SINGLE{ }** et **DOUBLE{ }** alors que le codage est fait directement en OMIKRON.

Les suffixes qui servent à préciser le type des variables sont différents dans les deux langages mais ces deux langages proposent tous deux la non écriture fastidieuse des suffixes grâce aux instructions **DEFxxx" "**, les xxx précisant le type des variables commençant par les initiales précisées en argument.

Voici un tableau récapitulatif et comparatif des suffixes de variables en OMIKRON et GFA :

Types de variable	Suffixe OMIKRON	Suffixe GFA
Booléen	%F	!
Octet	%B	I
Entier court	%W ou %	&
Entier long	%L ou rien	%
Réel simple	!	
Réel (Float)		# ou rien
Réel double	#	
Chaîne de caractères	$	$

Dans les deux langages, les chaînes de caractères sont limitées à 32767 caractères (un octet par caractère).

En OMIKRON cependant, il existe une restriction pour l'emploi des variables booléennes et octets ; ces dernières ne sont utilisables que sous forme de tableaux, à la différence du GFA.

Exemples :

Bool%F(12)=0

L'élément 12 du tableau booléen **Bool%( )** reçoit la valeur 0.

Les valeurs d'un tableau booléen ne pouvant recevoir que 0 (Faux) ou −1 (Vrai), toute valeur différente de 0 sera considérée comme vraie et sera codée comme −1 dans l'élément correspondant du tableau.

Aze!=42.56

La variable **Aze!** (réel simple) recevra la valeur 42.56 (réel simple). Comme en GFA, les conversions de type se font automatiquement.

Aze#=42.56

La variable **Aze#** (réel double) recevra la valeur 42.56 (réel simple).

Aze=42.56

La variable **Aze** (Entier long) recevra la partie entière de 42.56, soit 42.

Pour simplifier la lecture et l'écriture des programmes, l'instruction **DEFxxx** permet de définir des groupes de variables comme appartenant à un type déterminé. En OMIKRON, on ne peut définir ces groupes de variables que pour les variables qui peuvent être utilisées hors des tableaux :

DEFINT	: Entier court
DEFINTL	: Entier long
DEFSNG	: Réel simple *(single)*
DEFDBL	: Réel double
DEFSTR	: Chaîne de caractères

Le Basic GFA dispose lui des deux instructions supplémentaires **DEFBIT** et **DEFBYT** respectivement pour les variables booléennes et octets.

DEFSNG "E-R" définit toutes les variables commençant par une lettre comprise entre E et R comme des réels simples.

Attention, les appellations de variables commencent par défaut par une majuscule en OMIKRON et sont suivies par des minuscules. La réaffectation automatique en majuscules et minuscules se fait normalement par l'éditeur dès qu'une instruction est validée par la touche **Return**. Dans le cas des instructions **DEFxxx** cependant, il faut vraiment taper la première lettre en majuscule. A la différence du GFA, on ne peut taper de lettres ou caractères à la suite et définir ainsi des groupes de variables d'un certain type commençant par un groupe de lettres.

Exemples : **DEFDBL "Emi"** n'est pas possible en OMIKRON.
DEFDBL "e" est erreur de syntaxe.



Ces instructions n'affectent que les variables qui n'ont pas de suffixe : la variable **Emile** sera un réel simple (après le DEFSNG "E-R") alors que la variable **Emile#** restera un réel double, même après une instruction DEF. Attention, à la différence du GFA, les instructions du type DEFxxx doivent toujours être tapées en premier lieu au tout début du programme pour être prises en compte. Mais si vous décidez de rentrer une correspondance de type après avoir commencé à taper votre programme, vous pouvez tout de même récupérer les affectations de types en opérant comme suit :

— Insérer DEFxxx sur la première ligne et sauvegarder votre programme avec SAVE. Recharger votre programme avec LOAD BLOCK. Ce chargement (voir plus haut) précompile le programme instruction par instruction et vous permet de récupérer la signification de DEFxxx.

— Une seconde façon d'opérer et de sauver le programme en ASCII (SAVE,A ou SAVE BLOCK), de l'effacer de la mémoire par NEW, de taper sur les premières lignes les instructions de DEFxxx qui vous intéressent, et recharger le programme en ASCII par MERGE ou LOAD BLOCK à la suite des premières lignes tapées.

Les conversions de type

MKD$, MKS$, MKIL$, MKIS$, CVD, CVS, CVIL et CVI

Les fonctions MKD$, MKS$, MKIL$, MKI$ convertissent l'expression numérique argument en un chaîne de caractères. Les fonctions inverses sont respectivement CVD, CVS, CVIL et CVI. Ces fonctions opèrent de la même façon qu'en Basic GFA. Les chaînes de caractères relatives aux terminaisons de MK (abréviation de Make) doivent avoir respectivement des tailles de :

— 10 caractères pour un réel double (MKD$ et CVD) ;
— 6 caractères pour un réel simple précision (MKS$ et CVS) ;
— 4 caractères pour un entier long (MKIL$ et CVIL) ;
— 2 caractères pour un entier (MKI$ et CVI).

Exemple :
```
Wk# =CVD(MKD$(456.23))
PRINT Wk#
```
donnera 456.23.

ASC( ), CHR$( ), STR$( ), VAL( )

Lorsqu'on veut travailler sur un seul octet, comme en GFA, on dispose des deux fonctions ASC( ) et CHR$( ).

Les limites de ces deux fonctions sont les mêmes qu'en GFA :

```
PRINT ASC("AZERT")
```
donnera 65

Une autre façon de taper ces instructions en OMIKRON est la suivante :

```
A%B=ASC("AZERT")
```
Le résultat est 65

Les deux points sont un séparateur d'instruction. La limite du nombre d'instructions qu'on peut rentrer par ligne est fixée par le nombre maximum de caractères par ligne (255).

Les fonctions STR$( ) et VAL( ) fonctionnent exactement comme en GFA :

```
PRINT VAL(1212)
```
donne 1212.

```
PRINT STR$($FFFF)
```
donne 65535.

VAL( ) donne l'équivalent décimal de la chaîne argument. Cette chaîne peut être codée en octal, binaire ou hexadécimal :

```
PRINT VAL("$FF")
255
OK
PRINT VAL("&77")
63
OK
PRINT VAL("%10010")
18
OK
```

En OMIKRON on peut aussi convertir un nombre fractionnaire d'une base binaire, octale ou hexadécimale en décimal.

```
PRINT VAL("$1.4")
1.25
OK
```

Les préfixes utilisés en OMIKRON pour indiquer la base dans laquelle se codent les valeurs numériques sont les suivants :

- binaire % ou &b
- octal & ou &o
- décimal &d (par défaut tout est en décimal)
- hexadécimal $ ou &h

BIN$( ),OCT$( ),HEX$( )

Les trois fonctions pour passer de l'un à l'autre sont les mêmes qu'en GFA.

```
PRINT BIN$(51)
110011
OK
PRINT HEX$(65534)
FFFE
OK
PRINT OCT$(24) ;"      "; HEX$(&24)
30      14
OK
```

Les conversions de bases ne peuvent se faire que sur les expressions numériques qui sont dans les limites imposées aux entiers longs.



Les tableaux et les tris

Comme en GFA, il y a obligation de déclarer un tableau par l'instruction (DIM <tableau> (<valeur>...)) si on veut imposer une taille précise au tableau. Par contre on peut très bien utiliser en OMIKRON un tableau qui n'a pas été dimensionné. L'OMIKRON force 10 comme valeur sur chaque dimension :

Poids(1,2,3)=45

force le dimensionnement du tableau **Poids** à (10,10,10), si l'instruction **DIM Poids( )** n'a pas été utilisée auparavant.

Dans un tableau, l'index du premier élément est toujours zéro. On ne dispose pas de l'**OPTION BASE** du GFA.

La seule limite dans le nombre de dimensions est celle de l'écriture de la ligne (255 caractères). Pour le nombre d'éléments, il n'y a de limites que pour les tableaux à plusieurs dimensions :

65535 est la limite pour la première dimension et 65535 pour le produit des autres dimensions (un tableau a une structure d'arbre).

Exemple :

DIM Poids(12,49,9999,1) est interdit car le produit des deuxième, troisième et quatrième dimensions est de 100000.

Par contre, **DIM Poids(456,45,56,11,1)** est accepté, le produit des dimensions autres que la première étant inférieur à 65535.

Notez que l'erreur due au surdimensionnement n'est pas détectée avant la commande **RUN**

Mémoire nécessaire pour les tableaux :

Type du tableau	Suffixe	Nombre d'octets
Booléens	(%F)	1 octet pour 8 éléments
Octets	(%B)	1 octet par élément
Mots	(%)	2 octets par élément
Mots longs	(%L)	4 octets par élément
Réel simple	(!)	6 octets par élément
Réel double	(#)	10 octets par élément
Chaîne	($)	6 octets par élément * longueur + 10

Par rapport au GFA, il y a gain de place lorsqu'on utilise les tableaux booléens, du fait aussi qu'on ne peut utiliser les octets et les booléens qu'en tableaux ; en GFA, le codage des variables booléenes se fait sur un octet, ce qui peut faire perdre en place ce qu'on peut gagner en clarté de programmation.

Une particularité essentielle et unique de l'OMIKRON au sujet de l'instruction DIM est la possibilité de redimensionner un tableau en cours de programme, sans aucune perte de contenu ! Il faut simplement faire très attention lorsqu'on redimensionne des tableaux à plusieurs dimensions en inversant les index :

DIM Tableau(20,20)
DIM Tableau(20,30) : REM inchangé
DIM Tableau(30,20) : REM le contenu est inversé car les index de dimensions sont échangés.

En GFA, pour redimensionner un tableau, il faut auparavant l'effacer avec l'instruction **ERASE Tableau( )**.

CLEAR,CLEAR<GEMDOS><pile>,SWAP,SORT

CLEAR efface le contenu de toutes les variables, désactive les branchements pour traitement d'erreur (**ON ERROR...GOTO**), désactive les fonctions apparentées multi-tâches système OMIKRON (**ON TIMER...GOSUB**), ferme tous les fichiers ouverts et vérifie la régularité des boucles.

En OMIKRON, l'instruction **CLEAR** réserve la place mémoire nécessaire au GEMDOS (utilisée par les fichiers ressources) et à la pile processeur (tris et appels de toutes les procédures et sous programmes comme en assembleur).

CLEAR 1000,500 libère 1000 octets sur l'emplacement mémoire du GEMDOS, nombre qu'on peut retrouver par l'instruction **MEMORY(–1)** qui donne le nombre d'octets libres sur la mémoire réservée pour GEMDOS.

Si à la suite, vous tapez :

?MEMORY(–1)
vous aurez 1000.

Dans l'exemple plus haut, si vous ne réservez que 500 octets pour la pile processeur, vous ne pourrez certainement pas aller très loin sans planter la machine, vu le nombre impressionnant d'appels de routines que doit effectuer l'interpréteur de l'OMIKRON.

SWAP permet d'échanger les valeurs de deux variables à condition que ces dernières soient du même type. Il n'y a pas de conversion de type comme dans l'affectation de valeurs numériques à des variables entières ou réelles. On ne peut pas non plus échanger le contenu d'un élément de tableau de chaîne de caractères avec une variable chaîne de caractères. Ceci est dû au codage bien particulier des chaînes et tableau de chaînes. Par contre, on peut échanger une variable numérique et un élément d'un tableau numérique du même type.



Exemple :

```
A=35
B=52
SWAP A,B
? A
52
? B
35
OK
A$="aze"
B$="qsdfg"
SWAP A$,B$
? A$
qsdfg
? B$
aze
OK
SWAP Tab$(5),Tab$(45) :REM OK
SWAP A#,A! :REM impossible réels simple et double
SWAP Tab$(5),Tab$: REM impossible tableau de chaîne et chaîne
SWAP Tab(12),Tab(85):REM OK
```

SORT

A la différence du GFA qui possède les deux instructions de tri QSORT et SSORT, l'OMIKRON dispose de la seule instruction SORT. L'OMIKRON décide lui-même d'utiliser ou non le QUICK SORT ou le TRI PAR BULLE suivant la place mémoire dont il dispose. Tous les tableaux à une dimension peuvent être triés par l'OMIKRON. Par défaut, le tri s'effectue dans le sens des valeurs croissantes.

```
DIM Tab(100)
FOR I=0 to 100
  Tab(I)=RND(100)
NEXT I
SORT Tab(100)
FOR I=0 TO 100
  PRINT Tab(I),
NEXT I
```

Le tableau Tab est rempli par des valeurs aléatoires comprises entre 0 et 99. Les 100 premiers éléments du tableau sont ensuite triés par valeurs croissantes. Si vous désirez ne trier que les 20 premières valeurs, il faudra utiliser l'instruction SORT (20).

Il est à noter pour cette instruction un petit inconvénient : on ne peut trier plus de valeurs que la dimension du tableau ; or cette valeur représente l'indice maximum du tableau et non le nombre d'éléments, ce qui dans le cas du petit programme ci-dessus amènera le dernier élément du tableau à ne jamais être trié.

Pour les chaînes de caractères on peut utiliser la variante SORT ASC Tab$(50) qui permet de trier les éléments par le code ASCII de leur premie

caractère. Par défaut, le rangement sur les chaînes de caractères ne distingue pas les majuscules des minuscules.

On ne peut pas toutefois, comme en GFA, prendre un autre tableau comme référence de rangement, mais l'utilisation de SORT Tab1( ) to Tab2( ) permet de remplir le tableau avec l'indice d'occurence de la valeur rangée.

```
DIM Tab1(100),Tab2(100),Tab3(100)
FOR I=0 TO 100
  Tab1(I)=RND(1000):Tab3(I)=I
NEXT I
FOR I=0 to 100
  Tab2(I)=Tab1(I): REM on recopie Tab1( ) dans Tab2( )
NEXT I
SORT Tab2(50) TO Tab3(50) :REM Les 50 premiers éléments du
REM tableau Tab2( ) sont triés et l'indice correspondant est rangé
REM dans Tab3( ).
REM affichage du tableau rangé par valeurs croissantes
FOR I=0 to 49
PRINT Tab1(I),
NEXT I
REM affichage du tableau non rangé
FOR I=0 to 49
PRINT Tab2(I),
NEXT I
REM affichage de l'indice des valeurs trouvées en cours de rangement
FOR I=0 to 49
PRINT Tab1(I),
NEXT I
```

Vous noterez que l'OMIKRON, lorsqu'il trouve plusieurs fois la même valeur numérique dans le tableau Tab1( ), les range par indices décroissants dans le tableau Tab3( ).

En OMIKRON, il n'existe pas de possibilité simple pour modifier la table des codes ASCII en vue d'un tri utilisant les codes ASCII. Par défaut, ce sont les codes standard qui sont utilisés.

Il n'existe pas non plus actuellement (V3.01) d'instruction équivalente à INSERT Tab(i)=val ou DELETE Tab(i) qui permette de rajouter ou supprimer un élément dans un tableau.

La gestion de la mémoire

A=FRE(0),
A=FRE(A$),A=FRE("A:")

La gestion de la mémoire est l'une des parties les plus complexes d'un interpréteur Basic. Pour connaître la place disponible en mémoire, l'instruction FRE des basics standard peut être utilisée : PRINT FRE(0) donne la place disponible en mémoire.

La seconde syntaxe, FRE(A$) ou FRE("A") effectue une GARBAGE COLLECTION en mémoire avant d'imprimer la place disponible. La reallocation de mémoire est le problème principal d'un interpréteur. Les déclarations et réservations ne sont en effet pas faites en début de programme et dès qu'une



chaîne de caractères est modifiée, il y a perte et gaspillage de mémoire. Toute la qualité d'un interpréteur est de pouvoir récupérer et restituer rapidement la mémoire inutilisée.

Le problème peut se comprendre simplement sur une addition de chaînes.

A$ = "BONJOUR"

Le codage de cette chaîne se fait caractère par caractère à partir de l'adresse d'un seul tenant la plus basse disponible en mémoire. Le codage en mémoire est :

"BONJOUR0..."

Le 0 (octet nul ASCII ou caractère vide) en fin de chaîne indique toujours sur tous les basics la fin de chaîne.

Si le programme définit ensuite en mémoire une seconde chaîne :

B$ = "WEKA"

On trouvera en mémoire :

"BONJOUR0WEKA0..."

Si on remplace la variable A$ par la concaténation de A$ et de B$ soit A$ = A$ + B$, nous aurons en mémoire :

"BONJOUR0WEKA0BONJOURWEKA0..."

L'ancien codage de A$,"BONJOURO" est toujours en début de chaîne mémoire et n'a maintenant plus aucune utilité. Bien que prenant de la place mémoire à chaque modification d'une variable chaîne de caractères, cette méthode de codage est extrêmement rapide. Son inconvénient majeur est de provoquer tôt ou tard une opération de restitution de mémoire ou GARBAGE COLLECTION. C'est l'interpréteur qui décide de la nécessité d'une telle opération lorsqu'il se rend compte qu'il n'a plus de place pour travailler. Sur l'OMIKRON cette opération prend au maximum 5/200e de secondes et peut libérer suivant les cas plusieurs Ko de mémoire. C'est une des qualités premières de l'OMIKRON, lui conférant, sans doute sa vélocité.

Il est très simple de programmer un exemple qui mette ce procédé en valeur :

```
A$ = "WEKA"
FOR I = 0 TO 1000
  FOR J = 0 TO 200
    B$ = B$ + A$
  NEXT J
NEXT I
PRINT "AVANT";FRE(0),"APRES"; FRE("")
```

La variante FRE("A:") donne le nombre d'octets disponibles sur le lecteur spécifié (ici A) après la GARBAGE COLLECTION.

Ces instructions sont comparables à celles du GFA.

MEMORY(−1),
MEMORY(expression numérique),
MEMORY(chaîne)

MEMORY(−1) restitue le nombre d'octets disponibles sur la mémoire réservée pour GEMDOS.

Dans la mémoire laissée disponible par le Basic, il est possible de réserver un nombre d'octets égal à l'argument d'appel. Si le nombre d'octets demandé est trop important, le message pas assez de mémoire apparaît à l'écran.

MEMORY(2000) réserve 2000 octets.

En utilisant cette instruction avec une chaîne de caractères, la chaîne argument est placée dans la zone réservée.

A=MEMORY("WEKA"):REM réserve de la mémoire et place la REM chaîne "WEKA" en début de la zone réservée.

Les blocs de mémoire réservés par MEMORY sont automatiquement libérés par CLEAR, RUN, ou une modification du programme. En effet, l'allocation de mémoire en mode exécution se fait toujours de façon dynamique.

Ces instructions sont à rapprocher des instructions MALLOC, MSHRINK et MFREE du GFA et certains basics.

MEMORY_BLOCK N°,taille,adresse

C'est une innovation importante de la version 3.01.

Cette instruction réserve un bloc d'octets qui sera conservé lors de la sauvegarde du programme. Le bloc peut être numéroté de 0 à 99 (100 blocks au maximum) et la taille et l'adresse du bloc sont fixées par les deux derniers paramètres.

MEMORY_BLOCK 0,32000,adr

Cette instruction définit le bloc 0 comme de longueur 32000 octets et commençant à l'adresse adr. C'est par exemple le cas d'une image écran ou d'un programme en langage machine. L'instruction équivalente en GFA est INLINE qui existe aussi en OMIKRON mais qui n'a pas la même utilisation.

MEMORY_MOVE dep,longueur,TO arr

Cette instruction est équivalente à l'instruction BMOVE du GFA. A sa différence, elle ne peut faire que des transferts de mots et doit donc s'effectuer d'une adresse paire à une adresse paire.

Pour effectuer des déplacements mémoire octet par octet, il faut utiliser l'instruction

MEMORY_MOVEB dep,longueur, TO arr

qui permet d'utiliser des valeurs impaires pour les adresses dep et arr :

```
A=MEMORY(14)
B=MEMORY(14)
MEMORY_MOVE A,15,B
```

Le programme comporte deux erreurs de programmation : A et B sont des adresses de départ de réservation de 14 octets. Le transfert des 15 octets à partir de A va au delà de la zone mémoire réservée par MEMORY. Le 15e transfert ne pourra pas de toute façon s'effectuer car il correspond à une adresse impaire. Le programme correct devient :

```
A=MEMORY(15)
B=MEMORY(15)
MEMORY_MOVEB A,15,B
```



INLINE

Attention, c'est le faux-ami du GFA. En OMIKRON il permet d'introduire du langage machine codé sur une chaîne de caractères en hexadécimal. Lorsque le programme rencontre l'instruction **INLINE**, le Basic exécute le programme codé à l'adresse de l'instruction.

En GFA, l'instruction **INLINE** ne servait qu'à réserver une zone mémoire destinée à stocker des octets (nuls par défaut avant tout chargement de la zone) sans obligation d'exécution au passage de l'instruction.

Les variables système

Les variables système OMIKRON sont à peu de chose près semblables aux variables système GFA, en ce qui concerne la lecture de la souris, du timer et celle du curseur. Leur appellation diffère de temps en temps.

MOUSEBUT

Lit l'état des boutons de la souris : **0** pas de clic
1 clic gauche
2 clic droit
3 clic gauche et droit

MOUSEX et MOUSEY

Donnent respectivement l'abscisse et l'ordonnée de la souris.

TIMER et TIME$

Contiennent respectivement le temps écoulé depuis la mise en route de l'ordinateur en 1/200e de secondes et l'heure système.

DATE$

Contient la date système.

PI

Contient la valeur de π en double précision. Les temps de calcul étant plus longs en double précision, si vous avez plus de 19 chiffres significatifs, passez en simple précision en posant :

```
PI!=PI
```

CSRLIN, POS

Sont équivalentes aux variables **CRSLIN, CRSCOL** du GFA. On ne dispose pas directement de la faculté de positionner le curseur sur une ligne ou une colonne comme en GFA avec les instructions VTAB et HTAB.

LPOS

Donne le numéro de colonne sur laquelle se trouve la tête de l'imprimante (en réalité le numéro de la dernière colonne envoyée sur le buffer de l'imprimante et non la position réelle de la tête de l'imprimante).

LPOS ne fonctionne pas en mode graphique et ne retourne pas le bon numéro de colonne suite à des envois de codes **ESC**.

ERR, ERL, ERR$

ERR : correspond au numéro de l'erreur
ERL : affiche le numéro de ligne où s'est produite l'erreur
ERR$: contient le libellé de l'erreur.

```
ON ERROR GOTO Trait_err
INPUT "QUELLE ERREUR VOUS PLAIRAIT ?";Er
ERROR Er
END
Trait_err
PRINT ERR,ERL,ERR$
```

On demande un numéro d'erreur à l'utilisateur, on affiche ce numéro, le message qui correspond à cette erreur ainsi que la ligne où s'est produite cette erreur (ligne de simulation ici).

VERSION

Contient le numéro de version du Basic OMIKRON que vous utilisez. Dans le cas du 3.01, VERSION renverra la valeur 301. Par définition, le programme compilé ne fonctionne pas sous interpréteur ; cette variable n'existe donc pas en version compilée du programme.

Les opérateurs

Opérateurs arithmétiques

Les opérateurs arithmétiques sont les mêmes qu'en GFA et fonctionnent exactement de la même façon :

+	: Addition
−	: Soustraction
/	: Division
*	: Multiplication
^	: Puissance
\	: Division entière (résultat entier)
MOD	: Modulo (reste de la division entière)

Pensez à utiliser la division entière lorsque vous opérez sur des entiers. Tapez le programme ci-dessous et comparez les temps en secondes de 1000 divisions successives :

```
T=TIMER
FOR I=1 TO 1000
Y=98/65
NEXT I
PRINT (TIMER-T)/200
T=TIMER
FOR I=1 TO 1000
Y=98\65
NEXT I
PRINT (TIMER-T)/200
```

Opérateurs de comparaison

Ce sont exactement les mêmes qu'en GFA :

>	: supérieur à
<	: inférieur à
=	: égal à
>=	: supérieur ou égal à
<=	: inférieur ou égal à
><	: différent de

Les opérateurs doubles peuvent être tapés dans n'importe quel ordre : >< ou <>.

Cependant, l'OMIKRON ne dispose pas de l'opérateur = = du GFA (très peu différent de) qui permet la comparaison d'égalité entre deux valeurs numériques réelles à l'erreur d'arrondi près pour la partie fractionnaire. Cette erreur n'est imputable ni à l'OMIKRON ni au Basic, ni à l'Atari, mais simplement au fait de retranscrire toutes les valeurs numériques dans le système binaire. S'il est toujours possible de coder une partie entière en binaire, il n'en est pas de même pour les parties fractionnaires qui ne peuvent pas toujours s'exprimer comme somme de puissances négatives de deux. Essayez donc l'instruction suivante sur votre ordinateur :

```
PRINT 100000-100000.1
```



1/10 ne peut pas s'exprimer facilement comme somme de 1/16+ 1/32 + ... Attention aux égalités strictes de nombres réels qui devraient théoriquement être égaux !

Comme en GFA, on dispose en OMIKRON des fonctions **MIN( )** et **MAX( )** qui retournent respectivement la plus petite et la plus grande valeur des arguments spécifiés entre parenthèses.

Opérateurs logiques

L'OMIKRON dispose des opérateurs logiques disponible sur GFA et de la majorité des basics, mais propose aussi les opérateurs **NAND** et **NOR**. Le NAND est le contraire du **AND** et le **NOR** est le contraire logique du **OR**. En GFA il faudrait les définir comme suit :

B NAND C = NOT(B AND C)
B NOR C = NOT(B OR C)

Les fonctions mathématiques

C'est un autre point fort de ce langage. Il propose un nombre impressionnant de fonctions mathématiques qui sont d'autant plus intéressantes que cet interpréteur Basic est sans doute le plus rapide actuellement sur le marché.

INT(A)

C'est la même fonction qu'en GFA. Elle rend la partie entière du nombre A :

INT(7.6) donne 7.
INT(−7.6) donne −8 par arrondi au chiffre inférieur.

FIX(A)

Est équivalente à **TRUNC(A)** du GFA. Les nombres négatifs sont arrondis au nombre supérieur.

FIX(−7.6) donne −7.

FRAC(A),ABS(A), SGN(A),RND(A)

Ce sont les fonctions standards qui donnent respectivement la partie fractionnaire, la valeur absolue, le signe de A et qui génère un nombre semi-aléatoire compris entre 0 et A−1 :

3+RND(5) génère un nombre compris entre 3 et 7.

RND(1) génère une valeur comprise entre 0 et 0.99999999.

RND(0) retourne le dernier nombre aléatoire généré.

RND génère toujours les valeurs réelles converties en entiers (cf. **XBIOS(17)**).

DEG et RAD

L'OMIKRON effectue à la suite de l'une de ces deux instructions les calculs trigonométriques en degrés ou en radians.

Fonctions trigonométriques

L'OMIKRON dispose d'un nombre impressionnant de fonctions trigonométriques, du sinus et cosinus classiques **SIN(A) COS(A)** aux tangentes, cotangentes, sécantes, cosécantes ainsi que leurs fonctions inverses, les fonctions hyperboliques au grand complet avec leurs fonctions inverses, avis aux amateurs de belles courbes ! Il suffit pour ces dernières fonctions de rajouter le H en fin de nom (et d'enlever le C de ARC pour les fonctions inverses) à leurs homologues trigonométriques classiques.

Exemple :

COS(A)	COSH(A)
ARCSIN(A)	ARSINH(A)
SEC(A)	SECH(A)

Logarithmes et exponentielles

L'OMIKRON dispose du logarithme népérien et du logarithme de base quelconque.

LN(A) donne le logarithme népérien de A.

LOG(A,B) donne le logarithme de base A du nombre B.

Cette dernière fonction impose à A et B d'être strictement positifs.

EXP(A) donne l'exponentielle de A dans la base e.

Autres fonctions

SQR(A) retourne la racine carée de A(A>=0).

FACT(A) donne la factorielle de A(A>0 et entier).

Cette fonction factorielle n'existe pratiquement dans aucun Basic (pas en GFA par exemple), prenez donc garde si vous désirez exporter votre programme !

```
Rappel FACT(A)=A*(A-1)*(A-2)*...*1
INPUT "Rentrez une valeur entière supérieure à 10";A
B=FACT(A)
C=FACT(A-1)
PRINT B/C
END
REM donne A
```



4/2.4.1

Utilitaires Omikron

Voici une série de programmes utilitaires en Basic OMIKRON dont l'action va du traitement des fonctions GEM aux routines VDI et graphiques plus classiques. Si vous avez quelques problèmes pour vous remémorer la signification des variables et tableaux spécifiques au GEM AES et VDI, revoyez l'article traitant de la programmation du VDI et le petit programme d'initialisation qui est appliqué (Partie 5 chapitre 3).

Les appellations des fonctions sont volontairement proches des appellations en langage C, optique qui est reprise aussi dans tous les langages évolués actuels. Les tableaux d'adresses (control) et les tableaux de coordonnées (int_in et out) sont facilement repérables. Reportez-vous aussi le cas échéant à la bibliothèque des fonctions AES ou VDI pour retrouver le numéro de fonction ou de sous-fonction concernée. La fonction d'attente de message utilise le traitement de recherche de chaîne de caractères déjà vu dans les articles sur l'OMIKRON (SEGPTR+28). Les retours à la ligne au milieu d'une suite d'arguments dans la définition ou l'appel d'une fonction ne seront pas forcément admis par votre interpréteur OMIKRON. Ecrivez donc tout sur la même ligne sans oublier une virgule, même si vous ne voyez pas tous les caractères à l'écran. C'est encore l'un des avantages de mise en page du texte de programmation dans un langage comme le C. Le fichier de procédures ainsi complété pourra être inclus et utilisé dans votre programmation OMIKRON comme il a été précisé dans le cours sur l'OMIKRON (Partie 4 chapitre 2).

```
'          BIBLIOTHEQUE DE FONCTIONS GEM, AES, VDI,GRAPHIQUES
'
'
'-------------------------GEM AES-------------------------------
' Le résultat d'erreur à la suite de l'appel de chaque
'              fonction GEM AES se situe dans
'                 Int_Out%(0).
'                 Si Int_Out%(0)  = 0 alors ERREUR
'                 Si Int_Out%(0) <> 0 alors PAS D'ERREUR


'    OUVERTURE D'UNE APPLICATION
DEF PROC Appl_Init
  LOCAL I%
  FOR I%=0 TO 100: MOUSEON : NEXT : MOUSEOFF
    ' Dimensionner les tableaux pour l'AES et le VDI.
  DIM Global%(15),Int_In%(128),Int_Out%(128)
  DIM Addr_In!(3),Addr_Out!(1)
  DIM Contrl%(12),Intin%(128),Intout%(128)
  DIM Ptsin%(1,128),Ptsout%(1,128)
  AES (10,Global%(15),Int_In%(0),Addr_In!(0),
       Int_Out%(1),Addr_Out!(0))
  PRINT "";
  CLIP :Form_Dial(3,0,0,0,0,0,0,640,400)
  MOUSEON
RETURN
'-----------------------------
'-----------------------------

'LECTURE D'OCTETS DU FLUX DE MESSAGE SUPERIEUR A 16 OCTECTS)
'          INDENTIFICATEUR , NBRE OCTECT ,R TAMPON
DEF PROC Appl_Read(Int_In%(0),Int_In%(1),
                   R Text$,R Int_Out%(0))
  Int_In%(0)=Global%(2)
  Text$= CHR$(0)*Int_In%(1)
  Addr_In!(0)= LPEEK( VARPTR(Text$))+ LPEEK( SEGPTR +28)
  AES (11,Global%(15),Int_In%(2),Addr_In!(1),
       Int_Out%(1),Addr_Out!(0))
 RETURN
'-----------------------------
'-----------------------------

'    ECRITURE D'OCTETS DANS FLUX DE MESSAGE
'    INDENTIFICATEUR , NBRE OCTECT, TAMPON
DEF PROC Appl_Write(Int_In%(0),Int_In%(1),Text$)
  Text$=Text$+ CHR$(0)
  Addr_In!(0)= LPEEK( VARPTR(Text$))+ LPEEK( SEGPTR +28)
  AES (12,Global%(15),Int_In%(2),Addr_In!(1),
       Int_Out%(1),Addr_Out!(0))
RETURN
```



(suite)

```
'
DEF PROC Appl_Find(Text$)
   Text$= LEFT$(Text$+ SPC(8),8)+ CHR$(0)
   AES (13,Global%(15),Int_In%(0),Addr_In!(1),
        Int_Out%(1),Addr_Out!(0))
RETURN
'-------------------------

'    EXECUTION D'UNE SUITE D'ACTIONS
'
DEF PROC Appl_Tplay(Addr_In!(0),Int_In%(0),Int_In%(1))
   AES (14,Global%(15),Int_In%(2),Addr_In!(1),
        Int_Out%(1),Addr_Out!(0))
RETURN
'-------------------------

'    ENREGISTREMENT D'UNE SERIE D'ACTIONS
'
DEF PROC Appl_Trecord(Addr_In!(0),Int_In%(0))
   AES (15,Global%(15),Int_In%(1),Addr_In!(1),
        Int_Out%(1),Addr_Out!(0))
RETURN
'-------------------------

'    SORTIR DE L'APPLICATION
'
DEF PROC Appl_Exit
   MOUSEOFF
RETURN
'
'---------GESTION DES EVENEMENTS ----------
' ____________________________________
'    ATTENDRE UN EVENEMENT DU CLAVIER
'
DEF PROC Evnt_Keybd
   AES (20,Global%(15),Int_In%(0),Addr_In!(0),
                       Int_Out%(1),Addr_Out!(0))
RETURN
' ---------------------------

'    ATTENDRE UN ETAT DU BOUTON DE LA SOURIS
'
DEF PROC Evnt_Button(Int_In%(0),Int_In%(1),Int_In%(2),
                     R Int_Out%(1),R Int_Out%(2),
                     R Int_Out%(3),R Int_Out%(4))
   AES (25,Global%(15),Int_In%(3),Addr_In!(0),
        Int_Out%(5),Addr_Out!(0))
RETURN

'    ATTENDRE UNE POSITION DE LA SOURIS
'
```

(suite)

```
DEF PROC Evnt_Mouse(Int_In%(0),Int_In%(1),Int_In%(2),
            Int_In%(3),Int_In%(4),R Int_Out%(1),
            R Int_Out%(2),R Int_Out%(3),R Int_Out%(4))
   AES (22,Global%(15),Int_In%(5),Addr_In!(0),
         Int_Out%(5),Addr_Out!(0))
RETURN
'-------------------------------
'    ATTENDRE UN MESSAGE DU FLUX DE MESSAGE
'    (FERMETURE D'UNE FENETRE, OPTION D'UN MENU,..)
'
DEF PROC Evnt_Mesag(R Text$):Text$= CHR$(0)*16
    Addr_In!(0)= LPEEK( VARPTR(Text$))+ LPEEK( SEGPTR +28)
    AES (23,Global%(15),Int_In%(0),Addr_In!(1),
         Int_Out%(1),Addr_Out!(0))
RETURN
'
'-------------------------
'
'    ATTENDRE UN TEMPS DETERMINÉ
'
DEF PROC Evnt_Timer(T%):Int_In%(0)= LOW(T%):Int_In%(1)= HIGH(T%)
    AES (24,Global%(15),Int_In%(2),Addr_In!(0),
         Int_Out%(1),Addr_Out!(0))
RETURN
'
'-------------------------
'
'    ATTENDRE UN OU PLUSIEURS EVENEMENTS SIMULTANÉS
'    (SOURIS ET CLAVIER PAR EXEMPLE)
'
DEF PROC Evnt_Multi(Int_In%(0),Int_In%(1),Int_In%(2),
            Int_In%(3),Int_In%(4),X1%,Y1%,W1%,H1%,
            Int_In%(9),X2%,Y2%,W2%,H2%,
            Temp%,R Text$,R Int_Out%(0),R Int_Out%(1),
            R Int_Out%(2),R Int_Out%(3),
            R Int_Out%(4),R Int_Out%(5),R Int_Out%(6))
  Int_In%(5)=X1%:Int_In%(6)=Y1%
  Int_In%(7)=W1%:Int_In%(8)=H1%
  Int_In%(10)=X2%:Int_In%(11)=Y2%
  Int_In%(12)=W2%:Int_In%(13)=H2%
  Int_In%(14)= LOW(Temp%)
  Int_In%(15)= HIGH(Temp%)
  Text$= CHR$(0)*16
  Addr_In!(0)= LPEEK( VARPTR(Text$))+ LPEEK( SEGPTR +28)
  AES (25,Global%(15),Int_In%(16),Addr_In!(1),
       Int_Out%(7),Addr_Out!(0))
RETURN
'
'-----------------
```



(suite)

```
' GESTION DU TEMPS  DU DOUBLE CLIC DE LA SOURIS
'
DEF PROC Evnt_Dclick(Int_In%(0),Int_In%(1))
    AES (26,Global%(15),Int_In%(2),Addr_In!(0),
        Int_Out%(1),Addr_Out!(0))
RETURN
'
'------------GESTION DE MENUS ------------
'-----------------------------------
'    AFFICHE OU EFFACE LA LIGNE DES TITRES DU MENU
'    ADRESSE DE L'ARBRE DU MENU,  0:efface  1:affiche
DEF PROC Menu_Bar(Addr_In!(0),Int_In%(0))
   AES (30,Global%(15),Int_In%(1),Addr_In!(1),
        Int_Out%(1),Addr_Out!(0))
RETURN
'-------------------
'-------------------

'    AFFICHE OU EFFACE UN REPERE DEVANT UNE OPTION DU MENU
'    ADRESSE DE L'ARBRE, NUMERO DE L'OPTION,
'          0:efface  1:affiche
DEF PROC Menu_Icheck(Addr_In!(0),Int_In%(0),Int_In%(1))
   AES (31,Global%(15),Int_In%(2),Addr_In!(1),
        Int_Out%(1),Addr_Out!(0))
RETURN
' ------------------------------
'--------------------------------

'    AUTORISE OU EMPECHE LA VALIDATION D'UNE OPTION
'    ADRESSE DE L'ARBRE, NUMERO DE L'OPTION,
'          0:empeche  1:autorise
DEF PROC Menu_Ienable(Addr_In!(0),Int_In%(0),Int_In%(1))
   AES (32,Global%(15),Int_In%(2),Addr_In!(1),
        Int_Out%(1),Addr_Out!(0))
RETURN
' ------------------------
'------------------------
'    INVERSION DU TITRE D'UN MENU
'    ADRESSE MENU, NUMERO TITRE, 0:inversion 1:normal
DEF PROC Menu_Tnormal(Addr_In!(0),Int_In%(0),Int_In%(1))
   AES (33,Global%(15),Int_In%(2),Addr_In!(1),
        Int_Out%(1),Addr_Out!(0))
RETURN
'  -------------------------
```

(suite)

```
'     MODIFICATION DU TEXTE D'UN MENU (TITRE OU OPTION)
'     LA LONGUEUR DU TEXTE DOIT ETRE INFERIEURE OU EGALE
'           À CELLE D'ORIGINE
'     ADRESSE MENU, NUMERO ELEMENT, NOUVEAU TEXTE
DEF PROC Menu_Text(Addr_In!(0),Int_In%(0),Text$)
   Text$=Text$+ CHR$(0)
   Addr_In!(1)= LPEEK( VARPTR(Text$))+ LPEEK( SEGPTR +28)
   AES (34,Global%(15),Int_In%(1),Addr_In!(2),
        Int_Out%(1),Addr_Out!(0))
RETURN
'  ------------------------------
'-------------------------------------
'     INSTALLER LE TEXTE D'UNE OPTION DU MENU
'          "BUREAU" POUR UN ACCESSOIRE
DEF PROC Menu_Register(Int_In%(0),Text$)
   Addr_In!(0)= MEMORY(Text$)
   AES (35,Global%(15),Int_In%(1),Addr_In!(1),
        Int_Out%(1),Addr_Out!(0))
RETURN
'
'     ------- GESTION DES OBJETS APPARTENANTS
'              A LA RESSOURCE PRESENTE EN MEMOIRE
'
'     ADDITION D'UN OBJET
DEF PROC Objc_Add(Addr_In!(0),Int_In%(0),Int_In%(1))
   AES (40,Global%(15),Int_In%(2),Addr_In!(1),
        Int_Out%(1),Addr_Out!(0))
RETURN
'-------------------------
'-------------------------
'     EFFACER UN OBJET
DEF PROC Objc_Delete(Addr_In!(0),Int_In%(0))
   AES (41,Global%(15),Int_In%(1),Addr_In!(1),
        Int_Out%(1),Addr_Out!(0))
RETURN
'-----------------------
'     AFFICHER UN OBJET DE L'ARBRE
'     ADRESSE LA STRUCTURE, OBJET DE DEPART,
'       NBRE DE NIVEAUX, Xc,Yc,Wc,Hc
'    (Xc,Yc,Wc,Hc = RECTANGLE DE CLIPPING)
DEF PROC Objc_Draw(Addr_In!(0),Int_In%(0),Int_In%(1),
          Int_In%(2),Int_In%(3),Int_In%(4),Int_In%(5))
   AES (42,Global%(15),Int_In%(6),Addr_In!(1),
        Int_Out%(1),Addr_Out!(0))
RETURN
'
DEF PROC Objc_Find(Addr_In!(0),Int_In%(0),Int_In%(1),
                   Int_In%(2),Int_In%(3))
   AES (43,Global%(15),Int_In%(4),Addr_In!(1),
        Int_Out%(1),Addr_Out!(0))
RETURN
'-----------------------
```



(suite)

```
'     CHERCHE LA POSITION D'UN OBJET À L'ÉCRAN
'     ADRESSE STRUCTURE, NUMERO DE L'OBJET, R X, R Y
DEF PROC Objc_Offset(Addr_In!(0),Int_In%(0),
                     R Int_Out%(1),R Int_Out%(2))
   AES (44,Global%(15),Int_In%(1),Addr_In!(1),
        Int_Out%(3),Addr_Out!(0))
RETURN
'
DEF PROC Objc_Order(Addr_In!(0),Int_In%(0),Int_In%(1))
   AES (45,Global%(15),Int_In%(2),Addr_In!(1),
        Int_Out%(1),Addr_Out!(0))
RETURN
'
DEF PROC Objc_Edit(Addr_In!(0),Int_In%(0),Int_In%(1),Int_In%(2),
                   Int_In%(3),R Int_Out%(1))
   AES (46,Global%(15),Int_In%(6),Addr_In!(1),
        Int_Out%(2),Addr_Out!(0))
RETURN
'-------------------
'-------------------
'     MODIFICATION DE L'ETAT D'UN OBJET
'  (INVERSION VIDÉO PAR EXEMPLE)
'     ADRESSE STRUCTURE, NUMERO OBJET,
'     0, Xc,Yc,Wc,Hc, ETAT, Si 0 pas de réaffichage
'    Si 1 réaffichage
DEF PROC Objc_Change(Addr_In!(0),Int_In%(0),Int_In%(1),
                     Int_In%(2),Int_In%(3),Int_In%(4),
                     Int_In%(5),Int_In%(6),Int_In%(7))
   AES (47,Global%(15),Int_In%(8),Addr_In!(1),
        Int_Out%(1),Addr_Out!(0))
RETURN
'-------------------
'-------------------
'     ----------CENTRER UNE FORME DIALOGUE----------
'     ADRESSE STRUCTURE, R X, R Y, R W, R H
DEF PROC Form_Center(Addr_In!(0),R Int_Out%(1),
          R Int_Out%(2),R Int_Out%(3),R Int_Out%(4))
   AES (54,Global%(15),Int_In%(0),Addr_In!(1),
        Int_Out%(5),Addr_Out!(0))
RETURN
'
'
DEF PROC Form_Keybd(Addr_In!(0),Int_In%(0),
                    Int_In%(1),Int_In%(2),
                    R Int_Out%(1),R Int_Out%(2))
   AES (55,Global%(15),Int_In%(3),Addr_In!(1),
        Int_Out%(2),Addr_Out!(0))
RETURN
```

(suite)

```
'
DEF PROC Form_Button(Addr_In!(0),Int_In%(0),
                     Int_In%(1),Addr_In!(0))
   AES (56,Global%(15),Int_In%(2),Addr_In!(1),
        Int_Out%(1),Addr_Out!(0))
RETURN
'
'----------GESTION DES BOITES DEFORMABLES----------
'                ------------------------------

DEF PROC Graf_Rubberbox(Int_In%(0),Int_In%(1),Int_In%(2),
        Int_In%(3),R Int_Out%(1),R Int_Out%(2))
   AES (70,Global%(15),Int_In%(4),Addr_In!(0),
        Int_Out%(3),Addr_Out!(0))
RETURN
'
DEF PROC Graf_Dragbox(Int_In%(0),Int_In%(1),Int_In%(2),
           Int_In%(3),Int_In%(4),Int_In%(5),Int_In%(6),
           Int_In%(7),R Int_Out%(1),R Int_Out%(2))
   AES (71,Global%(15),Int_In%(8),Addr_In!(0),
        Int_Out%(3),Addr_Out!(0))
RETURN
'
DEF PROC Graf_Movebox(Int_In%(0),Int_In%(1),Int_In%(2),
                      Int_In%(3),Int_In%(4),Int_In%(5))
   AES (72,Global%(15),Int_In%(6),Addr_In!(0),
        Int_Out%(1),Addr_Out!(0))
RETURN
'
DEF PROC Graf_Growbox(Int_In%(0),Int_In%(1),Int_In%(2),
                      Int_In%(3),Int_In%(4),
                      Int_In%(5),Int_In%(6),Int_In%(7))
   AES (73,Global%(15),Int_In%(8),Addr_In!(0),
        Int_Out%(1),Addr_Out!(0))
RETURN
'
DEF PROC Graf_Shrinkbox(Int_In%(0),Int_In%(1),Int_In%(2),
                        Int_In%(3),Int_In%(4),Int_In%(5),
                        Int_In%(6),Int_In%(7))
   AES (74,Global%(15),Int_In%(8),Addr_In!(0),
        Int_Out%(1),Addr_Out!(0))
RETURN
'
DEF PROC Graf_Watchbox(Addr_In!(0),Int_In%(1),
           Int_In%(2),Int_In%(3))
   AES (75,Global%(15),Int_In%(4),Addr_In!(1),
        Int_Out%(1),Addr_Out!(0))
RETURN
'
```



(suite)

```
DEF PROC Graf_Slidebox(Addr_In!(0),Int_In%(0),
            Int_In%(1),Int_In%(2))
   AES (76,Global%(15),Int_In%(3),Addr_In!(1),
         Int_Out%(1),Addr_Out!(0))
RETURN
'
DEF PROC Graf_Handle(R Int_Out%(1),R Int_Out%(2),
                     R Int_Out%(3),R Int_Out%(4))
   AES (77,Global%(15),Int_In%(0),Addr_In!(0),
         Int_Out%(5),Addr_Out!(0))
RETURN
'   ------------------------

'    MODIFICATION DU CURSEUR GRAPHIQUE DE LA SOURIS
'              ------------------

DEF PROC Graf_Mouse(Int_In%(0),Addr_In!(0))
   AES (78,Global%(15),Int_In%(1),Addr_In!(1),
         Int_Out%(1),Addr_Out!(0))
RETURN
'
DEF PROC Graf_Mkstate(R Int_Out%(1),R Int_Out%(2),
                      R Int_Out%(3),R Int_Out%(4))
   AES (79,Global%(15),Int_In%(0),Addr_In!(0),
         Int_Out%(5),Addr_Out!(0))
RETURN
'
'
'---------- GESTION DES FENETRES
'              ----------

'    CREATION D'UNE FENETRE
DEF PROC Wind_Create(Int_In%(0),Int_In%(1),Int_In%(2),
                     Int_In%(3),Int_In%(4))
   AES (100,Global%(15),Int_In%(5),Addr_In!(0),
         Int_Out%(1),Addr_Out!(0))
RETURN
'-------------------
'-------------------------

'    OUVERTURE D'UNE FENETRE (PRECEDEMENT CRÉÉE)
DEF PROC Wind_Open(Int_In%(0),Int_In%(1),Int_In%(2),
                   Int_In%(3),Int_In%(4))
   AES (101,Global%(15),Int_In%(5),Addr_In!(0),
         Int_Out%(1),Addr_Out!(0))
RETURN
'  ----------------
```

(suite)

```
'     FERMETURE D'UNE FENETRE (OUVERTE)

DEF PROC Wind_Close(Int_In%(0))
   AES (102,Global%(15),Int_In%(1),Addr_In!(0),
        Int_Out%(1),Addr_Out!(0))
RETURN
'  ----------------
'--------------------
'     EFFACER UNE FENETRE (PRECEDEMENT CRÉÉE)

DEF PROC Wind_Delete(Int_In%(0))
   AES (103,Global%(15),Int_In%(1),Addr_In!(0),
        Int_Out%(1),Addr_Out!(0))
RETURN
'  ----------------
'----------------
'     CHERCHER LA FENETRE SITUÉE SOUS LA SOURIS
'     X,Y
DEF PROC Wind_Find(Int_In%(0),Int_In%(1))
   AES (106,Global%(15),Int_In%(2),Addr_In!(0),
        Int_Out%(1),Addr_Out!(0))
RETURN
'-----------------

'---------- GESTION DE FICHIER RESSOURCE
'
'     CHARGEMENT D'UN FICHIER RESSOURCE
'     NOM DU FICHIER RESSOURCE (TESTER INT_OUT%(0)
'PAR SECURITE POUR SAVOIR SI LE FICHIER RESSOURCE
' À ÉTÉ CHARGÉ CORRECTEMENT)
DEF PROC Rsrc_Load(Text$):Text$=Text$+ CHR$(0)
   Addr_In!(0)= LPEEK( VARPTR(Text$))+ LPEEK( SEGPTR +28)
   AES (110,Global%(15),Int_In%(0),Addr_In!(1),
        Int_Out%(1),Addr_Out!(0))
RETURN
DEF PROC Rsrc_Load(Text$,R Int_Out%(0))
   Text$=Text$+ CHR$(0)
   Addr_In!(0)= LPEEK( VARPTR(Text$))+ LPEEK( SEGPTR +28)
   AES (110,Global%(15),Int_In%(0),Addr_In!(1),
        Int_Out%(1),Addr_Out!(0))
RETURN

'                    ------------
'     LIBÉRER LA MEMOIRE PRÉCÉDEMENT OCCUPÉE
'          PAR LE FICHIER RESSOURCE
'          ------------------
'
DEF PROC Rsrc_Free
   AES (111,Global%(15),Int_In%(0),Addr_In!(0),
        Int_Out%(1),Addr_Out!(0))
RETURN
'
```



(suite)

```
'----------------------GEM VDI----------------------

'    OUVERTURE D'UNE STATION DE TRAVAIL VIRTUELLE
'    ---------------------------------------------
'
DEF PROC V_Opnvwk(R Contrl%(6))
LOCAL I%
  FOR I%=0 TO 9:Intin%(I%)=1: NEXT
  Intin%(10)=2
  VDI (100,Contrl%(12),Intin%(11),Ptsin%(0,0),
       Intout%(45),Ptsout%(0,6))
RETURN

'    FERMETURE DE LA STATION DE TRAVAIL VIRTUELLE
'    --------------------------------------------

DEF PROC V_Clsvwk(Contrl%(6))
  VDI (101,Contrl%(6),Intin%(0),Ptsin%(0,0),
       Intout%(0),Ptsout%(0,0))
RETURN
'
DEF PROC V_Clrwk(Contrl%(6))
  VDI (3,Contrl%(6),Intin%(0),Ptsin%(0,0),
       Intout%(0),Ptsout%(0,0))
RETURN

'    POSITIONNER UN RECTANGLE DE CLIPPING
'     0 est inhibé 1 est validé,X1,Y1,X2,Y2
'
DEF PROC Vs_Clip(Contrl%(6),Intin%(0),Ptsin%(0,0),Ptsin%(1,0),
                 Ptsin%(0,1),Ptsin%(1,1))
  VDI (129,Contrl%(12),Intin%(1),Ptsin%(0,4),
       Intout%(0),Ptsout%(0,0))
RETURN
'
```

(suite)

```
'-------------------------
'------FONCTIONS DE TRACÉS DE GRAPHIQUES-------
'            (RECTANGLE, CERCLE,ELLIPSE...)
'            -------------------------

DEF PROC V_Gtext(Contrl%(6),Ptsin%(0,0),
                 Ptsin%(1,0),Text$)
LOCAL I%
FOR I%=0 TO LEN(Text$)-1
Intin%(I%)= ASC( MID$(Text$,I%+1,1))
NEXT
  VDI (8,Contrl%(12),Intin%(I%),Ptsin%(0,1),
       Intout%(0),Ptsout%(0,0))
RETURN
'
DEF PROC V_Bar(Contrl%(6),Ptsin%(0,0),Ptsin%(1,0),
               Ptsin%(0,1),Ptsin%(1,1))
  Contrl%(5)=1
   VDI (11,Contrl%(12),Intin%(0),Ptsin%(0,2),
        Intout%(0),Ptsout%(0,0))
RETURN
'
DEF PROC V_Arc(Contrl%(6),Ptsin%(0,0),Ptsin%(1,0),
               Ptsin%(0,3),Intin%(0),Intin%(1))
  Contrl%(5)=2
  VDI (11,Contrl%(12),Intin%(2),Ptsin%(0,4),
       Intout%(0),Ptsout%(0,0))
RETURN
'
DEF PROC V_Pieslice(Contrl%(6),Ptsin%(0,0),Ptsin%(1,0),
          Ptsin%(0,3),Intin%(0),Intin%(1))
  Contrl%(5)=3
  Ptsin%(1,2)=0:Ptsin%(1,3)=0
  VDI (11,Contrl%(12),Intin%(2),Ptsin%(0,4),
       Intout%(0),Ptsout%(0,0))
RETURN
'
DEF PROC V_Circle(Contrl%(6),Ptsin%(0,0),
                  Ptsin%(1,0),Ptsin%(0,2))
  Contrl%(5)=4
  VDI (11,Contrl%(12),Intin%(0),Ptsin%(0,3),
       Intout%(0),Ptsout%(0,0))
RETURN
'
```

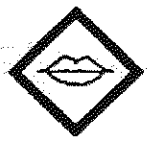


(suite)

```
DEF PROC V_Ellipse(Contrl%(6),Ptsin%(0,0),Ptsin%(1,0),
        Ptsin%(0,1),Ptsin%(1,1))
  Contrl%(5)=5
  VDI (11,Contrl%(12),Intin%(0),Ptsin%(0,2),
      Intout%(0),Ptsout%(0,0))
RETURN
'
DEF PROC V_Ellarc(Contrl%(6),Ptsin%(0,0),Ptsin%(1,0),
        Ptsin%(0,1),Ptsin%(1,1),Intin%(0),Intin%(1))
  Contrl%(5)=6
  VDI (11,Contrl%(12),Intin%(2),Ptsin%(0,2),
      Intout%(0),Ptsout%(0,0))
RETURN
'
DEF PROC V_Ellpie(Contrl%(6),Ptsin%(0,0),Ptsin%(1,0),
        Ptsin%(0,1),Ptsin%(1,1),Intin%(0),Intin%(1))
  Contrl%(5)=7
  VDI (11,Contrl%(12),Intin%(2),Ptsin%(0,2),Intout%(0),
      Ptsout%(0,0))
RETURN
'
DEF PROC V_Rbox(Contrl%(6),Ptsin%(0,0),Ptsin%(1,0),
                Ptsin%(0,1),Ptsin%(1,1))
  Contrl%(5)=8
  VDI (11,Contrl%(12),Intin%(0),Ptsin%(0,2),Intout%(0),
      Ptsout%(0,0))
RETURN
'
DEF PROC V_Rfbox(Contrl%(6),Ptsin%(0,0),Ptsin%(1,0),
        Ptsin%(0,1),Ptsin%(1,1))
  Contrl%(5)=9
  VDI (11,Contrl%(12),Intin%(0),Ptsin%(0,2),Intout%(0),
      Ptsout%(0,0))
RETURN
'
DEF PROC V_Justified(Contrl%(6),Ptsin%(0,0),Ptsin%(1,0),
        Text$, Ptsin%(0,1),Intin%(0),Intin%(1))
  LOCAL I%:Contrl%(5)=10
  FOR I%=1 TO LEN(Text$):Intin%(I%+1)= ASC( MID$(Text$,I%)):
  NEXT Intin%(I%+1)=0
  VDI (11,Contrl%(12),Intin%(I%+1),Ptsin%(0,2),Intout%(0),
      Ptsout%(0,0))
RETURN
'
```

(suite)

```
'
' FONCTIONS DE TRACÉS OU REMPLISSAGES
'          ----------------------
'
DEF PROC V_Pline(Contrl%(6),X%)
   VDI (6,Contrl%(12),Intin%(0),Ptsin%(0,X%),Intout%(0),
        Ptsout%(0,0))
RETURN
'
DEF PROC V_Pmarker(Contrl%(6),X%)
   VDI (7,Contrl%(12),Intin%(0),Ptsin%(0,X%),Intout%(0),
        Ptsout%(0,0))
RETURN
'
DEF PROC V_Fillarea(Contrl%(6),X%)
   VDI (9,Contrl%(12),Intin%(0),Ptsin%(0,X%),Intout%(0),
        Ptsout%(0,0))
RETURN
'
DEF PROC V_Cellarray(Contrl%(6),Ptsin%(0,0),Ptsin%(1,0),
            Ptsin%(0,1),Ptsin%(1,1),Contrl%(7),Contrl%(8),
            Contrl%(9),Contrl%(10))
   VDI (10,Contrl%(12),Intin%(Contrl%(8)*Contrl%(9)),
        Ptsin%(0,2),Intout%(0),Ptsout%(0,0))
RETURN
'
DEF PROC V_Contourfill(Contrl%(6),Ptsin%(0,0),
                       Ptsin%(1,0),Intin%(0))
   VDI (103,Contrl%(12),Intin%(1),Ptsin%(0,2),Intout%(0),
        Ptsout%(0,0))
RETURN
'
DEF PROC Vr_Recfl(Contrl%(6),Ptsin%(0,0),Ptsin%(1,0),
                  Ptsin%(0,1),Ptsin%(1,1))
   VDI (114,Contrl%(12),Intin%(0),Ptsin%(0,4),
        Intout%(0),Ptsout%(0,0))
RETURN
'
'COULEURS ET MODES DE TRACÉS
'
DEF PROC Vswr_Mode(Contrl%(6),Intin%(0))
   VDI (32,Contrl%(6),Intin%(1),Ptsin%(0,0),
        Intout%(1),Ptsout%(0,0))
RETURN
'
```



(suite)

```
DEF PROC Vs_Color(Contrl%(6),Intin%(0),Intin%(1),
                  Intin%(2),Intin%(3))
   VDI (14,Contrl%(6),Intin%(4),Ptsin%(0,0),Intout%(0),
        Ptsout%(0,0))
RETURN
'
DEF PROC Vsl_Color(Contrl%(6),Intin%(0))
   VDI (17,Contrl%(6),Intin%(1),Ptsin%(0,0),
        Intout%(1),Ptsout%(0,0))
RETURN
'
DEF PROC Vsl_Type(Contrl%(6),Intin%(0))
   VDI (15,Contrl%(6),Intin%(1),Ptsin%(0,0),Intout%(1),
        Ptsout%(0,0))
RETURN
'
DEF PROC Vsl_Udsty(Contrl%(6),R Intout%(0))
   VDI (113,Contrl%(6),Intin%(1),Ptsin%(0,0),
        Intout%(0),Ptsout%(0,0))
RETURN
'
DEF PROC Vsl_Width(Contrl%(6),Ptsin%(0,0))
   VDI (16,Contrl%(6),Intin%(0),Ptsin%(0,1),
        Intout%(0),Ptsout%(0,1))
RETURN
'
DEF PROC Vsl_Ends(Contrl%(6),Intin%(0),Intin%(1))
   VDI (108,Contrl%(6),Intin%(2),Ptsin%(0,0),Intout%(0),
        Ptsout%(0,0))
RETURN
'
DEF PROC Vsm_Type(Contrl%(6),Intin%(0))
   VDI (18,Contrl%(6),Intin%(1),Ptsin%(0,0),Intout%(1),
        Ptsout%(0,0))
RETURN
'
DEF PROC Vsm_Height(Contrl%(6),Ptsin%(1,0)):Ptsin%(0,0)=0
   VDI (19,Contrl%(6),Intin%(0),Ptsin%(0,1),Intout%(0),
        Ptsout%(0,1))
RETURN
'
DEF PROC Vsm_Color(Contrl%(6),Intin%(0))
   VDI (20,Contrl%(6),Intin%(1),Ptsin%(0,0),Intout%(1),
        Ptsout%(0,0))
RETURN
```

(suite)

```
'
DEF PROC Vst_Height(Contrl%(6),Ptsin%(1,0),R Ptsout%(0,0),
   R Ptsout%(1,0),R Ptsout%(0,1),R Ptsout%(1,1))
   VDI (12,Contrl%(6),Intin%(0),Ptsin%(0,1),Intout%(0),
        Ptsout%(0,2))
RETURN
'
DEF PROC Vst_Point(Contrl%(6),Intin%(0),R Ptsout%(0,0),
   R Ptsout%(1,0),R Ptsout%(0,1),R Ptsout%(1,1))
   VDI (107,Contrl%(6),Intin%(1),Ptsin%(0,0),Intout%(0),
        Ptsout%(0,2))
RETURN
'
DEF PROC Vst_Rotation(Contrl%(6),Intin%(0))
   VDI (13,Contrl%(6),Intin%(1),Ptsin%(0,0),Intout%(1),
        Ptsout%(0,0))
RETURN
'
DEF PROC Vst_Font(Contrl%(6),Intin%(0))
   VDI (21,Contrl%(6),Intin%(1),Ptsin%(0,0),
        Intout%(1),Ptsout%(0,0))
RETURN
'
DEF PROC Vst_Color(Contrl%(6),Intin%(0))
   VDI (22,Contrl%(6),Intin%(1),Ptsin%(0,0),
        Intout%(1),Ptsout%(0,0))
RETURN
'
DEF PROC Vst_Effects(Contrl%(6),Intin%(0))
   VDI (106,Contrl%(6),Intin%(1),Ptsin%(0,0),
        Intout%(1),Ptsout%(0,0))
RETURN
'
DEF PROC Vst_Alignment(Contrl%(6),Intin%(0),Intin%(1),
                       R Intout%(0),R Intout%(1))
   VDI (39,Contrl%(6),Intin%(2),Ptsin%(0,0),
        Intout%(2),Ptsout%(0,0))
RETURN
'
DEF PROC Vsf_Interior(Contrl%(6),Intin%(0))
   VDI (23,Contrl%(6),Intin%(1),Ptsin%(0,0),
        Intout%(1),Ptsout%(0,0))
RETURN
```



(suite)

```
'
DEF PROC Vsf_Style(Contrl%(6),Intin%(0))
   VDI (24,Contrl%(6),Intin%(1),Ptsin%(0,0),
        Intout%(1),Ptsout%(0,0))
RETURN
'
DEF PROC Vsf_Color(Contrl%(6),Intin%(0))
   VDI (25,Contrl%(6),Intin%(1),Ptsin%(0,0),
        Intout%(1),Ptsout%(0,0))
RETURN
'
DEF PROC Vsf_Perimeter(Contrl%(6),Intin%(0))
   VDI (104,Contrl%(6),Intin%(1),Ptsin%(0,0),
        Intout%(1),Ptsout%(0,0))
RETURN
'
DEF PROC Vsf_Updat(Contrl%(6),Planes%)
   VDI (112,Contrl%(6),Intin%(Planes% SHL4),
   Ptsin%(0,0),Intout%(0),Ptsout%(0,0))
RETURN
'
'----------FONCTIONS ESCAPE ---------
'          ----------------
'
DEF PROC V_Exit_Cur
   PRINT "";: RETURN
'
DEF PROC V_Enter_Cur
   PRINT "";: RETURN
'
DEF PROC V_Curup
   PRINT "";: RETURN
'
DEF PROC V_Curdown
   PRINT "";: RETURN
'
DEF PROC V_Curright
   PRINT "";: RETURN
'
DEF PROC V_Curleft
   PRINT "";: RETURN
'
DEF PROC V_Curhome
   PRINT "";: RETURN
'
DEF PROC V_Eeos
   PRINT "";: RETURN
```

(suite)

```
'
DEF PROC V_Eeol
   PRINT "";: RETURN
'
DEF PROC V_Curaddress(Y%,X%)
   PRINT @(Y%-1,X%-1);: RETURN
'
DEF PROC V_Curtext(Text$)
   PRINT Text$;: RETURN
'
DEF PROC V_Rvon
   PRINT "";: RETURN
'
DEF PROC V_Rvoff
   PRINT "";: RETURN
'
DEF PROC Vq_Curaddress(R Y%,R X%)
   Y%= CSRLIN :X%= POS(0)+1: RETURN
'
DEF PROC Vq_Tabstatus(Contrl%(6),R Intout%(0))
   Contrl%(5)=16
   VDI (5,Contrl%(6),Intin%(0),Ptsin%(0,0),
        Intout%(1),Ptsout%(0,0))
RETURN
'
DEF PROC V_Hardcopy
   XBIOS (,20): RETURN
'
DEF PROC V_Dspcur(Contrl%(6),Ptsin%(0,0),Ptsin%(1,0))
   Contrl%(5)=18
   VDI (5,Contrl%(6),Intin%(0),Ptsin%(0,1),
        Intout%(0),Ptsout%(0,0))
RETURN
'
DEF PROC V_Rmcur
   Contrl%(5)=19
   VDI (5,Contrl%(6),Intin%(0),Ptsin%(0,0),
        Intout%(0),Ptsout%(0,0))
RETURN
'
```



LA GESTION DES CHAINES DE CARACTERES

A quelques différences de syntaxe près, le Basic OMIKRON dispose à peu près des mêmes instructions de traitement de chaînes de caractères que le Basic GFA.

L'addition de chaînes s'effectue par l'opérateur **+** et la multiplication de chaînes par l'opérateur *. Attention ! on ne peut multiplier qu'une même chaîne un certain nombre de fois et l'opérateur * est placé après la chaîne à multiplier.

Exemple :

```
A$="WE"
B$="KA"
A$=A$+B$
PRINT A$*4
WEKAWEKAWEKAWEKA
OK
```

STRING$()

Cette opération fonctionne comme la multiplication des chaînes, mais sur un seul caractère à la fois. Par contre, il n'est pas possible de multiplier un caractère en donnant son code ASCII comme on peut le faire en GFA par cette même instruction.

Exemple :

```
REM En OMIKRON:

A$=STRING$(5,"A")
PRINT A$
AAAAA
OK

REM En GFA:

A$=STRING$(5,"A")
OK
PRINT A$
OK
AAAAA
OK
A$=STRING$(5,65)
OK
PRINT A$
OK
AAAAA
OK
```

SPACE$() et SPC()

SPACE$(X) construit une chaîne de caractères contenant x espaces. x ne peut excéder 32767.SPC(X), il fonctionne exactement comme SPACE$ et utilise en plus l'instruction PRINT pour générer des espaces à l'impression. En Basic GFA, cette instruction ne produit pas de chaîne de caractères mais affiche x espaces dans l'instruction PRINT. C'est le fonctionnement usuel de cette instruction dans la majorité des Basic.

Exemple :

```
EXEMPLE$="OMIKRON"
PRINT EXEMPLE$;SPC(2);"WK";
OMIKRON  WK
OK
BLANC$=SPC(2)
PRINT EXEMPLE$;SPC(2);BLANC$;"WK";
OMIKRON  WK
OK
```

CHR$() et ASC()

Ces deux instructions fonctionnent comme en GFA et renvoient respectivement le caractère dont le code ASCII est précisé en argument et le code ASCII du caractère argument. Toutefois, dans le cas où l'on recherche le code ASCII d'une chaîne, si la chaîne est vide, le code ASCII renvoyé n'est pas zéro comme en GFA, mais c'est le message d'appel illégal de fonction qui apparaît.

LEFT$(),RIGHT$(),MID$(), INSTR(),MIRROR$()

Ces fonctions ont exactement le même fonctionnement qu'en GFA mais ce dernier langage possède des extensions de ces fonctions qui permettent un maniement plus souple des chaînes de caractères. Les trois premières instructions permettent d'extraire des sous-chaînes d'une chaîne de départ en partant de la gauche (LEFT$()), de la droite (RIGHT$()), ou d'une position médiane (MID$()).

Exemple :

```
A$="COMMENT çà va?"
PRINT LEFT$(A$,5) :REM 5 caracteres a partir de la gauche
COMME
OK
PRINT RIGHT$(A$,5):REM 5 Caractères à partir de la droite
à va?
OK
PRINT MID$(A$,4,3) :REM 3 caractères à partir du 4ème
MEN
OK
```



En OMIKRON on ne dispose pas comme en GFA de la forme d'instruction **MID$()** qui permet d'insérer une chaîne de caractères dans une autre sur une position quelconque, ni des instructions **LSET** et **RSET** qui permettent d'insérer une chaîne à partir de la position gauche ou droite.

La fonction **INSTR(x,a$,b$)** recherche une chaîne **b$** dans une chaîne **a$** à partir du caractère x. Si cette valeur n'est pas précisée, la recherche s'effectue à partir du premier caractère de la chaîne **a$**. La recherche s'effectue de gauche à droite et on ne dispose pas de l'instruction **RINSTR()** comme en GFA.

La fonction **MIRROR$()** est unique au Basic OMIKRON et permet d'inverser une chaîne de caractères.

Exemple :

```
PRINT MIRROR$("AZE")
EZA
OK
REM:Petit exemple récapitulatif:
A$=STRING$(3,"?")
C$="1234"
B$=LEFT$(A$,2)+RIGHT$(C$,2)
PRINT B$
??34
OK
B$=MIRROR$(B$)
PRINT B$
43??
OK
B$=LEFT$(B$,3)+"!"
D$=LEFT$(MIRROR$(RIGHT$(B$,3)),2)
PRINT D$
!?
OK
```

UPPER$() et LOWER$()

Ces deux instructions transforment respectivement en majuscules et minuscules tous les caractères lettres d'une chaîne de caractères sans toucher aux chiffres ou caractères spéciaux. Les accents sont concernés par ces instructions. L'utilisation principale est la saisie de noms propres ou communs au clavier en vue de tests sur l'écriture de ces noms.

LEN(), VAL(), STR$()

LEN(a$) donne la longueur de la chaîne **a$**. Comme en GFA la longueur d'une chaîne vide est zéro.

Les fonctions **VAL()** et **STR$()** sont inverses l'une de l'autre. Elles servent à convertir une chaîne en une valeur numérique décimale et

inversement. Comme en GFA, l'opération de conversion s'arrête dès que l'on rencontre un caractère impossible à convertir. Lorsque la chaîne ne commence pas par un chiffre, la valeur zéro est renvoyée.

Exemple :

```
A$="$AA"
PRINT VAL(A$)
 176
OK
A$=STR$(176)
PRINT A$
 176
OK
PRINT STR$($AA)
 176
PRINT HEX$(STR$($AA))
$AA
OK
```

L'OMIKRON ne dispose pas des fonctions PRED(), SUCC() et TRIM$() qui permettent de traiter les codes ASCII ou valeurs numériques en séquences ascendantes ou descendantes et supprimer rapidement les espaces en bouts de chaînes.

D'une façon générale, l'OMIKRON dispose d'un jeu d'instructions moins complet que le GFA pour traiter les chaînes de caractères, à la différence de ce qui se produisait pour les fonctions mathématiques.

LES CONDITIONS ET LES BOUCLES

Instructions de décision

IF condition THEN Instructions ELSE Instructions ENDIF

Exemple :

```
INPUT "Rentre un nombre égal à un ou deux";a
IF a=1 THEN PRINT "GAGNE a égale 1":END
IF a=2 THEN PRINT "PERDU a égale 1":END

REM Ce programme peut s'écrire de la même façon:
INPUT "Rentre un nombre égal à un ou deux";a
IF a=1 THEN
PRINT "GAGNE a égale 1"
END
ENDIF
IF a=2
THEN PRINT "PERDU a égale 1"
END
ENDIF
```



Cette structure du **IF** est similaire à celle du GFA. Petites différences à noter, le **THEN** est ici obligatoire alors qu'il est facultatif en GFA, et le **ENDIF** peut être omis, comme en C. Il va sans dire que pour structurer le programme, il est préférable de séparer conditions et instructions sur des lignes distinctes afin de mieux repérer chaque étage logique de la programmation.

Une variante utilise le **ELSE** en supposant que l'opérateur ne rentre que 1 ou 2.

Exemple :

```
INPUT "Rentre un nombre égal à un ou deux";a
IF a=1
THEN PRINT "GAGNE a égale 1"
ELSE IF a=2
THEN PRINT "PERDU a égale 1"
ENDIF
ENDIF
```

Une autre écriture valide serait :

```
INPUT "Rentre un nombre égal à un ou deux";a
IF a=1 THEN
PRINT "GAGNE a égale 1"
ELSE IF a=2 THEN PRINT "PERDU a égale 1"
ENDIF
```

A la différence du GFA, on ne dispose pas de la variante **ELSEIF** de cette instruction, ce qui permettrait d'imbriquer de façon plus claire les instructions **IF** successives correspondant à plusieurs tests. Le **ENDIF** final est le seul à préciser dans ce cas ce qui permet de n'écrire que les différents cas de **ELSEIF** à l'intérieur du **IF** principal. Cette structure est tout à fait comparable à :

SELECT
CASE
ENDSELECT

du langage **C**, qu'on retrouve d'ailleurs aussi en GFA ; cette forme d'instruction n'existait pas dans les premières versions du Basic et ont été rajoutées dans le but de structurer le langage. Nul doute que ces formes de programmation des instructions précédentes apparaîtront dans les **UPDATES** à venir. Les branchements multiples sont par contre disponibles en OMIKRON grâce à l'instruction : **ON Expression numérique GOSUB Sous progr1,Sous prog2,...** Cette instruction

saute au sous-programme, à la procédure ou à la fonction dont le nom est précisé dans l'instruction à la suite de **GOSUB** ; à la différence de ce qui était possible en GFA, on peut transmettre des paramètres d'appel aux procédures. L'expression numérique doit donner un résultat entier. Dans le cas où l'expression numérique donne un résultat fractionnaire, les chiffres après la virgule seront ignorés. Si l'expression numérique est inférieure à un ou supérieure au nombre de procédure spécifiées après **GOSUB**, aucune fonction n'est appelée et on passe à l'instruction qui suit le **GOSUB**.

Exemple :

```
X=4
ON X GOSUB Prog1,Prog2,Prog3,Prog4
ON X-2 GOSUB Prog1,Prog2,Prog3,Prog4
```

Le programme (Procédure, Fonction) Prog4 sera d'abord appelé, puis le programme Prog2.

Les boucles

Le Basic OMIKRON dispose de trois types de boucles. La boucle itérative à bornes fixées **FOR NEXT** commune à tous les langages de programmation ainsi que les boucles conditionnelles **WHILE WEND** et **REPEAT UNTIL**. Par contre à ce jour, on ne dispose pas de forme de boucle inconditionnelle du style **DO LOOP** du GFA.

FOR...NEXT
FOR I=a to b STEP s
NEXT I

La boucle **FOR NEXT** sert à répéter le corps de la boucle (instructions comprises entre le **FOR** et le **NEXT**) un certain nombre de fois. La valeur de départ a est affectée à la variable **I**. On effectue ensuite les instructions du corps de la boucle. Suivant le signe du pas (la valeur de s peut être positive ou négative) précisé par **STEP**, on regarde si on a dépassé ou égalé la valeur limite d'arrivée donnée par b. Si c'est le cas, on passe à l'instruction suivant le **NEXT**, sinon on recommence à la première instruction de la boucle jusqu'à ce que la condition de sortie soit réalisée. Le corps de cette boucle est donc toujours traité au moins une fois.

Exemple :

```
FOR I=1 to 10 STEP 2
PRINT I;
NEXT I
PRINT
FOR I=10 TO 3 STEP -2
PRINT I;
NEXT
1 3 5 7 9
10 8 6 4
```



Notez que lorsque le NEXT n'est pas écrit sur la même ligne que le FOR, on peut omettre de préciser le nom de la variable d'itération. Le programme précédent aurait pu aussi s'écrire :

```
FOR I=1 to 10 STEP 2 : PRINT I;: NEXT I
PRINT
FOR I=10 TO 3 STEP -2 :PRINT I;: NEXT I
```

WHILE ...WEND
WHILE condition
Instructions
WEND

Les instructions WHILE et WEND encadrent le corps des instructions qui seront traitées aussi longtemps que la condition sera remplie. La condition est testée en début de boucle et si la condition n'est pas remplie, on peut très bien ne jamais effectuer cette boucle.

Exemple :

```
A$="W"
WHILE RIGHT$(INKEY$,1)<>A$
PRINT "BOO!";
WEND
END
REM: affiche BOO!BOO!.....
REM:jusqu'à ce que vous tapiez un W

AS="W"
WHILE RIGHT$(INKEYS,1)=A$
PRINT "BOO!";
WEND
END

REM Vous sortez tout de suite du programme.
```

REPEAT
Instructions
UNTIL Condition

A la différence de l'instruction conditionnelle précédente, la condition est testée après le corps de la boucle. Ce dernier est donc toujours traité au moins une fois, comme dans le cas de la boucle FOR NEXT.

Exemple :

```
REPEAT
A$=INPUT$(1)
UNTIL ASC(A$)=13
PRINT "C'EST FINI"
END
REM:On attend de frapper RETURN pour sortir du programme.

REPEAT
A$=INPUT$(1)
UNTIL LEN(INKEY$)
END
PRINT "C'EST FINI"
REM:On sort dès qu'on frappe sur le clavier

REPEAT
UNTIL LEN(INKEY$) OR TIMER
END
PRINT "C'EST FINI"
REM:On sort tout de suite
```

En OMIKRON, on dispose aussi d'une instruction qui permet de sortir d'une boucle même si la condition finale de sortie n'est pas encore réalisée. Il s'agit de l'instruction **EXIT**.

EXIT sans étiquette ferme toutes les boucles et continue le programme à la suite des boucles, (comparable au **EXIT IF condition=TRUE** du GFA).

Exemple :

```
MOUSEON
WHILE MOUSEDEBUT=0
IF ASC(RIGHT$(INKEY$,1))=13 THEN EXIT
WEND
REPEAT
REPEAT UNTIL MOUSEDEBUT
IF MOUSEDEBUT AND 3 THEN EXIT TO Fin
UNTIL Sortie
-Fin: PRINT:"C'EST LA FIN":RUN
END
```

Si on ne clique pas sur la souris, on sort de la première boucle en tapant **RETURN** ; la variable **Sortie** n'est jamais initialisée et sera toujours fausse car contenant la valeur zéro.



En utilisant une étiquette à la suite d'**EXIT**, on donne l'ordre au programme de terminer les boucles en cours puis de continuer à l'étiquette référencée. La syntaxe de l'instruction à ce moment là est **EXIT TO Etiquette**. La condition de sortie de boucle **UNTIL Sortie** ne sera jamais réalisée. Le moyen de sortir de cette boucle ici est de cliquer sur les deux touches de la souris. La commande **RUN** renvoie au début du programme. Pour sortir complètement du programme, il faut utiiser la commande **CONTROL C**.

LES PROCÉDURES ET LES FONCTIONS

Sous le Basic OMIKRON, les sous-programmes reçoivent des noms comme dans n'importe quel langage de programmation évolué. Ces sous-programmes peuvent recevoir des paramètres et être appliqués à des variables différentes. On peut aussi bien transmettre la valeur de la variable que la variable elle-même. A la différence du GFA, les procédures deviennent l'équivalent des mots clefs du langage dès lors qu'elles sont définies ; elles sont alors accessibles depuis l'éditeur standard sans avoir besoin de taper **RUN**.

La définition d'une procédure se fait entre les mots clefs **DEF PROC** et **RETURN**.

Exemple :

```
DEF PROC Carre(A)
    B=A^2
PRINT "LE CARRE DE:";A,"VAUT:";B
RETURN
```

La variable **A** est un paramètre d'appel de la procédure. Comme dans tous les langages évolués, il est considéré comme une variable locale à cette procédure. Si une variable **A** avait été définie avant l'appel de cette procédure, elle serait inchangée.

Exemple :

```
A=12
PRINT "AVANT L'APPEL DE CARRE:"A
Carre(A)
PRINT "APRES L'APPEL DE CARRE:"A
END
DEF PROC Carre(A)
    B=A^2
PRINT "LE CARRE DE:";A,"VAUT:";B
RETURN
12
OK
144
OK
12
OK
```

Si la variable A était utilisée dans la procédure sans être définie comme un paramètre d'appel, elle ne serait plus définie comme variable locale et pourrait être modifiée comme n'importe quelle autre variable globale.

Exemple :

```
A=10
PRINT "AVANT L'APPEL DE CARRE:"A
Carre
PRINT "APRES L'APPEL DE CARRE:"A
END
DEF PROC Carre
    A=A^2
PRINT "LE CARRE VAUT:";A
RETURN
10
OK
100
OK
100
OK
```

La définition de variables locales se fait comme en GFA par la commande LOCAL.

Exemple :

```
A=10
PRINT "AVANT L'APPEL DE CARRE:"A
Carre
PRINT "APRES L'APPEL DE CARRE:"A
END
DEF PROC Carre
LOCAL A
A=A^2
PRINT "LE CARRE VAUT:";A
RETURN
10
OK
0
OK
10
OK
```

En définissant une procédure, on peut définir plusieurs paramètres d'appel. Les variables passées doivent bien sûr appartenir au même type que les celles définies par la procédure sous peine de déclencher une erreur de « confusion de type ».



Exemple :

```
DEF PROC Affich_envers(T$,Nombre)
LOCAL I
IF Nombre<= LEN(T$) THEN
FOR I=LEN(T$) TO LEN(T$)-Nombre+1
PRINT MIDS(T$,I,1)
WAIT 10
NEXT I
ELSE
PRINT Nombre,;"EST TROP GRAND"
RETURN
REM:Vous pouvez taper ensuite en mode direct:
'Affich_envers("OMIKRON",3)'
REM:donne
NOR
OK
```

A la différence du GFA, les procédures peuvent aussi renvoyer des valeurs alors que cette spécificité est limitée aux fonctions dans ce langage. Plusieurs valeurs peuvent être renvoyées. Il suffit de noter les valeurs à retourner par un **R** dans la définition de la procédure. Mais attention, ce **R** n'est pas à écrire au moment de l'appel de la procédure. Dans l'exemple classique qui suit nous allons aussi utiliser la propriété de récursivité des procédures (propriété de s'appeler soi-même) pour retourner la valeur de la factorielle.

Exemple :

```
Facto(5,Total)
PRINT Total
120
OK
DEF PROC Facto(Nombre,R Resultat)
LOCAL J
IF Nombre=0 THEN Resultat=1
IF Nombre>0
Resultat=1
FOR J=1 TO Nombre
Resultat=Resultat*J
NEXT J
ENDIF
RETURN

Recur(5,Total)
PRINT Total
120
OK
DEF PROC Recur(Nombre,R Resultat)
IF Nombre<2 THEN Resultat=1
IF Nombre>2
Resultat=Resultat*Recur(Nombre-1)
ENDIF
RETURN
```

Comme le GFA, l'OMIKRON permet aussi de définir des fonctions. Ces dernières ne renvoient alors qu'un seul paramètre et peuvent être définies sur une ou plusieurs lignes.

DEF FN Nom(par1,par2,...)=Expression calculable

définit une fonction sur une ligne par le signe =.

Cette fonction s'appelle par FN(val1,val2,...)

Exemple :

```
DEF FN polynome(A,B,C,X)=A*X^2+B*X+C
```

Ce même exemple peut se programmer avec une fonction sur plusieurs lignes. Le signe égal disparaît et la définition du corps de la fonction se fait entre les instructions DEF FN et RETURN suivi de la valeur de retour.

Exemple :

```
DEF FN polynome(A,B,C,X,Y)
Y=A*X^2+B*X+C
RETURN Y
PRINT FN polynome(2,3,5,2)
19
OK
```

Cet exemple peut aussi se programmer :

```
DEF FN polynome(A,B,C,X)
RETURN (A*X^2+B*X+C)
PRINT FN polynome(2,3,5,2)
19
OK
```

Comme pour les procédures une ou plusieurs variables locales peuvent être définies. Si la fonction retourne une chaîne de caractères, le suffixe du nom de la fonction doit être un $. C'est le suffixe qui fixe le type de la fonction, comme c'était le cas dans les variables.

Exemple :

```
DEF FN Prem_der$(A$)=LEFT$(A$,1)+RIGHT$(A$,1)
B$=Prem_der$("OMIKRON")
PRINT B$
END
ON
OK
```



Autre programmation sur plusieurs lignes :

```
DEF FN Prem_der$(A$)
A$=LEFT$(A$,1)+RIGHT$(A$,1)
RETURN A$
PRINT Prem_der$("WEKA")
END
WK
OK
```

LES ENTRÉES SORTIES GÉNÉRALES

Les lignes de données

READ DATA RESTORE

DATA expression1,expression2,...

En **DATA** on peut évidemment spécifier des constantes numériques ou alphanumériques qui seront lues ensuite par l'instruction **READ** mais, innovation par rapport au GFA, on peut aussi spécifier des expressions numériques ou alphanumériques ainsi que des tableaux ce qui évite bien souvent de devoir retaper des **DATA** dont le mode de génération est le même. Les valeurs ou expressions sont séparées par des virgules. Lorsque c'est une chaîne de caractères contenant une virgule qui doit être lue, l'ensemble de la chaîne doit être placé entre guillemets. Comme dans tous les Basic, il faut vérifier que les **DATA** sont en assez grand nombre et que le type de **DATA** correspond au type de variables auxquelles ils vont être attribués. Les valeurs numériques peuvent être spécifiées sous forme hexadécimale, octale ou binaire par les préfixes appropriés (voir le début du cours sur l'OMIKRON).

Exemple :

```
DATA 22,Pot,FACT(12),&22,ARCSIN(A!)
REM: 5 VALEURS OU EXPRESSIONS NUMÉRIQUES
DATA $12,Aze$
REM:UNEVALEURNUMÉRIQUE,UNEVARIABLEALPHANUMÉRIQUE
DATA Aze$+Qsd$+" ",Toto(12)
REM: une expression alphanumérique , un tableau
DATA "",+"","+",""
REM: une valeur alphanumérique, une expression alphanumérique
```

Les instructions **DATA** et **READ** fonctionnent avec un pointeur de données. Ce pointeur désigne en permanence la prochaine valeur de **DATA** qui sera lue. Au début du programme, le pointeur est placé sur la première valeur de DATA et est incrémenté à chaque **READ**.

L'instruction RESTORE permet de placer ce pointeur à l'adresse suivant cette instruction. Cette adresse peut être soit un numéro de ligne soit une étiquette. Sans paramètre, RESTORE replace le pointeur de lecture au début des DATA.

On dispose aussi de l'option :

ON <expression numérique> RESTORE Etiq1,Etiq2,...
REM:Cette instruction fonctionne exactement comme :
ON <expression numérique> GOTO
ON <expression numérique> GOSUB

Exemple :

```
DIM Tab(5)
INPUT "PARITE:";Parite
Parite=MAX(1,Parite)
Parite=MIN(0,Parite)
REM: Rend Parite à 0 ou 1
REM: On prendra 0 pour pair et 1 pour impair
ON Parite+1 RESTORE ,Etpair,Etimpair
FOR I=1 TO 5
READ Tab(I)
NEXT I
FOR I=1 TO 5
PRINT Tab(I);
NEXT I
-Etpair:
DATA 0,2,4,6,8
-Etimpair:
DATA 1,3,5,7,9
REM si on tape une valeur différente de 0 en entrée
1 3 5 7 9
OK
REM Si on tape une valeur égale à zéro
0 2 4 6 8
OK
```

Les entrées de données

L'OMIKRON dispose de cinq instructions simples permettant l'entrée de données. Lorsqu'on veut lire ou envoyer des octets sur les ports MIDI ou autres, on ne dispose pas des instructions simplifiées du GFA :

INP(N° de périphérique) ou OUT(N° de périphérique)

Par contre, on peut accéder à ces fonctions par l'intermédiaire des fonctions INPUT$, BIOS, XBIOS et OPEN en ouvrant un canal de données vers un fichier ou un périphérique.



INKEY$

INKEY$ n'attend pas qu'une touche du clavier ait été frappée, mais renvoie une chaîne vide si aucune touche n'a été actionnée depuis le dernier test du clavier. Par rapport à la majorité des Basic (dont le GFA), OMIKRON ne retourne pas simplement le code ASCII de la touche mais une chaîne de quatre caractères octets définie comme suit :

N°Octet	1	2	3	4
	STATUT	SCANCODE	INUTILE	CODE-ASCII

Le premier octet contient bit par bit le statut des touches spéciales :

shift droit	bit 0
shift gauche	bit 1
[contr1]	bit 2
[alt]	bit 3
[caps lock]	bit 4

Il suffit de lire ASC(MID$(INKEY$,Numoctect,1)) pour connaître la valeur de l'octet correspondant.

INPUT

L'instruction INPUT peut être utilisée avec différentes variantes, pour entrer des variables ou listes de variables avec ou sans texte.

Par rapport à la majorité des Basic, l'OMIKRON peut :

– positionner à l'écran le texte du message et du point d'interrogation signifiant l'attente de la valeur à entrer par le clavier ;
– constituer un texte comme opération sur des chaînes de caractères. Le séparateur du texte et de la variable est le point virgule.

Par contre, comme dans toutes les opérations de saisie de données et d'affectations de ces données dans des variables, il faut que données et variables soient du même type. Dans le cas contraire, l'entrée doit être recommencée. Tant que la touche RETURN ou ENTER n'a pas été actionnée, les caractères peuvent être effacés à l'aide des touches DELETE et BACKSPACE. Chaque variable peut être confirmée séparément en actionnnant les deux touches précitées ou bien en les séparant par des virgules en entrée et en actionnant une seule fois la touche RETURN. Toutefois, pour que des virgules puissent être entrées comme faisant partie d'une chaîne de caractères, il faut utiliser l'instruction LINE INPUT ou INPUT USING.

Exemple :

```
F$="C'est BON "
D$="O.K"
INPUT A$
INPUT "";B$
INPUT "Entrez deux valeurs:";A,Bb$
INPUT C$+D$;E$
INPUT @(5,6);"Valeur 5eme ligne 6eme colonne";B
REM Le programme entre deux chaînes de caractères.
REM La première attente se présente avec ?
REM la seconde sans texte
REM Après le texte on attend une valeur numérique
REM puis une chaîne à valider en fin avec RETURN
REM L'avant dernier INPUT lit la valeur à affecter à E$
REM après le message C'est BON O.K
```

LINE INPUT

L'intérêt de cette instruction est de pouvoir entrer des virgules pour une variable alphanumérique. Par contre, la virgule n'est plus un séparateur de variables et cette instruction ne permet de saisir qu'un seule variable. Il faut utiliser un LINE INPUT par variable. Les explications données pour INPUT en ce qui concerne l'entrée d'une variable et la correction de l'entrée jusqu'à ce que la touche RETURN ou ENTER ait été actionnée sont valables dans ce cas.

Exemple :

```
LINE INPUT A$
INPUT B$
PRINT A$,B$
REM:Entrez deux fois le texte OMI,KRON
REM:Vous verrez apparaître sur le moniteur OMI,KRON et OMI.
```

INPUT$
A$=INPUT$(Nbre)
A$=INPUT$(Nbre,Numéro)

INPUT$ lit au clavier une chaîne de caractères composée de Nbre de caractères. L'indication de Numéro de fichier est optionnelle et permet de lire des caractères dans un fichier. L'expression Nbre indique dans les deux cas le nombre de caractères qui doivent être lus. Attention, dans les deux cas on attend d'avoir lu le nombre de caractères précisé avant de passer à l'instruction suivante.



Exemple :

```
OPEN "O",1,"VERSION.DAT"
REM: ouverture du fichier de nom VERSION.DAT
REM: en écriture sur le canal 1
PRINT #1,"OMIKRON VERSION 3.01"
REM: écriture du texte sur ce fichier
CLOSE #1
REM: fermeture du canal 1 correspondant
REM: au fichier ouvert
OPEN "I",#1,"VERSION.DAT"
REM: ouverture du fichier de nom VERSION.DAT
REM: en lecture sur le canal 1
Lu$=INPUT$(7,#1)
REM: On lit les 7 premiers caractères
REM: du fichier, soit OMIKRON
CLOSE #1
REM: ON ferme le canal de lecture
PRINT Lu$
REM: on imprime OMIKRON
PRINT "INDIQUER LE NUMERO DE LA VERSION!"
REPEAT
Test$= INPUT$(4)
REM: On attend 4 caractères rentrés au clavier(3.01)
UNTIL Test$="3.01"
REM: on ne sort de la boucle que lorsque
REM le numéro de version est correct
END
```

INPUT USING

INPUT@(X,Y);"texte";<chaîne> USING <Chaîne de commande> <Validation>,<longueur>,<Caractère de remplissage>,<Position du curseur>

Cette instruction est particulièrement complexe et puissante. Elle est originale au Basic OMIKRON et permet de filtrer et transformer les caractères lus de façon similaire à ce qui se passe au cours des traitements de lectures de fichiers alphanumériques en COBOL.

Les trois premiers paramètres de l'instruction sont les mêmes que précédemment.

- *Chaîne de commande*

Son rôle est de permettre ou d'interdire certains symboles et de convertir certains symboles en d'autres. C'est à l'aide de certains caractères de commande que se fait le codage de ces fonctions.

Codages des commandes de permission ou d'interdiction

Tous les symboles de commande peuvent s'inscrire en séquence, l'ordre n'a pas d'importance car c'est la globalité de la chaîne de commande qui est analysée.

0 Tous les chiffres sont autorisés.
a Toutes les lettres sont autorisées.
% Les signes spéciaux sont autorisés.
^ Autorise la combinaison **[CTRL][caractère]**.
\+ Le caractère qui suit est autorisé.
\- Le caractère qui suit est interdit.

Exemples de chaînes de commande

Commande$="+1+4+8%"

Les chiffres 1,4,8, sont autorisés ainsi que les signes spéciaux.

Commande$="0-9a+."

Les chiffres sont autorisés excepté le chiffre 9 ainsi que toutes les lettres et le point.

Codage des symboles de commande de transformation de caractères

C x1 x2 Conversion du caractère x1 en x2.
U Toutes les lettres passent en majuscules *(UPPER)*.
L Toutes les lettres passent en minuscules *(LOWER)*.

Les espaces et les majuscules ou minuscules ne sont pas pris en compte dans les chaînes de commande et facilitent leur lecture. Ne vous-en privez pas !

Commande$="0a%C?!CéeCèeCêe"

Les chiffres et les lettres sont acceptés. Le point d'interrogration est transformé en point d'exclamation et les e accentués sont désaccentués.

• *Validation*

Comme dans les **INPUT** classiques, on peut quitter la saisie par **ENTER** ou **RETURN**, mais on peut aussi paramétrer cette sortie. Les caractères de validation sont :

X Code ASCII

On spécifie le code ASCII de la touche qui permet la sortie de la saisie.



S Scancode

On spécifie le SCANCODE de la touche qui permet la sortie de la saisie. Cela rend équivalents les claviers AZERTY et QWERTY à ce niveau là puisque les codes de touches claviers sont les mêmes.

> Sortie à droite par la touche flèche droite.

< Sortie à gauche par la touche flèche gauche.

La variable apparaissant en **VALIDATION** contiendra quatre octets correspondant exactement à **INKEY$**, à l'exception des sorties faites par :

RETURN ou ENTER	**code 0**
Flèche droite	**code -1**
Flèche gauche	**code -2**

Les paramètres suivants sont facilement compréhensibles puisqu'ils codent la longueur maximale attendue, la définition d'un caractère de remplissage (par défaut le tiret souligné _) et la position de départ du curseur dans la chaîne à saisir.

Exemple :

```
Sortie$="X"+CHR$(32)+"X"+CHR$(48)
Masque$="a+-U"+Sortie$
INPUT @(4,4);Nom$ USING Masque$,Val,10
```

On attend dix caractères au maximum. Toutes les lettres sont autorisées (et transformées en majuscules) ainsi que le trait d'union. La validation se fait par la touche **RETURN** ou **ENTER** ou le zéro du clavier numérique.

Exemple :

```
Masque$="a0+/+-+_+.C./"
Masque$=Masque$+"S"+CHR$($48)+"S"+CHR$($50)
INPUT @(6,4);Nom$ USING Masque$,Val,20,"*",2
```

On attend vingt caractères au maximum à stocker dans la variable **Nom$**. Le caractère de remplissage est l'astérisque (*) et le curseur se positionne au départ sur le deuxième caractère de la chaîne à saisir. Les caractères permis sont les lettres et les chiffres ainsi que le **slash** (**/**) le tiret (-) et le point (.). Le point est transformé en slash. La sortie de la saisie se fait par **RETURN**, **ENTER** ou les flèches gauches ou droites. On retrouvera donc en sortie de saisie les valeurs 0 -1 ou -2 dans la variable **Val** suivant qu'on est sorti respectivement par les touches **RETURN** ou **ENTER**, flèche droite, flèche gauche.

Les sorties de données

PRINT

PRINT (Chaîne ou Expression numérique) (Caractère de contrôle)
PRINT @(Ligne,colonne);(Sortie)(cf PRINT AT du GFA).

Cette instruction est classique. Elle réalise l'affichage à l'écran de la chaîne ou expression numérique ou constante. La seule différence est la spécificité des caractères de contrôle.

Le point virgule (;) provoque un affichage sur la même ligne sans espace (excepté dans le cas de nombre ou un espace est généré). La virgule (,) provoque un saut de tabulation égal à huit caractères sur la ligne. Ces deux caractères de contrôle sont comparables à ceux du GFA, bien que ce dernier en dispose d'un plus grand nombre. Le point virgule est le séparateur des variables ou valeurs que vous voulez voir afficher. L'OMIKRON le positionne automatiquement. Le raccourci de PRINT est (?) comme dans tous les Basic.

PRINT TAB(X)

Ce PRINT positionne le curseur sur la colonne X avant de poursuivre l'affichage. Le maximum autorisé est la dernière colonne de l'écran, soit 79.

PRINT SPC(X)

Affichage de X expaces avant de continuer. SPC(X) peut aussi être considérée comme chaîne de caractères en OMIKRON (voir plus haut). La valeur maximale fait référence à la longueur maximale d'une chaîne de caractères à savoir 32765. Notez que cette valeur maximale est supérieure au nombre d'octets de la RAM écran (32000).

PRINT USING

PRINT USING <Masque$>,<Chaîne$> et USING <Masque$>

L'instruction USING permet de définir un masque de formattage des commandes à venir. La définition courante reste valide tant qu'un autre USING n'est pas exécuté. C'est le même type de masque qui est utilisé dans PRINT USING. Le masque d'édition des données numériques rappelle en quelques points celui du GFA, mais ce dernier langage dispose de plus de possibilités, surtout en ce qui concerne le formattage des chaînes de caractères :

Emplacement réservé pour un chiffre ou le signe du nombre ; si c'est le dernier chiffre qui est concerné, il sera arrondi.

. Sépare les chiffres décimaux parmi plusieurs caractères. Placé en début de chaîne le point sépare les milliers (groupes de trois chiffres).

, Son rôle est similaire à celui du point. Il ne faut pas toutefois que le point et la virgule puissent avoir la même fonction au même moment.

C'est celui des deux signes qui est placé en début de ligne qui sépare les milliers alors que l'autre servira de point décimal.



- Le signe moins ne sert qu'à préciser sa position d'affichage, car ce signe est affiché dès qu'un nombre est négatif.

+ Le signe plus n'est pas automatiquement affiché. L'affichage et l'emplacement du + est forcé, que le nombre soit positif ou négatif.

* Le caractère qui suit l'astérisque sert de caractère de remplissage (le tiret souligné _ n'est pas accepté).

- Le tiret souligné force l'affichage du caractère qui le suit sur sa position.

^ Chaque symbole d'exponentiation réserve un caractère pour l'affichage des nombres en notation scientifique. Les signes # déterminent la longueur de la mantisse (y compris les caractères E+ et E-). Pour les nombres négatifs, un signe doit absolument être prévu.

En OMIKRON, on peut aussi insérer dans le masque ce qu'on désire voir afficher à l'écran.

Exemple :

```
Masque1$="###.##^^^^"
Masque2$="#.####"
Masque3$="-###,#"
Masque4$="*$#####,##"
Masque5$="###. Kg ###"
FOR I= 1 TO 4
PRINT USING Masque1$,2^(2*I);
NEXT I
4,00E +0  1,60E +1  6,40E +1  2,56E +2
OK
PRINT USING Masque2$,PI
%.####
OK
REM ERREUR DE MASQUE
REM: CAR PAS D'EMPLACEMENT DE SIGNE
Masque2$="##.####"
PRINT USING Masque2$,PI
3.1415
OK
PRINT USING Masque3$,PI
3,1
OK
PRINT USING Masque4$,PI
$$$$$3,14
OK
PRINT USING Masque5$,PI
3, Kg 1415
OK
```

On ne dispose pas en OMIKRON d'instructions qui permettent de modifier facilement les masques d'affichage en cours de programme, comme **MODE** ou **DEFNUM** en GFA, ce qui oblige toujours à réécrire les nouveaux masques, même si l'on ne doit effectuer que des transformations minimes. Par contre, il est vraiment intéressant de pouvoir utiliser un masque de façon constante grâce à **USING** sans avoir à le repréciser de façon permanente à chaque affichage. Pour annuler tout formattage de chaîne il suffit d'ailleurs de taper **USING** sans argument.

LPRINT (Chaîne)

C'est l'équivalent du **PRINT** pour l'imprimante. On peut envoyer plusieurs arguments séparés par des virgules et abréger l'instruction par L?. On ne peut, par contre, spécifier le caractère de commande de mise en page @ du fait que toutes les imprimantes n'ont pas la même implémentation.

PRINT# Numcanal,Sortie

Cette instruction fonctionne en sortie comme **INPUT$** en entrée. Reportez-vous à l'exemple donné plus haut pour cette instruction. **PRINT#** écrit les données constantes, numériques ou alphanumériques sur le fichier correspondant au numéro de canal spécifié. Il faut bien sûr que ce fichier ait été ouvert précédemment en écriture. Il est possible d'abréger le code instruction par **?#**.

WRITE, WRITE# Numcanal,sortie

WRITE fonctionne syntaxiquement comme **PRINT** si ce n'est la reconnaissance des guillemets et de la virgule comme caractère et non comme séparateur. A l'affichage écran **(WRITE)**, on verra apparaître ces deux symboles et en écriture sur fichier **(WRITE#)**, on pourra écrire ces symboles et les relire grâce à l'instruction **INPUT#**. L'intérêt est de pouvoir lire plusieurs champs de données (équivalents aux sorties séparées par des virgules ou des guillemets) avec une seule instruction **INPUT#**.



Exemple :

```
OPEN "O",1,"VERSION.DAT"
REM: ouverture du fichier de nom VERSION.DAT
REM: en écriture sur le canal 1
WRITE #1,"OMIKRON","VERSION"," 3.01"
REM: écriture du texte sur ce fichier
CLOSE #1
REM: fermeture du canal 1 correspondant
REM: au fichier ouvert
OPEN "I",#1,"VERSION.DAT"
REM: ouverture du fichier de nom VERSION.DAT
REM: en lecture sur le canal 1
INPUT #1,Fich1$,FIch2$,Fich3$
REM Lecture de trois fichiers sur le canal 1
REM dans les trois variables réceptrices
CLOSE #1
WRITE "WEKA"
"WEKA"
OK
PRINT "WEKA"
WEKA
OK
```

CMD Numcanal

Cette commande n'existe pas en GFA ou dans les autres Basic. Elle permet de détourner le chemin des instructions de sortie.

Il suffit que le canal périphérique ait été ouvert précédemment pour que les sorties puissent être redirigées dans sa direction. Les périphériques se notent par des lettres en OMIKRON alors que des chiffres sont utilisés en Basic GFA, mais bien sûr il y en a toujours le même nombre !

OMIKRON PERIPHERIQUE GFA

"P"	PRINTER	0
"V"	RS232	1
"C"	CONSOLE	2
"M"	MIDI	3
"K"	KEYBOARD	4

Exemple :

```
OPEN "M",1
OPEN "V",2
CMD 1
REM ON PEUT VÉRIFIER SI LE PORT MIDI OUT
REM EST OPÉRATIONNEL
CMD 2
REM REDIRIGER PAR EXEMPLE LES INFORMATIONS
REM VERS LA RS232
CLOSE #1
CLOSE #2
OPEN "P",1
OPEN "I",2,"BUFFER"
OPEN "C",3
CMD 3
REM SI L'IMPRIMANTE N'EST PAS BRANCHÉE
REM ON PEUT ENVOYER LE FICHIER "BUFFER"
REM OUVERT EN LECTURE SUR L'ECRAN
CLOSE #1:CLOSE #2:CLOSE #3
END
```

4/3

L'assembleur

I - Introduction

Les ordinateurs Atari ST fonctionnent autour d'un microprocesseur très puissant (le MC 68000), et pourtant, très simple à utiliser.

Le MC 68000 est un microprocesseur 16/32 bits fabriqué par MOTOROLA en technologie VLSI (*Very Large Scale Integration,* ce qui signifie intégration (de composants) à très grande échelle). La dénomination 16/32 bits signifie que :

— les données qui circulent sur le bus des données ont une longueur de 16 bits (2 octets) ;

— les données manipulées par le microprocesseur ont une longueur maximale de 32 bits (4 octets).

La première version du MC 68000 est apparue en 1979. La conception de ce microprocesseur par Motorola correspondait à un besoin nouveau du marché : les microprocesseurs de l'époque étaient pour la plupart des 8 bits (6502, 6809, Z80, etc.). Les langages de programmation étaient souvent le Basic, et l'assembleur, quand le besoin s'en faisait sentir. Les langages tels que le Pascal ou le C n'existaient que très rarement. Lorsqu'ils existaient, ils demandaient de nombreuses manipulations de disquettes, et ne laissaient que très peu de place à l'utilisateur pour écrire ses programmes. En effet, le compilateur, le linker, le code source et le code objet devaient cohabiter dans l'espace adressable du microprocesseur 8 bits (64 Koctets).

Le 68000 a bouleversé la technique des microprocesseurs en introduisant :

— une capacité d'adressage linéaire de 16 MOctets (utilisation de 24 bits d'adresse) ;

— un bus de données sur 16 bits ;

— des registres internes sur 32 bits ;

— une vitesse d'horloge supérieure à celle de ses concurrents : 8 MHz (Méga-Hertz).

II - La structure d'un ordinateur

Avant de nous spécialiser dans le 68000, nous allons parler de la structure générale de la (ou des) carte(s) mère(s) d'un micro-ordinateur. Quel que soit son type, elle est constituée :

— de circuits mémoire ;

— d'une ou de plusieurs unité(s) de traitement ;

— d'une ou de plusieurs unité(s) de communication avec les organes périphériques.

Examinons en détail chacun de ces éléments.

LA MÉMOIRE

La mémoire des ordinateurs est généralement de deux types : la mémoire vive (ou RAM) et la mémoire morte (ou ROM).

La mémoire vive peut être accédée en lecture et en écriture (il est possible d'y lire et d'y stocker des données). Ce type de mémoire fonctionne jusqu'à ce que l'alimentation soit coupée. Si une panne de courant survient, ou si vous mettez hors tension l'ordinateur, les données qui se trouvaient en RAM sont effacées.

La mémoire morte ne peut être accédée qu'en lecture. Elle contient des données qui ont été écrites de manière permanente (même après l'arrêt de la tension d'alimentation) par son constructeur ou par une autre personne.

Les mémoires mortes peuvent être de l'un des types suivants : EPROM, EEPROM, EAROM, ou PROM. La différence entre ces quatre types de mémoires réside dans le moyen utilisé pour les effacer : les premières sont effaçables grâce à un bombardement photonique de lumière ultraviolette. Les secondes et troisièmes, grâce à une tension électrique. Les dernières ne sont pas effaçables.

Les mémoires mortes contiennent généralement les routines du système d'exploitation, et parfois un ou plusieurs langages, utilitaires ou jeux qui sont disponibles dès que la machine est sous tension.

LA OU LES UNITÉ(S) DE TRAITEMENT

L'unité de traitement est le cœur de l'ordinateur. Elle est souvent appelée unité centrale, microprocesseur ou CPU (*Central Processing Unit*).

Dans les Atari ST, l'unité de traitement est un microprocesseur 68000.

Certains ordinateurs possèdent plusieurs unités de traitement à but spécialisé ou non.

Les unités de traitement spécialisées déchargent le microprocesseur de certaines tâches (par exemple, la gestion du clavier, de l'écran ou du lecteur de disquettes).



Les unités de traitement non spécialisées sont des unités centrales qui accèdent aux mêmes ressources dans le but d'augmenter la vitesse de traitement de l'ordinateur. Ce type de structure est généralement implanté sur de gros ordinateurs.

LES UNITÉS SPÉCIALISÉES DANS LA COMMUNICATION AVES LES PÉRIPHÉRIQUES

Les unités de communication font partie des unités de traitement spécialisées dont nous venons de parler.

Les organes d'entrée de l'ordinateur sont (par exemple) le clavier, la souris, le joystick (en manette), le lecteur de codes barres, etc.

Les organes de sortie sont (par exemple) l'écran, l'imprimante, la table traçante, le modem, etc.

Certains périphériques fonctionnent en entrée et en sortie. Il s'agit par exemple des lecteurs de disquettes ou de disques durs. Ces unités de traitement des entrées/sorties sont la plupart du temps constituées d'un ou de plusieurs circuit(s) intégré(s) qui s'occupe(nt) des fonctions primaires de communication, et qui communique(nt) le résultat au microprocesseur.

III - La représentation des données en mémoire

Un ordinateur est avant tout une machine destinée à traiter des nombres (avant l'apparition de la micro-informatique, les ordinateurs portaient le nom de calculateurs). Toutes les données manipulées par un ordinateur (graphiques, sonores, textes, etc.) sont converties en nombres pour pouvoir être mémorisées.

Les mémoires d'un ordinateur sont composées d'un ensemble d'éléments binaires, appelés bits, regroupés par huit pour former des octets. Un bit a toujours l'une des deux valeurs suivantes : 0 ou 1.

Chacun des huit bits d'un octet a un *poids* (c'est-à-dire une valeur) différent.

Le premier bit est dit *bit de poids faible*. Il a un poids de 2^0 = 1 (^ signifie mise à la puissance).

Le second bit a un poids de 2^1 = 2, etc.

Le dernier bit dit *bit de poids fort* a un poids de 2^7 = 128.

Cette affectation des poids à chaque bit est représentée sur le schéma ci-dessous :

Poids	128	64	32	16	8	4	2	1
Position du bit dans l'octet	7	6	5	4	3	2	1	0

Pour stocker un nombre entier en mémoire, il faut au préalable le décomposer en une suite d'éléments binaires, c'est-à-dire le convertir en base deux. Par exemple, le nombre 170 sera décomposé comme suit : 170 = 128+32+8+2. Sa représentation binaire sera donc la suivante :

128	64	32	16	8	4	2	1
1	0	1	0	1	0	1	0

Il est facile d'en déduire que le plus grand nombre qui peut être stocké sur 8 bits est égal à 128+64+32+16+8+4+2+1, soit 255. Signalons au passage que ce nombre est aussi égal à 2^8−1, c'est-à-dire la puissance de deux supérieure à celle du bit de poids le plus fort (à un près).

Ce concept peut être étendu sur 16 bits. Le bit de poids le plus faible vaut 2^0=1. Le bit de poids le plus fort vaut 2^15=32768. De même que précédemment, le plus grand nombre qui peut être codé sur 16 bits est égal à la puissance de deux supérieure au poids du bit de poids fort (à un près), soit 2^17−1=65535.

Pour pouvoir manipuler plus simplement les nombres binaires, on utilise souvent le codage hexadécimal (base 16). Chaque chiffre de cette base est compris entre 0 et 15. Les quinze chiffres successifs sont : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E et F.

Voyons comment utiliser cette base de numération à partir d'un exemple. Le nombre 142 (inférieur à 255, donc codé sur 8 bits) se décompose de la manière suivante :

128	64	32	16	8	4	2	1
1	0	0	0	1	1	1	0

Le nombre 142 est donc codé 10001110 en binaire. Pour représenter ce nombre en base 16, séparons chaque quartet (un quartet est par définition égal à quatre bits successifs).

10001110 devient donc la succession des deux quartets 1000 et 1110. Affectons un poids compris entre 1 et 8 (2^0 et 2^3) à ces quatre bits. Les deux quartets sont donc codés comme suit :

8	4	2	1
1	0	0	0

8	4	2	1
1	1	1	0

Le premier quartet est donc égal à 8, et le second à 8+4+2=14, soit E en hexadécimal.

Finalement, le nombre 142 (10001110) est donc codé 8E en hexadécimal.



IV - Le microprocesseur MC 68000

Lors de la conception du 68000, Motorola s'est inspiré des techniques utilisées sur les calculateurs multi-utilisateurs de l'époque (mini-ordinateurs). Sur ce type de machine, tous les utilisateurs n'ont pas accès à la totalité de la mémoire.

Le 68000 possède deux modes de fonctionnement :

— le mode superviseur :

- toute la mémoire adressable (16 MOctets) peut être accédée ;
- toutes les instructions du microprocesseur sont utilisables.

Le mode superviseur est accessible au système d'exploitation, et au logiciel de base, mais non aux programmes écrits par les utilisateurs qui fonctionnent toujours dans le mode utilisateur.

— le mode utilisateur :

- seul un jeu d'instructions réduit est utilisable ;
- seule une partie de la mémoire adressable est accessible.

Si l'utilisateur tente d'accéder à une ressource (logicielle ou matérielle) qui lui est interdite, le 68000 se déroute automatiquement vers une procédure d'erreur.

A. L'architecture interne du 68000

Comme tous les microprocesseurs, le 68000 dispose de mémoires internes (qui se trouvent dans le circuit lui-même). Ces mémoires sont appelées *registres*. Le 68000 possède des registres de trois types : données, adresses et spéciaux.

Comme leur nom l'indique, les registres d'adresses permettent de manipuler des adresses mémoire, sur 16 ou 32 bits.

De même, les registres de données permettent de manipuler des données sur 8, 16 ou 32 bits.

Un registre spécial appelé *compteur de programme* contient l'adresse de la prochaine instruction à exécuter.

Un autre registre spécial appelé *registre d'état* contient divers indicateurs relatifs à la dernière opération effectuée. Ces informations sont du type résultat nul, dépassement de capacité, etc.

Le type des registres accessibles diffère en fonction du mode d'utilisation du 68000.

En mode utilisateur

Les registres internes disponibles sont les suivants :

— 8 registres de données de 32 bits à usage général (D0 à D7) ;

— 8 registres d'adresses de 32 bits à usage général (A0 à A7) ;

— 1 compteur de programme de 32 bits (PC) ;

— 1 pointeur de pile de 32 bits (registre d'adresse A7) ;

— 1 registre d'état de 8 bits (CCR).

En mode superviseur

Les registres internes disponibles sont les suivants :

— 8 registres de données de 32 bits à usage général (D0 à D7) ;

— 8 registres d'adresses de 32 bits à usage général (A0 à A6 et A7') ;

— 1 compteur de programme de 32 bits (PC) ;

— 1 pointeur de pile de 32 bits (registre d'adresse A7') ;

— 1 registre d'état de 16 bits (SR).

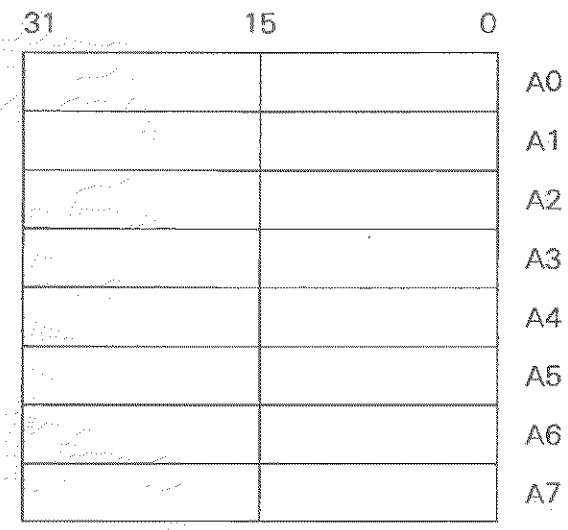
Remarque :

Le pointeur de pile étant différent en mode utilisateur et en mode superviseur, il est facile de gérer deux piles selon le mode d'utilisation du 68000.

Nous allons maintenant parler en détails de chaque type de registres du 68000.

LES REGISTRES D'ADRESSES

Les registres d'adresses sont au nombre de 8. Ils ont pour nom A0 à A7. Chacun d'entre eux peut être subdivisé en deux sous-registres de 16 bits comme le montre le schéma ci-dessous :



Ces registres de 32 bits sont destinés à gérer les 16 Méga Octets adressables du 68000 (seuls les 24 bits de poids faible des registres d'adresses sont utilisés).

Les registres d'adresses A0 à A6 peuvent également être utilisés comme pointeurs de pile. Dans ce cas, l'adressage sera du type post-décrémentation ou post-incrémentation.



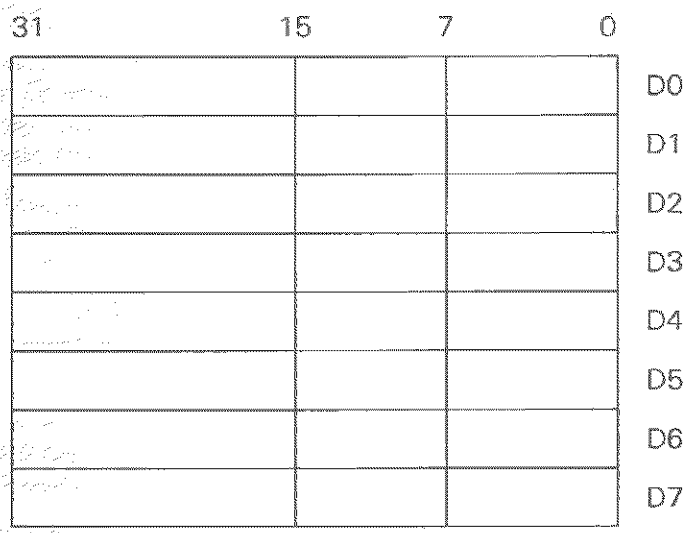
Les registres A7 (en mode utilisateur) et A7' (en mode superviseur) ne sont pas des registres d'adresses classiques : le registre A7 est le pointeur de la pile utilisateur USP (*User Stack Pointer*) et le registre A7' est le pointeur de la pile superviseur. Les piles utilisateur et superviseur sont des espaces mémoire destinés à sauvegarder le compteur de programme et le registre d'état lors de la gestion d'interruptions et lors de l'appel de sous-programmes.

Remarque :

Lorsqu'un registre d'adresse est utilisé comme pointeur de pile pour déplacer une donnée de 16 ou 32 bits, l'adresse pointée doit obligatoirement être paire.

LES REGISTRES DE DONNÉES

Les registres de données sont au nombre de 8. Ils ont pour nom D0 à D7. Chacun d'entre eux peut être subdivisé en deux sous-registres de 16 bits. Le sous-registre de poids faible peut lui-même être subdivisé en deux sous-registres de 8 bits, comme le montre le schéma ci-dessous :



Les registres de données permettent donc de manipuler des données de 8, 16 ou 32 bits. Le bit de poids le plus fort, souvent appelé msb (31 pour une donnée sur 32 bits, 15 pour une donnée sur 16 bits, et 7 pour une donnée sur 8 bits) contient le signe de la donnée.

Remarque :

Les registres de données peuvent également être utilisés comme index dans les modes d'adressage indexé.

LE COMPTEUR DE PROGRAMME PC

Ce registre pointe sur l'adresse mémoire de la prochaine instruction à exécuter. Sur les 32 bits dont est constitué le PC, seuls les 24 bits de poids faible sont utilisés pour accéder à une adresse en mémoire. C'est pour cette raison que la plage de mémoire adressable a pour étendue 2^24 octets, soit 16 Méga-Octets.

LE REGISTRE D'ÉTAT

Sa taille est différente en fonction du mode de fonctionnement du microprocesseur.

Mode utilisateur

En mode utilisateur, le registre d'état occupe un octet et est appelé CCR (*Condition Code Register*, soit en français : registre des codes condition). Les bits du registre d'état ont la dénomination suivante :

7	6	5	4	3	2	1	0
			X	N	Z	V	C

X eXtend : Indicateur d'extension
X se comporte comme C pour les instructions sur les opérandes étendus. Reportez-vous sur l'instruction qui vous intéresse pour avoir des détails à son sujet.

N Negate : Indicateur négatif
Ce bit est mis à un lorsque le résultat de l'opération courante est négatif.

Z Zéro : Indicateur Zéro
Ce bit est positionné lorsque le résultat de l'opération courante est nul.

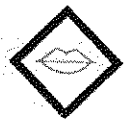
V oVerflow : Indicateur de débordement
Ce bit est utilisé sur les opérations arithmétiques. Lorsqu'il vaut un, il indique que le résultat dépasse les limites permises.

C Carry : Indicateur de retenue
Ce bit est utilisé sur les opérations arithméthiques. Lorsqu'il vaut un, il indique qu'une retenue s'est produite sur le bit de poids fort pendant l'opération courante.

Mode superviseur

En mode superviseur, le registre d'état occupe 16 bits. Il est divisé en deux : l'octet de poids fort est utilisé exclusivement en mode superviseur, l'octet de poids faible est utilisé dans les deux modes de fonctionnement. Les bits du registre d'état ont la dénomination suivante :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
T		S			I 2	I 1	I 0				X	N	Z	V	C



Les bits de l'octet de poids faible ont le même nom et sont utilisés de la même manière qu'en mode utilisateur (voir page précédente). Les bits de l'octet superviseur ont la signification suivante :

T — Trace : mode trace
Lorsqu'il vaut 1, le 68000 exécute le programme en mémoire pas à pas. Cela est souvent utile pour mettre au point des programmes complexes.

S — Supervisor : mode de travail
Ce bit permet de choisir le mode de travail : 0 pour le mode superviseur, 1 pour le mode utilisateur.

I0 à I2 — Interrupt : masque d'interruption
Lorsqu'une interruption se produit, elle n'est prise en compte que dans le cas où son niveau de priorité est supérieur au masque d'interruption contenu dans ces trois bits.

B. Les codes opératoires du 68000

Dans ce paragraphe, nous analysons en détails les 56 codes opératoires du 68000, par ordre alphabétique.

Les codes opératoires sont systématiquement représentés selon le modèle suivant :

Code Op — Résumé
Format
Fonction
Indicateurs modifiés

Le format de chaque code opératoire suit les conventions syntaxiques suivantes :

Dx = Registre de données source,
Dy = Registre de données destination,
Dn = Un des registres de données,
Ax = Registre d'adresses source,
Ay = Registre d'adresses destination,
An = Un des registres d'adresses,
<AE> = Adresse effective,
#<Donnée> = Donnée immédiate,
<Etiquette> = Adresse mémoire,
CCR = Condition Code Register,
SR = Registre d'état

Les indicateurs sont représentés selon les conventions de codage suivantes :

* = Positionné,
I = Indéfini,
– = Non affecté,
0 = Mis à zéro,
1 = Mis à un.

ABCD Addition décimale avec retenue

Format

ABCD Dy, Dx
ou ABCD −(Ay), −(Ax)

Fonction

L'instruction ABCD effectue une addition décimale avec le bit d'extension (bit X du CCR) entre :
— deux registres de données,
ou — deux registres d'adresses adressés par prédécrémentation.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			*	I	*	I	*

Exemple de codage :

```
C300    ABCD D0,D1
CB08    ABCD -(A0),-(A5)
```

ADD Addition binaire

Format

ADD <AE>, Dn
ou ADD Dn, <AE>

Fonction

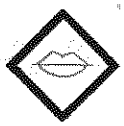
L'instruction ADD effectue une addition sans extension entre ses deux opérandes. Le résultat est rangé dans l'opérande destination. L'opérande source est adressé par un mode d'adressage quelconque. Par contre, l'opérande destination doit être adressé par un mode mémoire modifiable.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			*	*	*	*	*

Exemple de codage :

```
D51C    ADD.W D2, (A4)+
D424    ADD.B -(A4), D2
```



ADDA Addition binaire avec une adresse

Format ADDA <AE>, An

Fonction L'instruction ADDA effectue une addition sans extension entre le premier opérande et un registre d'adresses. Le résultat est rangé dans le registre d'adresses. L'opérande source est adressé par un mode d'adressage quelconque.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	—	—	—	—

Exemple de codage :

```
D7E4        ADDA.L  -(A4),A3
```

ADDI Addition immédiate

Format ADDI #<Donnée>, <AE>

Fonction L'instruction ADDI effectue une addition sans extension entre une donnée immédiate et l'opérande destination. Le résultat est rangé dans l'opérande destination. Ce dernier doit être adressé par un mode de type données modifiables.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			*	*	*	*	*

Exemple de codage :

```
065C 1234   ADDI.W  #$1234,(A4)+
```

ADDQ Addition rapide

Format ADDQ #<Donnée>, <AE>

Fonction L'instruction ADDQ effectue une addition rapide sans extension entre une donnée immédiate (comprise entre 1 et 8) et l'opérande destination. Le résultat est rangé dans l'opérande destination. Son mode d'adressage doit être du type modifiable.

Indicateurs modifiés 7 6 5 X N Z V C
* * * * *

Exemple de codage :

```
5210    ADDQ.B #1, (A0)
```

ADDX Addition binaire étendue

Format
ADDX Dy, Dx
ou ADDX −(Ay), −(Ax)

Fonction
L'instruction ADDX effectue une addition signée (bit X du CCR) entre ses deux opérandes. Les deux opérandes doivent être :
— tous deux des registres de données ;
— tous deux des registres d'adresses en mode prédécrémentation.

Indicateurs modifiés 7 6 5 X N Z V C
* * * * *

Exemple de codage :

```
D902    ADDX.B D2,D4
```

AND ET logique

Format
AND <AE>,Dn
ou AND Dn,<AE>

Fonction
L'instruction AND effectue un ET logique entre ses deux opérandes. Le résultat est rangé dans le second opérande. L'opérande source doit être de type donnée, et l'opérande destination de type mémoire modifiable.

Indicateurs modifiés 7 6 5 X N Z V C
— * * 0 0

Exemple de codage :

```
C39B    AND.L D1,(A3)+
C810    AND.B (A0),D4
```



ANDI ET logique immédiat

Format ANDI #<Donnée>, <AE>

Fonction L'instruction ANDI effectue un ET logique entre une donnée immédiate et le second opérande qui doit être de type mémoire modifiable. Le résultat est rangé dans le second opérande.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	*	*	0	0

Exemple de codage :

```
0222 FF      ANDI.B #$FF,-(A2)
023C 00FE    ANDI #$FE,CCR
027C 1234    ANDI #$1234,SR
```

ASL Glissement arithmétique à gauche

Format

ASL Dx,Dy
ou ASL #<Donnée>,Dy
ou ASL <AE>

Fonction L'instruction ASL effectue un glissement arithmétique vers la gauche. Les opérandes peuvent être d'un des types suivantes :

- deux registres de données : le premier contient le nombre de glissements (entre 0 et 63) à effectuer sur le second ;
- une adresse modifiable : le glissement fait toujours un bit.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			*	*	*	*	*

Exemple de codage :

```
E564         ASL.W D2,D4
EB04         ASL.B #3,D6
```

ASR Glissement arithmétique à droite

Format

ASR Dx, Dy
ou ASR #<Donnée>, Dy
ou ASR <AE>

Fonction

L'instruction ASR effectue un glissement arithmétique vers la droite. Les opérandes peuvent être d'un des types suivants :

— deux registres de données : le premier contient le nombre de glissements (entre 0 et 63) à effectuer sur le second.

— une adresse modifiable : le glissement fait toujours un bit.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			*	*	*	*	*

Exemple de codage :

```
E424    ASR.B   D2,D4
E644    ASR.W   #3, D4
```

Bcc Branchement conditionnel

Format

Bcc <Etiquette>
où cc= CC, CS, EQ, NE, GE, GT, HI, LE, LS, LT, MI, PL, VC, VS, T ou F

Fonction

L'instruction Bcc effectue un branchement conditionnel à l'adresse spécifiée. La condition exprimée dans l'instruction peut être l'une des suivantes :

Suffixe	Condition
CC	C = 0
CS	C = 1
EQ	Z = 1
GE	N OU exclusif V = 0
GT	Z ET (N OU exclusif V) = 0
HI	C OU Z = 1
LE	Z OU (N OU exclusif V) = 1
LS	C OU Z = 1
LT	N OU exclusif V = 1
MI	N = 1
NE	Z = 0
PL	N = 1
VC	V = 0
VS	V = 1



Indicateurs modifiés 7 6 5 X N Z V C
— — — — — —

Exemple de codage :

```
6715          BEQ $15
6700 1234     BEQ $1234
```

BCHG Test d'un bit et complémentation

Format

BCHG Dn, <AE>
ou BCHG #<Donnée><AE>

Fonction

L'instruction BCHG teste le bit du deuxième argument dont la position est spécifiée dans le premier argument. Le bit testé est recopié dans l'indicateur Z, puis inversé. Le premier argument peut être une donnée immédiate ou un registre de données. Le second argument peut être un registre de données ou d'adresses.

Indicateurs modifiés 7 6 5 X N Z V C
— — * — —

Exemple de codage :

```
075C          BCHG D3,(A4)+
0852 0001     BCHG #1,(A2)
```

BCLR Test d'un bit et remise à zéro

Format

BCLR Dn,<AE>
ou BCLR #<Donnée>, <AE>

Fonction

L'instruction BCLR teste le bit du deuxième argument dont la position est spécifiée dans le premier argument. Le bit testé est recopié dans l'indicateur Z et forcé à zéro. Le premier argument peut être une donnée immédiate ou un registre de données. Le second argument peut être un registre de données ou d'adresses.

Indicateurs modifiés 7 6 5 X N Z V C
— — * — —

Exemple de codage :

```
0982          BCLR D4,D2
0892 0001     BCLR #1, (A2)
```

BRA Branchement inconditionnel

Format BRA <Etiquette>

Fonction L'instruction BRA effectue un débranchement inconditionnel à l'adresse spécifiée dans l'instruction.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	—	—	—	—

Exemple de codage :

```
5000 1234     BRA $1234
```

BSET Test d'un bit et remise à un

Format

BSET Dn,<AE>
ou BSET #<Donnée>, <AE>

Fonction L'instruction BSET teste le bit du deuxième argument dont la position est spécifiée dans le premier argument. Le bit testé est recopié dans l'indicateur Z, puis forcé à un. Le premier argument peut être une donnée immédiate ou un registre de données. Le second argument peut être un registre de données ou d'adresses.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	—	*	—	—

Exemple de codage :

```
05C4          BSET D2,D4
```



BSR Branchement à un sous-programme

Format BSR <Etiquette>

Fonction L'instruction BSR sauvegarde l'adresse qui suit l'adresse courante dans la pile, puis effectue un débranchement inconditionnel vers un sous-programme dont l'adresse est spécifiée dans l'instruction.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	—	—	—	—

Exemple de codage :

```
612C          BSR $12
6100 1234     BSR $1234
```

BTST Test d'un bit

Format BTST Dn,<AE>
ou BTST #<Donnée>, <AE>

Fonction L'instruction BTST teste le bit du deuxième argument dont la position est spécifiée dans le premier argument. Le bit testé est recopié dans l'indicateur Z. Le premier argument peut être une donnée immédiate ou un registre de données. Le second argument peut être un registre de données ou d'adresses.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	—	*	—	—

Exemple de codage :

```
0802 0003     BTST #3,D2
```

CHK Test du contenu d'un registre

Format CHK <AE>,Dn

Fonction L'instruction CHK teste si la valeur qui se trouve dans le premier argument est comprise entre 0 et le mot de poids faible du second argu-

ment (registre de données). Une exception 6 est générée dans le cas contraire.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	*	I	I	I

Exemple de codage :

```
41B8 1234      CHK $1234, D0
```

CLR Mise à zéro de l'opérande

Format CLR <AE>

Fonction L'instruction CLR initialise à zéro le contenu de l'opérande qui lui est passé.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	0	1	0	0

Exemple de codage :

```
4238 1234      CLR.B $1234
```

CMP Comparaison

Format CMP <AE>, Dn

Fonction L'instruction CMP soustrait l'opérande source de l'opérande destination. L'opérande source peut avoir un mode d'adressage quelconque. Par contre, l'opérande destination est forcément un registre de données.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	*	*	*	*

Exemple de codage :

```
B052           CMP (A2),D0
```



CMPA Comparaison avec un registre d'adresses

Format CMPA <AE>,An

Fonction L'instruction CMPA soustrait l'opérande source de l'opérande destination qui est toujours un registre d'adresses. Les deux opérandes sont inchangés. Le résultat de la comparaison modifie les indicateurs en conséquence.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	*	*	*	*

Exemple de codage :

```
B5F8  1234      CMPA.L $1234,A2
```

CMPI Comparaison immédiate

Format CMPI #<Donnée>,<AE>

Fonction L'instruction CMPI soustrait une donnée immédiate de l'opérande spécifié. L'opérande est inchangé. Le résultat de la comparaison affecte les indicateurs.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	*	*	*	*

Exemple de codage :

```
0C03  0012      CMPI.B #$12,D3
0C43  1234      CMPI.W #$1234,D3
```

CMPM Comparaison de deux mémoires

Format CMPM (Ay)+,(Ax)+

Fonction L'instruction CMPM compare les deux opérandes accédés par post-incrémentation à l'aide de deux registres d'adresses. Les deux opérandes sont inchangés. Le résultat de la comparaison modifie les indicateurs en conséquence.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	*	*	*	*

Exemple de codage :

```
B90B    CMPM.B (A3)+,(A4)+
B94B    CMPM.W (A3)+,(A4)+
```

DBcc Test d'une condition, décrémentation et branchement en fonction du résultat du test

Format

DBcc Dn,<Etiquette>

Fonction

L'instruction DBcc teste la condition cc. Si cette condition est vraie, le programme continue en séquence. Si cette condition est fausse, le registre de données spécifié est décrémenté de un. Si sa valeur vaut −1, le programme continue en séquence. Dans le cas contraire, la prochaine instruction exécutée est celle qui est spécifiée dans l'instruction.

La condition exprimée dans l'instruction peut être l'une des suivantes :

Suffixe	Condition
CC	C = 0
CS	C = 1
EQ	Z = 1
GE	N OU exclusif V = 0
GT	Z ET (N OU exclusif V) = 0
HI	C OU Z = 1
LE	Z OU (N OU exclusif V) = 1
LS	C OU Z = 1
LT	N OU exclusif V = 1
MI	N = 1
NE	Z = 0
PL	N = 1
VC	V = 0
VS	V = 1

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	—	—	—	—

Exemple de codage :

```
56CB 1234    DBNE D3,$1234
```



DIVS Division signée

Format DIVS <AE>,Dn

Fonction L'instruction DIVS effectue une division arithmétique signée de l'opérande destination (32 bits) par l'opérande source (16 bits). Le quotient de la division est rangé dans les 16 bits de poids faible du registre destination. Le reste de la division est rangé dans les 16 bits de poids fort du registre destination.

Si l'opérande source vaut zéro, une exception 5 est générée.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	*	*	*	0

Exemple de codage :

```
82D2          DIVS (A2),D1
82E9  1234    DIVS $1234(A1),D1
```

DIVU Division non signée

Format DIVU <AE>,Dn

Fonction L'instruction DIVU effectue une division arithmétique non signée de l'opérande destination (32 bits) par l'opérande source (16 bits). Le quotient de la division est rangé dans les 16 bits de poids faible du registre destination. Le reste de la division est rangé dans les 16 bits de poids fort du registre destination.

Si l'opérande source vaut zéro, une exception 5 est générée.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	*	*	*	0

Exemple de codage :

```
8AD9      DIVU (A1)+,D5
```

EOR OU exclusif

Format EOR Dn,<AE>

Fonction L'instruction EOR effectue un OU exclusif entre les opérandes source et destination. Le résultat est placé dans l'opérande destination.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	*	*	0	0

Exemple de codage :

```
B339  0000  1234      EOR.B D1,$1234
B342                  EOR.W D1,D2
B3A9  0120            EOR.L D1,$120(AL)
```

EORI OU exclusif immédiat

Format EORI #<Donnée>,<AE>

Fonction L'instruction EOR effectue un OU exclusif entre la donnée immédiate spécifiée et l'opérande destination. Le résultat est placé dans l'opérande destination.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	*	*	0	0

Exemple de codage :

```
0A03  0012      EORI.B #$12,D3
0A44  1234      EORI.W #$1234,D4
```

EORI to CCR OU exclusif immédiat avec CCR

Format EORI #<Donnée>,CCR

Fonction L'instruction EORI to CCR effectue un OU exclusif entre la donnée immédiate 8 bits spécifiée et le poids faible du registre des indicateurs. Le résultat est placé dans les 8 bits de poids faibles du registre des indicateurs.



Indicateurs modifiés 7 6 5 X N Z V C
* * * * *

Exemple de codage :

```
0A3C  0012      EORI #$12,CCR
```

EORI to SR OU exclusif immédiat avec SR

Format EORI #<Donnée>,SR

Fonction L'instruction EORI to SR effectue un OU exclusif immédiat entre la donnée 16 bits spécifiée et le registre d'état. Le résultat est placé dans le registre d'état.

Indicateurs modifiés 7 6 5 X N Z V C
* * * * *

Exemple de codage :

```
0A7C  1234      EORI  #$1234,CR
```

EXG Echange de deux registres

Format EXG Rx,Ry

Fonction L'instruction EXG échange le contenu des deux registres 32 bits spécifiés.

Indicateurs modifiés 7 6 5 X N Z V C
_ _ _ _ _

Exemple de codage :

```
C549      EXG A1,A2
C78C      EXG D3,A4
```

EXT Extension signée de l'opérande

Format EXT Dn

Fonction L'instruction EXT effectue une extension signée de l'opérande 8 ou 16 bits sur 16 ou 32 bits.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	*	*	0	0

Exemple de codage :

```
4885    EXT.W D5
```

JMP Branchement inconditionnel

Format JMP <AE>

Fonction L'instruction JMP effectue un branchement inconditionnel à l'adresse spécifiée.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	—	—	—	—

Exemple de codage :

```
4EF9  0000  1234    JMP $1234
```

JSR Appel d'un sous-programme

Format JSR <AE>

Fonction L'instruction JSR effectue un branchement inconditionnel à l'adresse spécifiée. L'adresse de retour est placée dans la pile. Le retour sera effectué par l'instruction RTS.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	—	—	—	—



Exemple de codage :

```
4EB9  0000  1234      JSR $1234
4EB0  2802            JSR $2(A0,D2)
```

LEA Chargement d'une adresse effective

Format LEA <AE>,An

Fonction L'instruction LEA charge une adresse effective dans le registre d'adresses spécifié.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	—	—	—	—

Exemple de codage :

```
45F9  0000  1234      LEA $1234,A2
```

LINK Registre d'adresse dans la pile

Format LINK An,#<Offset>

Fonction L'instruction LINK est utilisée pour allouer une zone de mémoire dans la pile afin de ranger des paramètres lors de l'appel de programmes récursifs.

Elle effectue les opérations suivantes :

— registre d'adresses spécifié mis en pile,

— chargement du registre d'adresses avec SP,

— ajout du déplacement spécifié à SP.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	—	—	—	—

Exemple de codage :

```
4E52  1000      LINK A2,#$1000
```

LSL Décalage logique vers la gauche

Format LSL Dx,Dy ou LSL #<Donnée>,Dy ou LSL <AE>

Fonction L'instruction LSL comporte deux opérandes. Elle effectue un décalage logique vers la gauche de l'opérande destination. Les opérandes peuvent être d'un des types suivants :

— deux registres de données : le premier contient le nombre de décalages (entre 0 et 63) à effectuer sur le second,

— une adresse modifiable : le décalage fait toujours un bit.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			*	*	*	0	*

Exemple de codage :

```
E52C                LSL.B D2,D4
ED89                LSL.L #6,D1
E3F9  0000  1234    LSL $1234
```

LSR Décalage logique vers la droite

Format LSR Dx,Dy ou LSR #<Donnée>,Dy ou LSR <AE>

Fonction L'instruction LSR comporte deux opérandes. Elle effectue un décalage logique vers la droite de l'opérande destination. Les opérandes peuvent être d'un des types suivants :

— deux registres de données : le premier contient le nombre de décalages (entre 0 et 63) à effectuer sur le second,

— une adresse modifiable : le décalage fait toujours un bit.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	*	*	0	0

Exemple de codage :

```
E80B    LSR.B #4,D3
E26A    LSR.W D1,D2
```



MOVE Opérande source dans l'opérande destination

Format MOVE <AE>,<AE>

Fonction L'instruction MOVE transfère l'opérande source dans l'opérande destination. Lorsqu'elle concerne des données, elle permet de travailler sur des octets, des mots ou des mots longs. Lorsqu'elle concerne des adresses, elle permet de travailler sur des mots ou des mots longs.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			*	*	*	*	*

Exemple de codage :

```
1411    MOVE.B (A1),D2
36DC    MOVE.W (A4)+,(A3)+
```

MOVE to CCR Opérande source dans le registre des codes condition

Format MOVE <AE>,CCR

Fonction L'instruction MOVE to CCR transfère l'opérande source dans les 8 bits de poids faibles du registre d'état (CCR).

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	—	—	—	—

Exemple de codage :

```
44C3    MOVE D3,CCR
```

MOVE to SR Opérande source dans le registre d'état

Format MOVE <AE>,SR

Fonction L'instruction MOVE to SR transfère l'opérande source dans le registre d'état.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			*	*	*	*	*

Exemple de codage :

```
40F8  1234      MOVE SR,$1234
```

MOVE from SR Registre d'état dans le registre destination

Format MOVE SR,<AE>

Fonction L'instruction MOVE from SR transfère le registre d'état dans l'opérande source.

Indicateurs modifiés 7 6 5 X N Z V C
— — — — — — — —

Exemple de codage :

```
46C3      MOVE D3,SR
```

MOVE USP Pointeur de pile dans un registre d'adresses et inversement

Format MOVE USP,An ou MOVE An,USP

Fonction L'instruction MOVE USP copie le pointeur de pile dans le registre d'adresses spécifié (état superviseur) ou génère une exception (état utilisateur).

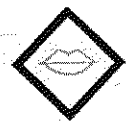
Indicateurs modifiés 7 6 5 X N Z V C
— — — — — — — —

Exemple de codage :

```
4E69      MOVE USP,A1
```

MOVEA Opérande source dans un registre d'adresses

Format MOVEA <AE>,An



Fonction L'instruction MOVEA transfère le registre source spécifié vers le registre d'adresses spécifié.

Indicateurs modifiés 7 6 5 X N Z V C
— — — — —

Exemple de codage :

```
3868  00A4      MOVEA.W $12(A0),A4
```

MOVEM Registres sources transférés à partir de l'adresse effective ou inversement.

Format MOVEM <registres>,<AE> ou MOVEM <AE>,<registres>

Fonction L'instruction MOVEM transfère le ou les registre(s) spécifié(s) à partir de l'adresse mémoire spécifiée.

Indicateurs modifiés 7 6 5 X N Z V C
— — — — —

Exemple de codage :

```
48A0  A0E0      MOVEM D0/D2/A0-A2,-(A0)
```

MOVEP Transfert d'un registre de données dans la mémoire indiquée et inversement.

Format MOVEP Dx,d(Ay) ou MOVEP d(Ay),Dx

Fonction L'instruction MOVEP transfère deux données 8 bits de ou vers un organe périphérique. Elle est particulièrement utile pour communiquer avec les périphériques de la famille 6800. Le premier octet émis ou reçu est lu ou stocké dans le poids fort du registre de données spécifié. Le second octet est donc lu ou stocké dans le poids faible du registre de données spécifié.

Si l'adresse spécifiée est paire, le transfert est effecté sur la moitié supérieure du bus de données.

Dans le cas contraire, le transfert est effectué sur la moitié inférieure du bus de données.

Indicateurs modifiés 7 6 5 X N Z V C
— — — — —

Exemple de codage :

```
0389  0100      MOVEP D1,$100(A1)
```

MOVEQ Transfert rapide d'un registre de données dans la mémoire indiquée

Format MOVEQ #<Donnée>,Dn

Fonction L'instruction MOVEQ effectue un chargement rapide d'un registre de données avec une valeur immédiate signée codée sur 8 bits. La donnée chargée est étendue à 32 bits avant d'être chargée.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	*	*	0	0

Exemple de codage :

```
7403      MOVEQ #3,D2
```

MULS Multiplication signée sur 16 bits

Format MULS <AE>,Dn

Fonction L'instruction MULS effectue une multiplication signée des deux opérandes 16 bits spécifiés. L'opérande destination doit toujours être du type registre de données. Le résultat de la multiplication est stocké dans le registre de données 32 bits de destination.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	*	*	0	0

Exemple de codage :

```
C5C1      MULS D1,D2
```



MULU Multiplication non signée sur 16 bits

Format MULU <AE>,Dn

Fonction L'instruction MULU effectue une multiplication non signée des deux opérandes 16 bits spécifiés. L'opérande destination doit toujours être du type registre de données. Le résultat de la multiplication est stocké dans le registre de données 32 bits de destination.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	*	*	0	0

Exemple de codage :

```
C4C1                    MULU D1,D2
C6F9  0000  1234        MULU $1234,D3
```

NBCD Négation décimale étendue

Format NBCD <AE>

Fonction L'instruction NBCD effectue la négation décimale étendue de l'opérande destination : soustraction de l'octet destination et du bit X à la valeur 0. Le résultat est rangé dans l'opérande destination.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			*	—	*	—	*

Exemple de codage :

```
4819        NBCD (A1)+
```

NEG Négation

Format NEG <AE>

Fonction L'instruction NEG effectue une négation logique de l'opérande destination : soustraction de l'octet, du mot ou du double mot destination à la valeur 0. Le résultat est rangé dans l'opérande destination.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			*	*	*	*	*

Exemple de codage :

```
4439  0000  1234      NEG.B $1234
```

NEGX Négation étendue

Format NEGX <AE>

Fonction L'instruction NEGX effectue une négation logique étendue de l'opérande destination : soustraction de l'octet, du mot ou du double mot destination et du bit d'extension à la valeur 0. Le résultat est rangé dans l'opérande destination.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			*	—	*	—	*

Exemple de codage :

```
4021      NEGX.B -(A1)
4041      NEGX.W D1
```

NOP Pas d'opération

Format NOP

Fonction L'instruction NOP n'effectue aucune opération et demande 4 périodes d'horloge pour s'exécuter. Connaissant la fréquence de l'horloge, il est pratique d'utiliser cette instruction pour créer des temporisations.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	—	—	—	—

Exemple de codage :

```
4E71      NOP
```



NOT Complémentation logique

Format NOT <AE>

Fonction L'instruction NOT effectue une complémentation à un de l'opérande destination spécifié.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	*	*	0	0

Exemple de codage :

```
4621    NOT.B -(A1)
4642    NOT.W D2
```

OR OU logique

Format OR <AE>,Dn ou OR Dn,<AE>

Fonction L'instruction OR effectue un OU logique entre les deux opérandes spécifiés. Le résultat est placé dans le registre destination.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	*	*	0	0

Exemple de codage :

```
8351    OR.W D1,(A1)
8399    OR.L D1,(A1)+
```

ORI OU logique immédiat

Format ORI #<Donnée>,<AE>

Fonction L'instruction ORI effectue un OU logique entre l'opérande destination et la donnée immédiate spécifiée. Le résultat est placé dans le registre destination.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	*	*	0	0

Exemple de codage :

```
0062  1234        ORI.W #$1234,-(A2)
```

ORI to CCR OU logique immédiat avec CCR

Format ORI #<Donnée>,CCR

Fonction L'instruction ORI to CCR effectue un OU logique entre le registre des indicateurs (CCR) et la donnée immédiate spécifiée. Le résultat est placé dans le registre des indicateurs.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			*	*	*	*	*

Exemple de codage :

```
003C  0012        ORI #$12,CCR
```

ORI to SR OU logique immédiat avec SR

Format ORI #<Donnée>,SR

Fonction L'instruction ORI to SR effectue un OU logique entre le registre d'état (SR) et la donnée immédiate spécifiée. Le résultat est placé dans le registre d'état.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			*	*	*	*	*

Exemple de codage :

```
007C  1234        ORI #$1234,SR
```



PEA Adresse effective dans la pile

Format PEA <AE>

Fonction L'instruction PEA place l'adresse effective spécifiée dans la pile.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	—	—	—	—

Exemple de codage :

```
4851                PEA (A1)
4879  0000  1234    PEA $1234
```

RESET Remise à zéro du 68000

Format RESET

Fonction L'instruction RESET fait passer à zéro la ligne de RESET du 68000, ce qui a pour effet une initialisation des circuits externes. L'état du 68000 n'est pas modifié, et le programme continue à s'exécuter en séquence.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	—	—	—	—

Exemple de codage :

```
4E70        RESET
```

ROL Rotation sans extension vers la gauche

Format ROL Dx,Dy ou ROL #<Donnée>,Dy ou ROL <AE>

Fonction L'instruction ROL peut comporter un ou deux opérandes. Elle effectue une rotation vers la gauche de l'opérande destination. Les opérandes peuvent être d'un des types suivants :

— deux registres de données : le premier contient le nombre de rotations (entre 0 et 63) à effectuer sur le second,

— une adresse modifiable : la rotation fait toujours un bit.

Remarque :

Le bit de retenue (C dans CCR) reçoit le bit de poids le plus fort de l'opérande destination.

Contrairement à l'instruction ROXL, le bit d'extension n'est pas pris en compte dans la rotation.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	*	*	0	*

Exemple de codage :

```
E33A    ROL.B D1,D2
E7BD    ROL.L D3,D5
E7D1    ROL (A1)
```

ROR Rotation sans extension vers la droite

Format ROR Dx, Dy ou ROR #<Donnée>,Dy ou ROR <AE>

Fonction L'instruction ROR peut comporter un ou deux opérandes. Elle effectue une rotation vers la droite de l'opérande destination. Les opérandes peuvent être d'un des types suivants :

— deux registres de données : le premier contient le nombre de rotations (entre 0 et 63) à effectuer sur le second,

— une adresse modifiable : la rotation fait toujours un bit.

Remarque :

Le bit de retenue (C dans CCR) reçoit le bit de poids le plus faible de l'opérande destination.

Contrairement à l'instruction ROXR, le bit d'extension n'est pas pris en compte dans la rotation.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	*	*	0	*

Exemple de codage :

```
E23A    ROR.B D1,D2
E6BD    ROR.L D3,D5
E6D2    ROR (A2)
```



ROXL Rotation avec extension vers la gauche

Format

ROXL Dx,Dy ou ROXL #<Donnée>,Dy ou ROXL <AE>

Fonction

L'instruction ROXL peut comporter un ou deux opérandes. Elle effectue une rotation vers la gauche de l'opérande destination et du bit d'extension. Les opérandes peuvent être d'un des types suivants :

— deux registres de données : le premier contient le nombre de rotations (entre 0 et 63) à effectuer sur le second,

— une adresse modifiable : la rotation fait toujours un bit.

Remarque :

Le bit de retenue (C dans CCR) et le bit d'extension reçoivent le bit de poids le plus fort de l'opérande destination. L'ancienne valeur du bit d'extension est placée dans le bit de poids le plus faible de l'opérande destination.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			*	*	*	0	*

Exemple de codage :

```
E372                ROXL.W D1,D2
E5F9  0000  1234    ROXL $1234
```

ROXR Rotation avec extension vers la droite

Format

ROXR Dx,Dy ou ROXR #<Donnée>,Dy ou ROXR <AE>

Fonction

L'instruction ROXR peut comporter un ou deux opérandes. Elle effectue une rotation vers la droite de l'opérande destination et du bit d'extension. Les opérandes peuvent être d'un des types suivants :

— deux registres de données : le premier contient le nombre de rotations (entre 0 et 63) à effectuer sur le second,

— une adresse modifiable : la rotation fait toujours un bit.

Remarque :

Le bit de retenue (C dans CCR) et le bit d'extension reçoivent le bit de poids le plus faible de l'opérande destination. L'ancienne valeur du bit d'extension est placée dans le bit de poids le plus fort de l'opérande destination.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			*	*	*	0	*

Exemple de codage :

```
E2B3      ROXR.L D1,D3
E4D1      ROXR (A1)
```

RTE — Retour d'exception (en mode superviseur), Génération d'une exception (en mode utilisateur)

Format RTE

Fonction L'instruction RTE est utilisée en fin d'exception pour retourner au programme interrompu. Le registre d'état SR et le compteur de programme sont initialisés avec les valeurs qui se trouvent au sommet de la pile système.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			*	*	*	*	*

Exemple de codage :

```
4E73      RTE
```

RTR — Retour et restauration des codes condition

Format RTR

Fonction L'instruction RTR est utilisée pour rendre le contrôle au programme appelant. Le premier mot de la pile système est placé dans le registre des indicateurs CCR. Le mot long qui suit est placé dans le PC.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			*	*	*	*	*

Exemple de codage :

```
4E77      RTR
```



RTS Retour d'un sous-programme

Format RTS

Fonction L'instruction RTS est utilisée pour rendre le contrôle au programme appelant après un appel de sous-programme. Le premier mot long de la pile système est placé dans le PC.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	—	—	—	—

Exemple de codage :

```
4E75      RTS
```

SBCD Soustraction décimale étendue

Format SBCD Dy,Dx ou SBCD −(Ay),−(Ax)

Fonction L'instruction SBCD soustrait l'octet source et le bit d'extension de l'octet destination. Les deux octets ainsi que le résultat sont des nombres codés en décimal. Les deux octets doivent se trouver dans des registres de données ou dans des mémoires accédées en pré-décrémentation.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			*	—	*	—	*

Exemple de codage :

```
8501      SBCD D1,D2
8308      SBCD -(A0),-(A1)
```

Scc Initialisation d'un octet en fonction d'une condition

Format Scc <AE>
où cc=CC, CS, EQ, NE, GE, GT, HI, LE, LS, LT, MI, PL, VC, VS, T OU F.

Fonction L'instruction Scc teste la condition cc. Si elle est vraie, l'octet qui occupe l'adresse effective spécifiée est mis à $FF. Il est mis à zéro dans le cas

contraire. La condition exprimée dans l'instruction peut être l'une des suivantes :

Suffixe	Condition
CC	C = 0
CS	C = 1
EQ	Z = 1
GE	N OU exclusif V = 0
GT	Z ET (N OU exclusif V) = 0
HI	C OU Z = 1
LE	Z OU (N OU exclusif V) = 1
LS	C OU Z = 1
LT	N OU exclusif V = 1
MI	N = 1
NE	Z = 0
PL	N = 1
VC	V = 0
VS	V = 1

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	—	—	—	—

Exemple de codage :

```
56D1          SNE (A1)
5BE9  0012    SMI $12(A1)
```

STOP Chargement du registre d'état et arrêt (en mode superviseur)

Format

STOP #<Donnée>

Fonction

L'instruction STOP interrompt le fonctionnement du microprocesseur en mode superviseur et fait pointer le compteur d'instruction sur l'instruction suivante. Le mot qui se trouve à la suite de l'instruction est placé dans le registre d'état SR.

L'exécution reprend lorsqu'un des événements suivants se produit :

— reset externe,

— exception de type trace dans le cas où le bit T du registre des indicateurs vaut 1 dans la donnée immédiate qui suit l'instruction,

— interruption externe d'un niveau supérieur au niveau courant.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			*	*	*	*	*



Exemple de codage :

```
4E72 04D2       STOP #1234
```

SUB Soustraction binaire

Format SUB <AE>, Dn ou SUB Dn,<AE>

Fonction L'instruction SUB soustrait sans extension l'opérande source de l'opérande destination.
Le résultat est rangé dans l'opérande destination. L'opérande source est adressé par un mode d'adressage quelconque. Par contre, l'opérande destination doit être adressée par un mode mémoire modifiable.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			*	*	*	*	*

Exemple de codage :

```
9319        SUB.B D1,(A1)+
96A2        SUB.L -(A2),D3
```

SUBA Soustraction d'une adresse

Format SUBA <AE>,An

Fonction L'instruction SUBA effectue une soustraction sans extension entre le premier opérande et un registre d'adresses. Le résultat est rangé dans le registre d'adresses. L'opérande source est adressé par un mode d'adressage quelconque.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	—	—	—	—

Exemple de codage :

```
96EA        SUBA.W -(A4),A3
```

SUBI Soustraction immédiate

Format SUBI #<Donnée>,<AE>

Fonction L'instruction SUBI effectue une soustraction sans extension entre une donnée immédiate et l'opérande destination. Le résultat est rangé dans l'opérande destination. Ce dernier doit être adressé par un mode du type données modifiables.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			*	*	*	*	*

Exemple de codage :

```
041C  0012      SUBI.B #$12,(A4)+
```

SUBQ Soustraction rapide

Format SUBQ #<Donnée>,<AE>

Fonction L'instruction SUBQ effectue une soustraction rapide sans extension entre une donnée immédiate (comprise entre 1 et 8) et l'opérande destination. Le résultat est rangé dans l'opérande destination. Son mode d'adressage doit être du type modifiable.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			*	*	*	*	*

Exemple de codage :

```
5311        SUBQ.B #1,(A1)
```

SUBX Soustraction étendue

Format SUBX Dy,Dx ou SUBX -(Ay),-(Ax)

Fonction L'instruction SUBX effectue une soustraction signée (bit X du CCR) entre ses deux opérandes. Les deux opérandes doivent être :
— tous deux des registres de données,
— tous deux des registres d'adresses en mode prédécrémentation.



Indicateurs modifiés	7	6	5	X	N	Z	V	C
				*	*	*	*	*

Exemple de codage :

```
9943        SUBX.W D3,D4
```

SWAP Echange des mots de poids fort et de poids faible d'un registre

Format SWAP Dn

Fonction L'instruction SWAP échange les mots de poids fort et de poids faible d'un registre de données.

Indicateurs modifiés	7	6	5	X	N	Z	V	C
				—	*	*	0	0

Exemple de codage :

```
4841        SWAP D1
4845        SWAP D5
```

TAS Test d'un octet, et mise à un de son bit 7

Format TAS <AE>

Fonction L'instruction TAS teste l'octet adressé par l'adresse effective spécifiée. Les bits C et V du registre des indicateurs sont mis à zéro. Le bit N est mis à un si le bit 7 de l'octet est à un, et à zéro dans le cas contraire. Le bit Z est mis à un si l'octet est nul, et à zéro dans le cas contraire. Le bit X n'est pas modifié. Après exécution de l'instruction, le bit 7 de l'octet adressé est mis à un.

Cette instruction permet de partager des ressources en mode multi-processeur. L'octet adressé joue le rôle de sémaphore.

Indicateurs modifiés	7	6	5	X	N	Z	V	C
				—	*	*	0	0

Exemple de codage :

```
4AC1            TAS D1
4AEE  0012      TAS $12(A6)
```

TRAP Exécution d'une exception

Format TRAP #<Vecteur>

Fonction L'instruction TRAP permet d'exécuter une procédure d'exception. Le numéro du vecteur d'exception est spécifié par les quatre bits de poids faibles de la donnée qui suit l'instruction.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	—	—	—	—

Exemple de codage :

```
4E45      TRAP #5
```

TRAPV Exécution d'une exception si débordement

Format TRAPV

Fonction L'instruction TRAPV génère l'exception dont le numéro est spécifié dans le cas où le bit V (*overflow*) du registre CCR est positionné. Dans le cas où ce bit n'est pas positionné, l'exécution se poursuit en séquence.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	—	—	—	—

Exemple de codage :

```
4E76      TRAPV
```



TST Test d'un opérande

Format TST <AE>

Fonction L'instruction TST teste la valeur de l'opérande octet, mot ou mot long spécifié. Le registre des indicateurs (mis à part le bit X) est positionné en fonction du résultat du test. L'opérande est forcément du type données modifiables.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	*	*	0	0

Exemple de codage :

```
4A02                    TST.B D2
4A79  0000  1234        TST.W $1234
```

UNLK Transfert d'un registre d'adresses dans le pointeur de pile

Format UNLK An

Fonction L'instruction UNLK transfère le registre d'adresses spécifié dans le pointeur de pile, puis la valeur pointée par la pile dans le registre d'adresses.

Indicateurs modifiés

7	6	5	X	N	Z	V	C
			—	—	—	—	—

Exemple de codage :

```
4E59      UNLK A1
4E5D      UNLK A5
```

4/3.1

Aide-mémoire des codes opératoires du 68000

Ce programme utilitaire écrit en GFA Basic rendra certainement de grands services aux programmeurs en assembleur. Il donne en effet de manière résumée la fonction de chaque code opérationnel du 68000 :

— code opérationnel ;

— résumé des opérations effectuées ;

— taille des opérandes ;

— effet du code op (en pseudo code) ;

— état des codes conditions après exécution du code op.

COMMENT UTILISER LE PROGRAMME

Lancez le programme. Les messages suivants sont affichés sur l'écran :

```
Aide-mémoire des codes opérationnels du 68000
---------- --- ----- ---------- -- -----

Entrez le code opérationnel à expliciter : ?
```

Les codes reconnus par le programme doivent faire partie des 84 suivants :

ABCD, ADD, ADDA, ADDI, ADDQ, ADDX, AND, ANDI, ANDI CCR, ANDI SR, ASL, ASR, Bcc, CBHG, BCLR, BRA, BSET, BSR, BTST, CHK, CLR, CMP, CMPA, CMPI, CMPM, DBcc, DIVS, DIVU, EOR, EORI, EORI CCR, EORI SR, EXG, EXT, JMP, JSR, LEA, LINK, LSL, LSR, MOVE, MOVE CCR, MOVE SR, MOVE USP, MOVEA, MOVEM, MOVEP, MOVEQ, MULS, MULU, NBCD, NEG, NEGX, NOP, NOT, OR, ORI, ORI CCR, ORI SR, PEA, RESET, ROL, ROR, ROXL, ROXR, RTE, RTR, RTS, SBCD, Scc, STOP, SUB, SUBA, SUBI, SUBQ, SUBX, SWAP, TAS, TRAP, TRAPV, TST, UNLK.

Si vous tapez (par exemple) MOVE, en majuscules ou en minuscules, les données concernant le code opérationnel MOVE sont affichées sur l'écran :

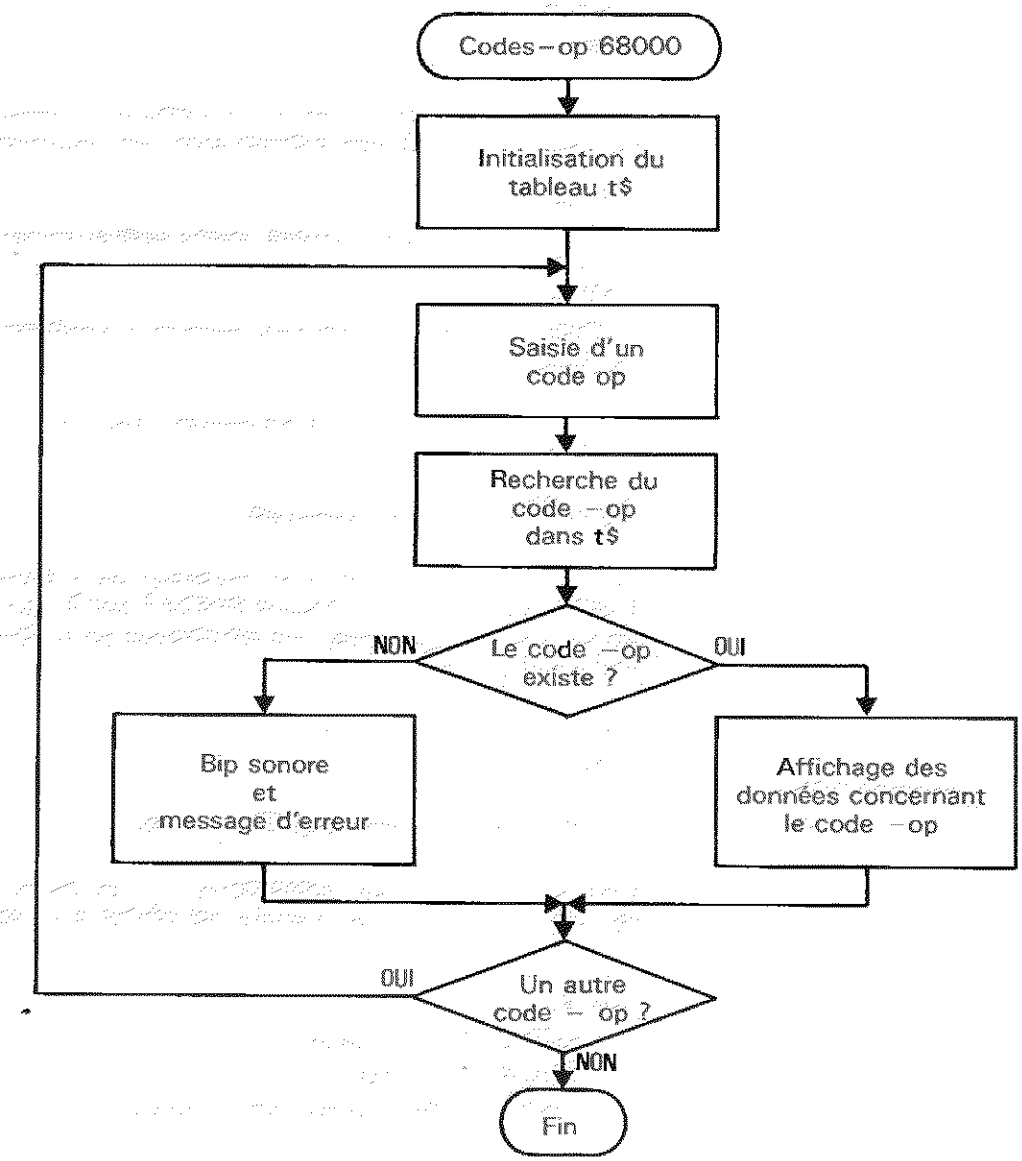
```
MOVE
Transfert de l'opérande source dans l'opérande destination
Taille des opérandes : B/W/L
Effet du code opératoire : s—d
Codes condition  X N Z V C
                 – * * 0 0
Un autre code opératoire (O/N) : ?
```

Répondez O pour avoir des informations sur un autre code opérationnel et N pour quitter le programme.



LE PROGRAMME EN DÉTAIL

La logique du programme apparaît dans l'ordinogramme suivant :



La première action effectuée par le programme consiste à stocker les données concernant les divers codes opérationnels dans le tableau **t$** :

```
DIM t$(200,5)
FOR i=1 TO 118
  FOR j=1 TO 5
    READ t$(i,j)
  NEXT j
NEXT i
```

Un écran de présentation est ensuite affiché. L'utilisateur doit entrer le code opérationnel pour lequel il désire avoir des informations :

```
CLS
PRINT "Aide-mémoire des codes opérationnels du 68000"
PRINT "------------------ ----- ------------ -- -----"
PRINT
PRINT "Entrez le code opérationnel à expliciter :";
INPUT codeop$
```

Le code opérationnel entré est ensuite converti en majuscules à l'aide de l'instruction UPPER$:

```
codeop$=UPPER$(codeop$)
```

Ce code opérationnel est alors recherché dans le tableau **t$**. La recherche s'effectue à l'aide d'une boucle REPEAT UNTIL qui prend fin lorsque toutes les données du tableau ont été décrites ou lorsque le code opérationnel recherché est trouvé :

```
i=0
REPEAT
  i = i+1
UNTIL (t$(i,1)=codeop$) OR (i=118)
```

Dans le cas où le code opérationnel n'a pas été trouvé, un signal sonore est émis et un message d'erreur est affiché sur l'écran :

```
PRINT
IF i=118 THEN
  SOUND 1,15,#284,20
  SOUND 1,0,0,0,0
  PRINT "Code opérationnel inconnu."
```

Dans le cas où le code opérationnel a été trouvé, les données le concernant sont affichées sur l'écran à l'aide d'une série d'instructions LOCATE / PRINT :



```
  ELSE
    CLS
    PRINT t$(i,1)
    LOCATE 3,1
    IF LEN(t$(i,5))>80 THEN
      j=80
      REPEAT
        j=j-1
      UNTIL (MID$(t$(i,5),j,1)=" ")
      PRINT MID$(t$(i,5),1,j)
      PRINT MID$(t$(i,5),j+1,LEN(t$(i,5))-j)
    ELSE
      PRINT t$(i,5)
    ENDIF
    LOCATE 5,1
    PRINT "Taille des opérandes : ";t$(i,2)
    LOCATE 7,1
    PRINT "Effet du code opératoire : ";t$(i,3)
    LOCATE 9,1
    PRINT "Codes condition X N Z V C"
    LOCATE 10,17
    FOR j=1 TO 5
      PRINT MID$(t$(i,4),j,1); " ";
    NEXT j
  ENDIF
```

Que le code opérationnel ait été trouvé ou non dans le tableau **t$**, le programme demande à l'utilisateur s'il désire des explications sur un autre code ;

```
LOCATE 14,1
PRINT "Un autre code opératoire (O/N) : ";
INPUT rep$
```

Dans le cas où la réponse est différente de "**O**", le programme se termine :

```
UNTIL (UPPER$(rep$)<>"O")
END
```

La fin du programme rassemble les données relatives aux codes opérationnels :

```
DATA ABCD,B,d10+s10+X-d,*U*U*
DATA Addition décimale avec le bit d'extension entre deux registres de
données ou deux cases mémoire
...
DATA UNLK,,"An-SP, (SP)+-An",--
DATA Transfert d'un registre d'adresses dans le pointeur de pile
```

Ces données sont organisées de la manière suivante :

DATA <Code-Op>, <Taille>, <Pseudo>, <Conditions>, <Effet>

où <Code-Op> est le code opératoire.

<Taille> est la taille du ou des opérande(s)
B = Byte, W = Word, L = Long Word

<Pseudo> résume les opérations effectuées en pseudo-code,

<Conditions> spécifie la modification des codes conditions
(*) après exécution du code opérationnel,

et <Effet> résume (en clair) l'effet du code opérationnel;

(*) L'affectation des codes condition est codifiée comme suit :

* : modifié,

– : non affecté,

0 : mis à zéro,

1 : mis à un

U : non défini;

Le listing du programme est le suivant :

```
' ==================================================
' AIDE MEMOIRE DES CODES OPERATIONNELS DU 68000
' ==================================================
'
' ____________________________________________
' Initialisation du tableau de données
' ____________________________________________
'
DIM t$(200,5)
FOR i=1 TO 118
  FOR j=1 TO 5
    READ t$(i,j)
  NEXT j
NEXT i
'
' ________________
' Présentation
' ________________
'
REPEAT
  CLS
  PRINT "Aide mémoire des codes opérationnels du 68000"
  PRINT "____________________________________________"
  PRINT
  PRINT "Entrez le code opérationnel à expliciter :";
  INPUT codeop$
  codeop$=UPPER$(codeop$)
  '
  ' ____________________________________________
  ' Recherche du code op dans le tableau
  ' ____________________________________________
  '
  i=0
  REPEAT
    i=i+1
  UNTIL (t$(i,1)=codeop$) OR (i=118)
  '
  ' ____________________________
  ' Affichage du résultat
  ' ____________________________
  '
  PRINT
  IF i=118 THEN
    SOUND 1,15,#284,20
    SOUND 1,0,0,0,0
    PRINT "Code opérationnel inconnu."
  ELSE
    CLS
    PRINT t$(i,1)
    LOCATE 3,1
    IF LEN(t$(i,5))>80 THEN
      j=80
      REPEAT
        j=j-1
```

(suite)

```
        UNTIL (MID$(t$(i,5),j,1)=" ")
        PRINT MID$(t$(i,5),1,j)
        PRINT MID$(t$(i,5),j+1,LEN(t$(i,5))-j)
      ELSE
        PRINT t$(i,5)
      ENDIF
      LOCATE 5,1
      PRINT "Taille des opérandes : ";t$(i,2)
      LOCATE 7,1
      PRINT "Effet du code opératoire : ";t$(i,3)
      LOCATE 9,1
      PRINT "Codes condition X N Z V C"
      LOCATE 10,17
      FOR j=1 TO 5
        PRINT MID$(t$(i,4),j,1);" ";
      NEXT j
    ENDIF
    LOCATE 14,1
    PRINT "Un autre code opératoire (O/N) : ";
    INPUT rep$
  UNTIL (UPPER$(rep$)<>"O")
  END
  '
  ' ------------------------
  ' ZONE DES DONNEES
  ' ------------------------
  '
  DATA ABCD,B,d10+s10+X->d,*U*U*
  DATA Addition décimale avec le bit d'extension entre deux
  registres de données ou deux cases mémoire
  DATA ADD,B/W/L,d+Dn->d ou Dn+s->Dn ou d+Dn->d ou Dn+s->Dn,*****
  DATA Addition binaire entre un opérande et le contenu d'un
   registre de données
  DATA ADDA,W/L,Ar+s->An,-----
  DATA Addition binaire entre un opérande et le contenu d'un
  registre d'adresses
  DATA ADDI,B/W/L,d+#->d,*****
  DATA Addition immédiate
  DATA ADDQ,B/W/L,d+#->d,*****
  DATA Addition rapide (constante entre 0 et 7)
  DATA ADDX,B/W/L,d+s+X->d,*****
  DATA Addition binaire avec le bit d'extension
  DATA AND,B/W/L,(d AND Dn)->d ou (Dn AND s)->d ou (d AND Dn)->d
  ou (Dn AND s)->Dn,-**00
  DATA ET logique
  DATA ANDI,B/W/L,(d AND #)->d,-**00
  DATA ET logique immédiat
  DATA ANDI CCR,B,(# AND CCR)-> CCR,*****
  DATA ET logique entre CCR et la donnée immédiate
  DATA ANDI SR,W,(# AND SR)->SR,*****
  DATA ET logique entre SR et la donnée immédiate en mode
  superviseur
  DATA ASL,B/W/L,"Dernier bit décalé->C et X, bits entrant à zéro"
  ,*****
```



(suite)

```
DATA Décalage arithmétique à gauche
DATA ASR,B/W/L,"Dernier bit décalé->C et X, bit de poids fort
conservé",*****
DATA Décalage arithmétique à droite
DATA BCC,B/W,"Si C=0, PC+Disp->PC",------
DATA Branchement conditionnel si retenue à zéro
DATA BCS,B/W,"Si C=1, PC+Disp->PC",------
DATA Branchement conditionnel si retenue à un
DATA BEQ,B/W,"Si Z=1, PC+Disp->PC",------
DATA Branchement conditionnel si égal
DATA BNE,B/W,"Si Z=0, PC+Disp->PC",------
DATA Branchement conditionnel si différent
DATA BGE,B/W,"Si (N OU Excl V)=0, PC+Disp->PC",------
DATA Branchement conditionnel si supérieur ou égal
DATA BGT,B/W,"Si Z+(N OU Excl V)=0, PC+Disp->PC",------
DATA Branchement conditionnel si supérieur
DATA BHI,B/W,"Si C+Z=0, PC+Disp->PC",------
DATA Branchement conditionnel si supérieur
DATA BLE,B/W,"Si Z + (N OU Excl V) = 1, PC+Disp->PC",------
DATA Branchement conditionnel si inférieur ou égal
DATA BLS,B/W,"Si C+Z=1, PC+Disp->PC",------
DATA Branchement conditionnel si inférieur ou égal
DATA BLT,B/W,"Si (N OU Excl V)=1, PC+Disp->PC",------
DATA Branchement conditionnel si inférieur
DATA BMI,B/W,"Si N=1, PC+Disp->PC",------
DATA Branchement conditionnel si négatif
DATA BPL,B/W,"Si N=0, PC+Disp->PC",------
DATA Branchement conditionnel si positif
DATA BVC,B/W,"Si V=0, PC+Disp->PC",------
DATA Branchement conditionnel si dépassement à zéro
DATA BCHG,B/L,Complément du bit # de d -> Z et Complément du
bit # de d -> bit # de d,--*--
DATA "Test d'un bit, affectation à Z, puis inversion"
DATA BCLR,B/L,Complément du bit # de d -> Z et 0 -> bit # de d,
--*--
DATA "Test d'un bit, affectation à Z, puis mise à zéro"
DATA BRA,B/W,PC+Disp->PC,------
DATA Branchement inconditionnel
DATA BSET,B/L,Complément du bit # de d -> Z et 1 -> bit # de d,
--*--
DATA "Test d'un bit, affectation à Z, puis mise à un"
DATA BSR,B/W,"PC->-(SP),PC+Disp->PC",------
DATA Branchement à un sous-programme
DATA BTST,B/L,Complément du bit # de d -> Z,--*--
DATA Test d'un bit et affectation de ce bit à Z
DATA CHK,W,Si Dn<0 ou Dn>(Bound) alors trap,-*UUU
DATA Comparaison du contenu d'une adresse à deux valeurs limites
DATA CLR,B/W/L,0->d,-0100
DATA Mise à zéro de l'opérande destination
DATA CMP,B/W/L,Dn-s,-****
DATA Comparaison des opérandes Source et Destination
DATA CMPA,W/L,An-s,-****
DATA Comparaison de l'opérande source et de l'opérande adresse
destination.
```

(suite)

```
DATA CMPI,B/W/L,d-#,-****
DATA Comparaison de l'opérande et de la donnée immédiate
DATA CMPM,B/W/L,d-s,-****
DATA Comparaison de l'opérande source et de l'opérande destination
DATA DBCC,W,Si CC=1 alors Dn-1->Dn. Si Dn<>-1 alors PC+Disp->PC,
-----
DATA Branchement conditionnel si retenue à un et Dn-1 <> -1
DATA DBCS,W,Si CC=0 alors Dn-1->Dn. Si Dn<>-1 alors PC+Disp->PC,
-----
DATA Branchement conditionnel si retenue à zéro et Dn-1 <> -1
DATA DBEQ,W,Si Z=0 alors Dn-1->Dn. Si Dn<>-1 alors PC+Disp->PC,
-----
DATA Branchement conditionnel si non égal et Dn-1 <> -1
DATA DBNE,W,Si Z=1 alors Dn-1->Dn. Si Dn<>-1 alors PC+Disp->PC,
-----
DATA Branchement conditionnel si égal et Dn-1 <> -1
DATA DBGE,W,Si (N OU EXCL)=1 alors Dn-1->Dn. Si Dn<>-1 alors
PC+Disp->PC,------
DATA Branchement conditionnel si non supérieur ou égal et
Dn-1 <> -1
DATA DBGT,W,Si Z+(N OU EXCL V)=1 alors Dn-1->Dn. Si Dn<>-1 alors
PC+Disp->PC,------
DATA Branchement conditionnel si non supérieur et Dn-1 <> -1
DATA DBHI,W,Si C+Z=1 alors Dn-1->Dn. Si Dn<>-1 alors PC+Disp->PC,
-----
DATA Branchement conditionnel si non supérieur et Dn-1 <> -1
DATA DBLE,W,Si Z+(N OU EXCL V)=0 alors Dn-1->Dn. Si Dn<>-1 alors
PC+Disp->PC,------
DATA Branchement conditionnel si non inférieur ou égal et Dn-1 <> -1
DATA DBLS,W,Si C+Z=0 alors Dn-1->Dn. Si Dn<>-1 alors PC+Disp->PC,
-----
DATA Branchement conditionnel si non inférieur ou égal et Dn-1 <> -1
DATA DBLT,W,Si (N OU EXCL V)=0 alors Dn-1->Dn. Si Dn<>-1 alors
PC+Disp->PC,------
DATA Branchement conditionnel si non inférieur et Dn-1 <> -1
DATA DBMI,W,Si N=0 alors Dn-1->Dn. Si Dn<>-1 alors PC+Disp->PC,
-----
DATA Branchement conditionnel si positif et Dn-1 <> -1
DATA DBPL,W,Si N=1 alors Dn-1->Dn. Si Dn<>-1 alors PC+Disp->PC,
-----
DATA Branchement conditionnel si négatif et Dn-1 <> -1
DATA DBVC,W,Si V=1 alors Dn-1->Dn. Si Dn<>-1 alors PC+Disp->PC,
-----
DATA Branchement conditionnel si dépassement à un et Dn-1 <> -1
DATA DIVS,W,Dn32/s16->Dn(r:q),-***0
DATA Division arithmétique signée
DATA DIVU,W,Dn32/s16->Dn(r:q),-***0
DATA Division arithmétique non signée
DATA EOR,B/W/L,d OU EXCL Dn -> d,-**00
DATA OU Exclusif
DATA EORI,B/L,d OU EXCL # -> d,-**00
DATA OU Exclusif immédiat
DATA EORI CCR,B,# OU EXCL CCR -> CCR,*****
DATA OU Exclusif entre CCR et la donnée immédiate
```



(suite)

```
DATA EORI SR,W,# OU EXCL SR -> SR,*****
DATA OU Exclusif entre SR et la donnée immédiate
DATA EXG,L,s<->d,------
DATA Echange de deux registres de 32 bits
DATA EXT,W/L,bit 7->bits 15 à 8 ou bit 15-> bits 31 à 16,-**00
DATA Extension sur 16 ou 32 bits
DATA ILLEGAL,,"PC->-(SSP),SR->-(SSP),(vecteur illegal)->PC",------
DATA Exception 4
DATA JMP,,d->PC,------
DATA Saut inconditionnel
DATA JSR,,"PC->-(SP),d->PC",------
DATA Saut au sous-programme dont l'adresse est spécifiée
DATA LEA,L,s->An,------
DATA Chargement d'un registre d'adresse avec l'adresse effective
DATA LINK,,"An->-(SP), SP->An,SP+depl->SP",------
DATA "Registre d'adresses dans la pile, chargement avec SP et
ajout du déplacement à SP"
DATA LSL,B/W/L,"dernier bit décalé -> C et X, 0 -> bits de poids
faibles décalés",***0*
DATA Décalage logique vers la gauche
DATA LSR,B/W/L,"dernier bit décalé -> C et X, 0 -> bits de poids
forts décalés",***0*
DATA Décalage logique vers la droite
DATA MOVE,B/W/L,s->d,-**00
DATA Transfert de l'opérande source dans l'opérande destination
DATA MOVE CCR,W,s->CCR,*****
DATA Transfert de l'opérande source de/vers CCR
DATA MOVE SR,W,s->SR ou SR->d,***** ou ------
DATA Transfert de l'opérande source de/vers SR
DATA MOVE USP,L,USP->An ou An->USP,------
DATA Transfert du contenu du pointeur de pile de/vers un registre
d'adresses
DATA MOVEA,W/L,s->An,------
DATA Transfert de l'opérande source vers un registre d'adresses
DATA MOVEM,W/L,Xn->d ou s->Xn,------
DATA Transfert d'un ou de plusieurs registres de/vers l'adresse
effective indiquée
DATA MOVEP,W/L,Dn->d ou s->Dn,------
DATA Transfert d'un ou de plusieurs registres de données de/vers
l'adresse effective indiquée
DATA MOVEQ,L,#->Dn,-**00
DATA Transfert rapide d'une donnée vers un registre de données
DATA MULS,W,Dnxs->Dn,-**00
DATA Multiplication signée de deux registres 16 bits
DATA MULU,W,Dnxs->Dn,-**00
DATA Multiplication non signée de deux registres de 16 bits
DATA NBCD,B,0-d10-X->d,*U*U*
DATA Négation décimale avec le bit d'extension
DATA NEG,B/W/L,0-d->d,*****
DATA Négation
DATA NEGX,B/W/L,0-d-X->d,*****
DATA Négation avec le bit d'extension
DATA NOP,,,------
DATA Aucune opération
```

(suite)

```
DATA NOT,B/W/L,d barre->d,-**00
DATA Complément à un de l'opérande destination
DATA OR,B/W/L,d OR Dn->d ou Dn OR s->Dn,-**00
DATA OU logique entre deux registres de données
DATA ORI,B/W/L,d OR #->D,-**00
DATA OU logique entre une donnée immédiate et le registre destination
DATA ORI CCR,B,# OR CCR->CCR,*****
DATA OU logique entre CCR et une donnée immédiate
DATA ORI SR,W,s OR SR->SR,*****
DATA OU logique entre SR et une donnée immédiate
DATA PEA,L,s->-(SP),-----
DATA Sauvegarde de l'adresse effective dans la pile
DATA RESET,B/W/L,activation du connecteur RESET,-----
DATA Activation de la broche RESET si le processeur est en état
superviseur
DATA ROL,B/W/L,"Rotation à gauche, dernier bit déplacé -> C",-**00
DATA Rotation à gauche
DATA ROR,B/W/L,"Rotation à droite, dernier bit déplacé -> C",-**00
DATA Rotation à droite
DATA ROXL,B/W/L,"Rotation à gauche, dernier bit déplacé dans C et X",
***0*
DATA Rotation à gauche en passant par le bit d'extension
DATA ROXR,B/W/L,"Rotation à droite, dernier bit déplacé dans C et X",
***0*
DATA Rotation à droite en passant par le bit d'extension
DATA RTE,,"(SP)+->SR, (SP)+->PC",*****
DATA Retour d'exception si le processeur est en état superviseur
DATA RTR,,"(SP)+->CCR,(SP)+->PC",*****
DATA Retour de sous-programme avec restitution des codes conditions CCR
DATA RTS,,(SP)+->PC,-----
DATA Retour de sous-programme
DATA SBCD,B,d10-s10-X->d,*U*U*
DATA Soustraction décimale des opérandes source et destination
DATA SCC,B,Si C=0 alors 1s->d sinon 0s->d,-----
DATA Si retenue à zéro alors l'octet spécifié est initialisé à $FF.
Sinon il est initialisé à $00
DATA SCS,B,Si C=1 alors 1s->d sinon 0s->d,-----
DATA Si retenue à un alors l'octet spécifié est initialisé à $FF.
Sinon il est initialisé à $00
DATA SEQ,B,Si Z=1 alors 1s->d sinon 0s->d,-----
DATA Si égal alors l'octet spécifié est initialisé à $FF.
Sinon il est initialisé à $00
DATA SNE,B,Si Z=0 alors 1s->d sinon 0s->d,-----
DATA Si différent alors l'octet spécifié est initialisé à $FF.
Sinon il est initialisé à $00
DATA SGE,B,Si (N OU EXCL V)=0 alors 1s->d sinon 0s->d,-----
DATA Si supérieur ou égal alors l'octet spécifié est initialisé à $FF.
Sinon il est initialisé à $00
DATA SGT,B,Si Z+(N OU EXCL V)=0 alors 1s->d sinon 0s->d,-----
DATA Si supérieur alors l'octet spécifié est initialisé à $FF.
Sinon il est initialisé à $00
DATA SHI,B,Si C+Z=0 alors 1s->d sinon 0s->d,-----
DATA Si supérieur alors l'octet spécifié est initialisé à $FF.
Sinon il est initialisé à $00
```



(suite)

```
DATA SLE,B,Si Z+(N OU EXCL V)=1 alors 1s->d sinon 0s->d,------
DATA Si inférieur ou égal alors l'octet spécifié est initialisé à $FF.
Sinon il est initialisé à $00
DATA SLS,B,Si C+Z=1 alors 1s->d sinon 0s->d,------
DATA Si inférieur ou égal alors l'octet spécifié est initialisé à $FF.
Sinon il est initialisé à $00
DATA SLT,B,Si (N OU EXCL V)=1 alors 1s->d sinon 0s->d,------
DATA Si inférieur alors l'octet spécifié est initialisé à $FF.
Sinon il est initialisé à $00
DATA SMI,B,Si N=1 alors 1s->d sinon 0s->d,------
DATA Si négatif alors l'octet spécifié est initialisé à $FF.
Sinon il est initialisé à $00
DATA SPL,B,Si N=0 alors 1s->d sinon 0s->d,------
DATA Si positif alors l'octet spécifié est initialisé à $FF.
Sinon il est initialisé à $00
DATA SVC,B,Si V=0 alors 1s->d sinon 0s->d,------
DATA Si dépassement à zéro alors l'octet spécifié est initialisé à $FF.
Sinon il est initialisé à $00
DATA STOP,,"#->SR, attente d'interruption",*****
DATA Arret du processeur s'il se trouve en mode superviseur
DATA SUB,B/W/L,d-Dn->d ou Dn-s->Dn,*****
DATA Soustraction de l'opérande source et de l'opérande destination
DATA SUBA,W/L,An-s->An,------
DATA Soustraction de l'opérande source et de l'opérande adresse
destination
DATA SUBI,B/W/L,d-#->d,*****
DATA Soustraction de la donnée immédiate et de l'opérande destination
DATA SUBQ,B/W/L,d-#->d,*****
DATA Soustraction rapide de la donnée immédiate (entre 0 et 7) et de
l'opérande destination
DATA SUBX,B/W/L,d-s-X->d,*****
DATA Soustraction de l'opérande source et de l'opérande destination
avec le bit d'extension
DATA SWAP,W,Dn(31:16)<->Dn(15:0),-**00
DATA Echange des 16 bits de poids fort avec les 16 bits de poids faible
DATA TAS,B,"test d->cc, 1 -> bit 7 de d",-**00
DATA "Test d'un octet, puis mis à 1 du bit 7 de cet octet"
DATA TRAP,,"PC->-(SSP),SR->-(SSP),Vecteur de trap->PC",------
DATA Exécution d'une procédure d'exception
DATA TRAPV,,"Si V=1 alors PC->-(SSP), SR->-(SSP),
vecteur de trapv -> PC",------
DATA Exécition d'une procédure d'exception si l'indicateur V est
positionné
DATA TST,B/W/L,test d->cc,-**00
DATA Comparaison de l'opérande par rapport à zéro
DATA UNLK,,"An->SP,(SP)+->An",------
DATA Transfert d'un registre d'adresses dans le pointeur de pile
```

4/3.2

L'assembleur par la pratique

4/3.2.1

Le fonctionnement de l'instruction MOVE

L'instruction MOVE est utilisée en assembleur pour déplacer des données depuis une adresse source vers une adresse destination.

La structure générale de cette instruction est **move Source,Destination**.

Après cette instruction, le contenu de la source est inchangé et l'ancien contenu de la destination est perdu.

Cette instruction de structure générale constante revêt des formes particulières suivant le format de la source et de la destination.

PREMIER CAS : LA SOURCE EST UNE VALEUR NUMÉRIQUE

Les valeurs numériques sont toujours précédées du symbole # :
MOVE #10,D0

On transfère la valeur numérique 10 dans le registre D0.

Par défaut, le 68000 est un processeur qui travaille sur des mots, cette instruction sera traduite par **move.w#10,D0**.

Avant transfert de la valeur #10, cette dernière est codée sur une valeur de mot (16 bits en binaire) soit :

#10 00000000 00001010

NB : les espaces entre les bits ne sont introduits que pour bien montrer la séparation du codage octet par octet, de huit bits en huit bits.

Supposons que le registre D0 (comme tous les registres du 68000 il contient 32 bits ou mot long) soit, avant l'instruction, rempli de 1 :

```
D0      11111111 11111111 11111111 11111111
          <poids fort>      <poids faible>
```

C'est comme si un registre était constitué par l'assemblage de deux mots. Par défaut, l'instruction **"move"** fonctionne sur des mots, si on ne le précise pas dans le code opération. Votre assembleur a toutes les chances de traduire immédiatement le **"move"** en **"move.w"** dès que vous aurez validé l'instruction. Le déplacement va donc s'opérer sur l'un des deux mots que comporte le registre D0. Par défaut c'est le mot des poids faibles qui est sélectionné dans le fonctionnement du 68000 (mais ce n'est pas le cas de tous les micro-processeurs).

L'instruction s'écrit maintenant **move.w #10,D0**.

Le mot poids fort du registre D0 n'est pas touché et le mot poids faible est remplacé par la valeur décimale 10.

Après l'instruction, D0 a la valeur binaire suivante :

```
D0      11111111 11111111 00000000 00001010
          <même poids fort> <    valeur 10    >
```

Il est plus facile de coder les valeurs précédentes en hexadécimal. Chaque demi-octet (quatre bits) représente alors un caractère hexadécimal. Les valeurs numériques que l'on veut directement déplacer dans une adresse de destination peuvent aussi être codées en hexadécimal à condition de les faire précéder d'un symbole dollar **($)**.

Contenu de D0 avant l'instruction : **ffffffff**

Valeur à transférer en hexadécimal : **a**

L'instruction peut aussi s'écrire : **move.w #$a,D0**

Le contenu du registre après l'instruction sera : FFFF000A

Il est possible aussi de transférer des valeurs sur le registre tout entier en le précisant dans le code instruction. Le "**move**" est alors suivi par le suffixe **l** (pour *long word* ou mot long).

L'instruction **move.l #$a,D0** va remplacer le mot long précédemment contenu dans D0 par la valeur numérique **$a** codée sur un mot long (4 octets ou 8 caractères hexadécimaux). Dans ce cas, tout le contenu du registre D0 sera remplacé : Valeur à transférer codée sur 4 octets : **0000000a**

Les zéros à gauche du a ne sont là que pour montrer comment opère l'assembleur dans sa logique.

Après l'instruction, le contenu du registre est **0000000a** en hexadécimal.

Il est aussi possible de transférer une valeur numérique octet dans le registre D0, à condition de le préciser dans le suffixe du code opération par un **b** (pour *byte* ou octet).



L'instruction **move.b #$a,D0** va transférer la valeur hexadécimale a codée sur un octet **(0a)** vers l'octet poids faible du registre **D0** sans modifier le reste du registre. Après l'instruction, le registre **D0** aura pour valeur ;

D0 ffffff0a

Tout ce qui a été fait sur le registre **D0** peut aussi l'être sur l'un quelconque des registres de données **(D0 à D7)** ou des registres d'adresses **(A0 à A7)**.

Le transfert d'une valeur numérique peut aussi s'effectuer suivant la même logique vers une adresse mémoire.

Les instructions suivantes :

```
move.b #$20,mem1
move.w #$20,mem1
move.l #$20,mem1
```

vont successivement transférer à l'adresse **mem1** la valeur hexadécimale 20 sur l'octet d'adresse **mem1**, la valeur **$20** codée sur un mot **(0020)** à l'adresse **mem1**, et la valeur **$20** codée sur un double mot ou mot long **(00000020)** à l'adresse **mem1**. Sur le 68000 comme sur tous les microprocesseurs, les adresses de mots ou mots longs doivent être paires sous peine de déclencher une erreur d'adresse impaire (3 bombes à l'écran). En supposant que **mem1** ait une valeur correcte et que son contenu ait été rempli par des 1 précédemment grâce à une instruction du type **move.l# $ffffffff,mem1** notons en regard de chaque instruction le contenu de **mem1** après le déroulement de l'instruction :

Instruction	Contenu de la mémoire à partir du mem1
	ffffffff
move.b #$20,mem1	20ffffff
move.w #$20,mem1	0020ffff
move.l #$20,mem1	00000020

Le petit programme suivant permet de recopier la valeur **$20** sur les quatre octets consécutifs à partir de l'adresse **mem1** :

```
move.l #$ffffffff,mem1          ;pour initialiser mem1
```

Instruction	Contenu des quatre octets à partir du mem1
	ffffffff
move.b #$20,mem1	20ffffff
move.b #$20,mem1+1	2020ffff
move.b #$20,mem1+2	202020ff
move.b #$20,mem1+3	20202020

Comme on s'en rend compte dans ce petit programme, si **mem1** est un nombre pair, **mem1+1** est un nombre impair. Comme l'opération s'effectue sur un octet (précisé par le **move.b**), l'adresse peut être impaire mais c'est le seul cas où cela est permis.

DEUXIÈME CAS : LA SOURCE EST LE CONTENU D'UN REGISTRE

Le registre concerné peut être indifféremment un registre de données ou un registre d'adresse. Suivant le suffixe qui suit le code **move**, la valeur à transférer sera un octet **(b)**, un mot **(w)** ou un double mot **(l)**.

La donnée est codée des bits poids fort aux bits poids faible en allant de gauche à droite dans le registre. Initialisons en premier lieu une valeur de registre, **a0** par exemple :

```
move.l    #$12345678,a0
```

La décomposition de ce codage est la suivante :

```
    12      34      56      78
mot poids fort        mot poids faible

octet poids fort          octet poids faible
```

Initialisons un autre registre qui nous servira de registre destination :

```
move.l   #$aaaaaaaa,a1
```

Les différentes possibilités de l'instruction **move a0** vers **a1** donneront les configurations suivantes dans le registre de destination **a1** :

Contenu des registres ;

a0	a1
$12345678	$aaaaaaaa
move.b a0,a1	$aaaaaa78
move.w a0,a1	$aaaa5678
move.l a0,a1	$12345678

De cette façon nous pouvons remarquer qu'on ne peut modifier séparément par l'instruction **move**, chacun des deux octets du mot poids fort du registre destination.

Dans le cas où l'adresse de destination est une adresse mémoire, le principe de transfert est toujours le même. La seule précaution à prendre sera de bien utiliser des adresses paires pour des transferts de mots ou de doubles-mots, cet alignement d'adresses n'étant pas à faire pour les transferts de registre à registre.

Le petit programme suivant s'amuse à effectuer quelques transferts de registres à mémoire et réciproquement. Les contenus situés à droite indiquent les changements qui interviennent à l'adresse concernée.



Instruction		Nouveau contenu de	
move.b	#$12,mem1	mem1	$12
move.w	#$34,d0	d0	$00000034
move.w	#$5678,d1	d1	$00005678
movea.l	#$3bcd,a1	a1	$00003bcd
move.l	#$ffffffff,d3	d3	$ffffffff
move.b	d1,d0	d0	$00000078
movea.w	a1,a0	a0	$00003bcd
move.w	a0,d3	d3	$ffffabcd
move.b	d3,d1	d1	$000056cd
move.l	a1,d2	d2	$0000abcd
move.b	d1,mem1	mem1	$cd
move.w	d1,mem1	mem1	$56cd
move.w	d2,mem1	mem1	$abcd
move.l	d0,d2	d2	$00000078
movea.w	d2,a1	a1	$00000078

Les contenus notés à partir de mem1 sont ceux concernés par l'instruction en cours, sans préjuger de la valeur des octets qui suivent.

Vous avez pu remarquer que lorsque l'adresse destination est un registre d'adresse, il y a automatiquement addition du suffixe a au code instruction. Certains assembleurs reconnaissent automatiquement les modes d'adressage :

move.w source,an au lieu de movea.w source,an

move.l source,an au lieu de movea.l source,an

La différence essentielle est que ce transfert ne peut s'effectuer sur des octets et que le signe de la valeur à transférer est étendu au mot long quelle que soit la taille du transfert ; nous verrons ce problème plus tard avec le codage des valeurs négatives et le positionnement des bits du registre code condition (CCR).

L'instruction :

movea.b a0,a1

ne sera donc pas acceptée quel que soit l'assembleur que vous utilisez.

TROISIÈME CAS : ADRESSAGE INDIRECT PAR UN REGISTRE D'ADRESSE

Dans les exemples précédents, rien ne distingue les registres données des registres adresses dans l'utilisation des transferts. Le gros intérêt des registres d'adresse est de pouvoir contenir eux-mêmes l'adresse d'un opérande. C'est le principe de l'adressage indirect, qui ne peut être réalisé que par des registres d'adresse. Cet adressage est utilisé lorsque l'adresse de la donnée se trouve quelque part dans la mémoire du système.

La saisie de la donnée à transférer se fait suivant les mêmes critères que dans les transferts précédents. Dans le cas où le transfert s'effectue sur des mots ou des doubles mots il faudra prendre soin que le contenu du registre adresse utilisé en adressage indirect contienne bien une valeur paire. Il faut aussi prendre garde à ce que l'adresse contenue dans le registre d'adresse ne soit pas une adresse réservée pour le système sous peine de déclencher une erreur de bus et voir apparaître deux bombes à l'écran pour tentative de violation de zone réservée sans passer en mode superviseur.

Petit programme d'exemple :

```
move.l   #$11111111,d1
move.l   #$00451256,a0
move.l   #$abcdef00,(a0)
move.b   (a0),d1
move.w   (a0),d1
move.l   (a0),d1
```

En raisonnant par analogie avec les exemples précédents, cela revient à dire que le sigle mem1 qui était utilisé est équivalent à la valeur $451256.

Etudions le petit programme instruction par instruction :

— la première instruction met la valeur mot long $451256 dans le registre a0 ;

— la seconde instruction transfère la valeur $abcdef00 en mot long à l'adresse contenue dans a0 soit en $451256. Ce transfert de valeur mot long peut s'effectuer sans problème car cette adresse est paire ;



— la troisième instruction transfère l'octet situé à l'adresse **$451256**, soit **$ab** sur l'octet poids faible du registre **d1** sans rien toucher aux autres bits de **d1**.

d1 contiendra alors la valeur : **$111111ab**

Les instructions suivantes effectuent les transferts de mot et de mot long suivant la même logique. Après les instructions, les contenus de **d1** seront donc respectivement :

$1111abcd

$abcdef00

Il faut remarquer que les octets sont notés en séquence de gauche à droite à partir de l'adresse indirecte ou l'adresse mémoire alors que cette classification se fait de droite à gauche dans les registres.

ADRESSAGE DIRECT PAR UN REGISTRE D'ADRESSE AVEC POST-INCRÉMENTATION OU PRÉDÉCRÉMENTATION

Pour bien comprendre l'adressage avec postincrémentation ou prédécrémentation revoyez l'article précédent traitant du fonctionnement des piles (Partie 4 chapitre 3.2.2). L'intérêt de ce type d'adressage en mode indirect réside dans les possibilités de transferts traitant de zones de mémoire consécutives. Ces zones correspondent aux champs, tableaux et chaînes de caractères suivant les langages de programmation.

Exemple :

La première instruction charge l'adresse de la zone source d'où démarre le transfert :

```
movea.l #$00404846,a0
```

La deuxième instruction charge l'adresse de départ de la zone destination du transfert :

```
movea.l #$00484850,a1
```

Si l'on veut faire un transfert du mot dont l'adresse est contenue dans **a0($404846)** à l'adresse contenue dans **a1 ($484850)**, il suffit d'écrire le transfert indirect :

```
move.w (a0),(a1)
```

Si l'on désire effectuer des transferts consécutifs, il faut que les contenus des registres **a0** et **a1** contiennent maintenant l'adresse du mot suivant, et d'utiliser la même instruction. L'incrémentation se fait automatiquement par le mode postincrémenté :

```
move.w (a0)+,(a1)+
```

Il suffit d'écrire autant de fois l'instruction que l'on désire de transferts.

Le programme suivant :

```
movea.l #$00404846,a0
movea.l #$00484850,a1
move.w  (a0)+,(a1)+
move.w  (a0)+,(a1)+
move.w  (a0)+,(a1)+
move.w  (a0)+,(a1)+
```

copie les quatre mots d'adresses $404846 à $40484c vers les quatre mots d'adresses $484850 à $484856.

Lorsque des transferts importants de zones de mémoire se font, il est bien sûr impensable de répéter l'écriture de la même instruction autant de fois qu'il y a de transferts. Nous verrons ultérieurement l'écriture d'une boucle de transfert avec tests de fin de boucle.

Les modes d'adressage postincrémenté et prédécrémenté peuvent être utilisés indifféremment dans les transferts de mémoire. En post-incrémentant, vous irez toujours vers les adresses croissantes et en pré-décrémentant vers les adresses décroissantes.

Le programme :

```
movea.l #$00404846,a0
movea.l #$00484850,a1
move.w  (a0)+,-(a1)
move.w  (a0)+,-(a1)
move.w  (a0)+,-(a1)
move.w  (a0)+,-(a1)
```

copie les quatre mots d'adresses $404846 à $40484c vers les quatre mots d'adresses $48484e à $484848.



Le programme :

```
movea.l #$00404846,a0
movea.l #$00484850,a1
move.w -(a0),(a1)+
move.w -(a0),(a1)+
move.w -(a0),(a1)+
move.w -(a0),(a1)+
```

copie les quatre mots d'adresses $404844 à $40483e vers les quatre mots d'adresses $484850 à $484856.

Le programme :

```
movea.l #$00404846,a0
movea.l #$00484850,a1
move.l -(a0),-(a1)
move.l -(a0),-(a1)
move.l -(a0),-(a1)
move.l -(a0),-(a1)
```

copie les quatre mots longs d'adresses $404842 à $404836 vers les quatre mots longs d'adresses $48484c à $484840.

Cependant, par une instruction du type move avec adressage indirect sur un registre d'adresse, on peut transférer directement des octets. L'instruction move.b(a0)+,(a1)+ est permise à la différence de movea.b a0,a1 ou move.b a0, a1.

Observez les transferts effectués par le petit programme suivant :

		adresse	contenu
movea.l	#$00404846,a0	a0	$00404846
movea.l	#$00484850,a1	a1	$00484850
movea.l	#$00484854,a2	a2	$00484854
move.l	#$1234,(a1)	$484850	$00001234
move.l	#$ffffffff,(a2)	$484854	$ffffffff
move.l	#$45ad,-(a0)	$404842	$000045ad
move.l	#$45ad,-(a0)	$40483e	$000045ad
move.l	#$1234,-(a1)	$48484c	$00001234
move.l	(a0)+,(a1)+	$48484c	$000045ad
move.l	(a0)+,(a1)+	$484850	$000045ad
move.l	(a1),-(a0)	$40483a	$ffffffff

Pour bien suivre les transferts, il est nécessaire de noter les contenus des registres d'adresse puisqu'ils varient sans arrêt dès qu'ils sont utilisés en adressage indirect postincrémenté ou prédécrémenté. Sous la colonne adresse vous trouverez la référence de l'adresse ou du registre dont le contenu est modifié par l'instruction. Le nouveau contenu est spécifié sous la colonne contenu.

Les contenus différents des registres **a0** et **a1** sont spécifiés ci-dessous :

Après les deux instructions prédécrémentées (instructions 6 et 7, le contenu de **a0** est diminué de la valeur 8 correspondant à deux doubles mots (notez que cette décrémentation se fait avant le transfert). Sa valeur passe donc de **$404846** à **$40483e**.

Avant d'effectuer le premier transfert prédécrémenté (instruction 8) le contenu de **a1** est diminué de 4 (valeur **$48484c**).

Après les deux instructions de transfert double mot postincrémentées (instructions 9 et 10), les contenus de **a1** et de **a0** sont augmentés chacun de la valeur 8(**a0** vaudra **$40483a** et **a1** **$484854**). Toutefois, le dernier transfert s'effectue vers l'adresse **$484850** du fait de la post-incrémentation utilisée dans l'adressage.



4/3.2.2

Le fonctionnement d'une pile

Les piles sont des structures de stockage de données du type LIFO *(Last In First Out)*. En français cela signifie que la dernière donnée entrée sera la première sortie. Pour gérer une telle structure sur un ordinateur, il faut disposer d'une zone mémoire intouchable et assez grande et d'un pointeur de pile. Sur le 68000, on dispose de deux types de piles complètement indépendantes : la pile système et la (ou les) pile(s) implantée(s) par l'utilisateur.

Le pointeur de la pile système est égal au contenu du registre **A7** noté **SP** *System Pile*. C'est pourquoi il faut se garder de toucher à ce registre sans l'avoir sauvegardé, sous peine de planter la machine à coup sûr. Si le bit **S** (mode superviseur) du registre d'état est à 1, la pile est la pile superviseur ; dans le cas contraire, la pile est la pile utilisateur.

La pile superviseur est utilisée par toutes les procédures d'exception ou d'interruption du système, et la pile utilisateur sera essentiellement utilisée par le programmeur au cours des appels de sous-programmes.

Voyons maintenant comment fonctionnent l'empilage et le dépilage des données.

EMPILAGE ET DÉPILAGE DE DONNÉES

Cette gestion de la pile se fait par l'intermédiaire des systèmes d'adressages pré-décrémentés et post-incrémentés. Il faut obligatoirement passer par un registre d'adresse en zone source ou destination dans une instruction du type **move**. L'adressage sera du type indirect, c'est-à-dire que le registre d'adresse contiendra l'adresse de la donnée source ou l'adresse de la destination.

Exemple :

move.w –(A0),mem

Cette instruction signifie :

— pré-décrémenter l'adresse contenue dans a0 de deux (instruction de mot) ;

— prendre le mot contenu à cette nouvelle adresse ;

— transférer ce mot à l'adresse **mem** (adresse paire obligatoirement).

Supposons l'état initial de **A0** comme suit en hexadécimal :

A0 0050a634

on pré-décrémente d'abord de 2 le contenu de **A0**, **A0** devient **0050a632**.

On prend le contenu du mot (2 octets) situé à l'adresse **0050a632** et on le transfère à l'adresse **mem**.

Remarque : il fallait que l'adresse contenue dans A0 au départ soit paire pour pouvoir ensuite faire un transfert de mot. Dans le cas contraire, vous généreriez trois bombes d'erreur d'adresse à l'écran !

La valeur de la décrémentation dépend de la taille de la donnée :

Si le registre d'adresse est l'un des registres A0-A6 la pré-décrémentation sera de 1,2,4 suivant que la donnée est un octet, un mot ou mot long.

Si le registre d'adresse est le registre A7, la valeur de la pré-décrémentation sera de deux pour les données octet ou mot et de quatre pour les données mot long.

Le principe de l'adressage post-incrémenté est le même, mais l'incrémentation s'effectue après le transfert :

move.w(A0)+,mem

Cette instruction signifie :

— prendre le mot dont l'adresse est contenue dans A0 ;

— transférer le mot à l'adresse mem ;

— incrémenter A0 de deux.

Si les valeurs initiales de A0 sont les mêmes que dans l'exemple précédent, le contenu du mot situé à l'adresse 0050a634 est transféré à l'adresse mem, et la nouvelle valeur du registre A0 devient 0050a636.

Les piles sont toujours utilisées par les programmeurs avec empilage des données dans les sens des adresses décroissantes.

L'instruction qui permet de remonter ainsi la pile vers les adresses basses est une prédécrémentation du pointeur de pile.

Après chaque exécution, le pointeur indique donc le sommet de la pile, c'est-à-dire la dernière donnée entrée. Le dépilage se fera donc dans le mode post-incrémenté afin de faire le chemin inverse pour les adresses.

Exemple :

Supposons que le pointeur de pile soit le registre A6. La pile est représentée mot par mot (4 caractères hexadécimaux par mot).

Pile au départ :

```
    libre                                   Adresses basses
    libre
    libre
(A6) -> sommet de la pile
    valeur précédente=$4545
    ...
    ...
    fond de la pile                         Adresses hautes
```

Regardons l'effet sur la pile des instructions suivantes, en pas à pas :

— Instruction **move.w #$1230,-(A6)**



```
Nouvel état de la pile :
        libre                                   Adresses basses
        libre
(A6) -> 1230
        premier sommet de la pile
        4545
        ...
        ...
        fond de la pile                         Adresses hautes
```

— Nouvelle instruction : move.l #$789a,-(sp)

```
Nouvel état de la pile :
(A6) -> 789a                                    Adresses basses
        0000
        1230
        premier sommet de la pile
        4545
        ...
        ...
        fond de la pile                         Adresses hautes
```

— Nouvelle instruction : add #6,A6

```
Nouvel état de la pile :
        libre                                   Adresses basses
        libre
        libre
(A6) -> sommet de la pile retrouvé
        4545
        ...
        ...
        fond de la pile                         Adresses hautes
```

La dernière instruction a simplement permis de récupérer le pointeur de pile sur sa position de départ.

Après une opération de transfert sur un mot puis sur un double mot, il suffit donc de rajouter le nombre d'octets correspondants (ici 6) pour annuler le nombre d'octets déduits au cours de la pré-décrémentation. Si le programme le nécessite, vous pouvez bien sûr récupérer aussi cette adresse de départ avec une suite d'instructions move avec adressage post-incrémenté sur A6 :

```
move.w (A6)+,mem
move.l (A6)+,mem
```

Vous pouvez aussi utiliser l'adressage post-incrémenté sur A6 en zone destination, cela dépend du traitement que vous devez effectuer dans votre programme. Le tout est, dans ce cas précis, de faire une post-incrémentation mot long et une post-incrémentation mot (l'ordre importe peu) pour rajouter 6 à la valeur de A6 et récupérer la valeur de départ. Dans l'exemple donné ci-dessus, il était supposé qu'on doive simplement récupérer la valeur de départ sans effectuer aucun traitement, ce qui explique la simple addition de six au contenu de A6.

4/4

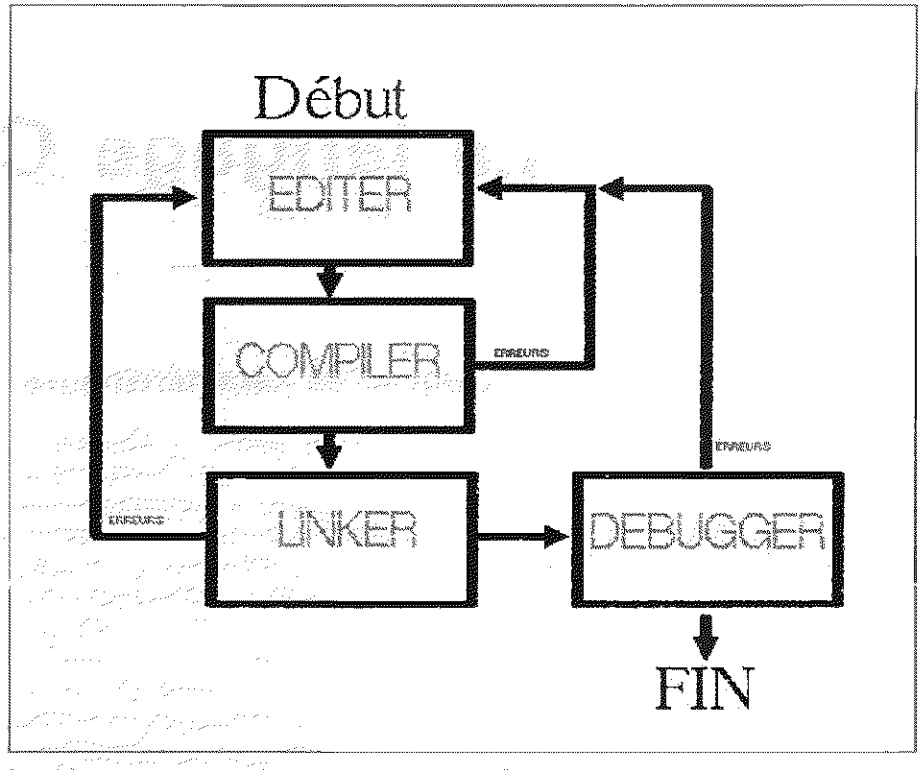
Le langage C

I - Origines et caractéristiques

Le langage C a été inventé au début des années 70 par Dennis Ritchie et Brian Kernigan. En effet, l'écriture d'un nouveau système d'exploitation (UNIX) nécessitait la conception d'un langage adapté. A cette époque, il n'existait aucun langage entre l'assembleur et des langages de haut niveau comme le Fortran ou le Cobol. Entre les délais de développement trop élevés de l'assembleur et les performances très ciblées des langages de haut niveau, il n'existait pas de langage orienté système. Le langage C a donc été conçu dans un souci de clarté et de simplicité, tout en conservant la puissance de l'assembleur pour l'accès aux niveaux les plus profonds de la machine. Il possède peu de mots clés et aucune instruction de gestion des Entrées/Sorties, telles que la gestion de fichiers ou l'affichage à l'écran. Mais l'usage systématique des librairies permet de pallier cet inconvénient.

Contrairement au Basic qui se présente habituellement sous la forme d'un interpréteur, le langage C est de type « compilé ». Bien que la société française d'édition LORICIEL édite un interpréteur C, la plupart des développeurs C sur Atari ST utilisent l'habituel cycle *Edition Compilation Linkage*.

La procédure de développement d'une application se déroule généralement de la façon suivante :



Le développement d'un programme en C.

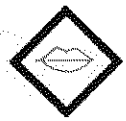
a) *Edition du fichier source* avec l'éditeur, qui est généralement fourni avec votre compilateur C ou n'importe lequel des éditeurs de textes existants. Ce fichier contient le programme sous forme d'un texte ASCII.

b) *Compilation* de ce même *fichier source* à l'aide du compilateur C (une description des différents compilateurs existants est donnée plus loin. Si le compilateur détecte une ou plusieurs erreurs, retourner à la phase Edition pour corriger ces erreurs.

c) *Linker* (édition de liens) le résultat de la compilation que l'on appelle *fichier objet* avec les *librairies*.

d) *Débugger* le programme. En cas d'erreurs, retourner à la phase Edition pour les corriger.

La seconde différence par rapport au Basic concerne la « modularité » du langage C. En effet, un programme Basic est écrit d'un seul bloc. Il n'y a donc qu'un seul fichier source. Le C permet pareillement une telle approche mais elle n'est pas vraiment conseillée. Il y a pour cela une excellente raison. Supposons que la taille de votre source C soit de 4 000 lignes, ce qui est assez important. La moindre modification entraînera l'édition de ces 4 000 lignes. Essayez de retrouver rapidement une



erreur parmi 4 000 lignes. Ensuite, il faut recompiler l'ensemble du fichier source. En fonction du compilateur utilisé, cette méthode pourra prendre entre 10 minutes et 1 heure.

L'approche conseillée pour le langage C consiste à diviser le programme en modules spécialisés. Vous pouvez avoir un module qui s'occupe du graphique, un autre pour le son, un autre pour les traitements de chaînes, etc. Cette façon de faire vous permettra de repérer plus facilement le module où se fera une modification et de ne recompiler que ce module. De plus, cela permet de débugger le programme module par module. Une modification, selon le compilateur utilisé, ne prendra pas plus de 10 minutes dans le pire des cas et 2 minutes pour un compilateur rapide.

Afficher un texte à l'écran

COMMENÇONS A PROGRAMMER EN C

Etudions maintenant l'aspect d'un programme écrit en langage C. Selon la coutume, prenons l'éternel exemple **PRINT BONJOUR** du Basic. Voyons ce que cela donne en langage C.

```
#include <stdio.h>
main( )
{ printf("BONJOUR\n");
}
```

Essayons d'analyser cet exemple.

• La première ligne contient une instruction un peu particulière du C : **#include <stdio.h>**, qui permet d'inclure un fichier texte au fichier source. Le compilateur, quand il arrivera sur cette ligne, la remplacera par le contenu du fichier dont le nom est donné entre <...>. Dans notre cas, il s'agit d'indiquer au compilateur que **printf** est une fonction de la librairie et de préciser ses paramètres. Le fichier **stdio.h** contient une liste de définitions dédiées aux Entrées/Sorties.

• Ensuite, nous arrivons au programme. Le **main( )** indique où se trouve le début du programme. L'accolade ouvrante « { » indique le début d'un bloc. Ici, le bloc constitue l'ensemble du programme. Mais nous verrons plus tard que cela n'est pas restrictif. L'accolade fermante « } » représente donc logiquement la fin d'un bloc. Ce bloc est constitué d'une unique instruction **printf("BONJOUR \ n");**.

• La fonction **printf** : elle représente, en langage C, le moyen d'afficher un texte à l'écran. Nous avons dit plus haut que le langage C ne possédait pas d'instructions d'Entrées/Sorties. Cela est vrai, mais les bibliothèques de fonctions offrent le moyen d'étendre les possibilités du langage. L'utilisation de ces bibliothèques est un grand avantage dans le sens où elle permet de redéfinir une fonction. Il est ainsi possible de se créer une fonction d'affichage personnalisée ou adaptée à des besoins spécifiques.

Dans le cas du **printf**, il s'agit d'une fonction standard du C et extrêmement puissante, chargée de l'affichage de données à l'écran. Dans l'exemple ci-dessus, on remarque que le texte à afficher est suivi des

deux caractères **\n**. Ils indiquent que le texte devra être affiché et suivi d'un retour à la ligne. Sans cette information, le prochain appel à cette fonction afficherait le texte juste derrière le **"BONJOUR"**.
Dernière information concernant cet exemple de **printf** : une chaîne de caractère est toujours encadrée par des guillemets. Vous avez dû remarquer que **printf** était suivi d'un point virgule. C'est le *séparateur* du langage C. Il permet d'indiquer la fin de l'instruction.

Il est tout à fait possible de mettre plusieurs instructions :

```
#include <stdio.h>
main( )
{ printf("BONJOUR,#"); printf("MONSIEUR");
  printf("\n");
  printf("IL FAIT BEAU CE MATIN");
}
```

Cet exemple affichera à l'écran :

```
BONJOUR, MONSIEUR
IL FAIT BEAU CE MATIN
```

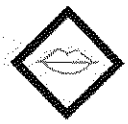
Réaliser des calculs complexes

UN EXEMPLE PLUS DIFFICILE

Le langage C ne sert pas simplement à afficher des textes à l'écran. Il est capable de réaliser des calculs plus ou moins complexes et possède toutes les structures de programmation indispensables. Un des aspects du C, qui rebute certains, concerne les variables. Une variable doit être déclarée avant son utilisation. Voici un exemple de programme qui affiche la table des carrés des 20 premiers nombres par pas de 0.5. Nous y verrons, par la même occasion, une des facilités que nous procure le langage C pour gérer les boucles.

```
#include <stdio.h>
#define LIMIT 10.0
float Square(Valeur)
float Valeur;
{ return Valeur*Valeur;
}
main( )
{ float Carre, Nombre;
  Nombre=0.0
  while (Nombre < = LIMIT)
  { Carre = Square(Nombre);
    printf("Le carre de %2.1f est %3.2f \n",Nombre,Carre);
    Nombre + = 0,5;
  }
}
```

Cet exemple doit vous paraître plus compliqué. C'est bien normal car il fait appel à de nouvelles notions.



• Nous avons déjà vu l'instruction **#include**. Le **#define** qui suit est de la même famille mais permet, quant à lui, de définir des constantes. Le compilateur, lors d'un premier passage, va remplacer tous les **LIMIT** qu'il rencontrera par la chaîne **10.0**. Nous verrons plus tard qu'il est également possible de définir des paramètres au **#define**.

• Nous avons ensuite une définition de fonction. Une fonction est un sous-programme à qui l'on peut passer des paramètres et qui peut renvoyer un résultat. Dans le cas présent, il s'agit de la fonction **Square**. Pour déclarer une fonction, il faut indiquer le type du résultat qui sera renvoyé. Cela peut être **float** (virgule flottante) ou n'importe lequel des types existants en C. Ensuite, on a le nom de la fonction suivi de la liste des paramètres entre parenthèses. Dans notre cas, nous avons :

```
float Square(Valeur)
float Valeur;
```

La fonction s'appelle **Square**, prend un seul paramètre de type float et renvoie un résultat de type float. Les types de paramètres doivent être définis avant le début de bloc définissant la procédure :

```
{ return Valeur*Valeur;
}
```

Nous voyons que cette fonction ne fait que renvoyer le carré du paramètre.

• Après la procédure, nous arrivons au **main** qui situe le début du programme. Voyons ce qui suit :

```
float Carre, Nombre;
```

Il s'agit de la déclaration de deux variables, le mot **float** indique que ces deux variables sont des variables réelles en virgule flottante, simple précision. Le nom de chaque variable est donc juste après l'indication de type. La fin de la déclaration est indiquée par le « ; ». Attention ! Contrairement aux paramètres d'une procédure qui sont déclarés avant le début de bloc, les variables locales sont déclarées après. On peut noter qu'il aurait été possible de séparer les deux variables et d'écrire ainsi :

```
float Carre;
float Nombre;
```

La ligne suivante initialise la variable Nombre à la valeur zéro :

```
Nombre = 0.0;
```

Nous arrivons maintenant à la boucle qui commence donc par :

```
while (Nombre <= 10.0)
```

Cela indique que le bloc, ou l'instruction, qui suit sera exécuté tant que la variable Nombre sera inférieure ou égale à la valeur 10. Attention ! Cette instruction ne se termine pas par un ;. Si c'était le cas, elle serait considérée comme une instruction nulle et le bloc suivant ne serait pas associé au **while**.

Le bloc suivant commence aussi par { et finit par }. La première ligne du bloc est la suivante :

```
Carre = Square(Nombre);
```

La variable **Carre** est évaluée comme étant égale au résultat de l'appel de la procédure **Square** avec comme paramètre Nombre.

```
printf("Le carre de %2.1f est %3.2 \n",Nombre,Carre);
```

Cet exemple est assez représentatif de la puissance de la fonction **printf**. Le programme appelle la fonction **printf**, qui est donc contenue dans la librairie standard du C, avec 3 paramètres. Le premier constitue ce que l'on appelle la *chaîne de formatage*. Elle permet d'indiquer le type d'affichage que l'on désire. Dans notre exemple, on désire que cette chaîne soit affichée, mais que le **%2.1f** soit remplacé par le contenu de la variable **Nombre** qui est une variable flottante et qu'elle soit affichée sur 2 caractères avec 1 chiffre après la virgule.

Le **%3.2f** concerne la variable **Carre** que l'on veut afficher sur 3 caractères plus 2 chiffres après la virgule. Le **\n** provoquera un retour chariot. Comme vous le voyez, cette fonction est extrêmement puissante et nous verrons plus tard l'ensemble de ses possibilités.

```
Nombre + =0.5;
```

Cette dernière ligne permet d'ajouter 0.5 à la variable Nombre. Il est tout à fait possible d'écrire cette instruction de la même manière que dans un autre langage tel que le Basic :

```
Nombre = Nombre + 0.5;
```

• Voici maintenant les premières lignes qui seront affichées par ce programme exemple :

```
Le carre de 0.0 est 0.00
Le carre de 0.5 est 0.25
Le carre de 1.0 est 1.00
Le carre de 1.5 est 2.25
Le carre de 2.0 est 4.00
...
```

L'étude de ces exemples simples vous ayant permis de mieux comprendre ce qu'est le langage C, nous allons maintenant étudier plus précisément la syntaxe de ce langage.

II - Les types de données

Le langage C reconnaît plusieurs types de données. Nous avons déjà vu le type de données à virgule flottante. Mais il existe d'autres types plus ou moins complexes et qui permettent d'envisager un grand nombre d'applications.

LES VARIABLES ENTIÈRES

Ce type de données est le plus simple sur lequel on puisse travailler. Dans le cas de l'Atari, il est de 8, 16 ou 32 bits. C'est celui que connaît le microprocesseur.



char correspond à un octet (8 bits) ; sa valeur est comprise entre −128 et +127. Il est généralement utilisé pour les chaînes de caractères.

short et **int** sont des entiers codés sur 16 bits et permettent la manipulation de nombres compris entre −32768 et +32767.

long est codé sur 32 bits et donne accès aux variables comprises entre −(2^31) et 2^31−1 (−(2^{31}) et 2^{30}).

Ces types sont signés par défaut mais peuvent être déclarés comme non signés **(unsigned)**. La déclaration de type sera comme suit :

```
unsigned char c;
unsigned short Index;
unsigned int Counter;
unsigned long Time;
```

LES VARIABLES FLOTTANTES

Comme nous l'avons vu dans l'exemple précédent, le langage C permet également de manipuler les nombres à virgule flottante. Ce type peut être en « simple précision » et « double précision ». Les caractéristiques sont généralement spécifiques à un compilateur donné. Les mots clés sont :

float pour la simple précision,
double pour la double précision.

LES CONSTANTES

Ce n'est pas à proprement parler un type de données mais il nous a semblé approprié de le citer dans cette partie. Dans une affectation comme :

`A=0;` A est une variable et **0** une constante.

Les constantes entières longues sont suivies d'un L majuscule ou minuscule :

`B=1L ;` ou `B=1l ;`

Les constantes flottantes peuvent être dans les formats suivants :

`Theta=1.55;` ou `Rho=1.2E-10;`

LES TABLEAUX

Un tableau est un ensemble de variables du même type. On peut, par exemple, avoir un tableau de 100 entiers :

```
int Tableau[100] ;
```

Il s'agit là d'un tableau à 1 dimension. Mais on peut avoir plus de dimensions. Voyons l'exemple d'un tableau à 3 dimensions :

```
int Cube[10][10][10];
```

Pratiquement, en langage C, on emploie le plus souvent les tableaux de caractères. Cela permet de manipuler des chaînes de caractères. Il est tout à fait possible de déclarer et initialiser une chaîne de caractères.

```
char Texte[ ]="BONJOUR";
printf(Texte);
```

Ici, nous définissons un tableau de caractères qui sera initialisé avec la chaîne "**BONJOUR**". Vous remarquerez que la taille a été omise. En effet, le compilateur la déduira automatiquement du texte qui suit. De plus, l'appel à la fonction **printf** ne nécessite que l'indication du nom du tableau. Cela permet d'indiquer l'adresse du tableau.

Texte[0] renvoie le premier caractère de la chaîne Texte mais **Texte** renvoie l'adresse de cette même chaîne.

III - Les expressions, variables et opérateurs

LES VARIABLES

D'une manière générale, la plupart des compilateurs limitent les noms de variables à 256 caractères mais la phase d'édition de liens ne reconnaît généralement que les 8 premiers. Cela tient au format du fichier objet utilisé. Prenez l'habitude de choisir des noms dont les 8 premières lettres soient différentes. Par exemple, les deux variables **Index_de_debut** et **Index_de_fin** risquent d'être acceptées par le compilateur comme étant différentes, mais elles peuvent poser des problèmes à l'édition de liens (Link).

LES DÉCLARATIONS

Une variable doit obligatoirement être déclarée avant son utilisation. Lors de la déclaration, on indique le type, le ou les noms et éventuellement une valeur d'initialisation :

```
int Index=1,Valeur;
```

Dans cet exemple, deux variables entières sont déclarées sous les noms **Index** et **Valeur.** Index est initialisé à 1 mais Valeur ne l'est pas.

Nous allons maintenant aborder la notion d'*espace de validité*. Il est possible de définir des variables externes. Cela permet d'accéder à une variable qui est définie dans un autre module ou même dans une librairie :

```
extern long Compteur;
```

Cette déclaration indique que la variable **Compteur** n'est pas dans le module mais ailleurs. C'est la phase d'édition de liens qui se chargera de trouver où. La plupart des variables locales sont dites « automatiques ». Cela signifie que non seulement la variable est locale à une procédure, mais elle a également une durée de vie limitée à l'exécution de la procédure. L'emplacement mémoire de cette variable est alloué lors de



l'entrée dans la procédure et il est libéré à la sortie. Par opposition aux variables automatiques, le langage C autorise les variables statiques qui possèdent une place fixe en mémoire.

Voyons en premier le cas d'un variable automatique :

```
#include <stdio.h>
int proc( )                    /* Procédure sans paramètres      */
{ int Variable=0;              /* Initialise Variable à 0        */
  Variable++;                  /* Incrémente Variable            */
  return Variable;             /* Renvoie Variable en résultat   */
}
main( )
{ printf("%d",proc( ));        /* Affiche le résultat du 1er appel */
  printf("%d",proc( ));        /* Affiche le résultat du 2e appel */
```

L'exécution de ce programme affichera : **1 1**. En faisant précéder la définition de **Variable** par **static**, nous aurons un résultat complètement différent : **1 2**. L'explication en est très simple. Dans le premier cas, l'initialisation de Variable est réalisée lors de l'entrée dans la procédure. Tandis que dans le 2e cas, ce n'est fait qu'une fois à la compilation du programme.

Revenons sur les variables automatiques. Il existe un moyen d'indiquer au compilateur qu'une variable doit être stockée dans un registre. Cela est particulièrement intéressant pour des questions de taille et de vitesse d'exécution. Il suffit de faire précéder la déclaration de la variable par register :

```
register int i;
```

Par opposition, il est possible d'indiquer de ne pas utiliser de registre avec **auto**. Notons qu'il est impossible d'avoir une variable qui soit static et register.

LES OPÉRATEURS

a) Les opérateurs de multiplication

expression1 * expression2 : multiplication de deux expressions.

expression1 / expression2 : division de deux expressions.

expression1 % expression2 : reste de la division.

b) Les opérateurs d'addition

expression1 + expression2 : addition de deux expressions.

expression1 − expression2 : différence de deux expressions.

c) Les opérateurs de décalage

expression1 < < expression2 : décale expression1 à gauche de expression2 fois.

expression1 > > expression2 : décale expression1 à droite de expression2 fois.

d) Les opérateurs de relation

expression1 < expression2 : teste si expression1 est plus petit que expression2.

expression1 > expression2 : teste si expression1 est plus grand que expression2.

expression1 < = expression2 : teste si expression1 est inférieur ou égal à expression2.

expression1 > = expression2 : teste si expression1 est supérieur ou égal à expression2.

expression1 = = expression2 : teste l'égalité de deux expressions.

expression1 != expression2 : teste la non égalité de deux expressions.

e) Les opérateurs logiques

expression1 && expression2 : renvoie 0 si les deux expressions sont nulles et 1 si elles sont toutes deux non nulles.

expression1 ¦ ¦ expression2 : renvoie 1 si au moins l'une des deux expressions n'est pas nulle.

f) Les opérateurs binaires

expression1 & expression2 : effectue un ET binaire entre deux expressions.

expression1 ¦ expression2 : effectue un OU INCLUSIF binaire entre deux expressions.

expression1 ^ expression2 : effectue un OU EXCLUSIF binaire entre deux expressions.

g) Les opérateurs d'affectation

Lvalue = expression ; affecte le résultat de l'expression à la variable indiquée par **Lvalue**. La terminologie du langage C utilise ce terme **Lvalue** pour dénommer l'entité qui est l'objet de cette affectation.

Tous les opérateurs que nous avons déjà vu auparavant peuvent être associés à l'opérateur d'affectation.

Lvalue + = expression ; ajoute expression à Lvalue.



Lvalue − = expression	; soustrait expression à Lvalue.
Lvalue * = expression	; multiplie Lvalue par expression et stocke le résultat dans Lvalue.
Lvalue / = expression	; divise Lvalue par expression et stocke le résultat dans Lvalue.
Lvalue % = expression	; divise Lvalue par expression et stocke le reste de la division dans Lvalue.
Lvalue > > = expression	; décale Lvalue à droite de expression unités et stocke le résultat dans Lvalue.
Lvalue < < expression	; décale Lvalue à gauche de expression unités et stocke le résultat dans Lvalue.
Lvalue & = expression	; effectue un ET entre Lvalue et expression et stocke le résultat dans Lvalue.
Lvalue | = expression	; effectue un OU INCLUSIF entre Lvalue et expression et stocke le résultat dans Lvalue.
Lvalue ^ = expression	; effectue un OU EXCLUSIF entre Lvalue et expression et stocke le résultat dans Lvalue.

h) Les opérateurs unaires

*** expression**	renvoie le contenu de la variable pointée par expression. Expression doit bien sûr être un pointeur.
& Lvalue	renvoie l'adresse de Lvalue.
− expression	renvoie la valeur négative de expression.
! expression	renvoie 0 si expression est nulle et 1 dans le cas contraire.
~ expression	renvoie le complément à 1 de expression.
+ + Lvalue	incrémente Lvalue et renvoie le résultat.
Lvalue + +	renvoie Lvalue et incrémente Lvalue.
− − Lvalue	décrémente Lvalue et renvoie le résultat.
Lvalue − −	renvoie Lvalue et décrémente Lvalue.
(type de donnée)expression	convertit expression en un autre type. Par exemple, vous désirez utiliser une variable de type long comme un pointeur de type caractère, puis effacer le premier caractère : ***(char *)P = 0;**
sizeof expression	calcule la taille, en octets, occupée par expression.

sizeof(type de donnée) calcule la taille occupée par un type. L'exemple classique est celui d'une structure dont on désire connaître la taille en octets pour pouvoir allouer la place nécessaire.

IV - La structure des programmes

FORMAT TYPE D'UN PROGRAMME C

D'une manière générale, un source C sera constitué ainsi :

— Inclusion des fichiers « include ». Cela dépendra du programme.

— Définition des constantes.

— Définition des variables globales. Ces variables pourront être atteintes depuis toutes les procédures ou même d'un autre module.

— Procédures. Il est préférable de travailler du niveau le plus bas vers le niveau le plus haut. Ecrire les routines de base au début du source et monter de niveau. La dernière procédure doit être le **main( )**.

INSTRUCTIONS DE CONTROLE EN LANGAGE C

a) Les tests conditionnels

if... L'instruction **if (condition) Instruction1;** permet d'exécuter une action Instruction1 si et seulement si la Condition est satisfaite.

Exemple :

```
if (A==1 && Square(X)>5) printf("C'est bon \n");
```

Si la variable est égale à 1 et que l'appel de la procédure Square avec le paramètre X renvoie un résultat supérieur à 5, on affiche la chaîne « C'est bon ».

if (Condition) Instruction1; else Instruction2; permet d'exécuter l'instruction1 si la Condition est satisfaite et d'exécuter Instruction2 dans le cas contraire. Il faut noter que l'action peut parfaitement être un bloc :

```
if (X>Y)
{ T=X;
  X=Y;
  Y=T;
}
else
  printf("%d est inférieur ou égal à %d \n",X,Y);
```

La dernière variante permet d'effectuer une succession de tests. Il s'agit du **else if** :

```
if (A<B)
   printf("%d est plus petit que %d \n",A,B);
else if (A==B)
```



```
  printf("%d est égal à %d \n",A,B);
else printf("%d est plus grand que %d \n",A,B);
```

L'opérateur conditionnel

Le langage C offre également la possibilité de réaliser des affectations conditionnelles. Prenons un exemple courant en programmation : il arrive souvent que l'on ait besoin de tester une variable et faire en sorte qu'elle n'excède pas certaines limites. En utilisant le if, cela pourrait ressembler à quelque chose comme ceci :

```
if (X<Xmin) X=Xmin;
```

Le C offre l'opérateur conditionnel avec lequel l'ensemble précédent est écrit ainsi :

```
X = (X<Xmin ? Xmin : X);
```

Nous voyons que le côté gauche de cette instruction comporte trois parties. La première contient la condition, la seconde contient la valeur ou l'expression d'affectation si le test est vrai, la dernière concerne un résultat négatif du test. Il est possible de combiner deux tests dans une affectation conditionnelle. Ainsi,

```
if (X<Xmin) X=Xmin;
else if (X>Xmax) X=Xmax;
```

devient

```
X=(X<Xmin?Xmin:(X>Xmax?Xmax:X);
```

Bien sûr, cela peut paraître moins clair mais ce n'est qu'une question d'habitude.

switch

L'instruction **switch** est une instruction de décision à plusieurs cas. Elle teste si une expression est égale à une valeur parmi un ensemble de constantes. Dans ce cas, un branchement est effectué en conséquence. Voyons la syntaxe :

```
switch (c) {
case 0 :
     ...;
     break;
case 5 :
     ...;
     break;
default :
     ...;
     break ;
}
```

Ici, la variable c est comparée avec la valeur 0 et la valeur 5. Dans chaque cas, un traitement approprié est effectué. De plus, un cas par défaut est également présent. Notons que chaque cas se termine par l'instruction **break** qui permet de sortir de la structure switch. Sans sa présence, les instructions seraient exécutées séquentiellement jusqu'à la fin du switch ou la rencontre d'un break.

b) Les boucles

Une boucle permet d'exécuter répétitivement une action et de moduler cette action. Le C offre plusieurs instructions pour y parvenir.

for

L'instruction **for** permet d'exécuter un bloc d'instruction en partant d'une situation initiale et de n'arrêter que lorsqu'une condition est satisfaite. Voyons la syntaxe :

```
for (Initialisation;Condition d'exécution;opération)
{ ...
}
```

Prenons un exemple précis : une boucle sur une variable i entre 0 et 10 inclus par pas de 1.

```
for (i=0; i<=10; i++)
```

Incrémentation de la variable i

La boucle est effectuée tant que i<=10

La variable i début à zéro

Comme vous le voyez, c'est bien classique. Maintenant, précisons les particularités de cette fonction. On peut avoir plusieurs initialisations, il suffit de les séparer par une virgule. Le test d'exécution de la boucle peut être très complexe et dépendre d'appel à des procédures et bien plus. La dernière partie, qui fait évoluer la boucle, n'est pas restrictive à une incrémentation ou une décrémentation mais permet également l'appel à une ou plusieurs fonctions.

```
for (i=0,j=Fonc1( );i<Fonc2(j)&&i>=0;i*=2,j=Fonc3(i))
{ ...
}
```

Cet exemple n'a aucun autre but que de vous monter la puissance du **for**. Ici, nous initialisons i à 0 et j est égal au résultat de l'appel de la procédure **Fonc1( )**. La boucle sera exécutée tant que i sera inférieur au résultat de **Fonc2(j)** et que i sera positif ou nul. L'évolution de la boucle étant une multiplication de i par 2 et j est égal au résultat de **Fonc3(i)**.

while

while est bien plus simple puisqu'elle ne dépend que d'une condition d'exécution de la boucle. On peut considérer que c'est, en fait, une instruction **for** avec une condition unique d'exécution.

while (i<10) est équivalent à **for (;i<10;)**.

do while

C'est une variante du while dans le sens où la boucle est exécutée au moins une fois. Dans le cas du while, le test de condition est réalisé avant l'exécution du bloc d'instructions. Dans le cas du do while, le test est réalisé à la fin. Par conséquent, le bloc sera exécuté au moins une fois.

```
do {
    ....;       /* Bloc d'instructions */
} while (i<10);
```



break Nous avons déjà aperçu cette instruction dans l'instruction switch. Elle permet de sortir d'une instruction while, do while ou switch.

continue Cette instruction est utilisée dans les instructions de gestion de boucles. Elle permet de sauter directement sur la fin de la boucle et d'éviter ainsi l'exécution d'un bloc d'instructions.

goto Elle permet d'effectuer un branchement sur une étiquette. Une étiquette se compose d'un nom suivi d'un double point. N'importe quelle instruction peut être précédée d'une étiquette.

V - Les pointeurs et les tableaux

Nous abordons maintenant un domaine où le langage C est extrêmement performant. Un pointeur est une variable qui... pointe sur une autre variable. D'une manière générale, un pointeur contient l'adresse d'une autre variable. On peut donc accéder à une variable directement, par exemple A=1, ou indirectement par l'intermédiaire de pointeurs. L'exemple de pointeurs le plus utilisé en langage C concerne les chaînes de caractères que nous avons déjà définies comme étant des tableaux de caractères. Il existe deux opérateurs spécifiques à la gestion de pointeurs :

— **&** renvoie l'adresse d'une variable.

Dans l'affectation **p=&a;** nous affectons au pointeur p l'adresse de la variable a. On dit, dans ce cas, que p pointe sur a.

— * : Cet autre opérateur permet l'accès à une variable par l'intermédiaire d'un pointeur. Dans ce cas, une affectation à la variable pointée par p s'écrit ***p=5** ;

Voyons maintenant quelques exemples supplémentaires de l'utilisation d'un pointeur.

```
p=&A;      /* p pointe sur la variable A                    */
*p=0;      /* efface la variable pointée par p              */
*p++=0;    /* efface la variable pointée par p et incrémente p */
(*p)++;    /* incrémente la variable pointée par p          */
```

Voyons comment déclarer une variable de type pointeur.

char *p; Cette ligne permet de déclarer p comme étant une variable pointant sur une autre variable de type caractère.

long *q; Dans ce cas, le type de variable pointée est de type long.

p++ et **q++** auront un comportement totalement différent car dans le premier cas, la variable est de type caractère et l'incrémentation fera pointer p sur le prochain caractère tandis que le second cas fera pointer q sur le prochain long. Un octet plus loin dans le premier cas et quatre dans le second.

Voyons maintenant, comme exemple, une fonction très utile permettant d'échanger le contenu de deux variables de type entier.

```
swap(px,py)
int *px, *py;
{ int temp;      /* Variable temporaire */
  temp=*px;
  *px=*py;
  *py=temp;
}

main( )
{ int A,B;
  A=5
  B=10;
  printf("Avant l'appel de swap A = %d, B = %d \n",A,B);
  swap(&A,&B);
  printf("Après l'appel de swap A = %d, B = %d \n",A,B);
}
```

Examinons maintenant un autre exemple d'utilisation des pointeurs. Nous avons vu précédemment que les pointeurs s'utilisaient le plus souvent lors de la manipulation de chaînes de caractères. En effet, une chaîne de caractères est constituée comme un tableau de caractères. Examinons d'un peu plus près ces tableaux.

```
char Texte[ ]="BONJOUR";
```

Comme nous l'avons déjà vu, **&Texte[0]** renvoie l'adresse du premier caractère de la chaîne et par conséquent l'adresse de la chaîne. Nous savons que Texte renvoie également cette adresse. D'une manière générale, et pas seulement dans le cas d'un tableau de caractères, **Tableau+i** est équivalent à **&Tableau[i]**. Vous devez vous demander ce qui peut bien se passer pour un tableau à plusieurs dimensions. C'est tout à fait identique. Examinons le cas d'un tableau d'entiers à 2 dimensions.

```
int Cube[10][10];
```

Cube[0][0] représente le premier élément du tableau.

&Cube[0][0] donne l'adresse du premier élément et peut s'écrire **Cube[0]** ou tout simplement **Cube**. Mais **Cube[3]** représente l'adresse de la 3e ligne du tableau et est bien sur équivalent à **&Cube[3][0]**.

Revenons à nos chaînes de caractères avec quelques procédures simples de manipulations de chaînes de caractères.

```
strcpy(Source,Destination)
char *Source;
char *Destination;
{while (*Destination++ =*Source++) ;
}
```

Malgré sa taille très réduite, cette routine met en œuvre plusieurs concepts très puissants du langage C. Toute d'abord, elle reçoit deux pointeurs sur des caractères en paramètres. La boucle principale se charge de copier une chaîne de caractères. Dans le champ condition du while, la variable pointée par source est copiée à l'adresse donnée par destination, puis



ces deux pointeurs sont incrémentés pour pointer sur le prochain caractère. De plus, le caractère qui a été copié sert de condition d'exécution de la boucle. Cela signifie que la boucle sera exécutée tant que le caractère copié sera non nul. C'est ce que l'on appelle de la concision. Attention ! Il ne faut pas oublier que le pointeur Destination doit pointer sur une zone mémoire qui aura été réservée auparavant et d'une taille suffisante. Le langage C n'est pas comme le Basic qui se charge de trouver la place nécessaire à la gestion des chaînes de caractères.

Voyons un autre exemple qui consiste à concaténer deux chaînes de caractères :

```
strconc(Source,Destination)
char *Source;
char *Destination;
{ while (*Destination++);          /* Va à la fin de la chaîne */
  strconc(Source,Destination) ;    /* Copie à la suite         */
}
```

VI - Les structures

LES STRUCTURES DE DONNÉES EN LANGAGE C

La notion de structure est particulièrement intéressante, car elle permet d'associer plusieurs variables ensemble pour en faire une entité à part entière. Un exemple maintenant classique consiste à la comparer avec une fiche descriptive d'une personne dans un fichier d'adresses. On y trouve le nom et le prénom qui sont des chaînes de caractères, mais l'âge pourrait être un nombre entier. On pourrait également y ajouter un salaire qui serait de type réel.

Pour reprendre l'exemple ci-dessus, une structure se déclare de la manière suivante :

```
struct Fiche {
   char Nom[25];
   char Prenom[25];
   int Age;
   float Salaire;
};
```

Nous venons ainsi de définir un type de structure nous permettant de travailler sur des fiches. Mais nous n'avons, à aucun moment, défini de variable. Il faut pour cela l'indiquer dans une déclaration identique à celles que nous avons vu plus haut.

```
struct Fiche Adresse[200];
```

C'est ainsi que Adresse est un tableau de 200 fiches du type décrit plus haut. L'accès à l'un des éléments utilise l'opérateur . Ainsi :

`Adresse[0].Age = 25;` permet d'affecter la valeur 25 au champ Age de la première fiche du tableau Adresse.

Vous pourriez penser que l'on peut affecter une chaîne de la même manière, il n'en est rien. Le C ne possède pas de gestion dynamique des chaînes de caractères. C'est pourquoi nous avons défini, dans la structure Fiche, le Nom et le Prénom comme des tableaux de taille fixe. Le C ne permet pas d'affecter une chaîne avec une seule instruction, il faut utiliser une fonction de la bibliothèque nommée **strcpy**. Vous pouvez également utiliser celle décrite dans le chapitre sur les pointeurs (Partie 4, Chap. 4.1.IV).

```
strcpy ("WEKA",Adresse[0].Nom); ou
strcpy ("WEKA",&Adresse[0].Nom[0]);
```

LES TABLEAUX DE STRUCTURES

Il est également possible de déclarer un pointeur sur une structure :

```
struct Fiche *AdrPtr;
```

Attention ! Il ne faut pas oublier que seul le pointeur existe. Il faudra allouer au système un bloc de mémoire qui contiendra les fiches.

L'accès à la fiche pointée par AdrPtr se fera ainsi :

```
AdrPtr -> Salaire = 8000.00;
```

Nous avons vu dans les paragraphes dédiés aux pointeurs qu'il était possible de considérer le pointeur comme étant la base d'un tableau. Il est ainsi possible d'accéder à une fiche de la même manière que dans un tableau :

```
AdrPtr[20].Age = 55;
```

LES CHAMPS

Un champ est un moyen d'organiser des variables de taille inférieure à l'octet. Par exemple, sur l'Atari, la date et l'heure sont codées sur 32 bits :

bits 0-4 : secondes en incréments de 2 secondes
bits 5-10 : minutes
bits 11-15 : heures
bits 16-20 : jour
bits 21-24 : mois
bits 25-31 : année depuis 1980.

Il serait difficile de manipuler ce type de données par des décalages et des masquages. Possible mais probablement très illisible. Le C permet ce type de données.

```
struct TosDate {
unsigned Seconde : 5;
unsigned Minute : 6;
unsigned Heure : 5;
unsigned Jour : 5;
unsigned Mois : 4;
unsigned An : 7;
} Date;
```



Pour initialiser cette date au 17 Mai 1988 21:32:50, on peut écrire ainsi :

```
Date.An = 1988-1980;
Date.Mois = 5;
Date.Jour = 17;
Date.Heure = 21;
Date.Minute = 32;
Date.Seconde = 50/2;
```

LES UNIONS

Il arrive souvent que l'on ait besoin de considérer une variable de plusieurs façons différentes. Prenons l'exemple de la date citée plus haut. Le système renvoie un long et ne se préoccupe pas de savoir qu'il est composé d'une heure, etc.

Nous pourrons définir la date ainsi :

```
union {
  struct TosDate {
    unsigned Seconde : 5;
    unsigned Minute : 6;
    unsigned Heure : 5;
    unsigned Jour : 5;
    unsigned Mois : 4;
    unsigned An ; 7;
  };
  long LongDate;
} Date;
```

L'effacement de cette variable se ferait simplement par :
`Date.LongDate = 0 ;`
mais l'effacement de l'heure serait effectué ainsi :
`Date.TosDate.Minute = 0 ;`

L'INSTRUCTION typedef

Cette instruction est probablement une des plus méconnues par les utilisateurs du langage C. Elle permet de créer de nouveaux types de données.

`typedef int ENTIER;` créera un synonyme de int pour les francophones.

`typedef struct {` créera un nouveau type équivalent à une fiche. Il suffira d'écrire :

```
typedef struct {
char *Nom;
char *Prenom;
} Fiche;

Fiche Adresse[200];
```

Vous remarquerez que, dans notre cas, cela apporte simplement la possibilité d'omettre d'écrire **struct** en en-tête. Heureusement **typedef** ne s'arrête pas là ! On peut, par exemple, définir un type permettant d'accéder à un tableau à plusieurs dimensions.

```
typedef int TABLEAU[100];
void main ( )
{  TABLEAU *P;
   P=malloc(1000L);
   P[50][50]=0;
}
```

Ce moyen permet de manipuler des tableaux à plusieurs dimensions sans avoir à allouer la mémoire lors de la compilation.

Un dernier exemple concerne les types basés sur les fonctions.

```
typedef int (*PFI) ( );
PFI Func1,Func2,Func3;
```



4/4.1

Les compilateurs présents sur le marché

Voici maintenant une revue critique des différents compilateurs C disponibles sur le marché aujourd'hui.

ALCYON

C'est le compilateur fourni avec le package de développement vendu par Atari France. Il est complet, génère un code très propre sans être très optimisé. Son principal défaut concerne sa vitesse de compilation et la nécessité de posséder un disque dur.

LATTICE C

Ce compilateur représente le choix de tous ceux qui programment dans un souci de portabilité vers une machine comme le PC. Il possède une bibliothèque de fonctions sans égale sur les autres compilateurs. Par contre, il ne compile pas vraiment rapidement et génère un code particulièrement gros et lent. Il est préférable, comme pour l'ALCYON, de posséder un disque dur.

MARK WILLIAMS

Cette implémentation nous vient du PC et apporte un avantage par rapport à ses concurrents. CSD est un debugger symbolique permettant de mettre au point un programme C en exécutant pas à pas la source C lui-même. La compilation n'est pas vraiment rapide mais la qualité est excellente. La documentation est très complète, mais ressemble plus à un manuel de référence.

LASER C

Ce compilateur représente la nouvelle mouture du Megamax C dont nous n'avons pas parlé. C'est sans aucun doute le plus rapide de tous. Par

contre, le résultat de la compilation n'est pas des meilleurs et semble même être parmi les plus mauvais. Il est, toutefois, le seul à permettre d'insérer du source assembleur au cœur du source C et de plus, il ne nécessite pas de disque dur.

PROSPERO C

Il s'agit également d'un compilateur ne nécessitant pas de disque dur. Par contre, la qualité de la compilation est pratiquement la plus mauvaise qu'il nous ait été donné de voir. Par contre, le manuel est très bien organisé et constitue un excellent ouvrage de référence sur l'Atari ST.

TURBO C

Il s'agit de l'implémentation ST du célèbre compilateur de Borland. En réalité, il a été réalisé par la filiale allemande de Borland, indépendante de la maison mère aux USA. Il est pratiquement aussi rapide que LASER C, tout en compilant à la limite de la perfection. Malheureusement, à l'heure où ces lignes sont écrites, il n'est disponible qu'en Allemagne.

4/4.2

Le langage C par les exemples de programmation

Après avoir vu les instructions et formes principales du langage C, nous vous proposons maintenant d'apprendre à programmer ce langage par l'écriture de programmes. Le but est de pouvoir exposer les principes de base de la programmation en vous permettant le plus vite possible de programmer. Les exceptions et cas particuliers feront l'objet d'un traitement ultérieur en même temps que tous les outils qui peuvent vous permettre de programmer de façon plus élaborée, par exemple les structures et les pointeurs.

Nous essaierons de préciser au maximum les modifications que vous pourrez apporter aux petits programmes proposés, ainsi que les règles de programmation qui se rattachent aux exemples choisis pour comprendre et éviter les erreurs avant la compilation.

Le programme le plus simple par lequel on peut commencer est celui qui réalise l'affichage d'un texte, par exemple : bonjour, WK.

En langage C, le programme qui réalise cet affichage est le suivant :

PETIT PROGRAMME DE DÉPART

Exemple 1

```
main()
{
    printf("bonjour,WK\n");
}
```

Commentons ce petit programme.

Un programme en C, quel que soit sa taille et sa complexité, se compose d'une ou plusieurs fonctions similaires aux fonctions ou procédures des langages Basic. Dans l'exemple traité, **main** est le nom d'une telle fonction ; en langage C, les fonctions peuvent avoir un nom de votre choix mais tout programme doit commencer par la fonction main qui est un nom imposé du langage C. Dans un programme plus sophistiqué que ce premier exemple,

la fonction main appellera d'autres fonctions qui pourront être soit écrites dans le même programme, soit appartenir à certaines bibliothèques de fonctions. Ici main est le nom de la fonction et cette fonction est sans argument. Dans un cas plus général, les arguments sont précisés entre les parenthèses qui suivent le nom de la fonction, comme le font les paramètres d'appel des procédures ou fonctions du Basic. Le corps de la fonction est constitué par la suite des instructions comprises entre les accolades { et }. C'est la première chose à taper lorsque vous définissez une fonction, afin de toujours conserver un nombre égal de parenthèses ou accolades ouvrantes et fermantes.

La fonction main est constituée par une seule instruction qui est un appel de la fonction printf. Cette fonction n'est pas un mot clef du langage C mais une fonction qui fait partie de la bibliothèque standard et qui imprime l'argument précisé entre les guillemets sur le terminal (il n'y a aucune entrée sortie définie dans le langage C). Dans le cas du programme tel qu'il était écrit plus haut, il serait impossible au compilateur de trouver la fonction printf car aucune définition de cette fonction n'est faite dans le programme. Tout fichier source qui fait référence à une fonction de la bibliothèque standard doit inclure à son début la ligne :

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

Ce fichier stdio.h définit certaines macro-instructions et variables utilisées par la bibliothèque des entrées sorties. L'utilisation de < et > indique au compilateur de rechercher le fichier dans le répertoire d'en tête de la bibliothèque standard. Afin que la compilation puisse s'effectuer sans erreur, le programme doit maintenant s'écrire :

Exemple 2

```
#include <stdio.h>
main()
{
    printf("bonjour,WK\ n");
}
```

L'argument de la fonction précisé entre guillemets dans l'exemple est appelé chaîne de caractères ou chaîne constante. Le caractère\n positionne le terminal sur la position gauche et effectue donc un retour à la ligne, et c'est le seul caractère à le faire. D'autres caractères permettent un positionnement différent du terminal :

\t tabulation

\b espacement arrière *(backspace)*

\r retour chariot *(Return)*

L'instruction correspondant à l'appel de la fonction printf est délimitée à droite par le point virgule.

Les instructions qui suivent permettent aussi de réaliser le même affichage :



Exemple 3

```
#include <stdio.h>

main( )
{
    printf("bonjour,");
    printf("WK");
    printf("\ n");
}
```

Les deux premiers appels de **printf** effectuent l'affichage en suivant sur la même ligne et le retour à la ligne est réalisé par le dernier **printf**.

LES VARIABLES

Supposons que vous ayez besoin d'afficher une table des valeurs de la fonction définie par Y=(7/8)(X+6) pour les valeurs de X variant de 0 à 100 par pas de 5.

Le programme suivant permet de réaliser cet affichage :

Exemple 4

```
/* impression de la table des valeurs
de la fonction Y=(7/8)(X+6)
pour les valeurs de X=0,5,10,....,100 */

#include <stdio.h>

main( )
{
    int borneinf, bornesup, pas;
    float X,Y;

    borneinf = 0;     /* limite inférieure des X de la table */
    bornesup = 100; /* limite supérieure des X de la table */
    pas = 5;          /* valeur du pas d'incrémentation */

    X=borneinf;
    while (X<=bornesup) {
      Y=(7.0/8.0)*(X+6);
      printf("%5.0f %5.2f\ n",X,Y);
      X=X+pas;
    }
}
```

Les trois premières lignes constituent un commentaire qui peut être placé n'importe où dans le programme, sur un blanc ou sur un caractère d'interligne, mais qu'il vaut mieux tabuler pour des raisons de clarté.

Les symboles **/*** et ***/** sont les séparateurs de début et de fin de commentaire. Le compilateur ignorera tous les caractères situés entre ces deux séparateurs.

Le programme devant réaliser un affichage avec la fonction **printf**, la première ligne du programme inclut la référence de compilation à **<stdio.h>**.

Toutes les variables doivent être déclarées avant d'être utilisées dans le langage C. Si une variable n'est pas déclarée, le compilateur enverra un message d'erreur. Ces déclarations se font en début de fonction, avant les instructions proprement dites.

La déclaration se compose du type de la variable qu'on définit et de la liste des variables qui appartiennent à ce type, en séparant ces variables par une virgule. La déclaration se termine par une virgule. Le compilateur faisant la différence entre les majuscules et les minuscules, bien faire attention d'écrire ces variables toujours de la même façon !

Avant de choisir un nom pour une variable, il est bon de connaître les quelques restrictions qui s'imposent :

• Tout identificateur de variable peut comporter des lettres et des chiffres mais doit commencer par une lettre. Seuls les huit premiers caractères sont significatifs. Le tiret bas ou souligné (_), est analysé comme une lettre. Par exemple, les deux identificateurs de variables **azerty_uio** et **azerty_uoi** seront considérés comme les mêmes, les caractères différant à partir de la neuvième position.

• Comme dans tout langage, les identificateurs de mots clefs ne peuvent être utilisés comme identificateurs de variables. Les mots clefs sont les suivants :

auto	**double**	**if**	**static**
break	**else**	**int**	**struct**
case	**entry**	**long**	**switch**
char	**extern**	**register**	**typedef**
continue	**float**	**return**	**union**
defaut	**for**	**short**	**unsigned**
do	**goto**	**sizeof**	**while**

« **int** » indique que les variables sont entières.

« **float** » indique que les variables sont des nombres qui peuvent avoir une partie fractionnaire. La précision des entiers et des flottants n'est pas une donnée du langage C mais dépend de l'ordinateur sur lequel on travaille. Sur l'Atari, un entier est un nombre de 16 bits, c'est-à-dire compris entre +32767 et −32768. Un nombre en virgule flottante est codé sur 32 bits, ce qui permet d'avoir sept chiffres significatifs avec un ambitus variant de 10^(−38) à 10^(+38).

Les types de variables sont :

char caractère codé sur un octet.

int type entier codé sur 16 bits.

short type entier court codé sur 16 bits.

long type entier long codé sur 32 bits.

float type nombre en virgule flottante codé sur 32 bits.

double type nombre en virgule flottante et double précision codé sur 64 bits.



Les nombres en virgule flottante, ont un exposant codé sur huit bits. Le calcul de la table des Y commence par les affectations des valeurs initiales aux variables. Chaque affectation se termine par un point virgule puisqu'elle représente une instruction à part entière.

La boucle

```
while(X<= bornesup){
...
}
```

permet d'exécuter le corps de la boucle, c'est-à-dire la suite d'instructions comprises entre les deux accolades { et }, tant que la condition **(X<=bornesup)** est réalisée. Dès que la condition n'est plus réalisée, on passe à l'instruction qui est placée juste après la boucle. Ici c'est la fin du programme. Il est préférable, comme dans l'exemple des commentaires, de préparer vos tabulations afin d'aligner les mêmes niveaux logiques de programmation surtout lorsque vous utilisez des bouclers et des tests imbriqués. Cela facilite grandement la lecture des programmes. Notez aussi que le langage C est très tolérant au niveau des espaces séparant les opérateurs des variables : usez-en systématiquement pour bien séparer les opérandes des opérateurs. Il est possible aussi d'écrire plusieurs instructions par ligne. Cela ne pose aucun problème de compilation, mais par souci de clarté nous vous conseillons de n'écrire qu'une instruction par ligne.

Dans le corps de la boucle, notez l'utilisation de **7.0/8.0** au lieu de **7/8** car, en langage C, la division d'entiers donne un résultat entier, donc 0 dans le cas qui nous intéresse (partie entière du résultat). Le point décimal caractérise une constante en virgule flottante, ce qui convient au calcul de fonction que nous proposons ici. Notez par contre que l'on peut écrire **X+6** ; **X** étant déclaré comme variable de type flottant, 6 sera automatiquement converti en constante de type flottant avant l'addition. Il est tout à fait possible aussi d'écrire **X+6.0**, le résultat sera identique et cela peut aider à la clarté de lecture du programme. Dans le test de la boucle **do while**, l'entier **bornesup** est aussi converti en flottant avant le test.

Nous avons ensuite un autre exemple d'utilisation de la fonction **printf**, avec des formats d'affichage :

```
printf("%5.0f %5.2f\ n",X,Y);
```

Le premier argument de la fonction représente la chaîne de contrôle qui précise le format et la position d'écriture des deuxième et troisième arguments (X et Y).

Le **%** introduit une spécification de conversion. Cette conversion se termine par un caractère de conversion, ici **f**. Entre les deux **%5.0f** indique que X doit être écrit en virgule flottante sur cinq positions et aucun chiffre après le point décimal.

%5.2f indique que Y doit être écrit en virgule flottante sur cinq positions au moins et avec deux chiffres après le point décimal.

Le nombre de **%** dans le premier argument est de deux et correspond bien au nombre d'arguments à afficher. Attention, car **printf** utilise son premier argument (chaîne de contrôle) pour connaître le nombre d'arguments qui

suivent ainsi que leur type. Il faut absolument respecter le nombre et le genre des arguments sous peine d'avoir des affichages qui n'auront aucun sens.

On peut aussi définir les constantes BORNEINF, BORNESUP, PAS en début de programme grâce à **#define** (notez l'absence du point virgule en fin de ligne), afin qu'elles deviennent une référence dans tout ce programme. Il n'est plus besoin à ce moment-là de les déclarer, le compilateur remplaçant le nom de la constante par la chaîne de définition.

Exemple 5

```
#include <stdio.h>

#define    BORNEINF 0 /* limite inf des X de la table */
#define    BORNESUP 100 /* limite sup des X de la table */
#define    PAS = 5          /* valeur du pas d'incrémentation */

main( )
{

    float X,Y;

    X = BORNEINF;
    while (X< = BORNESUP){
      Y=(7.0/8.0)*(X+6);
      printf("%5.0f %5.2f\n",X,Y);
      X=X+PAS;
    }
}
```

Une programmation plus succinte du même résultat peut être réalisée grâce à l'utilisation de la boucle **for** et des définitions de constantes comme précédemment. Le calcul de Y n'étant pas nécessaire on utilise directement le résultat du calcul dans le second argument de **printf**.

Exemple 6

```
#include <stdio.h>

#define    BORNEINF 0 /* limite inf des X de la table */
#define    BORNESUP 100 /* limite sup des X de la table */
#define    PAS=5            /* valeur du pas d'incrémentation */

main( )
{

    int X;

    for(X=BORNEINF;X< =BORNESUP;X=X+STEP;)
    printf("%5.0f %5.2f\ n":,(7.0/8.0)*(X+6));

}
```

LECTURE ET COMPTAGE DE CARACTERES

La bibliothèque standard fournit des fonctions permettant la lecture ou l'écriture de caractères un par un. **getchar( )** va lire le caractère qui se trouve à l'entrée et en renvoie la valeur correspondante.



`c=getchar( );`

getchar( ) renvoie la valeur EOF *(end of file)* quand elle rencontre la fin du fichier en entrée de lecture. La constante symbolique **EOF** est définie égale à **–1** (à l'aide d'un **#define** dans le fichier **stdio.h**).

Les tests doivent utiliser **EOF** et non **–1** pour être indépendants d'une valeur précise.

putchar(c) écrit le caractère **c** sur le terminal.

Le programme suivant renvoie sur la sortie le caractère lu en entrée :

Exemple 7

```
#include <stdio.h>

main( )
{
    int c;
    c=getchar( );
    while(c!=EOF){
      putchar(c);
      c=getchar( );
    }
}
```

En C, l'affectation d'un caractère à la variable **c** peut s'intégrer dans l'expression du test de la boucle **while**. Le programme devient :

Exemple 8

```
#include <stdio.h>

main( )
{
    int c;

    while((c = getchar( ))!=EOF){
      putchar(c);
    }
}
```

Dans ce cas, le corps du **while** ne contient qu'une instruction. Les accolades ne deviennent alors plus nécessaires. Le programme peut maintenant s'écrire :

Exemple 9

```
#include <stdio.h>

main( )
{
    int c;

    while((c=getchar( ))!=EOF)
      putchar(c)
}
```

Le programme qui suit va compter le nombre de fois où apparaît un caractère, ici la tabulation \t, dans le fichier de lecture :

Exemple 10

```
#include <stdio.h>
main( )
{
    int c, nbtab;
    nbtab=0;
    while((c=getchar( ))!=EOF
      if (c= = '\t')
        + +ntab;
    printf("%d \n",ntab);
}
```

La variable **nbtab** qui compte le nombre de tabulations est déclarée ici comme type **int** ; il faut supposer ne pas avoir plus de 32767 tabulations à lire. Dans le cas contraire, il faudrait la déclarer comme variable de type long et transformer le **printf** en conséquence. Dans le cas où le caractère lu n'est pas celui de fin de fichier, on le compare au caractère de tabulation à l'intérieur du test **if**. Si ce test est vérifié, (égalité par le double signe égal = =), on effectue l'instruction ou le groupe d'instructions entre accolades qui suit. De la même façon que dans le programme précédent, une seule instruction ne nécessite pas l'emploi d'accolades. L'instruction + +**ntab** utilise l'opérateur + + qui sert à incrémenter de une unité. Cela revient à écrire **ntab**=**ntab**+1, mais l'écriture **ntab**+ + est plus concise. L'opérateur – sert à décrémenter de la même façon d'une unité. Ces deux opérateurs peuvent se placer devant ou derrière la variable. Dans tous les cas, la variable est incrémentée (ou décrémentée). La différence est la suivante :

+ +**ntab** incrémente **ntab** avant que sa valeur ne soit utilisée alors que **ntab**+ + l'incrémente après que sa valeur ait été utilisée.

Par exemple, si **ntab** vaut dix, **y**=**ntab**+ + affecte 10 à la variable **y** alors que **y**= + +**ntab** affecte la valeur 11. Dans les deux cas, **ntab** deviendra égal à 11. Dans le cas où seule une incrémentation de compteur entre en jeu, l'une ou l'autre des formules peut être employée.

Tout caractère peut être écrit entre apostrophes. Sa valeur est une constante de type caractère et est égale au code ASCII de ce caractère. Par exemple, 'B' est une constante de type caractère dont le code ASCII est 66. Si on voulait compter le nombre de lettres B et en supposant qu'on puisse en obtenir un grand nombre, le programme s'écrirait maintenant :

Exemple 11

```
#include <stdio.h>
main( )
{
  int c;
  long nbreB;
  nbreB=0;
  while((c=getchar( ))!=EOF)
    if(c= ='B')
      + +nbreB;
    printf("%.0f \n,nbreB);
}
```



printf utilise **%f** à la fois pour le type **float** et le type **double** ; **%.0f** supprimera de l'écriture la partie fractionnaire qui est inexistante dans **nbre B** puisque ce dernier est déclaré comme long. Si on voulait compter à la fois les tabulations et les B, le programme deviendrait :

Exemple 12

```
#include <stdio.h>
main( )
{
   int c;
   long nbretot;
   nbretot=0;
   while((c=getchar( ))!=EOF)
      if(c=='B' || c=='\t')
         ++nbretot;
   printf("%.0f\n",nbretot);
}
```

L'opérateur **||** représente **le** ou **logique**. La ligne : **if(c=='B'||c=='\t')** indique donc : Si le caractère est un B ou une tabulation, exécuter les instructions du corps du **if**. Le test **du** ou **logique** s'effectue de gauche à droite. Dans notre cas, si on trouve que le caractère est égal à B, on ne va pas ensuite le comparer à la tabulation. L'évaluation de la proposition logique cesse dès qu'on sait si cette proposition est vraie ou fausse.

L'exemple suivant compte séparément les tabulations et les B à l'aide de deux compteurs déclarés tous deux en long.

Notez l'initialisation des valeurs à zéro par l'instruction : **nbretot=nbretab=0;**

C'est dû au fait que l'affectation (grâce au symbole =) prend la valeur située à droite pour la mettre à gauche. On pourrait avoir plusieurs affectations similaires et en chaîne sur la même ligne. L'exemple utilise aussi l'instruction **else** qui précise l'instruction à exécuter lorsque la condition de l'instruction **if** n'est pas réalisée. Dans **if-else**, une seule des instructions ou groupe d'instructions est exécutée. Dans l'exemple, l'instruction qui suit le **else** est un autre **if** qui contrôle une seule instruction entre accolades. Ces dernières n'ont été mises ici que par souci de séparation logique.

Exemple 13

```
#include <stdio.h>
main( )
{
   int c;
   long nbreB,nbretab;
   nbretot=nbretab=0;
   while((c=getchar( ))!=EOF){
      if(c=='B')
         ++nbreB;
      else if(c=='\t'){
         ++nbretab;
      }
   }
      printf("%.0f %.0f\n",nbreB,nbretab);
}
```

LES TABLEAUX

Cherchons maintenant à écrire un programme qui compte les occurences de chaque lettre (que l'on supposera minuscules pour l'instant) dans la même lecture d'entrée que précédemment. Il y a donc 26 sortes de données en entrée et au lieu de définir 26 variables pour compter l'apparition de chaque lettre différente, nous allons définir un tableau qui va regrouper le nombre d'apparitions de chaque lettre. Le tableau doit avoir une dimension de 26 ; en C, comme en Basic, les indices commencent à zéro mais la dimension de définition est bien un nombre d'éléments et non une valeur maximum d'indice de tableau.

Exemple 14

```
#include <stdio.h>

main( )
{

    int c,i;
    int nbreapp[26];

    for(i=0;i<26;i++)
      nbreapp[i]=0;

    while((c=getchar( ))!=EOF)
      if(c>='a'&&c<='z')
        ++nbreapp[c-'a'];

    for(i=0;i<26;++i;)
      printf(" %d",nbreapp[i]);
}
```

Contrairement au Basic, les tableaux ne sont pas mis à zéro par défaut. Il faut toujours penser à initialiser un tableau ou une variable. C'est le rôle de la première boucle **for**.

c- 'a' est une expression entière dont la valeur varie entre 0 et 25 suivant le caractère contenu dans **c**. Reportez-vous à ce qui a été dit précédemment sur l'affectation du code ASCII à la valeur du caractère écrit entre guillemets. La valeur du code ASCII de **a** est 97 et les indices correspondants vont permettre de ranger les nombres d'apparition des lettres dans l'ordre alphabétique puisque aucun caractère autre que les lettres minuscules n'est compris entre **'a'** et **'z'**.

Le programme suivant reprend la comptabilisation précédente en ne distinguant pas les minuscules des majuscules. Le premier élément du tableau **nbreapp[]** contiendra donc en fin de fichier le nombre d'apparitions de la lettre **'a'**, minuscules et majuscules confondues. La configuration

```
if(condition1)
        instructions
else if(condition2)
        instructions
```



est un moyen classique de réaliser un test à plusieurs sorties. Un autre moyen est l'utilisation de la fonction **switch** que nous verrons ultérieurement. Dans le cas où plusieurs instructions se suivent dans le corps d'un **if** ou d'un **else if**, il ne faut pas oublier de mettre les accolades ouvrantes et fermantes.

Exemple 15

```
#include <stdio.h>

main( )
{
    int c,i;
    int nbreapp[26];

    for(i=0;i < 26;i++)
      nbreapp[i]=0;

    while((c=getchar( ))!=EOF)
      if(c >= 'a' && c <= 'z')
        ++nbreapp[c- 'a'];
      else if (c >= 'A' && c <= 'Z')
        ++nbreapp[c- 'A'];

    for(i=0;i <26; ++i;)
      printf("%d",nbreapp[i]);
}
```

LES FONCTIONS

En C, les fonctions sont équivalentes aux fonctions et procédures du Basic. Les fonctions que nous avons utilisées jusqu'à présent étaient celles fournies par la bibliothèque des entrées sorties **stdio.h**, **printf**, **getchar** et **putchar**. Pour créer une fonction, il faut la déclarer suivant une structure standard propre au langage C :

— Nom (liste éventuelle d'arguments)

— Déclarations d'arguments s'ils existent

— { (accolade ouverte)

— Déclarations

— Instructions

— } (accolade fermée)

Supposons que nous voulions définir une fonction linéaire en introduisant comme paramètres de cette fonction la pente et l'ordonnée à l'origine d'une droite. Appelons cette fonction **dtelin(a,b,x)** qui fait correspondre à la variable **x** la valeur **a*x+b**. Dans le programme qui suit, on n'est intéressé que par des définitions de valeurs entières de variables et de fonctions, mais il suffit de transformer toutes les déclarations de type en **float** pour travailler avec des valeurs réelles et de transformer aussi la boucle qui affiche quelques

valeurs de cette fonction. Penser à transformer aussi la chaîne de contrôle du **printf** en conséquence avec la précision fractionnaire désirée.

Voici cette fonction et le programme principal qui l'utilise.

Exemple 16

```
#include <stdio.h>

main( )
{
  int i;

  for(i=0;i<=20;i++)
    printf("%d %d %d \n",i,dtelin(4,5,i),dtelin(-2,3,i+5));
}

dtelin(a,b,x)     /* fonction linéaire */
int a,b,x;

{
int y;

y=a*x+b;
return(y);
}
```

Supposons que nous voulions maintenant, pour compliquer un peu le problème, définir une fonction polynôme du second degré ou parabole. Soit la fonction parabole (a,b,c,X) qui fasse correspondre à X la valeur $a*X^2+b*X+c$. La partie linéaire de cette fonction reprend le **dtelin** précédent, mais nous ne disposons pas de la fonction d'élévation à une puissance positive, (cas général de l'élévation au carré). Il nous faut donc définir d'abord cette fonction puissance qui sera elle-même appelée dans la fonction parabole. En langage C, les valeurs des arguments d'appel de fonction sont transmises par des variables provisoires, l'équivalent des variables locales du Basic GFA. La fonction ne peut pas modifier une variable du programme qui appelle cette fonction, mais seulement modifier sa variable locale.

Cette remarque permet souvent de compacter des programmes.

Exemple 17

```
#include <stdio.h>

main( )
{
  int i;

  fort(i=0;i<=20;i++)
      printf("%d %d \n",i,parabole(-3,4,5,i));
}

puissance(x,n)     /* élévation de x à la puissance n */
int x,n;
{
  int pui;
```



```
(suite)
    for(pui = 1;n > 0; - - n)
        pui = pui * x;
    return(pui);
    }

parabole(a,b,c,x) /* polynôme du second degré */
int a,b,c,x;
{
    int y;

    y = a * puissance(x,2) + b * x + c;
    return(y);
}
```

LES TABLEAUX DE CARACTÈRES

Les tableaux de caractères sont les plus utilisés dans le langage C. Nous vous proposons un petit exemple de traitement de tableaux de caractères. La fonction définie supprime un caractère **(caract)** donné d'une chaîne de **ch** et renvoie le nombre de caractères supprimés. Comme précédemment, la chaîne lue se termine par un '\n'.

Exemple 18

```
int supprime(ch, caract)
char ch[],carac;
{
int j,i;
j = 0;
i = 0;
while (ch[i] != '\n'){
        if(ch[i] != carac){
                ch[j] = ch[i];
                j + +;
                }
        i + +;
        }
ch[j] = '\n';
return(i - j);
}
```

Il existe une multitude de problèmes qui peuvent être traités en utilisant les tableaux de caractères. Les exemples les plus pointus traitent les textes et analysent des mots ou autres successions de caractères. Vous

pourrez en trouver facilement les applications dans toutes les fonctions des traitements de textes qui comptent les mots et vous proposent d'afficher les occurrences de mots, de caractères et de symboles. Les principes de bases de ces types d'analyse et de traitements peuvent être énoncés et résolus dans les quelques exemples qui suivent :

Trouver si les lettres d'un premier mot appartiennent ou non à un second mot.

Si nous nous basons sur l'orthographe française, nous supposerons qu'il n'y a pas de mot de longueur supérieure à 25 (le fameux anticonstitutionnellement).

Chaque mot doit être frappé et confirmé par la touche RETURN. Pour que le programme ait une allure structurée et puisse être réutilisé ultérieurement, il sera nécessaire de définir une fonction qui fournira la longueur d'un mot entré au clavier, puisqu'il faudra tester et comparer tous les caractères des deux mots un à un, du premier au dernier. Le mot rentré ne sera validé que lorsqu'on aura tapé sur RETURN. Cela veut dire aussi que si l'utilisateur n'a pas validé le mot avant le 25e caractère, ce dernier ne sera pas considéré comme valable. Il faudra tout de même attendre la validation par la touche RETURN pour sortir de la fonction donnant la longueur du mot.

Le premier exemple lit les deux mots caractère par caractère et les représente sous forme de chaîne de caractères dont le codage se termine par le caractère 0. Si les mots sont supposés avoir la longueur maximale de 25 caractères, il faut penser à coder le caractère de fin de chaîne, donc considérer 26 caractères à coder par mot. Notez que ce codage de chaîne de caractères n'est pas spécifique au C et qu'on le retrouve dans les autres langages traités sur Atari (voir Basic GFA et Basic OMIKRON).

A cette fin, il est pratique d'utiliser la fonction GETS de la bibliothèque standard STDIO.H qui lit les caractères sur l'entrée standard (stdin). La lecture s'interrompt à la rencontre du symbole de fin de ligne (\ n) ou d'une fin de fichier (EOF). Les caractères lus sont rangés dans la chaîne dont l'adresse optionnelle est l'argument de la fonction, en remplaçant le caractère de fin de ligne (\ n) par le caractère de fin de chaîne (\ 0). Si une erreur apparaît ou si le caractère de fin de fichier est lu alors qu'aucun caractère n'a été frappé, la valeur NULL est renvoyée.

La seconde fonction utilisée dans cet exemple est STRLEN qui renvoie la longueur de la chaîne de caractères qui est précisée en argument. C'est l'équivalent de LEN(a$) en GFA ou OMIKRON. Cette fonction fait partie de la bibliothèque standard STRING.H qui gère les manipulations de chaînes de caractères.

Ces deux bibliothèques doivent être signifiées au préprocesseur à l'aide de la directive #include en début de programme.



Exemple 19

```
#include <stdio.h>
#include <string.h>
#define  MOTMAX  25

main()
{
      char mot1 [MOTMAX+1],
           mot2 [MOTMAX+1];
      int i,j ;

/* lecture du premier mot*/
printf ("rentrez un premier mot et tapez RETURN:  ");
gets ( mot1);

/* lecture du second mot*/
printf ("rentrez un second mot et tapez RETURN:  ");
gets ( mot2);

/*comparaison des mots caractère par caractère*/

for (i=0 ; i<strlen(mot1) ; i++)

     for (j=0 ; j<strlen(mot2) ; j++)

          if ( mot1[i] == mot2[j] )

             { printf ("%c appartient aux deux mots\n", mot1[i]);

               mot2[j] =' ';

               break;
             }
          else

             printf ("%c n'appartient pas aux deux mots\n", mot1[i]);
}
```

Remarques :

— Lorsqu'on imprime le caractère commun trouvé, il se trouve dans **mot1[i]** car c'est le enième testé du premier mot.

— On remplace le caractère testé par un blanc dans le second mot pour être sûr de ne pas le retester au cours d'un passage ultérieur. Il faut donc supposer que les mots sont tapés sans erreur et ne comportent pas de caractères fantaisistes.

— L'instruction break permet de sortir de la boucle dès qu'on a trouvé le caractère du premier mot une fois dans le second mot. Cela évite d'avoir un message du type :

a appartient aux deux mots
a appartient aux deux mots
a appartient aux deux mots

si le premier mot est madame et le second barbara.

Notez que cela évite aussi de reconsidérer le a une seconde fois dans le message de sortie, mais que le traitement est effectué à nouveau pour cette même lettre, puisqu'en début de boucle, on ne teste pas sur le mot 1 si le caractère que l'on va comparer a déjà été considéré au cours d'une précédente itération.

Le second exemple reprend la comparaison précédente en utilisant la fonction getchar( ) de la bibliothèque STDIO.H et en créant la fonction lecture( ) d'un mot de longueur maximale de 25 lettres. Cette fonction compte et renvoie le nombre de lettres du mot qui est lu. Les mots ne sont plus codés en chaîne de caractères et n'ont plus de 0 en fin de chaîne, ce qui était une caractéristique de la fonction gets( ). Il suffit maintenant de définir des tailles de tableaux de caractères égales à la longueur maximale des mots traités.

Exemple 20

```
#include <stdio.h>

#define MOTMAX 25

main()
{
     int (lecture);
     char mot1 [MOTMAX],
          mot2 [MOTMAX];
/*longueur des mots 1 et 2*/
     int longmot1, longmot2,
         i,j ;

/* lecture du premier mot*/
printf ("rentrez un premier mot et tapez RETURN: ");
longmot1 = lecture ( mot1);

/* lecture du second mot*/
printf ("rentrez un second mot et tapez RETURN: ");
longmot2 = lecture ( mot1);

/*comparaison des mots caractère par caractère*/
for (i=0 ; i<longmot1 ; i++)
     for (j=0 ; j<longmot2 ; j++)
          if ( mot1[i] == mot2[j] )
```



(suite)

```
                { printf ("%c appartient aux deux mots\n", mot1[i]);
                  mot2[j] = ' ';
                  break;
                }
            else
                printf ("%c n'appartient pas aux deux mots\n", mot1[i]);
}

int lecture (mot)
char mot [MOTMAX];
{
    int i;
    char c;

    i=0;
      /* tant qu'on ne trouve pas le RETURN ou le nombre
      maximum de caractères, on lit on remplit le tableau
      mot[]. Comme on affecte toujours getchar dans c
      pour tester la condition de boucle, c contiendra toujours
      le premier caractère non validé */

          while ((c=getchar) != '\n' && i<MOTMAX+1)
              mot[i++] = c;

      /* ici on recherche \n pour traiter le cas où
      on dépasse la longueur sans avoir tapé sur
      la touche RETURN. On ne fait rien {}, et i
      correspond dans tous les cas au nombre de lettres entrées.*/

          if (c!= '\n')
              while (getchar() != '\n' {}
          return(i);
      }
```

Remarques

— La fonction lecture( ) a été écrite en suivant les conventions du C standard ; en Turbo C dont la concision est remarquable, les deux premières lignes auraient pu être remplacées par int lire (char*mot). Comme en GFA ou OMIKRON, l'astérisque représente l'adresse du tableau de caractères ou de la variable. La définition précédente permet de stipuler en une ligne que l'argument de lire est un pointeur (*) de caractère.

— Le programme source (non compilé) est composé du programme principal main( ) et de la fonction lecture( ) ; tant que la compilation s'effectue sur l'ensemble de ces deux fonctions, la valeur de MOTMAX est définie pour l'ensemble du source. Si vous décidiez d'utiliser lecture( ) dans un autre contexte, il serait préférable de constituer un fichier séparé de symboles communs à plusieurs sources et de la référencer par #include à chaque utilisation, comme une des fonctions standards de la bibliothèque.

Nous allons maintenant cumuler les deux programmes précédents avec le programme qui comptait les occurrences d'un caractère à l'intérieur d'un mot. Le programme que nous vous proposons se propose de compter les occurrences de chaque lettre de l'alphabet à l'intérieur d'un mot, puis de réaliser l'affichage du résultat.

Dans un premier temps, nous supposerons que les mots ne sont composés que de lettres minuscules.

Exemple 21

```
#include <stdio.h>

#define MOTMAX 25

main()
{
    int (lecture);
    char mot [MOTMAX];

        /*longueur du mot */
    int ocurlettre[26];
    int longmot,
        numerolettre,
        i;

/* initialisations du tableau */

for (i=0 ;i<26; i++) ocurlettre[i]=0;

/* lecture du  mot*/
printf ("rentrez un mot et tapez RETURN:  ");
longmot = lecture ( mot);

for (i=0 ; i<longmot; i++)
        {
/* on donne le rang 0 au a minuscule */
        numerolettre = mot[i] -'a';
        ocurlettre[numerolettre]++;

/*affichage du résultat */
printf ("\n\n\n le mot comporte %d caractères \n,logmot);
for (i=0; i< 26 ; i++)
    printf("%d fois la lettre %c\n,ocurlettre[i],'a'+i);
```



(suite)

```
}

int lecture (mot)
char mot [MOTMAX];
{
    int i;
    char c;

    i=0;

    while ((c=getchar) != '\n' && i<MOTMAX+1)
        mot[i++] = c;


    if (c!= '\n')
        while (getchar() != '\n' {}
    return(i);
}
```

Remarques :

— Le tableau de caractères que nous analysons est supposé parfait, à savoir ne contenant pas de caractères autres que les lettres minuscules.

— Si on voulait effectuer ce programme sur plusieurs mots consécutifs il suffirait de définir autant de tableaux que de mots à traiter de façon similaire à ce qui a été fait pour les deux mots dont on voulait comparer les lettres communes. Mais nous verrons plus loin que ce n'est pas la façon la plus élégante de traiter le problème.

L'exemple se complique un peu lorsqu'on veut effectuer cette opération non pas sur un mot mais sur un texte entré au clavier. Cette fois, on ne stockera pas le texte dans un tableau de caractères (la dimension de ce tableau dépendrait d'ailleurs de la longueur du texte entré). Le traitement se fera à la lecture des caractères au clavier, ce qui nous oblige dans le traitement à devoir considérer les caractères autres que des lettres, erreurs de frappe ou autres que nous pouvons éventuellement afficher.

La première méthode peut consister à lire le texte caractère par caractère comme précédemment en éliminant les caractères de fin de ligne. La seule fonction à utiliser est le **getchar( )** précédent inclus dans la bibliothèque standard **stdio.h**.

La deuxième méthode peut étendre ce traitement à un traitement ligne par ligne. Chaque traitement de ligne correspond à un traitement caractère par caractère des mots de la ligne. Il faut dans ce cas utiliser la fonction **gets( )** de la bibliothèque **stdio.h** pour lire une ligne et s'arrêter en fin de ligne. La fonction **strlen( )** de la bibliothèque **string.h** nous donnera la longueur de la ligne entrée. Comme l'analyse n'est pas poussée jusqu'au mot, nous n'avons pas à traiter de façon particulière l'espace

ou les signes de ponctuation divers. Nous pourrions d'ailleurs dans ce cas proposer des résultats du type : dans le texte, il y a tant de fois le mot « ... ». En moyenne les mots ont « ... » lettres et les occurrences des lettres sont les suivantes : « ... ». En fait les bases de ces traitements sont contenues dans les programmes ci-dessous et précédents ; il suffit simplement de développer les traitements à l'intérieur de boucles de mots au lieu de boucles de lignes.

Première méthode :

Exemple 22

```
#include <stdio.h>

main()
{
        char c,
        canter;
 /* c lit un caractère frappé au clavier
 canter est le caractère précédent car la fin du
texte est validée par une ligne vide donc
deux \n à la suite (c== canter == \n) qu'il faut éliminer*/
     int ocurlettre[26];
/*comptage des ocurrences de lettres*/
     int  numerolettre,
/*numero de la lettre dans l'alpahabet*/
        nbcaract,
/* nombre de caractères du texte*/
        nbpaslettre,
/* nombre de caractères autres que des lettres*/
        i ;

/* initialisations */
canter ='a'; /* autre chose que \n */
nbcaract = nbpaslettre =0;
for (i=0 ;i<26; i++) ocurlettre[i]=0;

/* lecture du  texte et analyse caractère par caractère*/
printf ("rentrez votre texte avec une ligne vide en fin \n  ");
while ((c=getchar!())='\n' || canter != '\n')
     {
     if   (c !='\n')
        { numerolettre = c-'a';
             if (numerolettre >=0 && numerolettre<26){
                  ocurlettre[numerolettre]++;
                  else  nbpaslettre  ++;
                                                    }
             nbcaract++;
        }
```



(suite)

```
            canter = c;

            }

    /*affichage du résultat */
    printf ("\n\n\n le mot comporte %d caractères et \n, nbcaract);
    for (i=0; i< 26 ; i++)
            printf("%d fois la lettre %c\n,ocurlettre[i],'a'+i);
            printf("\n et %d fois d'autres caractères\n, nbpaslettre);
    }
```

Exemple 23

Deuxième méthode :

```
#include <stdio.h>
#include <string.h>

main()
{
        char ligne[128];/* ligne entrée au clavier max 128
caractères par gets()*/
      int ocurlettre[26];
/*comptage des ocurrences de lettres*/
     int  numerolettre,
/*numero de la lettre dans l'alpahabet*/
        nbcaract,
/* nombre de caractères du texte*/
        nbpaslettre,
/* nombre de caractères autres que des lettres*/
        i ;

/* initialisations */
nbcaract = nbpaslettre =0;
for (i=0 ;i<26; i++) ocurlettre[i]=0;

/* lecture du  texte et analyse */
printf ("rentrez votre texte avec une ligne vide en fin \n  ");
do
    {gets(ligne) ;
     for (i=0; i<strlen(ligne) ; i++, nbcaract++)
        { numerolettre =ligne[i]- 'a';
            if (numerolettre >=0 && numerolettre<26){
                ocurlettre[numerolettre]++;
                else  nbpaslettre  ++;
                                                    }
        }
```

```
(suite)     }
        while (strlen(ligne));

        /*affichage du résultat */
        printf ("\n\n\n le mot comporte %d caractères et \n, nbcaract);
        for (i=0; i< 26 ; i++)
            printf("%d fois la lettre %c\n,ocurlettre[i],'a'+i);
            printf("\n et %d fois d'autres caractères\n, nbpaslettre);
        }
```

Remarque :

Les deux programmes ne traitent pas les actions des utilisateurs peu respectueux des règles affichées pour l'entrée des textes au clavier (lignes trop longues ou textes avec plus d'une ligne vide etc.).

Un pas supplémentaire peut être franchi dans l'analyse d'un texte si on s'attache à compter le nombre de mots contenus dans un texte. Comme dans l'exemple précédent, la fin du texte sera signalée par une ligne vide. Il nous reste à définir les séparateurs de mots ; dans cette première analyse nous considérerons les « vrais » séparateurs :

fin de ligne, espace, ponctuation(:;,.?!), parenthèses( ), guillemets " et apostrophe '. Par contre nous ne considérerons pas le tiret - séparateur de mots coupés en fin de ligne.

Pour déterminer si un mot se termine, il faut savoir si le caractère testé est un séparateur et incrémenter le compteur de mots dans le cas d'une réponse positive. Il devient indispensable d'écrire une fonction logique (testesp( )) qui renvoie la valeur vrai (1) ou faux (0) suivant le résultat du test sur le caractère.

Comme précédemment, nous allons opérer de deux manières en effectuant l'analyse soit caractère par caractère, soit ligne par ligne. Il faut penser à inclure le \ n si on travaille avec getchar( ) ou le \ 0 si on travaille avec gets( ). La logique du traitement est d'incrémenter le compteur de mots dès qu'on a trouvé un caractère différent d'un séparateur puis un caractère égal à un séparateur. Il est astucieux d'utiliser un opérateur logique (mot) qui nous dise en permanence si on est en train de traiter un mot. Il nous suffit maintenant d'avoir la démarche logique suivante :

— si mot = VRAI et caractère < > séparateur = > Ne rien faire car on est toujours dans le même mot ;
— si mot = VRAI et caractère = séparateur = > On termine un mot. Additionner un dans le compteur de mots et mettre mot = FAUX ;
— si mot = FAUX et caractère > < séparateur = > On commence un mot. Mettre mot = VRAI ;
— si mot = FAUX et caractère = séparateur = > On rajoute un séparateur à un séparateur. Ne rien faire.

Comme dans le programme précédent, la condition d'arrêt sera de trouver deux \ n pour le traitement caractère par caractère et une ligne vide pour le traitement ligne par ligne. Reportez-vous aux programmes précédents pour plus d'explications à ce sujet.



Exemple 24

Première méthode :

```
#include <sdio.h>
#define VRAI 1
#define FAUX 0

main()
{
      int testsep();
      char c,
           carant;
      int  nbremot,
           mot;

/*initialisations*/
carant ='a';
mot =FAUX;
nbremot = 0;
printf ("rentrez votre texte avec une ligne vide en fin \n  ");

while ((c=getchar!())='\n' || canter != '\n')
      { if ( testsep(c))
           { if ( mot)
                 { nbremot++ ;
                   mot = FAUX;
                 }
           }
        else mot =VRAI;
        canter = c;
      }

printf ("\n\n\n le texte a %d mots:\n, nbremot);

}

/* fonction de test de caractère*/

int testsep(c)
char(c);
{
     char separateur[12]={
/* le \n est considéré ici comme caractère séparateur */
                          '\n',
                          ' ' ' ' ' ' ' ' ' '
                          ' ' ' ' ' ' ' ' ' '
                          '?' '!' '(' ')'
                          '"' '"' };
     int i;
```

(suite)

```
        i=0 ;
        while ( c!= separateur[i] && i++<11 );
        if (i == 12) return (0) ;
                else      return (1);
    }
```

Deuxième méthode :

Exemple 25

```
#include <stdio.h>
#include <string.h>
#define VRAI 1
#define FAUX 0

main()
{
    int testsep();
    char ligne[128];
    int   nbremot,
          mot,i;

/*initialisations*/
mot =FAUX;
nbremot = 0;

printf ("rentrez votre texte avec une ligne vide en fin \n  ");

do
    {gets (ligne) ;
/* on traite le 0 de fin de ligne comme tout autre caractere.
on va donc de 0 à stren() et non à stren()-1. */

      for (i=0; i<=stren(ligne) ; i++)
        if ( testsep(ligne[i]) )
        { if ( mot)
                { nbremot++ ;
                  mot = FAUX;
                }
        }
      else mot =VRAI;

    }
while (stren (ligne) );

printf ("\n\n\n le texte a %d mots:\n, nbremot);

}
```



(suite)

```
/* fonction de test de caractère*/

int testsep(c)
char(c);
{
    char separateur[12]={
/* le 0 est considéré ici comme caractère séparateur*/
                        '\0',
                        ' ', ',', ',', ';', ':',
                        '?', '!', '(', ')'
                        '"'", '"' };
    int i;

    i=0 ;
    while ( c!= separateur[i] && i++<11 );
    if (i == 12) return (0) ;
        else    return (1);
}
```

CRÉATION D'UN PROGRAMME EN LANGAGE C

Après avoir manié les principales instructions de base en langage C, il est intéressant de bien comprendre la logique de la compilation et de la mise au point d'un programme en C. Ces différentes phases et terminologies semblent souvent obscures lorsqu'on vient du Basic.

Les trois phases principales de la réalisation d'un programme en C sont :

— L'édition

— La compilation

— L'édition de liens

L'édition

C'est la phase d'écriture du programme telle que vous la voyez dans l'ouvrage des éditions WEKA. Une partie du texte du programme est créée à partir du clavier. On parle de texte source ou programme source. La pratique courante consiste à conserver ce fichier dans un fichier intitulé « fichier source ».

La plupart du temps, en langage C, les fichiers sources porteront l'extension .C (sous DOS ou UNIX par exemple). Les autres programmes écrits pourront utiliser l'assembleur ou d'autres fonctions qui seront définies par les #include.

La compilation

La compilation consiste à traduire en langage machine interne le contenu du fichier source grâce à un programme appelé Compilateur qui fait évidemment partie du pack C que vous achèterez. Cette opération s'effectue en deux étapes, dues à la définition des directives précédées de # qui concernent spécialement le préprocesseur. Ce passage permet de produire un source sans aucune directive # puisque toutes les macros correspondantes et les prototypes des fonctions utilisées sont alors consultées par le préprocesseur. La correspondance se fait sans générer de code objet (traduction effective en langage machine) pour ces macros ou fonctions. C'est à ce premier niveau que vous pourrez discerner les erreurs d'appel et les oublis de définition des directives.

Ensuite vient la compilation proprement dite qui consiste à traduire en langage machine votre texte complet en langage C. Le module ainsi créé se nomme module objet.

L'éditions de liens

Il reste maintenant au compilateur à créer les modules objets qui correspondent aux fonctions standards des bibliothèques (printf( ), gets( )... qui sont utilisées par votre programme ainsi que tous les autres modules que vous aurez pu compiler de façon indépendante pour les regrouper au moment de la phase finale. Vu le temps que prend une compilation, il est indispensable de penser à une conception modulaire dans la mise au point d'un programme en C. C'est le rôle dévolu à l'éditeur de liens d'aller chercher dans les bibliothèques standards. Notez que ces objets standards (souvent eux aussi dans des fichiers avec extension .h) sont déjà compilés (écrits en langage machine) et rangés dans des fichiers. Il y a un nombre minimal de fonctions qui doivent être contenues dans ces fichiers ; c'est la norme ANSI *(American National Standard Institute)* qui définit ces fonctions.

A la suite de cette édition de liens, vous obtiendrez un programme indépendant de son contexte de mise au point en C et complètement exécutable en langage machine.

LES OPÉRATEURS ET LES EXPRESSIONS

Le C est certainement un des langages qui possède le plus grand nombre d'opérateurs. Nous avons déjà un peu exploré les opérateurs classiques (arithmétiques et logiques), mais il est intéressant de se pencher un peu plus sur les opérateurs de manipulation de bits, d'affectation et d'incrémentation.

Les opérations de conversion

A l'inverse du Pascal, le C permet le mélange des genres dans une expression. Il y a conversion automatique de la part du compilateur suivant la règle suivante :



Si un opérateur porte sur deux opérandes de types différents, il y a conversion du type le plus faible dans le type le plus fort. Ce principe permet de donner un sens aux expressions, comme par exemple la conversion d'un entier en réel, virgule flottante dans un calcul arithmétique : i-f.

Exemple :

```
/* définitions */
int a,b;
long c,d;
float x;
/* expressions */
a+b+c
```

Les valeurs de a et b sont converties en **long** avant d'être additionnées à c. Le résultat est de type **long**.

```
a+b*c
```

b est converti en **long** avant d'être multiplié à c. a est converti en **long** avant d'être additionné à b*c. Le résultat est de type **long**.

```
a*x+a*c
```

a est converti en **float** avant d'être multiplié à x. a est converti en **long** avant d'être multiplié à c. Ce résultat est converti en **float** avant d'être additionné au produit précédent. Le résultat est en **float**.

Par contre, les expressions qui n'auraient aucun sens (par exemple indice=float) sont purement et simplement rejetées à la compilation.

- Les types **char** et **int** peuvent se mélanger sans problème dans des expressions arithmétiques **char** étant immédiatement converti en entier. Deux cas sont à envisager suivant que le caractère est signé ou non signé.

Si l'attribut est **unsigned**, la conversion produira un nombre positif.

Si l'attribut est **signed**, la conversion produira un nombre positif ou négatif suivant le bit qui est considéré comme bit de signe (en général ce sera le bit de poids fort). Mais des configurations de bits arbitraires peuvent donner des conversions de variables caractères positives sur certaines machines et négatives sur d'autres. L'exemple le plus connu est la représentation de EOF *(End Of File)* par -1. Soit la séquence :

```
char c;
c=getchar( );
if(c==EOF)
...
```

Sur une machine sans extension de signe, c est toujours positif car de type **char**. Le test ne peut donc jamais être positif. Il faudrait ranger la valeur lue par **getchar** dans une variable de type **int**.

Les lois de conversion implicites pour le calcul sont les suivantes :
— **char** et **short** sont convertis en **int** et **float** en **double** ;
— si un des deux opérandes est **double**, l'autre est converti en **double** et le résultat est **double** ;

— si un des deux opérandes est **long**, l'autre est converti en **long**, et le résultat est **long** ;

— si un des deux opérandes est **unsigned**, l'autre est converti en **unsigned** et le résultat est du type **unsigned** ;

— sinon, les opérandes doivent être du type **int** et le résultat sera du type **int**.

Dans la compilation, les arguments qui figurent dans un appel de fonction sont différents suivant que la fonction a été déclarée ou non sous forme de prototype. (Le prototype représente le format de déclaration d'une entête de fonction et d'arguments).

Si ce prototype est présent, il précisera au compilateur le type des arguments et il y aura conversion dans le type spécifié.

Si le prototype n'est pas présent, les arguments suivront les règles précédemment énoncées (**char** en **int**, **float** en **double**). C'est le cas aussi de la fonction **printf( )** pour laquelle il y a systématiquement conversion de **float** en double. Nous verrons plus tard les erreurs qui peuvent être générées lorsque les codes de format sont différents des types des expressions à afficher.

Les opérateurs d'incrémentation et de décrémentation

L'opérateur d'incrémentation **(+ +)** ou de décrémentation **(− − −)** peut être utilisé devant ou derrière une variable.
+ + est un opérateur de préincrémentation lorsqu'il est placé à gauche de la variable à laquelle il se réfère et de postincrémentation lorsqu'il est placé à droite de la variable à laquelle il se réfère. La définition est semblable pour l'opérateur de décrémentation.
− − − est un opérateur de prédécrémentation lorsqu'il est placé à gauche de la variable à laquelle il se réfère et de postdécrémentation lorsqu'il est placé à droite de la variable à laquelle il se réfère.
Pour bien comprendre la logique de ce calcul et de l'affectation, il suffit de faire les opérations sous-jacentes de gauche à droite. **i + +** affecte à i la valeur avant l'incrémentation puis effectue l'incrémentation.

```
i=5;
c=i++-2;
```

donne comme résultat d'affectation 3 dans c et 6 dans i ;
alors que :

```
i=5;
c=++i-2;
```

donne 4 dans **c** et 6 dans i.
Cet opérateur unaire (qui ne porte que sur un seul opérande), ne peut être utilisé qu'avec des variables et une expression telle que **a=(b*c)++** n'a aucun sens.



Comme tous les opérateurs unaires ont des priorités élevées (revoir le tableau Partie 4 chapitre 4 pages 9 à 12), on peut compacter énormément les instructions, bien que leur compréhension nécessite un peu plus d'habitude. N'hésitez pas à ce moment-là à utiliser des espaces entre un opérateur et les opérandes auxquels il se rapporte. La lisibilité s'en trouvera grandement améliorée.

tab[i++]=getchar();

permet de lire le clavier et de l'affecter à l'élément i du tableau tab[ ]. L'incrémentation de 1 prépare la lecture de l'élément suivant.

Exemple 26

```
main()
{
int i=4;
int j=5;
int k=3,l;
/*l'expression :*/
l=3*i--*++j+k--;
/*est valable et calculable du fait que * a une pririté inférieure
aux opérateurs unaires.
Pour augmenter la lisibilité, les formes suivantes auraient pu
être tapées:*/
l = 3* i-- * ++j + k--;
/*simulation du calcul et de l'affectation
l = 3* 4    *    6 + 3
l = 75
i = 3
j = 6
k = 2 */
}
```

Les opérateurs de manipulation de bits

A priori, les opérateurs pouvant travailler sur le motif binaire des valeurs ne peuvent porter que sur des types entiers. Dans le cas où ils s'appliquent à des caractères, le résultat sera converti en entier.

Liste des opérateurs :

& : « et » (binaire bit à bit)

| : « ou » inclusif (binaire bit à bit)

^ : « ou » exclusif (binaire bit à bit)

>> : décalage à droite

<< : décalage à gauche

~ : complément à un (unaire bit à bit)

Exemples :

Le & (et) est souvent utilisé pour mettre à zéro des bits suivant un masque donné. Ainsi, l'expression :

a=b & OxFF (valeur hexadécimale)

permet de ne prendre en compte que les huit bits de droite de b.

De même :

a=b & O177 (valeur octale)

met à zéro tous les bits de b sauf les 7 bits de poids les plus faibles.

L'opérateur de bits | (ou) sert à mettre des bits à un.

l'expression :

a=a | O55

met à un dans a tous les bits qui étaient à un dans O55 à savoir les bits 10, 12, 13, 15.

Les autres bits gardent leur valeur d'origine.

Il faut soigneusement distinguer les opérateurs & et | des opérateurs de relation logique && et || qui analysent une expression booléenne de gauche à droite en cherchant une valeur vraie.

Si x est égal à 2 et y à 5, x & y vaut 0 (aucun bit égal à un sur des positions égales dans leur codage binaire), alors que x && y vaut 1 (proposition vraie car x et y sont tous deux différents de 0 et donc vrais).

Les opérateurs de décalage > > et < < effectuent des décalages à droite et à gauche sur l'opérande située à gauche du symbole d'un nombre de bits égal à l'opérande de droite. Ainsi :

a > > 5 décale le contenu de a de 5 bits vers la droite. Si a n'est pas signé, les 5 bits de gauche sont remplis par des 0 alors que si a est signé, on remplira ces bits par des bits de signe. Dans le cas du décalage à gauche, des 0 sont générés sur les positions décalées.

L'opérateur unaire ~ est utilisé généralement dans des expressions du type : a & ~ OxFF où l'on veut masquer par exemple les huit derniers bits en les mettant à zéro. Cette forme est indépendante de la longueur de la variable puisque la constante sera codée et convertie au moment de la compilation.

Voici un exemple qui teste la parité d'un entier :



Exemple 27

```
#include <stdio.h>
main()
{
    int i ;
    printf ("entrez un entier: ");
    scanf ("%d", &i) ;
    if ( i & 1 )
        printf ("entier pair");
      else
        printf ("entier impair");
}
```

Remarque :

De manière semblable à **printf**, la fonction **scanf** possède un format en argument. Il est écrit sous forme de chaîne de caractères (ici **%d**) ce qui correspond à une valeur entière. La fonction **scanf** doit être capable de ranger les valeurs lues dans des emplacements de mémoires précis correspondant à l'adresse de la variable i, ce qui explique l'utilisation de **&** devant le i. Cet opérateur signifie « adresse de ». Si vous oubliez le symbole **&**, la fonction **scanf** rangera les valeurs lues dans des emplacements aléatoires de mémoire.

Les opérateurs d'affectation élargie

Nous avons vu précédemment comment les opérateurs d'incrémentation et de décrémentation permettaient de simplifier l'écriture des affectations. Des expressions comme i = i – 2 dans lesquelles il y a calcul sur une variable et réaffectation du résultat dans la même variable peuvent être écrites de façon plus concise sous la forme : i – = 2 en utilisant un opérateur d'affectation tel que – =.

Les opérateurs binaires arithmétiques et de manipulation de bits possèdent un opérateur d'affectation et permettent une écriture plus concise des instructions.

Les affectations de la forme :

lvalue = lvalue opérateur expression

sont équivalentes à la forme :

lvalue opérateur = expression

en voici la liste complète :

+ = – = * = / = % = l = ^ = & = < < = > > =

Attention à ne pas confondre < = ou > = avec un opérateur d'affectation élargie !

L'opérateur séquentiel

La notion d'expression est très générale dans le langage C. Cet opérateur séquentiel (,) permet de faire plusieurs calculs successifs dans une même expression.

```
i--,j=i+k
```

postdécrémente i et évalue ensuite l'affectation j=i+k. La valeur de i utilisée est celle de i après la décrémentation. On peut utiliser cet opérateur dans une suite séquentielle de plusieurs expressions :

```
i--,j=i+k,j++
```

Les deux écritures suivantes sont équivalentes :

```
i--,j=i+k,j++;
i--,j=i+k;j++;
```

UTILISATION DES STRUCTURES

Après avoir effectué des exercices faisant appel aux instructions de base du langage C, nous allons maintenant faire intervenir les tableaux de structures et leur initialisation, une des particularités du langage C.

Soit un programme qui se propose de traduire en langage morse un texte entré au clavier. Les caractères reconnus et traduits seront les 26 lettres de l'alphabet, les chiffres de 0 à 9, le point et la virgule, soit 38 symboles. Les autres symboles de ponctuation comme le point virgule ou l'espace seront ignorés et affichés comme une succession de points d'interrogation. La structure que nous allons définir contiendra un caractère et un pointeur sur une chaîne de caractères qui contiendra le code morse correspondant. Une seconde structure contiendra les 38 emplacements de taille 3 (un pour le caractère et deux pour le pointeur).

L'emplacement mémoire des chaînes est réservé à la compilation, l'initialisation étant faite au moment de la déclaration. Chaque chaîne n'occupera donc en mémoire que la place qui lui est nécessaire. Le programme doit établir la correspondance entre un code morse et un caractère entré au clavier. Chaque élément ou structure du tableau contiendra un caractère et le code morse correspondant sous forme d'une chaîne de caractères. Pour coder le caractère, il suffira de le localiser dans le tableau liste[ ].

La concision et la logique du langage C permettent de rechercher le symbole dans le tableau liste[ ] uniquement par la seule instruction : **while (ligne[i]! = liste[j].symbole && j++ < Nbcaract-1);**

Cette dernière instruction possède un avantage évident de concision et permet de savoir directement si la recherche de l'élément a été fructueuse ou non. Souvenez-vous que l'opérateur && (et logique) n'évalue le second membre de l'opération logique que lorsque c'est nécessaire. La postincrémentation de j n'aura donc lieu que si le premier membre de l'expression est vrai, c'est-à-dire si on n'a pas trouvé le symbole dans la liste



(ligne[i] doit être différent de liste[j].symbole). Dans le cas contraire, on va incrémenter j et sortir de la boucle while pour passer aux impressions de symboles. Notez bien que cette remarque est valable quel que soit le compilateur que vous utiliserez en C, puisque c'est une spécificité structurelle du langage.

Exemple 28

```
#include <stdio.h>
#include <string.h>
#define  Nbcaract  38        /*nbre de caractères codés*/
main()
{
    struct caract { char symbole ;
                    char * morse ;
                  } ;
    struct  caract  liste[  Nbcaract  ]=
/* liste des codes morse */
                  {  'A', "._",      'B', "_...",    'C', "_._.",
                     'D', "_..",     'E', ".",       'F', ".._.",
                     'G', "__.",     'H', "....",    'I', "..",
                     'J', ".___",    'K', "_._",     'L', "._..",
                     'M', "__",      'N', "_.",      'O', "___",
                     'P', ".__.",    'Q', "__._",    'R', "._.",
                     'S', "...",     'T', "_",       'U', ".._",
                     'V', "..._",    'W', ".__",     'X', "_.._",
                     'Y', "_.__",    'Z', "__..",    ' ', "_____",
                     ',', "__..__",  '0', "_____",   '1', ".____",
                     '2', "..___",   '3', "...__",   '4', "...._",
                     '5', ".....",   '6', "_....",   '7', "__...",
                     '8', "___..",   '9', "____."
                  };
    char  ligne[128];       /* lecture de ligne clavier */
    char code[7];
    int i, j ;
              /* lecture de la phrase à traduire */
    ptintf ("tapez votre phrase sur une ligne: \n");
    gets (ligne);
    printf ("\n\n\n voici votre phrase en Français \n");
          /* la traduction se fait symbole par symbole*/
    i=0;
    while (i<strlen(ligne))
        {
/* on teste les espaces pour les ignorer dans l'analyse*/
            while (ligne[i] ==' ') i++ ;
            if (i< strlen(ligne)) /* avant fin de ligne */
               { sscanf (&ligne[i], "%6s",code);
                 i +=strlen(code);
                 j =0;
/* test et comparaison des chaînes*/
                 while  (stricmp  (code,  liste[j].symbole)  &&
                         j++<Nbcaract-1);
```

(suite)

```
    /* le symbole est trouvé*/
                    if (j<Nbcaract) printf ("%2c",liste[j].symbole);
    /* le symbole n'est pas trouvé*/
                            else printf (" ?");
                    }
            }
    }
```

L'exemple suivant est la réciproque du précédent. Il doit permettre de réaliser le décodage d'un message écrit en morse. Les symboles décodés seront le point et le tiret. Les symboles seront séparés par des espaces à volonté et l'ensemble du message devra tenir sur une ligne. Nous allons utiliser la structure de programme précédente. Cette fois, pour reconnaître le symbole, il ne nous faudra plus explorer la partie caractère, mais la partie chaîne du tableau ; toutefois la logique de recherche pourra rester la même. Le seul problème supplémentaire qui se pose est de pouvoir accéder à des symboles de longueur variable (regardez le tableau **symbole[ ]**) et séparés par un nombre variable de symboles d'espace. Si nous remarquons qu'aucun code n'excède la longueur 6, il nous suffira de lire une chaîne de longueur maximale 6 grâce à **scanf** avec le code format **%6s**. Ce code interrompt l'exploration de la suite des symboles dès qu'on atteint 6 symboles ou que l'on rencontre un séparateur. Dans cet exemple, nous supposerons que l'utilisateur ne donne aucun code de plus de 6 symboles. Pour parer à une telle éventualité, il suffirait de vérifier que chaque code analysé est bien suivi d'un espace ou d'un symbole de fin de chaîne. Par souci de clarté de présentation, notez que les symboles décodés le sont tous sur deux positions, la première étant un espace (format **%2c**) dans le **printf**. La logique de recherche au moment de la comparaison est la même que dans l'exemple précédent. Rappelons que la fonction **stricmp** (comparaison de chaînes) compare sans tenir compte des majuscules ou minuscules (pas utile dans l'exemple présent) les deux chaînes dont les adresses sont fournies en argument. Cette fonction restitue une valeur non nulle si les deux chaînes sont différentes et la valeur nulle si elles sont égales. Ces deux valeurs numériques sont interprétées dans l'exemple comme les valeurs vrai et faux.



Exemple 29

```
#include <stdio.h>
#include <string.h>
#define  Nbcaract  38        /*nbre de caractères codés*/

main()
{
    struct caract { char symbole ;
                    char * morse ;
                  } ;
    struct  caract  liste[  Nbcaract  ]=
 /* liste des codes morse */

                  {  'A',  ".-",        'B',  "-...",       'C', "-.-.",
                     'D',  "-..",       'E',  ".",          'F', "..-.",
                     'G',  "--.",       'H',  "....",       'I', "..",
                     'J',  ".---",      'K',  "-.-",        'L', ".-..",
                     'M',  "--",        'N',  "-.",         'O', "---",
                     'P',  ".--.",      'Q',  "--.-",       'R', ".-.",
                     'S',  "...",       'T',  "-",          'U', "..-",
                     'V',  "...-",      'W',  ".--",        'X', "-..-",
                     'Y',  "-.--",      'Z',  "--..",       '.', ".-.-.",
                     ',',  "--..--",    '0',  "-----",      '1', ".----",
                     '2',  "..---",     '3',  "...--",      '4', "....-",
                     '5',  ".....",     '6',  "-....",      '7', "--...",
                     '8',  "---..",     '9',  "----."
                  };

    char  ligne[128];        /* lecture de ligne clavier */
    int i, j ;

                /* lecture de la phrase à traduire */
    ptintf ("tapez votre phrase sur une ligne:  \n");
    gets (ligne);
    printf ("\n\n\n voici votre phrase en morse \n");
/* la traduction se fait symbole par symbole*/
    for (i=0; i<strlen(ligne); i++)
        { j=0;
            while (ligne[i] != liste[j].symbole && j++< Nbcaract-1);
            if (j<Nbcaract) printf ("%7s",liste[j].morse);
                else printf ("?????");
/* provoquer un retour à la ligne tous les huit */
/* symboles par l'opérateur % modulo " */
            if ( !((i+1)%8))printf "\n");
        }

}
```

4/4.3

Exemple de programme en C

Ce programme reprend un exemple qui avait été donné en Basic GFA sur la génération automatique d'accompagnement d'après la lecture d'une grille. La logique d'analyse est restée la même ainsi que l'algorithme de fonctionnement. Veuillez vous reporter à l'article précédent pour plus de détails (Partie 6 chapitre 1.5). La partie du programme qui sélectionne les motifs musicaux aléatoirement est sensiblement la même en C qu'en Basic et nous en donnerons un exemple ultérieurement, la fonction **switch** faisant office de select en Basic. La construction des motifs est la même, et le programme correspondant ne reviendrait presque qu'à remplir du listing pour le plaisir (ou presque) ce qui n'est pas notre but. Le rôle ingrat de cette opération est dévolu à la fonction **pfct** définie comme fonction externe. C'est la fonction **generer( )** qui crée vraiment la chaîne d'accompagnement en choisissant les motifs sur une grille, soit 48 symboles maximum sur un accompagnement. En termes de programmation pure cela revient à distribuer suivant la génération d'une fonction externe **(pfct)**, avec certains critères de choix aléatoires, une suite de symboles chaînes de caractères.

Le début du programme définit les structures nécessaires. La majorité des variables définies ensuite ont les mêmes significations et appellations que dans le programme en Basic.

Les nouvelles variables **argts[ ]**, **nargts** introduisent la possibilité de regrouper les patterns à la suite les uns des autres dans un mode arrangement, ce nombre maximal étant fixé par MAXARRANG.

Le nom du pattern initial est défini à partir des 4 premiers paramètres de la structure PATERN : 4 temps, 2 mesures, style 0 (ou premier style des choix de style), nom (max 10 caractères). Les autres variables sous le label variables spécifiques ont le même sens qu'en Basic. La fenêtre de présentation est simplement repérée de façon supplémentaire pour son handle et ses coordonnées. Les tableaux d'accords et de tensions représentent successivement les successions d'intervalles par rapport à la tonique des accords courants et les tensions permises sur ces accords (8 de chaque).

La première valeur représente le nombre d'éléments codés, chose qu'on ne peut faire aussi facilement en Basic.

Les fonctions utilitaires définies au début du traitement correspondent aux procédures de tirage aléatoire dans une tranche distribuée entre deux valeurs comprises entre 0 et 99. Ces zones visent à affecter une probabilité différente à chaque tension, ou en termes plus théoriques fixer le critère de choix de chaque élément du tableau **tabtension[ ]** par une fonction de densité de probabilité.

La fonction **motdep** (duree, transpo) effectue une translation verticale sur l'accord courant repéré dans la variable **ring**. La recopie en changeant le paramètre temps évite de refaire le calcul d'une cellule similaire sur une position décalée physiquement.

La fonction **generer( )** teste toutes les positions de la grille mais à la différence du programme en Basic n'initialise une solution accompagnée en mesure (4 temps successifs d'un même accord, ou dans un autre contexte 4 caractères semblables qui formeront une seule entité) qu'en début de mesure. Dans ce programme, si l'on trouvait Do Majeur entre les 3e et 4e temps de la première mesure et les 1er et 2e temps de la deuxième mesure, on analyserait comme deux suites d'accords de deux temps chacune. Ce n'est pas dû, rassurez-vous, à la différence des deux langages mais à l'évolution du concepteur du programme. Pour toutes les mesures et pour tous les temps, le principe d'analyse et de génération de silence noire est équivalent au traitement en Basic. C'est peut-être le passage du programme le plus accessible aux débutants en C qui ont déjà programmé en Basic GFA ou OMIKRON.

Les dernières fonctions du programme :

surgrille( ) Calcule les rectangles temps à l'intérieur de chaque grille.

lagrille( ) Calcule la position du curseur correspondant au nombre de temps correspondants.

afflepas( ) Inscrit le nom de l'accord ou du silence sur la case curseur à condition de ne pas être déjà sur un symbole d'état courant (variable **ring**). Dans le cas contraire on affiche du silence. Cette fonction appelle la fonction précédente pour afficher au bon emplacement.

grille Initialise une nouvelle grille en appelant avec le mode **video** égal à 0 (effacer) **lagrille( )** et **afflepas( )** pour toutes les cases possibles.



GENERATION DE SUITE DE MOTIFS(DEBUT)

```
#include <stdio.h>
#include <osbind.h>
#include <gemlib.h>
#include <stdlib.h>
#include <portab.h>
#include "gemexten.h"

                /* ACCES EXTERNES */

extern WORD hwhandle, xh,yh,wh,hh,xdesk,ydesk,wdesk,hdesk;
extern WORD handle;
extern draw_man();
extern char texte[] ;
extern OBJECT *menu;
extern WORD hlettre;
extern WORD a_couleur;
extern int whopn;
extern WORD mesure;
extern UBYTE *remonte( NSYMB *);
extern NSYMB *nosymb( );
extern WORD typmes;
extern char *textes[];
#define TEXT( i ) textes[i]


#define RANDOM(x ) (Xbios( 17 ) % x)

#define MAXNBM 12
#define MAXSTEP 4
#define MAXPATTERN 16
#define MAXARRANG  16
#define AIDE( n ) if( vmanuel ) { manuel( n ); break;}

#define AFFPATNOM( i )
sprintf(texte ,"%8s",tabpat[i].patname);
\objc_text( root,tabinpat[i],texte,(WORD)1)

#define AFFARGMNT( i )
sprintf(texte,"%8s", tabpat[argts[i]].patname);\
objc_text( root, tabinarg[i],texte ,(WORD)1)

#define EFFARGMNT( i )
objc_text( root, tabinarg[i],tirets,(WORD)1)

#define NSA bufpat.nsa
#define NSM bufpat.nsm
#define ACC_COURANT bufpat.accomp[NSA]
#define MELO_COURANT bufpat.melo[NSM]
```

(suite)

```
#define TEMPS           bufpat.temps
#define NBMESURE        bufpat.nbm
#define STYLE           bufpat.style
#define PATNAME         bufpat.patname
#define TON_COURANT( i ) bufpat.tabstep[i].ton
#define TYP_COURANT( i ) bufpat.tabstep[i].typ
#define GENERE          bufpat.ok
#define POUR_TOUTES_LES_MESURES( i ) for( i=0;i<NBMESURE;i++)
#define POUR_TOUS_LES_TEMPS( i ) for( i=0;i<4;i++)
#define PAUSE( x ) evnt_timer( x & 0xFFFF,x>>16 )

#define BONTON( x )  (60+((((x)%12)+1)%12))


#define  ARRET( i )      motdep( i ,128-hauteur_acc )

#define DBLE_CR         RESOLUTION/4
#define CROCHE          RESOLUTION/2
#define NOIRE           RESOLUTION
#define TRIOLET_CROCHE RESOLUTION/3



typedef struct {
        BYTE ton;
        BYTE typ;
        } GSTEP;

typedef struct {
        BYTE temps;      /* BIN/TER */
        BYTE nbm ;      /* nombre de mesure */
        BYTE style;
        char patname[10];
        GSTEP tabstep[MAXNBM*MAXSTEP];
        WORD  nsa;
        NSYMB * accomp;
        WORD nsm;
        NSYMB * melo  ;
        WORD ok ;
        BYTE  chksum  ;
        } PATERN ;


typedef struct{
   BYTE nbn;
   BYTE interv[11];
   } ACCORD ;
```



(suite)

```
static WORD argts[MAXARRANG];
static WORD nargts ;
static PATERN tabpat[MAXPATTERN] ;
static nombre_pattern ;
static PATERN patinit  ={4,2,0,"PASDENOM" };

PATERN bufpat ;

UBYTE  hauteur_acc ;
ACCORD *accour ;
WORD accmes ;
WORD prem_haut ;
extern WORD tempo ,tempobeat;
#define TEMPOBEAT 3840

            /* VARIABLES SPECIFIQUES */

static WORD tonique,type;
static char tirets[]="------";
static WORD awhandle;
/* Handle de fenetre */
static WORD axh,ayh,awh,ahh;
 /* Coordonnées de fenêtre */
static OBJECT *root;
 /* Objet */
static int dernote;

#define MAXTONIQUE 22
static char **toniques = &textes[82];

#define MAXSTYLE 12

static char ** styles = &textes[70];

#define MAXLMES 4
static BYTE tabmesures[]={ 2,4,8,12 };

#define MAXTYPE 9
#define RUNNING 9

char **types = & textes[ 61 ];

#define SIL 8
```

(suite)

```
ACCORD tabaccord[]={
        {2,4,7},
        {2,3,7},
        {2,3,6},
        {2,4,8},
        {2,5,7},
        {3,4,7,10},
        {3,3,7,10},
        {3,3,6,9}
        } ;

ACCORD tabtension[]={
        {6,0,2,4,7,9,12},
        {7,0,2,3,5,7,10,12},
        {7,0,3,5,6,8,10,12},
        {6,0,2,4,8,10,12},
        {7,0,2,5,7,9,10,12},
        {7,0,2,4,7,9,10,12},
        {7,0,2,3,5,7,10,12},
        {7,0,3,4,6,8,9,12}
        };

/*fonction de génération de suites de motifs*/
extern void ( *pfct[4][12])();

#define NOM_D_ACCORD(i) types[i]
#define TAIL_ACC(i)  tabaccord[i].nbn

static BYTE tabpas[MAXSTEP*MAXNBM];
/*nbre de temps et nbre de mesures de la grille*/

/* static char *sortie[]={"MIDI","ATARI"}; */
static WORD ring;
static WORD comptsil;

cfree( s ) char *s;
{
free( s );
}
char *cmalloc( t ) unsigned t;
{
char *s;
s = malloc( t );
return( s );
}
```



(suite)

```
abs( a ) int a ;
{ if( a >= 0) return( a );
  else return( -a );
}

static tirage( )
{
int tirage,i;

tirage=random( 100 );
for( i = 0;i<tabtension[ring].nbn;i++){
      if( tirage <= proba[ring][i] ){
            break;
            }
      }
return( i );
}

/*
    cree un symbole de hauteur tonique ( tranpsosée )
    longueur durée à l'heure courante
    dans accomp
*/
void motdep( duree, transpo )
int duree , transpo;
{
    register NSYMB *nb = &ACC_COURANT ;

    if( NSA > 0) nb->heure = (nb-1)->heure+(nb-1)->duree;
    else nb->heure = 0L;
    NOTE( nb->note ) =  ( UBYTE ) (hauteur_acc + transpo);
    VELO( nb->note ) = 80;
    nb->duree = duree ;
    NSA ++;
}

/*
     analyse TYP_COURANT && TEMPS_COURANT et crée la chaîne
     accomp du bufpat
*/
static void generer( )
{
register WORD t;
register WORD temps_joue ;
WORD longacc = 0;
WORD ok = 0;
```

(suite)

```
/* allouer 4 * 48 +1 symboles ( maximum ) pour acc  */
if( bufpat.accomp )

        free((char *) bufpat.accomp);

bufpat.accomp = ( NSYMB *)malloc( (1+4*(NBMESURE*TEMPS))*
                    sizeof( NSYMB ));
NSA = 0;
comptsil = 0;

if( ! bufpat.accomp ) erreur_fatale( 1 );

prem_haut = 0;
hauteur_acc = 128;
ring  = SIL ;
dernote = 0;


POUR_TOUTES_LES_MESURES( accmes ){
    POUR_TOUS_LES_TEMPS( t ) {
        temps_joue = accmes*4+t;
        if( ! tabpas [ temps_joue ] ) break ;


        if( TYP_COURANT( temps_joue ) == RUNNING ){
            if( ring == SIL ) {
                ARRET( NOIRE );


                }
            else
                longacc  ++;
            }
        else{
            if( TYP_COURANT( temps_joue ) == SIL ){
                if( longacc ) {
                  /*on appelle la génération des motifs*/
                    (*pfct[longacc-1][STYLE])();
                    longacc = 0;
                    }
                ring = SIL;
                ARRET( NOIRE );

                }
            else{
                ok = 1;
                if( longacc ){
                    (*pfct [longacc-1][STYLE])();
                                        }
```



(suite)

```
                        longacc = 1;
                        hauteur_acc = BONTON( TON_COURANT( temps_joue ));
                        if ( ! prem_haut ) prem_haut = hauteur_acc ;
                        ring = TYP_COURANT( temps_joue );
                        accour = &tabaccord[ring];
                        }
                    }
                }

        if( longacc ){
            (*pfct[longacc-1][STYLE])();
                        longacc = 0;
            }
        }

    /* baliser avec FIN */
    hauteur_acc = FIN;
    motdep( 0,0);

    /* reallouer la taille réelle */

    bufpat.accomp = ( NSYMB *)realloc((char *)bufpat.accomp ,
    (NSA)*sizeof( NSYMB ));

    if( ! bufpat.accomp ) if( NSA ) erreur_fatale( 1 );
    else bufpat.accomp = nosymb();

    bufpat.ok = ok ;

    }

    static WORD surgrille ( x1,y1)
    WORD x1,y1;
    {
    WORD g = (NBMESURE * 4) -1;
    WORD x , y;
    WORD m,t ;

    while ( g >= 0){
        if ( tabpas[g] ){

            /* calcul des x,y de lagrille du pas */
            x = 150;
            m = g/4;
            t = g%4;

            if( m & 1 ) x += 5+4*60;
    /* 2ème colonne */
```

(suite)

```
            x += 60*t;
            y =  70 +  ( (30+6)  *  (m/2));

            if ( x1>=x && x1<=x1+60 && y1>= YCOORD(y)
                && y1<= YCOORD(y+30))
                    break;
            }
        g --;
        }
return( g );
}

static void lagrille( t , mode )
WORD t,mode;
{
register WORD m,j;
WORD x,y,w,h;

w = 60;
h = 30;

m = t/4;
/* t = 4 = > mesure 1 */
j = t%4;
/*  t  =  4  ->    j    = 0;  on passe à la mesure suivante*/

if( m < NBMESURE ){

     MOUSE_OFF;

     x = 150 ;
     if( m  &  1 ) x  += 5+4*w;

/* 2ème colonne */

     x += w*j;
     y = 70 + ( (h+6)  *  (m/2));

     if( j < TEMPS )
          box (x, y,w,h);
     MOUSE_ON;

     }

}
```



(suite)

```
static void afflepas( lepas,mode )
WORD lepas,mode ; /* mode video */
{

WORD x,y;
register WORD i;
register WORD pas = lepas;

if(tabpas[ pas] ) {
     switch( i = TYP_COURANT( pas )){
          case RUNNING :
               strcpy(  texte  ,  "          " );
               break;
          case SIL  :  /* Silence */
               sprintf( texte,"%-2s%-3s","  ",types[i] );
               break;
          default :
               sprintf( texte,"%-2s%-3s",
toniques[TON_COURANT(pas)], types[i]);
               break;
          }

     y = 90 + 36*(pas/8);
     pas %= 8;
     if( pas >= 4 ){
          x = 245;
          pas -=4;
          }
     else x = 0;
     x += 155 + 60*pas;
    MOUSE_OFF;
    lagrille( lepas,mode );
    if( mode ) vswr_mode( handle, 4);
    v_gtext( handle, x , YCOORD(y),texte);
    if( mode ) vswr_mode( handle,1);
    MOUSE_ON;
    }
}

static void grille()
{
register WORD j;

/* effacer */
eff_box (150,YCOORD(70),490,YCOORD(210));
MOUSE_OFF;
for( j = 0;j<48;j++){
     lagrille( j,0 );
     afflepas( j,0 );
     }
MOUSE_ON;
}
```



4/4.3.1

Exemple de programme en C *(suite)*

Ce programme est le complément de l'exemple de programme en C paru précédemment. Nous n'avons donné que quelques exemples témoins d'agencements différents de motifs pour que vous saisissiez bien les définitions des structures en C. Les fonctions qui génèrent les motifs doivent choisir à chaque pas d'intervalle le ième élément du tableau **interv[]** ou dans le motif qui sélectionne aléatoirement le numéro de l'élément **interv[(random(accour→nbn)]**. Les fonctions **MINMAJ** et **QMINMAJ** permettent de choisir entre deux éléments, suivant la valeur précédente de **interv[0]**. La répétition de cette fonction de choix en rend la définition de fonction presque obligatoire par un define. La résolution de base contient 96 divisions, ce qui correspond à la précision maximale de l'unité de base. Les divisions entières par 2, 3, 4 sont ensuite définies afin de pouvoir servir de base de comptage dans les agencements de motifs par la fonction **motdep(i,accour→interv[i])**, chaque division étant associée à un élément du tableau **interv[]**. La sélection du motif à choisir se fait par l'intermédiaire du tableau **pfct[]** à deux indices dont le premier représente la longueur du motif (de 1 à 4), en nombre de divisions de l'unité de base, et le second le nombre de catégories différentes (12). Les variables qui fixent le choix dans le tableau sont **longacc** et **STYLE**.

Ensuite viennent les définitions des fonctions utilisées. Pour respecter la structure et les définitions des dimensions définies précédemment, il faudra en définir 48 ou remplacer 12 par 3 dans l'instruction **void (*pfct[4][12])()**= et ne garder dans la liste que les trois fonctions définies dans le listing à titre d'exemple. Reprenez la logique du programme en Basic GFA référencé en début de l'article 4/4.3 pour compléter le programme le cas échéant. La ventilation des divers choix à l'intérieur d'une fonction se fait par la fonction **switch** qui est l'équivalent du **select** du Basic. Le random est directement celui donné par le XBIOS(17), le même qu'en Basic, mais vous pouvez définir une autre fonction de génération de nombres au hasard par un algorithme de votre choix. Toutes les bibliothèques en contiennent qu'il vaut mieux d'ailleurs tester avant par un **printf()**. Reportez-vous aux articles traitant du C par l'exemple pour en voir une application pratique.

LISTING DU PROGRAMME EN C

```
#include <portab.h>

typedef struct {
        BYTE ton;
        BYTE typ;
        } GSTEP;

typedef struct {
        BYTE temps;       /* BIN/TER */
        BYTE nbm ;      /* nombre de mesures */
        BYTE style;
        char patname[10];
        GSTEP tabstep[ 48 ];
        WORD  nsa;
        char * accomp;
        WORD nsm;
        char * melo  ;
        WORD ok ;
        BYTE  chksum  ;
        } PATERN ;

extern long Xbios( );
extern void insere( int,int);
extern PATERN bufpat ;

extern UBYTE hauteur_acc;
extern WORD accmes ;
extern WORD prem_haut ;

#define RESOLUTION 96
#define RANDOM( x )  (Xbios( 17 ) % x)

#define  TIERCE( i )       motdep(i,accour->interv[0])
#define  QUINTE( i )       motdep(i,accour->interv[1])
#define  SEPTIEME( i )     motdep(i,accour->interv[2])
#define  MOTIF(i )         motdep(i,accour->interv[random( accour->nbn)])
#define TROIS_NOTES    accour->nbn==2
#define  MINMAJ( i )       if( accour->interv[0] == 3)\
                               motdep(i,5);\
                           else \
                               motdep(i,9)

#define  QMINMAJ( i )      if( accour->interv[0] == 3)\
                               motdep(i,8);\
                           else \
                               motdep(i,9)

#define  ARRET( i )        motdep( i ,128-hauteur_acc )
#define  DOUBLE            RESOLUTION/4
```



(suite)

```
#define CROCHE          RESOLUTION/2
#define NOIRE           RESOLUTION
#define TRIOLET         RESOLUTION/3

typedef struct{
        BYTE nbn;
        BYTE interv[11];
        } ACCORD ;

extern ACCORD *accour ;

static  void        joue1acc1(),joue1acc2(),joue1folk1(),joue1folk2(),

joue1disco1(),joue1disco2(),joue1latin1(),joue1latin2(),
                    joue1rock1(),joue1rock2(),joue1slow(),joue1valse();

static  void        joue2acc1(),joue2acc2(),joue2folk1(),joue2folk2(),

joue2disco1(),joue2disco2(),joue2latin1(),joue2latin2(),
                    joue2rock1(),joue2rock2(),joue2slow(),joue2valse();

static  void        joue3acc1(),joue3acc2(),joue3folk1(),joue3folk2(),

joue3disco1(),joue3disco2(),joue3latin1(),joue3latin2(),
                    joue3rock1(),joue3rock2(),joue3slow(),joue3valse();

static  void        joue4acc1(),joue4acc2(),joue4folk1(),joue4folk2(),

joue4disco1(),joue4disco2(),joue4latin1(),joue4latin2(),
                    joue4rock1(),joue4rock2(),joue4slow(),joue4valse();

void        (*pfct[4][12]) ()={
        {joue1acc1,joue1acc2,joue1folk1,joue1folk2,
         joue1disco1,joue1disco2,joue1latin1,joue1latin2,
         joue1rock1,joue1rock2,joue1slow,joue1valse
         },
        {joue2acc1,joue2acc2,joue2folk1,joue2folk2,
         joue2disco1,joue2disco2,joue2latin1,joue2latin2,
         joue2rock1,joue2rock2,joue2slow,joue2valse

        },
        {joue3acc1,joue3acc2,joue3folk1,joue3folk2,
         joue3disco1,joue3disco2,joue3latin1,joue3latin2,
         joue3rock1,joue3rock2,joue3slow,joue3valse
        },
        {joue4acc1,joue4acc2,joue4folk1,joue4folk2,
         joue4disco1,joue4disco2,joue4latin1,joue4latin2,
         joue4rock1,joue4rock2,joue4slow,joue4valse
        }
};
```

(suite)

```
/* pour se brancher sur la fonction voulue :

    (*pfct[longacc][STYLE])() ;
il faudra définir toutes les fonctons qui manquent
dans la liste ci-dessous en prenant modèle sur la structure
et en créant vos propres suites de motifs
*/

static void joue1acc1()
{
switch( random( 3 )){
      case 0:
      case 1:
            motdep( DOUBLE,0);
            TIERCE(DOUBLE);
            QUINTE(DOUBLE);
            if( TROIS_NOTES )
                  TIERCE( DOUBLE );
            else
                  SEPTIEME( DOUBLE );
            break;

      case 2:
            motdep( DOUBLE ,0 );
            TIERCE( DOUBLE );
            MOTIF ( DOUBLE );
            if( TROIS_NOTES )
                  QUINTE( DOUBLE );
            else
                  SEPTIEME(DOUBLE);
            break;

      }

}
static void joue2acc1()
{
WORD a;
switch( a=random( 3)){
      case 0:
            joue1acc1();
            joue1acc1();
            break;
      default:
            motdep( DOUBLE ,0);
            TIERCE( DOUBLE );
            QUINTE( DOUBLE );
```



(suite)

```
            if( a == 1)
                TIERCE( DOUBLE );
            else
                motdep( DOUBLE ,12);
            if( TROIS_NOTES )
                motdep( DOUBLE ,12);
            else
                SEPTIEME( DOUBLE );
            QUINTE( DOUBLE );
            TIERCE( DOUBLE );
            QUINTE( DOUBLE );
            break;
        }
}

static void joue3acc1()
{
switch( random( 3 )){
    case 0:
        joue1acc1();
        joue1acc1();
        joue1acc1();
        break;
    case 1:
        joue2acc1();
        joue1acc1();
        break;
    case 2:
        joue1acc1();
        motdep( DOUBLE ,0);
        motdep( DOUBLE ,2);
        TIERCE( DOUBLE );
        QUINTE( DOUBLE );
        motdep( DOUBLE ,6);
        if( TROIS_NOTES )
            motdep( DOUBLE ,12);
        else
            SEPTIEME( DOUBLE );
        QUINTE( DOUBLE );
        TIERCE( DOUBLE );
        break;
    }
}
static void joue4acc1()
{
```

(suite)

```
    switch( random(3)){
        case 0:
            joue2acc1();
            joue2acc1();
            break;
        case 1:
            joue3acc1();
            joue1acc1();
            break;
        case 2:
            motdep( DOUBLE ,0);
            motdep( DOUBLE ,2);
            TIERCE( DOUBLE );
            QUINTE( DOUBLE );
            motdep( DOUBLE ,6);
            if( TROIS_NOTES )
                motdep( DOUBLE ,12);
            else
                SEPTIEME( DOUBLE );
            motdep( DOUBLE ,12);
            MOTIF ( DOUBLE );
            joue2acc1();
            break;
        }
    }



    static void joue1folk10
    {
    switch( random(6)){
        case 0:
            motdep(CROCHE,0);
            TIERCE( DOUBLE );
            QUINTE( DOUBLE );
            break;

        case 1:
            motdep( DOUBLE ,0);
            TIERCE( DOUBLE );
            QUINTE( CROCHE);
            break;

        case 2:
            motdep( DOUBLE ,0);
            QUINTE( DOUBLE );
            TIERCE( DOUBLE );
            QUINTE( DOUBLE );
            break;
```



(suite)

```
        case 3:
                motdep( DOUBLE ,0);
                motdep( DOUBLE ,2);
                TIERCE( DOUBLE );
                QUINTE( DOUBLE );
                break;

        case 4:
                motdep( DOUBLE ,0);
                TIERCE( DOUBLE );
                QUINTE( DOUBLE );
                motdep( DOUBLE ,9);
                break;

        case 5:
                motdep( DOUBLE ,0);
                TIERCE( DOUBLE );
                QUINTE( DOUBLE );
                if( TROIS_NOTES )
                        motdep( DOUBLE ,12);
                else
                        SEPTIEME( DOUBLE );
                break;

        }
}

static void joue2folk1()
{
switch( random(3)){
        case 0:
                joue1folk1();
                joue1folk1();
                break;

        case 1:
        case 2:
                motdep(CROCHE,0);
                TIERCE( DOUBLE );
                QUINTE( DOUBLE );
                QUINTE( DOUBLE );
                TIERCE( DOUBLE );
                motdep( DOUBLE ,0);
                QUINTE(DOUBLE);
                break;
         }
}

static void joue3folk1()
{
```

(suite)

```
 switch(random(2)){
      case 0:
           joue2folk1();
           joue1folk1();
           break;

      case 1:
           motdep(CROCHE,0);
           motdep( DOUBLE ,2);
           TIERCE( DOUBLE );
           QUINTE( DOUBLE );
           MINMAJ( DOUBLE );
           QUINTE( DOUBLE );
           TIERCE( DOUBLE );
           if( random( 2))
                QUINTE(NOIRE);
           else
                motdep(NOIRE,0);
           break;

       }
}

static void joue4folk1()
{
switch( random(3)){
      case 0:
           joue2folk1();
           joue2folk1();
           break;

      case 1:
           joue3folk1();
           joue1folk1();
           break;

      case 2:
           motdep(CROCHE,0);
           TIERCE( DOUBLE );
           QUINTE( DOUBLE );
           motdep( DOUBLE ,2);
           motdep( DOUBLE ,0);
           motdep( DOUBLE ,2);
           TIERCE( DOUBLE );
           motdep( CROCHE ,0);
           TIERCE( DOUBLE );
           QUINTE( DOUBLE );
           motdep( DOUBLE ,12);
           QUINTE( DOUBLE );
           motdep( DOUBLE ,10);
```



(suite)

```
            if( TROIS_NOTES )
                QUINTE( DOUBLE );
            else
                SEPTIEME( DOUBLE );
            break;

        }
}

    /*------ latin ---------*/
static void joue1latin1()
{
switch( random(8)){
    case 0:
    case 1:
        motdep(3* DOUBLE ,0);
        QUINTE( DOUBLE );
        break;

    case 2:
    case 3:
        QUINTE(3* DOUBLE );
        motdep( DOUBLE ,0);
        break;

    case 4:
        motdep( DOUBLE ,0);
        TIERCE(CROCHE);
        QUINTE( DOUBLE );
        break;

    case 5:
        motdep( DOUBLE ,0);
        motdep(CROCHE,12);
        if( TROIS_NOTES )
            QUINTE( DOUBLE );
        else
            SEPTIEME( DOUBLE );
        break;

    case 6:
        motdep( DOUBLE ,0);
        TIERCE(CROCHE);
        if( TROIS_NOTES )
            QUINTE( DOUBLE );
        else
            SEPTIEME( DOUBLE );
        break;
```

(suite)

```
        case 7:
                motdep( DOUBLE ,0);
                motdep(CROCHE,0);
                motdep( DOUBLE ,12);
                break;
        }
}

static void joue2latin1()
{
switch( random(4)){
        case 0:
                joue1latin1();
                joue1latin1();
                break;

        case 1:
                motdep(3* DOUBLE ,0);
                QUINTE( DOUBLE );
                QUINTE(3* DOUBLE );
                motdep( DOUBLE ,0);
                break;

        case 2:
                motdep(3*DOUBLE ,0);
                QUINTE( DOUBLE );
                motdep(3*DOUBLE ,12);
                QUINTE( DOUBLE );
                break;

        case 3:
                motdep(3* DOUBLE,0);
                TIERCE( DOUBLE );
                QUINTE(3* DOUBLE );
                if( TROIS_NOTES )
                        motdep( DOUBLE ,12);
                else
                        SEPTIEME( DOUBLE );
                break;
        }
}

static void joue3latin1()
{
joue2latin1();
joue1latin1();
}
```



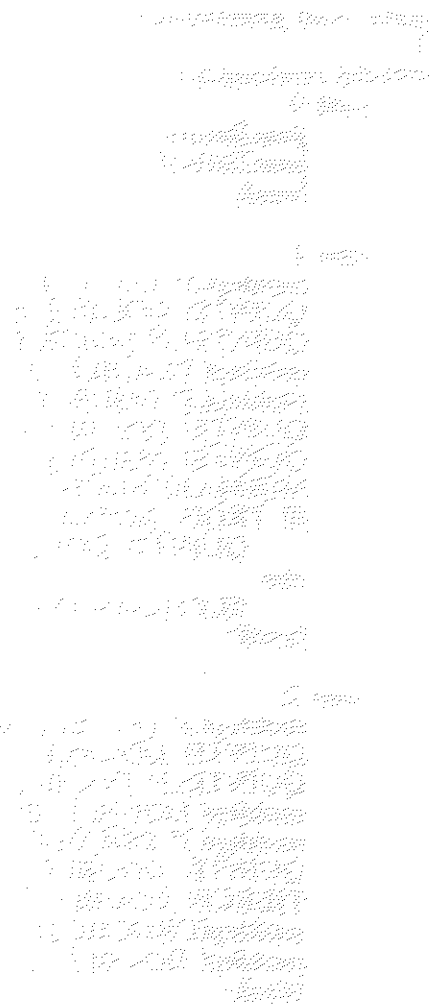
(suite)

```
static void joue4latin1()
{
switch( random(3)){
      case 0:
            joue2latin1();
            joue2latin1();
            break;

      case 1:
            motdep(3* DOUBLE ,0);
            QUINTE( DOUBLE );
            QUINTE(3* DOUBLE );
            motdep( DOUBLE ,0);
            motdep(3* DOUBLE ,0);
            QUINTE( DOUBLE );
            QUINTE( DOUBLE );
            MINMAJ(CROCHE);
            if( TROIS_NOTES )
                  QUINTE( DOUBLE );
            else
                  SEPTIEME(DOUBLE);
            break;

      case 2:
            motdep(3* DOUBLE ,0);
            QUINTE( DOUBLE );
            QUINTE(3* DOUBLE );
            motdep( DOUBLE ,0);
            motdep(3* DOUBLE ,0);
            QUINTE( DOUBLE );
            TIERCE( DOUBLE );
            motdep(CROCHE,2);
            motdep( DOUBLE ,0);
            break;

      }
}
```



4/4.3.2

Génération de motifs

Ce programme utilise un algorithme de choix dans les éléments de tableaux à une et plusieurs dimensions. En C, l'identificateur d'un tableau, lorsqu'il est employé seul, désigne en fait l'adresse de début du tableau, donc l'adresse de son premier élément. Si on déclare le tableau **int interv[n]** par exemple, la notation **interv** est équivalente à **&interv[0]**. Cet identificateur est considéré comme de type pointeur sur le type qui correspond aux éléments du tableau, ici **int**. Voici quelques exemples d'équivalences :

interv+1 **&interv[1]**
interv+n **&interv[n]**
interv[i] ***(interv+i)**

Les fonctions qui génèrent les motifs d'accompagnement doivent choisir à chaque pas d'intervalle le ième élément du tableau **interv[]** ou dans le motif qui sélectionne aléatoirement le numéro de l'élément **interv[(random(accour->nbn)]**. Les fonctions **MINMAJ** et **QMINMAJ** permettent de choisir entre deux éléments, suivant la valeur précédente de **interv[0]** qui est le premier élément de la suite à être choisi. Les agencements de motifs se font par la fonction **motdep (i,accour->interv[i])**.

Chaque division est associée à un élément du tableau **interv[]**. Dans le cas de tableaux à plusieurs dimensions, l'identificateur lorsqu'il est employé seul, représente toujours l'adresse de début, mais de nouvelles possibilités apparaissent du fait du multi-dimensionnement. Si l'on déclare **int insere[3][5]** par exemple, **insere** est effectivement égal à **&insere[0][0]**, mais les notations :
insere[0] **insere[1]** **insere[n]**
ont aussi un sens. Le tableau peut être considéré comme une succession de trois tableaux de cinq éléments. En C, **insere[0]** représente l'adresse du premier de ces tableaux à une dimension, **insere[1]** est l'adresse de début du second et **insere[n]** du nième.

Les notations suivantes sont équivalentes :

insere[0] **&insere[0][0]**
insere[1] **&insere[1][0]**

En outre les notations de la forme **insere[i]** sont des constantes pointeurs sur des entiers. Les deux expressions suivantes sont donc équivalentes :

```
insere[i]+1     &insere[i][0]+1
```

Compte tenu des priorités relatives d'opérateurs **&**, **[]** et **+**, aucune parenthèse n'est nécessaire dans la deuxième expression. La sélection du motif à choisir se fait par l'intermédiaire du tableau **pfct[]** à deux indices dont le premier représente la longueur du motif (de 1 à 4), en nombre de divisions de l'unité de base, et le second le nombre de catégories différentes (12). Les variables qui fixent le choix dans le tableau sont **longacc** et **STYLE**. La fonction **TROIS_NOTES** est un indicateur logique vrai si le nombre d'intervalles **(accour)** est égal à deux. Il permet un choix différent du quatrième élément à générer dans les suites de motifs, chaque élément devant respecter la structure de base à laquelle il est rattaché.

Les tableaux peuvent être aussi transmis en arguments de fonctions.

Lorsqu'il apparaît en argument d'appel d'une fonction, l'identificateur de tableau représente l'adresse de ce tableau. C'est cette adresse et non son contenu qui sera transmise à la fonction. Dans une fonction, il sera possible de modifier les valeurs d'un tableau transmis en argument. L'exemple suivant met une même valeur dans tous les éléments du tableau **interv[]**.

```
void fct (int interv[20])
{
        int i;
        for (i=0; i<20; i++) interv[i]=1;
}
```

Voici deux façons possibles d'appeler la fonction définie pus haut :

```
int tab1[20], tab2[20];
.......
fct(tab1);
.......
fct(tab2);
```

La fonction peut aussi s'écrire sans faire apparaître de tableau, en utilisant simplement des pointeurs sur des entiers :

```
void fct(int *i)
{
        int j;
        for (j=0; j<20; j++, i++) *i=1;
}
```



Le programme définit ensuite les fonctions utilisées. Pour respecter la structure et les définitions des dimensions définies précédemment, il faut en définir 48, mais vous pouvez transformer cette valeur pour augmenter le nombre de choix possibles. Le dimensionnement du tableau **pfct[][]** devra bien sûr être modifié en conséquence. Suite à l'exemple précédent, vous pouvez aussi transformer les tableaux en cours de programme pour éventuellement aboutir à une génération dynamique et évolutive de génération de motifs.

LISTING DU PROGRAMME EN C

```
#include <portab.h>

typedef struct {
        BYTE ton;
        BYTE typ;
        } GSTEP;

typedef struct {
        BYTE temps;       /* BIN/TER */
        BYTE nbm ;       /* nombre de mesure */
        BYTE style;
        char patname[10];
        GSTEP tabstep[ 48 ];
        WORD  nsa;
        char * accomp;
        WORD nsm;
        char * melo  ;
        WORD ok ;
        BYTE  chksum  ;
        } PATERN ;


extern long Xbios( );
extern void insere( int,int);
extern PATERN bufpat ;

extern UBYTE hauteur_acc;
extern WORD accmes ;
extern WORD prem_haut ;

#define RESOLUTION 96
#define RANDOM( x )  (Xbios( 17 ) % x)
#define  TIERCE( i )       motdep(i,accour->interv[0])
#define  QUINTE( i )       motdep(i,accour->interv[1])
#define  SEPTIEME( i )     motdep(i,accour->interv[2])
#define  MOTIF(i )         motdep(i,accour->interv[random( accour->nbn)])
#define TROIS_NOTES     accour->nbn==2
#define  MINMAJ( i )       if( accour->interv[0] == 3)\
                               motdep(i,5);\
                           else \
                               motdep(i,9)

#define  QMINMAJ( i )      if( accour->interv[0] == 3)\
                               motdep(i,8);\
                           else \
                               motdep(i,9)
```



(suite)

```
#define ARRET( i )      motdep( i ,128-hauteur_acc )

#define DOUBLE          RESOLUTION/4
#define CROCHE          RESOLUTION/2
#define NOIRE           RESOLUTION
#define TRIOLET         RESOLUTION/3

typedef struct{
        BYTE nbn;
        BYTE interv[11];
        } ACCORD ;

extern ACCORD *accour ;

static  void        joue1acc1(),joue1acc2(),joue1folk1(),joue1folk2(),

joue1disco1(),joue1disco2(),joue1latin1(),joue1latin2(),
                    joue1rock1(),joue1rock2(),joue1slow(),joue1valse();

static  void        joue2acc1(),joue2acc2(),joue2folk1(),joue2folk2(),

joue2disco1(),joue2disco2(),joue2latin1(),joue2latin2(),
                    joue2rock1(),joue2rock2(),joue2slow(),joue2valse();

static  void        joue3acc1(),joue3acc2(),joue3folk1(),joue3folk2(),

joue3disco1(),joue3disco2(),joue3latin1(),joue3latin2(),
                    joue3rock1(),joue3rock2(),joue3slow(),joue3valse();

static  void        joue4acc1(),joue4acc2(),joue4folk1(),joue4folk2(),

joue4disco1(),joue4disco2(),joue4latin1(),joue4latin2(),
                    joue4rock1(),joue4rock2(),joue4slow(),joue4valse();


void        (*pfct[4][12]) ()={
           {joue1acc1,joue1acc2,joue1folk1,joue1folk2,
            joue1disco1,joue1disco2,joue1latin1,joue1latin2,
            joue1rock1,joue1rock2,joue1slow,joue1valse
            },
           {joue2acc1,joue2acc2,joue2folk1,joue2folk2,
            joue2disco1,joue2disco2,joue2latin1,joue2latin2,
            joue2rock1,joue2rock2,joue2slow,joue2valse

           },
           {joue3acc1,joue3acc2,joue3folk1,joue3folk2,
            joue3disco1,joue3disco2,joue3latin1,joue3latin2,
            joue3rock1,joue3rock2,joue3slow,joue3valse
           },
```

(suite)

```
            {joue4acc1,joue4acc2,joue4folk1,joue4folk2,
             joue4disco1,joue4disco2,joue4latin1,joue4latin2,
             joue4rock1,joue4rock2,joue4slow,joue4valse
            }
};

/* pour se brancher sur la fonction voulue :

   (*pfct[longacc][STYLE])() ;

*/

static void joue1acc1()
{
switch( random( 3 )){
      case 0:
      case 1:
            motdep( DOUBLE,0);
            TIERCE(DOUBLE);
            QUINTE(DOUBLE);
            if( TROIS_NOTES )
                  TIERCE( DOUBLE );
            else
                  SEPTIEME( DOUBLE );
            break;

      case 2:
            motdep( DOUBLE ,0 );
            TIERCE( DOUBLE );
            MOTIF ( DOUBLE );
            if( TROIS_NOTES )
                  QUINTE( DOUBLE );
            else
                  SEPTIEME(DOUBLE);
            break;

      }

}
static void joue2acc1()
{
WORD a;
switch( a=random( 3)){
      case 0:
            joue1acc1();
            joue1acc1();
            break;
```



(suite)

```
        default:
            motdep( DOUBLE ,0);
            TIERCE( DOUBLE );
            QUINTE( DOUBLE );
            if( a == 1)
                TIERCE( DOUBLE );
            else
                motdep( DOUBLE ,12);
            if( TROIS_NOTES )
                motdep( DOUBLE ,12);
            else
                SEPTIEME( DOUBLE );
            QUINTE( DOUBLE );
            TIERCE( DOUBLE );
            QUINTE( DOUBLE );
            break;
        }
}

static void joue3acc1()
{
switch( random( 3 )){
    case 0:
        joue1acc1();
        joue1acc1();
        joue1acc1();
        break;
    case 1:
        joue2acc1();
        joue1acc1();
        break;
    case 2:
        joue1acc1();
        motdep( DOUBLE ,0);
        motdep( DOUBLE ,2);
        TIERCE( DOUBLE );
        QUINTE( DOUBLE );
        motdep( DOUBLE ,6);
        if( TROIS_NOTES )
            motdep( DOUBLE ,12);
        else
            SEPTIEME( DOUBLE );
        QUINTE( DOUBLE );
        TIERCE( DOUBLE );
        break;
    }
}
```

(suite)

```
static void joue4acc1()
{
switch( random(3)){
      case 0:
            joue2acc1();
            joue2acc1();
            break;
      case 1:
            joue3acc1();
            joue1acc1();
            break;
      case 2:
            motdep( DOUBLE ,0);
            motdep( DOUBLE ,2);
            TIERCE( DOUBLE );
            QUINTE( DOUBLE );
            motdep( DOUBLE ,6);
            if( TROIS_NOTES )
                  motdep( DOUBLE ,12);
            else
                  SEPTIEME( DOUBLE );
            motdep( DOUBLE ,12);
            MOTIF ( DOUBLE );
            joue2acc1();
            break;
      }
}



static void joue1acc2()
{
switch( random(5)){
      case 0:
            motdep( DOUBLE ,0);
            TIERCE( DOUBLE );
            QUINTE( DOUBLE );
            if( TROIS_NOTES )
                  motdep( DOUBLE ,12);
            else
                  SEPTIEME( DOUBLE );
            break;
      case 1:
            motdep(NOIRE,0);
            break;
```

(suite)

```
        case 2:
            motdep(CROCHE,0);
            TIERCE(CROCHE);
            break;

        case 3:
            TIERCE(NOIRE);
            break;

        case 4:
            SEPTIEME(NOIRE);
            break;
        }
}

static void joue2acc2()
{
switch( random(8)){
        case 0:
            joue1acc2();
            TIERCE(NOIRE);
            break;
        case 1:
        case 2:
        case 3:
            motdep(3*CROCHE,0);
            TIERCE(CROCHE);
            break;
        case 4:
        case 5:
            if( TROIS_NOTES ){
                motdep(NOIRE,0);
                QUINTE(NOIRE);
                }
            else{
                motdep(NOIRE,12);
                SEPTIEME(NOIRE);
                }
            break;
        case 6:
            motdep(NOIRE,0);
            TIERCE(CROCHE);
            QUINTE(CROCHE);
            break;
```

(suite)

```
        case 7:
            motdep(NOIRE,0);
            QUINTE(CROCHE);
            TIERCE(CROCHE);
            break;
        }
}

static void joue3acc2()
{
motdep(3*CROCHE,0);
TIERCE(CROCHE);
QUINTE(NOIRE);
}

static void joue4acc2()
{
switch( random(3)){
    case 0:
        motdep(3*CROCHE,0);
        TIERCE(CROCHE);
        QUINTE(NOIRE);
        if( TROIS_NOTES ){
            TIERCE(CROCHE);
            QUINTE(CROCHE);
            }
        else{
            motdep(CROCHE,12);
            SEPTIEME(CROCHE);
            }
        break;

    case 1:
        motdep(3*CROCHE,0);
        TIERCE(CROCHE);
        QUINTE(CROCHE);
        /* ???? */
        MINMAJ( CROCHE );
        if( TROIS_NOTES ){
            QUINTE(CROCHE);
            TIERCE(CROCHE);
            }
        else{
            SEPTIEME(CROCHE);
            QUINTE(CROCHE);
            }
        break;
```



(suite)

```
        case 2:
            motdep(NOIRE,0);
            TIERCE(CROCHE);
            QUINTE(CROCHE);
            if( TROIS_NOTES )
                motdep(NOIRE,12);
            else
                SEPTIEME(NOIRE);
            QUINTE(CROCHE);
            TIERCE(CROCHE);
            break;

        }
}

static void joue1folk1()
{
switch( random(6)){
        case 0:
            motdep(CROCHE,0);
            TIERCE( DOUBLE );
            QUINTE( DOUBLE );
            break;

        case 1:
            motdep( DOUBLE ,0);
            TIERCE( DOUBLE );
            QUINTE( CROCHE);
            break;

        case 2:
            motdep( DOUBLE ,0);
            QUINTE( DOUBLE );
            TIERCE( DOUBLE );
            QUINTE( DOUBLE );
            break;

        case 3:
            motdep( DOUBLE ,0);
            motdep( DOUBLE ,2);
            TIERCE( DOUBLE );
            QUINTE( DOUBLE );
            break;

        case 4:
            motdep( DOUBLE ,0);
            TIERCE( DOUBLE );
```

(suite)

```
            QUINTE( DOUBLE );
            motdep( DOUBLE ,9);
            break;

      case 5:
            motdep( DOUBLE ,0);
            TIERCE( DOUBLE );
            QUINTE( DOUBLE );
            if( TROIS_NOTES )
                  motdep( DOUBLE ,12);
            else
                  SEPTIEME( DOUBLE );
            break;

      }
}

static void joue2folk1()
{
switch( random(3)){
      case 0:
            joue1folk1();
            joue1folk1();
            break;

      case 1:
      case 2:
            motdep(CROCHE,0);
            TIERCE( DOUBLE );
            QUINTE( DOUBLE );
            QUINTE( DOUBLE );
            TIERCE( DOUBLE );
            motdep( DOUBLE ,0);
            QUINTE(DOUBLE);
            break;
        }
}

static void joue3folk1()
{

 switch(random(2)){
      case 0:
            joue2folk1();
            joue1folk1();
            break;
```



(suite)

```
        case 1:
            motdep(CROCHE,0);
            motdep( DOUBLE ,2);
            TIERCE( DOUBLE );
            QUINTE( DOUBLE );
            MINMAJ( DOUBLE );
            QUINTE( DOUBLE );
            TIERCE( DOUBLE );
            if( random( 2))
                QUINTE(NOIRE);
            else
                motdep(NOIRE,0);
            break;

        }
}

static void joue4folk1()
{
switch( random(3)){
    case 0:
        joue2folk1();
        joue2folk1();
        break;

    case 1:
        joue3folk1();
        joue1folk1();
        break;

    case 2:
        motdep(CROCHE,0);
        TIERCE( DOUBLE );
        QUINTE( DOUBLE );
        motdep( DOUBLE ,2);
        motdep( DOUBLE ,0);
        motdep( DOUBLE ,2);
        TIERCE( DOUBLE );
        motdep( CROCHE ,0);
        TIERCE( DOUBLE );
        QUINTE( DOUBLE );
        motdep( DOUBLE ,12);
        QUINTE( DOUBLE );
        motdep( DOUBLE ,10);
        if( TROIS_NOTES )
            QUINTE( DOUBLE );
```

(suite)

```
            else
                SEPTIEME( DOUBLE );
            break;

        }
    }
        /* ------ FOLK 2 ------- */
    static void joue1folk2()
    {
    switch( random(3)){
        case 0:
            motdep(NOIRE,0);
            break;

        case 1:
            motdep(CROCHE,0);
            TIERCE(CROCHE);
            break;

        case 2:
            motdep(CROCHE,0);
            QUINTE(CROCHE);
            break;

        }
    }

    static void joue2folk2()
    {
    switch( random(6)){
        case 0:
            joue1folk2();
            joue1folk2();
            break;

        case 1:
        case 2:
            motdep(NOIRE,0);
            TIERCE(CROCHE);
            QUINTE(CROCHE);
            break;

        case 3:
        case 4:
            motdep(NOIRE,0);
            motdep(CROCHE,2);
            TIERCE(CROCHE);
            break;
```



(suite)

```
        case 5:
            motdep(NOIRE,0);
            QUINTE(CROCHE);
            TIERCE( DOUBLE );
            motdep( DOUBLE ,2);
            break;

        }
}
static void joue3folk2()
{
switch( random(7)){
        case 0:
        case 1:
            motdep(CROCHE,0);
            motdep(CROCHE,12);
            MINMAJ(CROCHE);
            motdep(CROCHE,0);
            joue1folk2();
            break;

        case 2:
            joue1folk2();
            if( random(2)==1){
                MINMAJ(CROCHE);
                QUINTE(CROCHE);
                }
            else {
                motdep(CROCHE,0);
                QUINTE(CROCHE);
                }
            joue1folk2();
            break;

        case 3:
        case 4:
            motdep(NOIRE,0);
            motdep(CROCHE,2);
            TIERCE(CROCHE);
            MINMAJ(CROCHE);
            QUINTE(CROCHE);
            break;
        case 5:
        case 6:
            motdep(NOIRE,0);
            TIERCE(CROCHE);
            QUINTE(CROCHE);
            if( TROIS_NOTES )
                motdep(CROCHE,9);
```

(suite)

```
            else
                SEPTIEME(CROCHE);
            QUINTE(CROCHE);
            break;

        }
    }
    static void joue4folk2()
    {
    switch( random(9)){
        case 0:
            joue3folk2();
            joue1folk2();
            break;

        case 1:
            joue2folk2();
            joue2folk2();
            break;

        case 2:
            joue1folk2();
            joue2folk2();
            joue1folk2();
            break;

        case 3:
        case 4:
        case 5:
            motdep(NOIRE,0);
            motdep(CROCHE,2);
            TIERCE(CROCHE);
            QUINTE(NOIRE);
            MINMAJ(CROCHE);
            QUINTE(CROCHE);
            break;

        case 6:
        case 7:
            motdep(NOIRE,0);
            motdep(CROCHE,12);
            QUINTE(CROCHE);
            if( TROIS_NOTES ){
                MINMAJ(CROCHE);
                QUINTE(CROCHE);
                }
            else{
                SEPTIEME(CROCHE);
                MINMAJ(CROCHE);
                }
```

(suite)

```
                TIERCE(CROCHE);
                motdep(CROCHE,2);
                break;

        default :
                motdep(NOIRE,0);
                TIERCE(CROCHE);
                motdep(CROCHE,2);
                motdep(NOIRE,0);
                QUINTE(CROCHE);
                MINMAJ(CROCHE);
                break;
        }

}

        /* ---- Disco ----*/
static void joue1disco1()
{
switch( random(5)){
        case 0:
                motdep(CROCHE,0);
                motdep(CROCHE,12);
                break;

        case 1:
                motdep(CROCHE,0);
                motdep( DOUBLE ,12);
                motdep( DOUBLE ,12);
                break;

        case 2:
                motdep(CROCHE,0);
                motdep( DOUBLE ,12);
                MOTIF ( DOUBLE );
                break;

        case 3:
                motdep( DOUBLE ,0);
                motdep( DOUBLE ,0);
                motdep( DOUBLE ,12);
                motdep( DOUBLE ,12);
                break;

        case 4:
                motdep(NOIRE,0);
                break;
        }

}
```

(suite)

```
static void joue2disco1()
{
switch( random(3)){
      case 0:
            joue1disco1();
            joue1disco1();
            break;

      case 1:
            motdep(NOIRE,0);
            motdep( DOUBLE ,0);
            motdep( DOUBLE ,0);
            QUINTE( DOUBLE );
            QUINTE( DOUBLE );
            break;

      case 2:
            motdep(CROCHE,0);
            motdep(CROCHE,12);
            TIERCE(CROCHE);
            QUINTE(CROCHE);
            break;

      }
}

static void joue3disco1()
{
      joue2disco1();
      joue1disco1();
}

static void joue4disco1()
{
switch( random(3)){
      case 0:
            joue2disco1();
            joue2disco1();
            break;

      case 1:
            motdep(NOIRE,0);
            motdep( DOUBLE ,0);
            motdep( DOUBLE ,0);
            motdep( DOUBLE ,12);
            motdep( DOUBLE ,12);
            motdep( CROCHE ,0);
```



(suite)

```
                motdep( DOUBLE ,0);
                motdep( DOUBLE ,0);
                motdep(NOIRE ,12);
                break;

        case 2:
                motdep(NOIRE,0);
                motdep( DOUBLE ,0);
                motdep( DOUBLE ,0);
                motdep( DOUBLE ,12);
                motdep( DOUBLE ,12);
                motdep( DOUBLE ,0);
                motdep( DOUBLE ,12);
                motdep( DOUBLE ,2);
                motdep( DOUBLE ,12);
                TIERCE( DOUBLE );
                motdep( DOUBLE ,12);
                QUINTE( DOUBLE );
                motdep( DOUBLE ,12);
                break;

        }
}

        /* ---- Disco 2 -------*/
static void joue1disco2()
{
switch( random(4)){
        case 0:
        case 1:
                motdep(NOIRE,0);
                break;
        case 2:
                motdep(CROCHE,0);
                motdep(CROCHE,12);
                break;
        case 3:
                motdep(CROCHE,0);
                QUINTE(CROCHE);
                break;
        }
}

static void joue2disco2()
{
```

(suite)

```
switch( random(4)){
      case 0:
      case 1:
            motdep(NOIRE,0);
            motdep(NOIRE,12);
            break;

      case 2:
            motdep(CROCHE,3);
            TIERCE(CROCHE);
            motdep(NOIRE,12);
            break;

      case 3:
            if( TROIS_NOTES )
                  motdep(CROCHE,6);
            else
                  SEPTIEME(CROCHE);
            QUINTE(CROCHE);
            motdep(NOIRE,12);
            break;

      }
}

static void joue3disco2()
{
      joue2disco2();
      joue1disco2();
}
static void joue4disco2()
{
switch( random(2)){
      case 0:
            joue1disco2();
            joue1disco2();
            joue2disco2();
            break;

      case 1:
            motdep(NOIRE,0);
            motdep(NOIRE,12);
            TIERCE(CROCHE);
            motdep(CROCHE,12);
            TIERCE(CROCHE);
            motdep(CROCHE,2);
            break;
      }
}
```



(suite)

```
        /*------ latin ---------*/
static void joue1latin1()
{
switch( random(8)){
        case 0:
        case 1:
                motdep(3* DOUBLE ,0);
                QUINTE( DOUBLE );
                break;

        case 2:
        case 3:
                QUINTE(3* DOUBLE );
                motdep( DOUBLE ,0);
                break;

        case 4:
                motdep( DOUBLE ,0);
                TIERCE(CROCHE);
                QUINTE( DOUBLE );
                break;

        case 5:
                motdep( DOUBLE ,0);
                motdep(CROCHE,12);
                if( TROIS_NOTES )
                        QUINTE( DOUBLE );
                else
                        SEPTIEME( DOUBLE );
                break;

        case 6:
                motdep( DOUBLE ,0);
                TIERCE(CROCHE);
                if( TROIS_NOTES )
                        QUINTE( DOUBLE );
                else
                        SEPTIEME( DOUBLE );
                break;

        case 7:
                motdep( DOUBLE ,0);
                motdep(CROCHE,0);
                motdep( DOUBLE ,12);
                break;
        }
}
```

(suite)

```
static void joue2latin1()
{
switch( random(4)){
      case 0:
            joue1latin1();
            joue1latin1();
            break;

      case 1:
            motdep(3* DOUBLE ,0);
            QUINTE( DOUBLE );
            QUINTE(3* DOUBLE );
            motdep( DOUBLE ,0);
            break;

      case 2:
            motdep(3*DOUBLE ,0);
            QUINTE( DOUBLE );
            motdep(3*DOUBLE ,12);
            QUINTE( DOUBLE );
            break;

      case 3:
            motdep(3* DOUBLE,0);
            TIERCE( DOUBLE );
            QUINTE(3* DOUBLE );
            if( TROIS_NOTES )
                  motdep( DOUBLE ,12);
            else
                  SEPTIEME( DOUBLE );
            break;
      }
}

static void joue3latin1()
{
joue2latin1();
joue1latin1();
}


static void joue4latin1()
{
switch( random(3)){
      case 0:
            joue2latin1();
            joue2latin1();
            break;
```



(suite)

```
        case 1:
            motdep(3* DOUBLE ,0);
            QUINTE( DOUBLE );
            QUINTE(3* DOUBLE );
            motdep( DOUBLE ,0);
            motdep(3* DOUBLE ,0);
            QUINTE( DOUBLE );
            QUINTE( DOUBLE );
            MINMAJ(CROCHE);
            if( TROIS_NOTES )
                QUINTE( DOUBLE );
            else
                SEPTIEME( DOUBLE );
            break;

        case 2:
            motdep(3* DOUBLE ,0);
            QUINTE( DOUBLE );
            QUINTE(3* DOUBLE );
            motdep( DOUBLE ,0);
            motdep(3* DOUBLE ,0);
            QUINTE( DOUBLE );
            TIERCE( DOUBLE );
            motdep(CROCHE,2);
            motdep( DOUBLE ,0);
            break;

        }
    }

        /* ------ latin 2 ---------*/
    static void joue1latin2()
    {
    switch( random(2)){
        case 0:
            motdep(NOIRE,0);
            break;

        case 1:
            motdep(3* DOUBLE ,0);
            QUINTE( DOUBLE );
            break;
         }
    }

    static void joue2latin2()
    {
    switch( random(4)){
```

(suite)

```
        case 0:
                motdep(3*CROCHE,0);
                QUINTE(CROCHE);
                break;

        case 1:
                QUINTE(3*CROCHE);
                motdep(CROCHE,0);
                break;

        case 2:
                motdep(CROCHE,0);
                MOTIF (NOIRE);
                MOTIF (CROCHE);
                break;

        case 3:
                motdep(CROCHE,0);
                motdep(NOIRE,0);
                motdep(CROCHE,12);
                break;
        }
}
static void joue3latin2()
{
joue2latin2();
joue1latin2();
}
static void joue4latin2()
{
switch( random(5)){
        case 0:
                motdep(NOIRE,0);
                QUINTE(NOIRE);
                motdep(CROCHE,0);
                QUINTE(NOIRE);
                QUINTE(CROCHE);
                break;

        case 1:
                motdep(NOIRE,0);
                QUINTE(NOIRE);
                motdep(CROCHE,0);
                TIERCE(NOIRE);
                QUINTE(CROCHE);
                break;
```



(suite)

```
        case 2:
            motdep(NOIRE,0);
            motdep(NOIRE,12);
            QUINTE(CROCHE);
            if( TROIS_NOTES )
                MINMAJ(NOIRE);

            else
                SEPTIEME(NOIRE);
            QUINTE(CROCHE);
            break;

        case 3:
            motdep(NOIRE,0);
            QUINTE(NOIRE);
            TIERCE(CROCHE);
            QUINTE(NOIRE);
            if( TROIS_NOTES )
                MINMAJ(CROCHE);

            else
                SEPTIEME(CROCHE);
            break;

        case 4:
            motdep(CROCHE,0);
            QUINTE(CROCHE);
            ARRET(CROCHE);
            TIERCE(CROCHE);
            QUINTE(CROCHE);
            motdep(CROCHE,12);
            if( TROIS_NOTES )
                MINMAJ(NOIRE);

            else
                SEPTIEME(NOIRE);
            break;
        }
    }

    /* ------ rock ------*/
    static void joue1rock1()
    {
    switch( random(2)){
        case 0:
            motdep(CROCHE,0);
            motdep(CROCHE,0);
            break;
```

(suite)

```
        case 1:
                motdep(CROCHE,0);
                motdep(CROCHE,12);
                break;

        }
}
static void joue2rock1()
{
switch( random(2)){
        case 0:
                motdep(CROCHE,0);
                motdep(CROCHE,0);
                TIERCE(CROCHE);
                QUINTE(CROCHE);
                break;

        case 1:
                motdep(CROCHE,12);
                motdep(CROCHE,12);
                QUINTE(CROCHE);
                TIERCE(CROCHE);
                break;

        }

}

static void joue3rock1()
{
joue2rock1();
joue1rock1();
}

static void joue4rock1()
{
switch( random(6)){
        case 0:
                motdep(CROCHE,0);
                motdep(CROCHE,0);
                TIERCE(CROCHE);
                TIERCE(CROCHE);
                QUINTE(CROCHE);
                QUINTE(CROCHE);
                MINMAJ(CROCHE);
                QUINTE(CROCHE);
                break;
```

(suite)

```
        case 1:
            motdep(CROCHE,12);
            motdep(CROCHE,12);
            QUINTE(CROCHE);
            MINMAJ(CROCHE);
            motdep(CROCHE,12);
            motdep(CROCHE,12);
            MINMAJ(CROCHE);
            QUINTE(CROCHE);
            break;

        case 2:
            motdep(CROCHE,12);
            motdep(CROCHE,12);
            QUINTE(CROCHE);
            MINMAJ(CROCHE);
            if( TROIS_NOTES )
                motdep(CROCHE,12);
            else
                SEPTIEME(CROCHE);
            if( TROIS_NOTES )
                motdep(CROCHE,12);
            else
                SEPTIEME(CROCHE);
            MINMAJ(CROCHE);
            QUINTE( CROCHE );
            break;

        case 3:
            motdep(CROCHE,0);
            TIERCE(CROCHE);
            QUINTE(CROCHE);
            MINMAJ(CROCHE);
            if( TROIS_NOTES )
                motdep(CROCHE,12);
            else
                SEPTIEME(CROCHE);
            MINMAJ(CROCHE);
            QUINTE(CROCHE);
            TIERCE(CROCHE);
            break;

        case 4:
            if( TROIS_NOTES ){
                motdep(CROCHE,0);
                TIERCE(CROCHE);
                QUINTE(CROCHE);
                MINMAJ(CROCHE);
```

(suite)

```
                if( TROIS_NOTES )
                    motdep(CROCHE,12);
                else
                    SEPTIEME(CROCHE);
                MINMAJ(CROCHE);
                QUINTE(CROCHE);
                TIERCE(CROCHE);
                }
            else{
                motdep(CROCHE,2);
                motdep(CROCHE,4);
                motdep(CROCHE,5);
                motdep(CROCHE,6);
                QUINTE(CROCHE);
                motdep(CROCHE,5);
                TIERCE(CROCHE);
                motdep(CROCHE,2);
                }
            break;

        case 5:
            QUINTE(CROCHE);
            QUINTE(CROCHE);
            QMINMAJ( CROCHE );
            QMINMAJ( CROCHE);
            motdep( CROCHE ,10);
            motdep( CROCHE ,10);
            QMINMAJ( DOUBLE );
            motdep( DOUBLE ,10);
            QMINMAJ( DOUBLE );
            TIERCE( DOUBLE );
            break;

        }
}

static void joue1rock2()
{
switch( random(3)){
    case 0:
        motdep(2*TRIOLET,0);
        motdep(TRIOLET,0);
        break;

    case 1:
        motdep(2*TRIOLET,0);
        motdep(TRIOLET,12);
        break;
```



(suite)

```
        case 2:
            motdep(2*TRIOLET,0);
            QUINTE(TRIOLET);
            break;

        }
}

static void joue2rock2()
{
switch( random(2)){
        case 0:
            motdep(2*TRIOLET,0);
            motdep(TRIOLET,0);
            TIERCE(2*TRIOLET);
            QUINTE(TRIOLET);
            break;

        case 1:
            motdep(2*TRIOLET,12);
            motdep(TRIOLET,12);
            QUINTE(2*TRIOLET);
            TIERCE(TRIOLET);
            break;

        }
}

static void joue3rock2()
{
 joue2rock2();
 joue1rock2();
}

static void joue4rock2()
{
switch( random(5)){
        case 0:
            motdep(2*TRIOLET,0);
            motdep(TRIOLET,0);
            TIERCE(2*TRIOLET);
            TIERCE(TRIOLET);
            QUINTE(2*TRIOLET);
            QUINTE(TRIOLET);
            MINMAJ(2*TRIOLET);
            QUINTE(TRIOLET);
            break;
```

(suite)

```
        case 1:
                motdep(2*TRIOLET,12);
                motdep(TRIOLET,12);
                QUINTE(2*TRIOLET);
                MINMAJ(TRIOLET);
                motdep(2*TRIOLET,12);
                motdep(TRIOLET,12);
                MINMAJ(2*TRIOLET);
                QUINTE(TRIOLET);
                break;

        case 2:
                motdep(2*TRIOLET,12);
                motdep(TRIOLET,12);
                QUINTE(2*TRIOLET);
                MINMAJ(TRIOLET);
                if( TROIS_NOTES )
                        motdep(2*TRIOLET,12);
                else
                        SEPTIEME(2*TRIOLET);
                if( TROIS_NOTES )
                        motdep(TRIOLET,12);
                else
                        SEPTIEME(TRIOLET);
                MINMAJ(2*TRIOLET);
                QUINTE(TRIOLET);
                break;

        case 3:
                motdep(2*TRIOLET,0);
                TIERCE(TRIOLET);
                QUINTE(2*TRIOLET);
                MINMAJ(TRIOLET);
                if( TROIS_NOTES )
                        motdep(2*TRIOLET,12);
                else
                        SEPTIEME(2*TRIOLET);
                MINMAJ(TRIOLET);
                QUINTE(2*TRIOLET);
                TIERCE(TRIOLET);
                break;

        case 4:
                if( (accour->nbn == 3) &&
                        (accmes == bufpat.nbm -1)
                                        &&
                        ( hauteur_acc-prem_haut==7 ||
                          prem_haut - hauteur_acc == 5)
                )
```



(suite)

```
                    {
                    motdep(2*TRIOLET,2);
                    motdep(TRIOLET,4);
                    motdep(2*TRIOLET,5);
                    motdep(TRIOLET,6);
                    QUINTE(2*TRIOLET);
                    motdep(TRIOLET,5);
                    TIERCE(2*TRIOLET);
                    motdep(TRIOLET,2);
                    }
               else{

                motdep(2*TRIOLET,0);
                TIERCE(TRIOLET);
                QUINTE(2*TRIOLET);
                MINMAJ(TRIOLET);
                if( TROIS_NOTES )
                     motdep(2*TRIOLET,12);
                else
                     SEPTIEME(2*TRIOLET);
                MINMAJ(TRIOLET);
                QUINTE(2*TRIOLET);
                TIERCE(TRIOLET);
                }
           break;
      }
}

/* ------ slow ---------*/
static void joue1slow()
{
switch( random(7)){
      case 0:
      case 1:
      case 2:

           motdep(NOIRE,0);
           break;

      case  3:
      case 4:
           motdep(2*TRIOLET,0);
           QUINTE(TRIOLET);
           break;
```

(suite)

```
        case 5:
            motdep(TRIOLET,0);
            motdep(TRIOLET,2);
            TIERCE(TRIOLET);
            break;

        case 6:
            motdep(TRIOLET,0);
            TIERCE(TRIOLET);
            QUINTE(TRIOLET);
            break;
        }
    }

    static void joue2slow()
    {
    switch( random(3)){
        case 0:
            motdep(TRIOLET,0);
            TIERCE(TRIOLET);
            QUINTE(TRIOLET);
            motdep(TRIOLET,12);
            QUINTE(TRIOLET);
            TIERCE(TRIOLET);
            break;

        case 1:
            motdep(TRIOLET,0);
            TIERCE(TRIOLET);
            QUINTE(TRIOLET);
            motdep(TRIOLET,12);
            TIERCE(TRIOLET);
            QUINTE(TRIOLET);
            break;

        case 2:
            motdep(TRIOLET,0);
            motdep(TRIOLET,2);
            TIERCE(TRIOLET);
            QUINTE(TRIOLET);
            if( TROIS_NOTES ){
                if(accour->interv[0]==3 && accour->interv[1]==7 )
                    motdep(TRIOLET,8);
                else
                    motdep(TRIOLET,9);

                }
```

(suite)

```
                else
                    SEPTIEME(TRIOLET);
                QUINTE(TRIOLET);
                break;

        }
}

static void joue3slow()
{

switch( random(5)){
        case 0:
                joue2slow();
                joue1slow();
                break;
        case 1:
        case 2:
                motdep(TRIOLET,0);
                QUINTE(TRIOLET);
                TIERCE(TRIOLET);
                motdep(TRIOLET,12);
                QUINTE(TRIOLET);
                if( TROIS_NOTES ){
                    if(accour->interv[0]==3 && accour->interv[1]==7 )
                        motdep(TRIOLET,8);
                    else
                        motdep(TRIOLET,9);
                    }
                else
                    SEPTIEME(TRIOLET);
                TIERCE(TRIOLET);
                QUINTE(TRIOLET);
                motdep(TRIOLET,2);
                break;

        default:
                motdep(TRIOLET,0);
                QUINTE(TRIOLET);
                TIERCE(TRIOLET);
                motdep(TRIOLET,12);
                QUINTE(TRIOLET);
                if( TROIS_NOTES )
                    if( accour->interv[0]==3 && accour->interv[1]==7)
                        motdep(TRIOLET,8);
                    else
                        motdep(TRIOLET,9);
```

(suite)

```
                else
                    SEPTIEME(TRIOLET );
                joue1slow();
                break;

        }
    }

    static void joue4slow()
    {
    switch( random(10)){
        case 0:
            joue3slow();
            joue1slow();
            break;

        case 1:
            joue2slow();
            joue2slow();
            break;

        case 2:
        case 3:
            motdep(TRIOLET,0);
            TIERCE(TRIOLET);
            QUINTE(TRIOLET);
            motdep(TRIOLET,12);
            QUINTE(TRIOLET);
            TIERCE(TRIOLET);
            motdep(TRIOLET,0);
            TIERCE(TRIOLET);
            motdep(TRIOLET,2);
            QUINTE(TRIOLET);
            if( TROIS_NOTES )
                if( accour->interv[0]==3 && accour->interv[1]==7)
                    motdep(TRIOLET,8);
                else
                    motdep(TRIOLET,9);
            else
                SEPTIEME(TRIOLET);
            QUINTE(TRIOLET);
            break;

        case 4:
        case 5:
            motdep(TRIOLET,0);
            TIERCE(TRIOLET);
            QUINTE(TRIOLET);
```

(suite)

```
            motdep(TRIOLET,12);
            TIERCE(TRIOLET);
            QUINTE(TRIOLET);
            motdep(TRIOLET,0);
            TIERCE(TRIOLET);
            QUINTE(TRIOLET);
            if( TROIS_NOTES )
                if( accour->interv[0]==3 && accour->interv[1]==7 )
                    motdep(2*TRIOLET,8);
                else
                    motdep(2*TRIOLET,8);
            else
                SEPTIEME(2*TRIOLET);
            QUINTE(TRIOLET);
            break;

        case 6:
        case 7:
            motdep(5*TRIOLET,0);
            TIERCE(TRIOLET);
            QUINTE(NOIRE);
            TIERCE(2*TRIOLET);
            QUINTE(TRIOLET);
            break;

        case 8:
        case 9:
            motdep(TRIOLET,0);
            TIERCE(TRIOLET);
            QUINTE(TRIOLET);
            TIERCE(TRIOLET);
            QUINTE(TRIOLET);
            motdep(TRIOLET,12);
            if( TROIS_NOTES )
            if( accour->interv[0]==3 && accour->interv[1]==7 )
                    motdep(TRIOLET,8);
                else
                    motdep(TRIOLET,9);
            else
                SEPTIEME(TRIOLET);
            QUINTE(TRIOLET);
            TIERCE(TRIOLET);
            QUINTE(TRIOLET);
            TIERCE(TRIOLET);
            motdep(TRIOLET,2);
            break;
        }
    }
```

(suite)

```
/* - ---- valse ------*/
static void joue1valse()
{
switch( random(2)){
      case 0:
            motdep(NOIRE,0);
            break;
      case 1:
            motdep(CROCHE,0);
            QUINTE(CROCHE);
            break;
      }
}

static void joue2valse()
{
switch( random(3)){
      case 0:
            motdep(CROCHE,0);
            TIERCE(CROCHE);
            QUINTE(CROCHE);
            TIERCE(CROCHE);
            break;

      case 1:
            motdep(NOIRE,0);
            TIERCE(CROCHE);
            QUINTE(CROCHE);
            break;

      case 2:
            motdep(NOIRE,0);
            QUINTE(CROCHE);
            TIERCE(CROCHE);
            break;

      case 3:
            motdep( CROCHE ,0);
            TIERCE( CROCHE );
            QUINTE( CROCHE );
            if( TROIS_NOTES )
                  MINMAJ( CROCHE );
            break;
      }
}
```



(suite)

```
static void joue3valse()
{
switch( random(7)){
    case 0:
          motdep(CROCHE,0);
          TIERCE(CROCHE);
          QUINTE(CROCHE);
          TIERCE(CROCHE);
          QUINTE(CROCHE);
          TIERCE(CROCHE);
        break;

    case 1:
        motdep(CROCHE,0);
        TIERCE(CROCHE);
        QUINTE(CROCHE);
        QMINMAJ(CROCHE);
        QUINTE(CROCHE);
        TIERCE(CROCHE);
        break;

    case 2:
        motdep(CROCHE,0);
        TIERCE(CROCHE);
        QUINTE(CROCHE);
        motdep(CROCHE,12);
        QUINTE(CROCHE);
        TIERCE(CROCHE);
        break;

    case 3:
        if( TROIS_NOTES ){
            motdep(CROCHE,0);
            motdep(CROCHE,2);
            TIERCE(CROCHE);
            QUINTE(CROCHE);
            QMINMAJ(CROCHE);
            TIERCE(CROCHE);
            }
        else{
            motdep(CROCHE,0);
            TIERCE(CROCHE);
            QUINTE(CROCHE);
            SEPTIEME(CROCHE);
            QUINTE(CROCHE);
            TIERCE(CROCHE);
            }
        break;

    case 4:
        motdep(NOIRE,0);
        TIERCE(NOIRE);
```

(suite)

```
            QUINTE(NOIRE);
            break;

        case 5:
            motdep(NOIRE,0);
            QUINTE(NOIRE);
            TIERCE(NOIRE);
            break;

        case 6:
            motdep(CROCHE,0);
            motdep(CROCHE,2);
            TIERCE(CROCHE);
            QUINTE(CROCHE);
            TIERCE(CROCHE);
            motdep(CROCHE,2);
            break;


        }
    }

    static void melo1()
    {
    switch( random(8)){
        case 0:
            insere(1,0);
            break;

        case 1:
        case 2:
            insere(1,1);
            break;

        case 3:
        case 4:
            insere(1,2);
            break;

        case 5:
            insere(1,3);
            break;

        default:
            insere(1,4);
        }
    }
```



(suite)

```
static void melo2()
{
switch( random(16)){

      case 0:insere(2,0);
             break;
      case 1:
      case  2:
      case 3: insere(2,1);
             break;

      case 4:
      case 5:
      case 6: insere(2,2);
             break;

      case 7:
             insere(2,3);
             break;
      case 8:
      case  9:
      case 10:insere(2,4);
             break;

      case 11:
      case 12:insere(2,5);
             break;

      default:
             insere(2,6);
             break;

      }
}
static void melo3()
{
switch( random(8)){
      case 0: insere(3,0);
             break;

      case 1:
      case 2:
             insere(3,1);
             break;

     case 3:
      case 4:
             insere(3,2);
             break;
```

(suite)

```
        case 5: melo2();
                melo1();
                break;

      default:
                insere(3,3);
      }
}
static void melo4()
{
switch( random(27)){
      case  0:
      case 1: insere(4,0);
                break;

      case 2:
      case 3:
      case 4: insere(4,1);
                break;

      case 5:
                insere(4,2);
                break;
      case 6:
      case 7:
                insere(4,3);
                break;

      case 8: insere(4,4);
                break;

      case  9:
      case 10:
      case 11:
      case 12:insere(4,5);
                break;

      case 13:
      case 14: insere(4,6);
                break;

      case  15:
      case 16:
      case 17:
      case 18:insere(4,7);
                break;
```



(suite)

```
        case 19:
        case 20:
        case 21:insere(4,8);
                break;

        case 22:
        case 23:insere(4,9);
                break;

        case 24:insere(4,10);
                break;

        default :
                insere(4,11);
                break;
        }
}
```



4/4.3.3

Jeux musicaux en C

Les programmes suivants ont été présentés simplement dans la partie 6 de l'ouvrage de référence, en Basic GFA. La version en C, plus élaborée, reprend aussi les jeux musicaux des dictées mélodiques et rythmiques. Pour la compréhension des techniques de programmation mises en oeuvre dans cette version, il est bon d'avoir assimilé les exemples fournis dans tous les programmes du chapitre « le C par les exemples de programmation » ainsi que les paragraphes traitant de la compilation et de la structuration d'un programme modulaire. Ces programmes peuvent aussi être complétés avec le programme de génération de motifs référencé dans les exemples en C.

Voici néanmoins les points de repère importants qui vous permettront de vous y retrouver dans les trois programmes présentés en C.

PROGRAMME PRINCIPAL

Ce programme englobe tous les appels aux jeux traités en Basic au moyen de la définition de la variable **quoi**. **devine_quoi(quoi)** est la fonction qui référence le jeu stocké dans cette variable, puisque la logique en est toujours la même. Les variables logiques du GFA sont remplacées ici par les mots **solu** et **donne_solu**, le C ne reconnaissant pas de type booléen spécifique à part entière. L'objet sélectionné lorsque l'ordinateur donne la bonne réponse est visualisé avec un clignotement au moyen de la fonction **clignote(ob,index)**. L'index caractérise ici le numéro d'objet dans la liste des objets affichés.

Les huit accords et les treize intervalles sont définis sur **tti[]** et **ttj[]**.

Le titre du jeu et son objet sont ensuite affichés au moyen de ressources faisant référence aux niveaux et nombres de cases possibles définies pour les réponses. En général, les cases **jouer**, **ecouter**, **quitter** seront présentes dans tous les jeux.

Dans le cas de la sélection des jeux sur les intervalles, la variable **interasc** fixe le sens des intervalles et sélectionne l'affichage de ce sens. Tant que l'on ne quitte pas le jeu(**while (! fin)**), on se branche sur la sélection d'une fonction de jeu ou d'un changement de paramètre de jeu suivant la valeur de **i** par un switch. On réinitialise comme en Basic la variable de validation du jeu (**solu=-1**), dans le cas où un nouveau niveau est sélectionné. Le cas TBJOUER correspond à la place de la procédure **genere** en Basic. La sélection aléatoire de la tonique (**ton**) et

du sens de la déviation des hauteurs suivantes, dans le cas des intervalles (**if** (quoi)), s'effectue par l'intermédiaire de la variable **sens**. Le champ **repi[]** contient les réponses des jeux sur les intervalles et, par défaut, **repi[2]** vaut toujours **FIN** pour arrêter le jeu dès que deux notes sont rentrées. Dans le cas des jeux sur les accords, le pointeur de tableau **tabaccord[]** est transféré en **ac** et **repi[0]** contiendra toujours la tonique aléatoire de départ (**RANDOM(13)**). La réponse est stockée en **rep** et **nbrep** est décrémenté à chaque essai infructueux à partir de la valeur **nbmax-1**.

Dès que **nbrep** est négatif, on le ramène positif par **nbrep++** et on donne la solution du jeu toujours sélectionné par la variable **quoi**. Les *tempo* des notes arpégées sont fixés par la variable **tempojeux** et sont incrémentés ou décrémentés lors des situations correspondant à la procédure **ilfallait** du Basic.

La fonction de filtre sert à ne considérer que les notes et isole les hauteurs des notes, sans leur vélocité éventuelle , au cours de l'enregistrement des quizz.

La fonction **analyse(buf,n)** gère tous les renversements d'accords, supprime les doublons et les octaves, remet les notes dans un ordre ascendant.

L'analyse pourra se faire dès lors que toutes les hauteurs seront ramenées à l'intérieur d'une octave. La note de hauteur la plus faible sera par défaut considérée comme fondamentale et tonique de l'accord.

La fin du programme principal gère les affichages et les messages de solutions du type «Vous avez joué (ou sélectionné) ...», «Vous auriez dû jouer ...», «La solution est ...», etc.

```
/* LISTING DE JEUX PROGRAMME PRINCIPAL */
#include <stdio.h>
#include <osbind.h>
#include <portab.h>
#include <gemlib.h>
#include <stdlib.h>
#include "pistes.h"
#include "gemexten.h"
#include "filtre.h"

#define PAUSE( x ) evnt_timer( x & 0xFFFF,x>> 16 )
#define RANDOM( x ) ( Xbios( 17 ) %x )
#define EFF_BOX eff box( 50,YCOORD(66),620,YCOORD(160) )
#define VTEXT( x,y,s ) MOUSE_OFF;v_gtext(handle,(x),(y),(s));MOUSE_ON

extern char *textes[];
extern WORD metronome,mode,taille_mesure;

#define ttc(i) textes[i+17]
#define ttna(i) textes[i+21]
```



(suite)

```
#define ttni(i) textes[i+29]
#clefine TEXT( i ) textes[i]
WORD enjeux;

typedef struct{
 BYTE nbn;
 BYTE interv[11];
 } ACCORD;

extern ACCORD tabaccord[];
extern char **types;
extern WORD xdesk,ydesk,wdesk,hdesk,clipping,pxyarray[];
extern int vmanuel;
extern char texte[];
extern WORD handle,a_couleur;
extern WORD hlettre;
extern OBJECT * menu;
#define AIDE( n ) if( vmanuel ) { manuel( n ); break;}

WORD tempojeux;
WORD niveau;
static WORD renversement;

void play ( s ,n ) UBYTE *s;int n; {
unsigned long time = 500 * 60;
 time l= tempojeux; /* 1000*tempo 1120( croche ) */
 while( (*s != FIN) && n )
 {
    n --;
    noton( 0,*s );
    PAUSE( time -100);
    notof(0 ,*s );
    PAUSE( 99 );
    s ++;
 }
}


#define DENIVEAU_2A( x ) ( x==2 || x==3 || x==4 || x==7)
#define DENIVEAU 2I( x ) ( x==1 || x==2 || x==6 || x==8 || x==9 || x==11)
#define DENIVEAU_2( x ) ( quoi ? DENIVEAU_2I( x ): DENIVEAU_2A( x ))

static void reinit( ob, quoi,tt,nn )
OBJECT *ob;
WORD quoi;
WORD tt[];
WORD nn;
{
```

(Suite)

```
            /* valider, invalider *l
         reinit( tob[2], quoi,ttx,nbmax );
         objc_draw( tob[2],0,7,xdesk,ydesk,wdesk,hdesk);
         solu = -1;
         PAUSE( 100 );
         EFF BOX;
         break;

      case 1:
         switch( o ){

            case TBJOUER:
               reinit( tob[2], quoi, ttx,nbmax );

               objc_draw( tob[2],0,7,xdesk,ydesk,wdesk,hdesk)-
               solu = random( nbmax )
               if( ! niveau) {
                  while ( DENIVEAU 2( solu ) ) {
                     solu ++;
                     if( solu >= nbmax ) solu = 0;
                     }
                  }
               ton = random( 11 );
               ton += 60;
               /* generer la suite de notes:
                  si accord -> ton + les suivantes +ff
                  sinon ton +/- ( ton + solu ) +ff

               */
               if ( quoi ) {
                  if( sens == 1){
                     repi[0] = ton;
                     repi[1] = ton + solu;
                     }
                  else {
                     ton += 12;
                     repi[0] = ton;
                     repi[1] = ton -solu;
                     }
                  repi[2]=FIN;
                  }
               else {
                  ACCORD *ac;

                  ac = & tabaccord[ solu ];
                  repi[0] = ton;
                  for( j = 0;j< ac->nbn ,j++){
                     repi[ j+1 ] = ton + ac->interv[j];
```



(suite)

```
sprintf( texte ,TEXT(43),(quoi) ? TEXT(44):TEXT(45) );
barre_texte( menu, texte, hlettre );

rsrc_gaddr( R_TREE,(WORD)NIVEAU ,&tob[0]);
rsrc_gaddr( R_TREE,(WORD)TROISBOUT,&tob[1]);
rsrc_gaddr( R_TREE,( quoi) ? (WORD)TYPINTER:(WORD)TYPACC ,&tob[2]);

strcpy( (char *)(tob[1][TBECOUTE].ob_spec),ttc(0) );

set_coord( tob[0],540,ydesk+4 );
set_coord( tob[1],10 ,ydesk + hdesk - tob[1]->ob_height );
set_coord( tob[2],10 ,ydesk +4 );

if( quoi ){
      /* nbsol= 8+5*niveau; */
      nbmax= 13;
      ttx = ttj;
      sens = tob[2][INTERASC].ob_state & SELECTED;
      }
else {
      /* nbsol = 4 + 4*niveau ;*/
      nbmax = 8;
      ttx = tti;
      }

 reinit( tob[2], quoi, ttx ,nbmax);

/* dessinner */
for ( i=0;i<3;i++) {
           objc_draw( tob[i],0,7,xdesk,ydesk,wdesk,hdesk);
           }

wind_update( BEG_MCTRL );

while ( ! fin ){
    graf_mkstate( &x,&y,&k,&v );
    if( k != 1 ) continue;
    for(i=0;i <3;i++){
         if( (o = objc_find(tob[i],0,7,x,y )) > 0 ) break;
         }

    if( o && i<3) {
    switch( i ){

      /* changement de niveau */
      case 0: niveau ^= 1;
        sprintf(texte,"% ld",niveau + 1 );
        objc_text( tob[0],o,texte,(WORD)1 )
```

(suite)

```
                }
            if( rep < 0 ) break;
            if( tob[2][o].ob_state & (SELECTE|DIDISABLED)) break;
            select_ob( tob[2],o,(WORD) 1);
            nbrep ++;

            if (solu !=rep) {
               char repo[10];
               printf("\7");
               PAUSE(500);
/* si trop de réponses: donner la solution et solu = -1*/
               EFF_BOX;
               VTEXT( 50,YCOORD(80),TEXT(46));
               if( quoi ){
                  if( nbrep > ( (niveau ) ? 2: 3 )) donne solu = 1;
                  if( sens == 1){
                  repo[0] = ton;
                  repo[1] = ton + rep;
                        }
                  else{
                  repo[0] = ton;
                  repo[1] = ton - rep; }
                  repo[2] = FIN;
                  }
            else {
                     ACCORD *ac;

                     if( nbrep > ((niveau) ? 2: 1))donne_solu == 1;
                        ac = & tabaccord[ rep ];
                        repo[0] = ton;
                        for( j = 0;j< ac->nbn ;j++){
                        repo[ j+1 ] = ton + ac->interv[j];
                                    }
                        repo[j+1] = FIN;
                        }
                     tempojeux -= 20;
                     play ( repo,-1 );
                     tempojeux += 20;
                     EFF_BOX;
                     PAUSE ( 1000);
                     VTEXT( 50,YCOORD(80),TEXT(47));

                     play( repi,-1 );
                     PAUSE( 500 );
                     EFF_BOX;
                     }
               else {
```



(suite)

```
                    }
                    repi[j+1] = FIN;
                    }
                EFF_BOX;
                play( repi,-1 );
                nbrep = 0;
                break;

            case TBECOUTE:
                if ( solu < 0 ) printf("\7" );
                else play( repi,-1 );
                EFF_BOX;
                break;

            case TBQUITTE:
                fin = 1;
                break;
            }
        break;

    case 2:
        if ( quoi ){
            if( o == INTERASC){
                if(sens!=1){
                sens= 1;
                select_ob(tob[2],o,(WORD)1);
                deselect_ob(tob[2],(WORD)INTERDES(WORD)1);
                solu = -1;
                EFF_BOX;
                }
            break;
            }
            else if( o == INTERDES ){
                if(sens!=-1){
                sens = -1
                select ob(tob[2],o,(WORD)1);
                deselect_ob(tob[2],(WORD)INTERASC(WORD)1);
                solu = -1;
                EFF_BOX;
                }
            break;
            }
        }
        if( solu < 0 ) printf("\7" );
        else{

            for( rep = nbmax-1; rep>=0;rep--) {
                    if( o == ttx[rep] ) break;
```

(suite)

```
{
EFF_BOX;
if( quoi )
    if( rep < 0 )
       strcpy( texte,TEXT(115));
    else
       sprintf(texte,TEXT(52), ttni(rep) );
 else {
       if(rep >=8||rep < 0)
          sprintf( texte,TEXT(53));
       else if( ! renversement )
          sprintf(texte,TEXT(54),ttna(rep));
       else
          sprintf(texte,TEXT( 1 22),renversement,ttna(rep));
       }
    VTEXT(50,YCOORD( 80 ),texte);

}

WORD enregistre ( buf,n,pfct, emode)
WORD buf[],n;
int (*pfct)();

WORD emode;
{
WORD fin=0;
WORD n1=0;
int c;
long usp;
long a;
/*extern WORD shift;*/

static WORD f;
static WORD tm,mm;


if( emode & 1 ){
         enjeux= 1;
         /* R.A.Z file MIDI */
         tm = taille_mesure;
         taille_mesure = 0;
         mm = metronome;
         metronome = 0;
         mode = 2-get_piste( 0,PI_MODE ); /* MIDI, 2 = ATARI */

         installe_vecteurs( );

/* FILTRE F_NOTES */
```



(suite)

```
                    donne_solu= 1;
                    EFF_BOX;
                    VTEXT( 50,YCOORD( 80 ),TEXT(48));

                    }
                  if( donne_solu ){
                    donne_solu = 0;
                    MOUSE_OFF;
                    v_gtext(handle, 50,YCOORD( 100 ),TEXT(49));

                    if( ! quoi ) {
            sprintf( texte," %s %s ", TEXT( 93+(ton-72) ), types[solu] );
                    v_gtext(handle,50,YCOORD( 120 ),texte);
                    for( j = 0; repi[j]!=FIN;j++)
            v_gtext( handle,50+j*50,YCOORD(140),strnote( repi[j] ));

                 else {
            v_gtext( handle,50,YCOORD(120), ttni(solu) );
            v_gtext( handle, 50,YCOORD(140), stmote( repi[0] ));
            v_gtext( handle, 100,YCOORD(140), stmote( repi[1] ));
                    }
                    MOUSE_ON;
                      /* forcer o à bonne valeur */
                    clignote( tob[2], ttx[solu] );
                    solu = -1;
                    }
                  }
              break;
            }
          }
    }
wind_update( END_MCTRL );
form_dial( FMD_FINISH,0,0,0,0,xdesk,ydesk,wdesk,hdesk);
sonje( 0 ); }

static void affquest( quoi ,solu)
WORD quoi ,solu;
{
EFF_BOX;
if( quoi )
    sprintf(texte,TEXT(50), ttni(solu) );
else
    sprintf(texte, TEXT(51),ttna(solu) ,tabaccord[solu].nbn + 1);
VTEXT( 50,YCOORD( 80 ),texte);
}

static void affmorep( quoi,rep )
WORD quoi ,rep;
```

(suite)

```
rsrc_gaddr( R_TREE,(WORD)TROISOUT,&tob[1]);

strcpy( (char *)(tob[1][TBECOUTE].ob_spec),ttc(1) );

set_coord(tob[0],540,ydesk+4 )
set_coord( tob[1],10 ,jdesk + hdesk - tob[1]->ob_height );
if( quoi ){
      nbmax= 13;
      }
else {
      nbmax = 8;
      }

for ( i=0;i < 2;i++) {
      objc_draw( tob[i],0,7,xdesk,ydesk,wdesk,hdesk);
      }

/* enregistre( 0,0,0,1 ); */
while ( ! fin ){
/*

    while (( dc = get_midin( )) != 0xfe ){
      long a ,usp;
      a = Kbshift( -1 );
      if( a&16 || a&1 ) dc += 12;
      if(mode==2) {
        usp= Super( 0L );
        synth_in( dc );
        Super( usp );
        }
    }
*/
 graf_mkstate( &x,&y,&k,&v );
 if( k != 1 ) continue;
 for(i=0;i <2,i++){
    if( (o = objc_find(tob[i],0,7,x,y )) > 0 ) break;
    }

    if(o&&i<2) {
      switch( i ){

        /* changement de niveau *
        case 0: niveau ^= 1;
             sprintf(texte,"%ld",niveau + 1);
             objc_text( tob[0],o,texte,(WORD)1);
             solu = -1;
             PAUSE( 200 );
             break;
```



(suite)

```
          f = get_filtre( F_NOTES );
          }
if( emode & 8 ) reset_filtre( f );
if(emode&2){
          while ( ! fin ) {
             /* a = Kbshift( -1 );
             if ( a &17) shift = 12;
             else shift = 0;
             */
             if( (c = get_midin( )) != 0xfe ){
                  if( mode == 2 ) {
                       usp = Super( 0L );
                       synth_in( c );
                       Super( usp );
                       }
                  if( ! filtrer(f,(UBYTE) c ) && ! ISSTAT( c )){
                  if( ! fgetinfo( f, (WORD )F_INMES )){
                       /* message terminé ? */
                       /* SI NOTOF */
             if( ((fgetinfo( f,(WORD)F_RUNST )>>4)==8)|| !c ){
                            buf[n1]= fgetinfo(f,(WORD)F_L2DAT)
                            n1 ++:
                            }
                         }
                       }
                  }
             if (kbhit( )){
                  switch( getch()){
                            case 8:
                              n1 = -1;
                            case 13:
                              fin=1;
                            }
                       }
               if( (n1==n) ||(pfct && (*pfct)()) )
                  fin= 1;
               }
          }

if( emode & 4 ){
       mode = 0; /* ne pas recuperer l'enregistrement */
       taille_mesure = tm;
       metronome = mm;

       reinit_vecteurs( );
       enjeux = 0;
       put filtre( f );
       }
```

(suite)

```
                        }

                    else rep = analyse( bufmid, rep );

                    if( rep != solu ){

                    /* afficher la reponse donnée */
                       affmorep( quoi, rep );
                       printf("\7");
                       PAUSE( 2000 );
                       Affquest( quoi,solu );
                       }
                    else { /* c'est tout bon */

                       VTEXT( 50,YCOORD( 80 ),TEXT(60));
                       if( renversement ){
          sprintf( texte ,TEXT( 1 22),renversement,ttna(solu))
                          VTEXT( 50,YCOORD( 100 ),texte);
                          }
                       solu = -1;
                       }
                  }
               }
          else printf("\7");
          break;

        case TBQUITTE: fin =1; break;
        }
      break;

     }

   }
 }
/*enregistre(0,0,0,4);*/
wind_update( END_MCTRL )
form_dial( FMD_FINISH,0,0,0,0,xdesk,ydesk,wdesk,hdesk);
sonje( 0 );
}
```



(suite)

```
                if( tabaccord[a].interv[j] != bb[j] ) break;
                }

        if( j==n ) {trouve=1;break;} /* SI TOUS BON: OK */

                    /* RENVERSER */
        nn = bb[0];
        bb[0] = bb[1] - bb[0];
        if( n == 3 ) {
          bb[1] = bb[2] - nn;
          bb[2] = 12 - nn;
          }
        else {
          bb[1]=12 - nn;
          }
        }
      if (trouve ) break;
      }
n++;
renversement = (n-i) %n;
return( a );
}

jeu( quoi )
WORD quoi;
{

OBJECT *tob[ 2 ];
WORD i;
WORD fin =0, x,y,k,v;
WORD o;
WORD rep;
WORD bufmid[40];
WORD nbnotes,nbmax;
WORD solu;
int dc;


MOUSE_OFF;
printf("\033E" );
MOUSE_ON;
sonje( 1 );
wind_update( BEG_MCTRL );

sprintf(texte,TEXT(55),(quoi) ? TEXT(56): TEXT(57));
barre_texte( menu, texte, hlettre );

rsrc_gaddr( R_TREE,(WORD)NIVEAU ,&tob[0]);
```



(suite)

```
          case 1:
               switch( o ){

                  case TBJOUER
                     solu = random( nbmax );
                     if( ! niveau) {
                        while ( DENIVEAU_2( solu ) ) {
                           solu ++;
                           if( solu >= nbmax ) solu = 0;
                           }
                        }
                     nbnotes = ( quoi ) ? 2: tabaccord[solu].nbn+1;
                     EFF_BOX;
                     affquest( quoi ,solu );
                     while( clic_sur_objet( tob[i],o));
                     break;

               case TBECOUTE: /* ENREGISTREMENT de Quoi */

                     if( solu >= 0) {
                        select_ob( tob[i],o,(WORD)1);
                        VTEXT( 50,YCOORD( 100 ),TEXT(58));
                        /* enregistre( 0,0,0,8);      */
                        rep = enregistre( bufrnid, nbnotes,NULL, 7 );
                        if(rep > 1) {
                           sqsort( bufmid,rep );
                           }
                        deselect_ob( tob[i],o,(WORD) 1 );
                        EFF_BOX;

                        if ( rep != nbnotes ){
                             printf("\7");
                             VTEXT( 50,YCOORD( 100 ),TEXT(59));
                             PAUSE( 2000);
                             EFF_BOX;
                             affquest(quoi,solu);
                           }

                        /* analyser bufmid selon quoi et solu,
                           renvoyer rep */
                        else {

/* GERER OCTAVE, % 12 -> UNISSON */

                        if( quoi ){
                             rep = bufrmid[1] - bufmid[0];
                             if( rep > 12 )
                                rep = -1;
```

(suite)

```
int  filesel(masque,  def,  affichage)        /* def = default, */
     char masque[], def[], affichage[]; /*   réservé   en   C        */
{
static  char path[64],
            fnam[13];
short       back,
            button,
            i;

   strcpy (fnam, def);

   path[0] = 'A' + Dgetdrv(); /* Nom du lecteur courant */
   path[1] = ':';

   Dgetpath  (&path[2],  0);        /* Chemin du lecteur courant */

   strcat  (path,  "\\");           /* \\ signifie \ */
   strcat (path, masque);

   back = fsel_input (path, fnam, &button);
   if (button == 0 || fnam[0] == '\0')
     return (0); /* La fonction = 0 si quitte ou erreur de sélection*/

   for  (i  =  strlen(path)  -  1;     path[i]  !=  '\\';     path[i--] = '\0');
   strcat (path, fnam);
   strcpy (affichage, path);

   return (1);
}
```

Le second utilitaire vous permet de connaître la longueur d'un fichier en utilisant le DTA. Notez qu'il vous est facile de transformer ce programme pour avoir d'autres renseignements sur le fichier en modifiant l'instruction dta + =26 et en prenant l'offset qui vous intéresse dans la liste des paramètres.



4/4.4

Utilitaires de programmation en C

Le premier programme utilitaire de gestion de fichier vous permettra facilement de consulter, nommer, renommer et réaliser des back-up (backer( )) de fichiers. Le lecteur sélectionné est le lecteur courant fournit par la fonction Dgetdrv( ) et les deux antislash sont synonymes de simple antislash.

```
/*****************************************************************/
/*     Consulter un nom de fichier dans le sélecteur de fichiers  */

#include <osbind.h>
#include <gemlib.h>
#include <stdlib.h>
#include <string.h>
#include <dos.h>

void backer( s ) char *s;

{

static char node[15],ext[5],buf[64];
/*
static char chemin[64],drive[14];
*/



        strsfn( s,NULL,NULL,node,ext );
        ext[0] = 'B';
        strmfn(  buf,NULL,NULL,node,ext  );

        rename( s,buf );

}
```

```
#include <osbind.h>
#include <portab.h>
#include <gemlib.h>
#include <stdlib.h>

extern    WORD handle;
extern    WORD a_couleur;


#define objc_d( x, y, z ) objc_draw( x,y,z,x->ob_x,
x->ob_y,x->ob_width,x->ob_height )
#define set_coord( ob, x,y ) ob->ob_x = x;
ob->ob_y = y
#define menu_off( ) objc_d(menu,0,1)

#define MOUSE_ON  graf_mouse(M_ON,0 )
#define MOUSE_OFF graf_mouse(M_OFF,0 )


static WORD kbon;

static void gkbclick( )
{
char *s=0;
 s += 0x484;
 *s &= 0xfe;
 *s |= kbon;

}

void kbdclick( on )
WORD on;
{
kbon = (on)?1:0;
Supexec( gkbclick );
}


char *objc_gettext( ob,index)
OBJECT *ob;
WORD index;
{
TEDINFO *ted;

switch( ob[index].ob_type ){
      case G_TEXT:
      case G_BOXTEXT:
      case G_FTEXT:
      case G_FBOXTEXT:
                  ted = ( TEDINFO *) ob[index].ob_spec;
                  return( ( char *) ted->te_ptext );
```



```
#include <osbind.H>
#include <portab.h>


int filelength( s )
char *s;
{
 long l= 0;
 char *dta ;

 if( ! Fsfirst( s, 0 )){
 dta =(char *)Fgetdta( );
 if( dta ) {
      dta += 26;
      l  =*(( long  *)dta);
      }
 }
return( l );
}
```

UTILITAIRES GEMEXTEN

Ces utilitaires ont pour but de gérer plus facilement les menus, les sélections d'objets et de formes ainsi que les clipping d'écran. Les fonctions de dessin et prise d'objets sont relativement classiques, si ce n'est la transformation des objets par leurs adresses relatives -> et pointeurs *. La gestion de ressources classiques par textes, boîtes et boutons en est grandement facilitée. La valeur de non-saisie d'objet est −1 et liée à l'existence du caractère s.

(suite)

```
        default : return( -1 );
        }
if( red )
        objc_d( ob,index,1);

return( 0 );
}

WORD get_car( )
{
register long a;
register WORD b;

a = Crawcin( );
b = a>>8;
a &= 0xff;
b+=a;
return( b );
}

swappe_ob( ob, index ,red)
OBJECT *ob;
WORD index, red;
{
register WORD mode ;
mode = (ob+index)->ob_state;
mode ^= SELECTED ;
objc_change(ob,index,0,ob->ob_x,ob->ob_y,
ob->ob_width,ob->ob_height,mode ,red  );
return( mode & SELECTED );
}



/* selectionne un objet :
            entree : l'arbre , l' indice , un flag dessin
            sortie : rien
            traitement  :       appel objc_change avec
tous ses parametres

            INTERET : gain de place , perte de
vitesse sans incidence
*/
void select_ob( ob, index ,red)
register OBJECT *ob;
register WORD index, red;
{
ob[index].ob_state |=SELECTED;
if (red)
   objc_draw(ob,index,7,ob->ob_x,ob->ob_y,
ob->ob_width,ob->ob_height);
}
```



(suite)

```
        case G_BUTTON:
        case G_STRING:
        case G_TITLE:
                return( (char *) ob[index].ob_spec );
                break;

        }
return( NULL );
}

WORD objc_text( ob,index,s,red )
OBJECT *ob;
char *s;
WORD index, red;
{
TEDINFO *ted;
register WORD l,l1;
register char *d;

if( ! s ) return( -1 );
switch( ob[index].ob_type ){
        case G_TEXT:
        case G_BOXTEXT:
        case G_FTEXT:
        case G_FBOXTEXT:
                ted = ( TEDINFO *) ob[index].ob_spec;
                d = ( char *) ted->te_ptext;
                l = strlen( s );
                l = ( l < ted->te_txtlen )? l: ted->te_txtlen;
                strncpy( d, s, l);
                d[l] = 0;

                break;

        case G_BUTTON:
        case G_STRING:
        case G_TITLE:
                d = (char *) ob[index].ob_spec ;
                l = strlen( d );
                l1 = strlen( s );
                strncpy( d , s , (l1>l)?l:l1  );
                while( l < l1 ) d[l++]=' ';
                d[l1] = 0 ;
                break;

        case G_BOXCHAR:
                ob[index].ob_spec &= 0x0fff ;
                l = *s;
                ob[index].ob_spec |= l << 24;
                break;
```

(suite)

```
char    *m_realloc( s , n)
char    *s;
unsigned n;
{
UBYTE *d;
if( ! s  ) d= malloc( n );
else d= realloc( s,n);
return( d );
}


void barre_texte( menu ,texte ,hlettre)
OBJECT *menu;
char *texte;
WORD hlettre ;
{
WORD y;

objc_d(menu,0,1);
y = 4*(80 - strlen(texte));
v_gtext(handle,y,hlettre,texte  );



}


/* redessiner la zone x,y,w,h si elle est chez nous */

void redraw( wh,x,y,w,h,pfct)

WORD wh, x, y, w, h;
/* Nouvelle zone à dessiner */
void (*pfct)() ;

{
WORD rx, ry, rw, rh;
/* Variables pour la liste des rectangles */
extern WORD pxyarray[4],clipping;


  wind_update  (1);
/* Désactiver souris, bloquer la liste des rectangles */

  /* Appel du premier rectangle de la liste: (11) */

  wind_get (wh, 11, &rx, &ry, &rw, &rh);
```



(suite)

```
/* idem dans le principe que ci-dessus
mais réalise le contraire ! */
void deselect_ob( ob, index ,red)
register OBJECT *ob;
register WORD index, red;

{
ob[index].ob_state &= ~SELECTED;
if( red )
objc_draw(ob,index ,7,ob->ob_x,ob->ob_y,
ob->ob_width,ob->ob_height);
}


/* calcule un rectangle intersection dans pxyarray */
WORD intersect (x1, y1, w1, h1, x2, y2, w2, h2, pxyarray)
WORD x1, y1, w1, h1, x2, y2, w2, h2, pxyarray[];

{
WORD x, y, w, h;

   w = (x2+w2 < x1+w1) ? x2+w2 : x1+w1;
   h = (y2+h2 < y1+h1) ? y2+h2 : y1+h1;
   x = (x2 > x1) ? x2 : x1;
   y = (y2 > y1) ? y2 : y1;

   pxyarray[0] = x;
   pxyarray[1] = y;
   pxyarray[2] = w - 1;
   pxyarray[3] = h - 1;

   return ((WORD) (w > x && h > y) );
}

clic_sur_objet( arbre, index)
OBJECT *arbre;
WORD index;
{
WORD x,y,b,k;
graf_mkstate( &x,&y,&b,&k);
if ( b && objc_find(arbre,0,7,x,y)==index){
    return( b );
    }
return( 0 );
}
```



(suite)

```
  while ( rw )       /* Tant que largeur > 0... */
  {
    if (intersect (x, y, w, h, rx, ry, rw, rh, pxyarray))
    {
      vs_clip (handle, clipping=1, pxyarray);

      (*pfct)(pxyarray[0],pxyarray[1],
pxyarray[2]-pxyarray[0]+1,pxyarray[3]-pxyarray[1]+1 );
    }
  wind_get (wh , 12, &rx, &ry, &rw, &rh);
/* Rectangle suivant */
  }

  vs_clip  (handle,  clipping=0,  pxyarray);
  wind_update  (0);      /* Nous avons termine */

}

void line( x,y,x2,y2)
WORD x,y,x2,y2 ;
{
WORD pxyarray[4];

pxyarray[0] = x;
pxyarray[1] = a_couleur ? y/2: y;
pxyarray[2] = x2;
pxyarray[3] = a_couleur ? y2/2:y2;
MOUSE_OFF;
v_bar( handle,pxyarray );
MOUSE_ON;

}

void box( x,y,w,h )
WORD x,y,w,h;
{
WORD pxyarray[4];

y = a_couleur? y/2 : y;
h = a_couleur? h/2 : h;
pxyarray[0] = x;
pxyarray[1] = y;
pxyarray[2] = x+w-1;
pxyarray[3] = y+h-1;

MOUSE_OFF;
v_bar( handle,pxyarray );
MOUSE_ON ;
}
```



4/5

Le langage Pascal

GÉNÉRALITÉS

Avant d'entrer dans les détails du langage, il convient de rappeler quelques éléments fondamentaux relatifs aux structures manipulées par le Pascal.

Quel que soit son constructeur, le Pascal possède des instructions qui permettent de :

— faire des entrées/sorties sur l'écran, formatées ou non ;

— structurer le programme (boucles de répétition interactives contrôlées, procédures, fonctions) ;

— travailler sur de nombreux types de données.

Il est recommandé de ne faire figurer qu'une instruction par ligne pour augmenter la lisibilité des programmes.

Chaque ligne doit obligatoirement se terminer par un caractère terminateur : le point virgule.

ENTRÉES/SORTIES SUR L'ÉCRAN

Pour écrire un texte ou la valeur d'une variable sur l'écran, on utilise l'une des deux instructions suivantes :

— **Writeln** si l'on désire un passage à la ligne après affichage,
— **Write** dans le cas contraire.

Ces deux instructions doivent être suivies d'une parenthèse ouvrante pour délimiter le début des données à afficher, et d'une parenthèse fermante pour délimiter la fin des données à afficher.

Par exemple : **Writeln('A =',A);**

permet d'afficher le texte **A =** suivi de la valeur contenue dans la variable A.

Le résultat d'une instruction **Writeln** (ou **Write**) peut être formaté s'il représente un nombre. On peut, en effet, donner le nombre de chiffres désirés avant et après le point décimal.

Par exemple, si nous désirons trois chiffres avant le point décimal et deux après, nous ferons :

```
Writeln('A=',A:3:2);
```

Pour lire une valeur entrée au clavier et la stocker dans une variable, on utilise l'une des deux instructions suivantes :

— **Readln** si l'on désire un passage à la ligne après la saisie,
— **Read** dans le cas contraire.

Ces deux instructions doivent être suivies d'une parenthèse ouvrante pour délimiter le début des données à afficher, et d'une parenthèse fermante pour délimiter la fin des données à afficher.

Par exemple :

```
Readln(A,B);
```

permet de lire deux valeurs tapées au clavier et de les stocker dans les variables A et B.

STRUCTURE DU PROGRAMME

Boucles répétitives :

Nous distinguerons cinq types de boucles répétitives :

Repeat/Until
For/To/Do
If/Then/Else
Case
Goto <label>

Repeat/Until

La structure

```
Repeat
   <instruction>;
   ...
   <instruction>;
Until <condition>;
```

permet de répéter une ou plusieurs instructions jusqu'à ce qu'une condition soit réalisée.

For/To/Do

La structure

```
For<variable>:=<valeur 1> To <valeur 2> Do
begin
   <instruction>;
   ...
   <instruction>;
end;
```



permet de répéter une ou plusieurs instructions une ou plusieurs fois. Le nombre de répétitions est égal à <valeur 2> − <valeur 1>.

If/Then/Else

La structure

```
If <condition>Then
begin
   <instruction>;
   ...
   <instruction>;
end
else
begin
   <instruction>;
   ...
   <instruction>;
end;
```

permet d'exécuter une ou plusieurs instructions si la condition spécifiée est réalisée, et éventuellement une ou plusieurs instructions dans le cas contraire.

Case

La structure

```
Case <variable> of
   <Valeur 1> : <une ou plusieurs instructions>;
   ...
   <Valeur N> : <une ou plusieurs instructions>;
   ELSE <une ou plusieurs instructions>;
end;
```

permet d'effectuer un traitement spécifique pour chaque valeur spécifiée d'une variable, et éventuellement un autre traitement si la variable a une valeur différente de toutes celles spécifiées.

Goto<label>

La structure

```
Goto <Label>
```

ne doit être utilisée qu'en cas d'absolue nécessité. En effet, elle supprime l'aspect « structuré » inhérent à la programmation en Pascal. L'instruction Goto permet de donner le contrôle à une instruction qui se situe à l'étiquette spécifiée.

TYPES DE DONNEES

La plupart des Pascal permettent de manipuler des données complexes dont le type est l'un des suivants :

- tableaux mono et multi dimensionnels ;
- chaînes de caractères ;
- types définis par le programmeur ;
- fichiers à accès séquentiel, indexé ou direct ;
- pointeurs.

4/6

Le langage Logo

Le Logo est un langage évolué particulièrement destiné aux débutants et aux enfants. Il met à la disposition des non-informaticiens les possibilités graphiques de l'ordinateur sur lequel il fonctionne.

La version de Logo livrée avec votre Atari est très évoluée et présente des avantages bien plus attrayants que sur ses ancêtres qui fonctionnaient sur Apple II ou Commodore C64. Cette version permet non seulement de dessiner avec la tortue, mais également de manipuler des expressions logiques, numériques ou chaînes, des fichiers sur disquette ou disque dur, et d'activer le générateur sonore AY3.

Avantages

Un des grands avantages de ce langage réside dans le fait que l'utilisateur a la possibilité de créer ses propres instructions sous la forme de mots clés. Ces mots clés peuvent être utilisés par la suite comme faisant partie du langage. Cette démarche est également employée dans les langages de haut niveau comme le Pascal ou le Lisp dans lequel on définit des procédures qui sont activées par leur nom.

Inconvénients

Le langage Logo utilise la plupart des possibilités de l'Atari. Par contre sa portabilité[1] est très faible.

De plus, ce langage interprété est particulièrement lent, et, par là-même, il ne pourra pas être utilisé pour des applications professionnelles ou qui demandent des temps de traitement optimal. Le Logo doit être considéré comme un premier langage. Il est souvent utilisé par les débutants en informatique et représente un tremplin efficace vers des langages plus évolués comme le Basic, le Pascal ou le C...

1) La portabilité est la faculté d'exécuter un programme écrit dans le même langage sur deux machines différentes.

COMMENT EXPLOITER TOUTES LES RESSOURCES ET AUGMENTER LES PERFORMANCES DE VOTRE

ATARI®

ST 520/1040 et Méga

5e Complément

Pierre Bedrossian
Jean-Marc Campaner
Jean-Paul Carré
Philippe Casier
Didier Cackel
John Faure
Jean Guérard
J.-M. Guisiano
Michel Martin
Michel Riquart

EDITIONS WEKA
Paris - Kissing - Zurich - Milan - Amsterdam - Vienne - Londres - New York

© Copyright

Atari 520 ST, 1040 ST, MEGA ST2, MEGA ST4 : Atari Corp.
Basic GFA : GFA
Basic ST : Atari Corp.
Calcomat 2 : Micro Application
Cyber studio : Upgrade
Degas Elite : Electronics Arts
GEMDOS : Digital Research Inc.
IBM : IBM Corporation
K-RAM : Kuma Software
K-Ressource : Kuma
GST Holdings Limited
Timeworks Publisher
Lattice C : Lattice
LDW Power : Upgrade
Le Rédacteur : Libération
Lotus 1.2.3 : Lotus
MAD MAC : Atari Corp.
Motorola : Motorola
Mutil : Michtron
Turbo C : Borland International Inc.
Spectrum 512 : Upgrade
Wercs : Hisoft
ZZ lazy paint : Human Technologies

Tous droits de traduction, d'adaptation et de reproduction de la version française par tous procédés réservés pour tous pays

La loi du 11 mars 1957 n'autorisant, aux termes des alinéas 2 et 3 de l'article 41, d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale, ou partielle, faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause, est illicite » (alinéa 1er de l'article 40).

Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles 425 et suivants du Code pénal.

5e Complément

© 1990 Editions WEKA, 82, rue Curial, 75935 Paris cedex 19
Tél. : (1) 40 37 01 00 - Télex 210 504

Editions WEKA - Paris - Kissing - Zurich - Milan - Amsterdam - Vienne - Londres - New York

En Suisse : WEKA Verlag, Hermetschloostr. 77, 8010 Zurich

Editeur : Brigitte Morvant
Secrétariat d'édition : Brigitte Chevallier
Fabrication : Tina Lê Xuân

Composition : Compo-Méca - 64990 Mouguerre
Corlet imprimeur s.a. - 14110 Condé-sur-Noireau
Reproduction interdite - Tous droits réservés
Imprimé en France, 1990

Dépôt légal (ouvrage de base) : 2e trimestre 1989

ISBN 2-7337-0048-0

COMMENT EXPLOITER TOUTES LES RESSOURCES ET AUGMENTER LES PERFORMANCES DE VOTRE

ATARI®

ST 520/1040 et Méga

6e Complément

Pierre Bedrossian
Jean-Marc Campaner
Jean-Paul Carré
Philippe Casier
Didier Cackel
John Faure
Jean Guérard
J.-M. Guisiano
Michel Martin
Michel Riquart

EDITIONS WEKA
Paris - Kissing - Zurich - Milan - Amsterdam - Vienne - Londres - New York

© Copyright

Atari 520 ST, 1040 ST, MEGA ST2, MEGA ST4 : Atari Corp.
Basic GFA : GFA
Basic ST : Atari Corp.
Calcomat 2 : Micro Application
Cyber studio : Upgrade
Degas Elite : Electronics Arts
GEMDOS : Digital Research Inc.
IBM : IBM Corporation
K-RAM : Kuma Software
K-Ressource : Kuma
GST Holdings Limited
Timeworks Publisher
Lattice C : Lattice
LDW Power : Upgrade
Le Rédacteur : Libération
Lotus 1.2.3 : Lotus
MAD MAC : Atari Corp.
Motorola : Motorola
Mutil : Michtron
Turbo C : Borland International Inc.
Spectrum 512 : Upgrade
Wercs : Hisoft
ZZ lazy paint : Human Technologies

Tous droits de traduction, d'adaptation et de reproduction de la version française par tous procédés réservés pour tous pays

La loi du 11 mars 1957 n'autorisant, aux termes des alinéas 2 et 3 de l'article 41, d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale, ou partielle, faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause, est illicite » (alinéa 1er de l'article 40).

Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles 425 et suivants du Code pénal.

6e Complément

© 1990 Editions WEKA, 82, rue Curial, 75935 Paris cedex 19
Tél. : (1) 40 37 01 00 - Télex 210 504

Editions WEKA - Paris - Kissing - Zurich - Milan - Amsterdam - Vienne - Londres - New York

En Suisse : WEKA Verlag, Hermetschloostr. 77, 8010 Zurich

Editeur : Brigitte Morvant
Secrétariat d'édition : Brigitte Chevallier
Fabrication : Tina Lê Xuân

Composition : Compo-Méca - 64990 Mouguerre
Corlet imprimeur s.a. - 14110 Condé-sur-Noireau
Reproduction interdite - Tous droits réservés
Imprimé en France, 1990

Dépôt légal (ouvrage de base) : 2e trimestre 1989

ISBN 2-7337-0048-0

COMMENT EXPLOITER TOUTES LES RESSOURCES ET AUGMENTER LES PERFORMANCES DE VOTRE

ATARI®

ST 520/1040 et Méga

7e Complément

Pierre Bedrossian
Jean-Marc Campaner
Jean-Paul Carré
Philippe Casier
Didier Cackel
John Faure
Jean Guérard
J.-M. Guisiano
Michel Martin
Michel Riquart

EDITIONS WEKA
Paris - Kissing - Zurich - Milan - Amsterdam - Vienne - Londres - New York