

ATARI
ST+STE

LE LIVRE
OMIKRON®
BASIC

EDITIONS MICRO APPLICATION



LIVRE DATA BECKER

Michael MAIER

Le Livre de
L'OMIKRON[®]
BASIC

EDITIONS MICRO APPLICATION

Copyright

© 1988

DATA Becker GmbH
Merowingerstraße, 30
4000 Düsseldorf

© 1990

Micro Application
58, rue du Faubourg Poissonnière
75010 Paris

Auteur

Michael MAIER

Traducteur

M. et Mme BAUDIN

Toute représentation ou reproduction, intégrale ou partielle, faite sans le consentement de MICRO APPLICATION est illicite (Loi du 11 Mars 1957, article 40, 1er alinéa).

Cette représentation ou reproduction illicite, par quelque procédé que ce soit, constituerait une contrefaçon sanctionnée par les articles 425 et suivants du Code Pénal.

La Loi du 11 Mars 1957 n'autorise, aux termes des alinéas 2 et 3 de l'article 41, que les copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à l'utilisation collective d'une part, et d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration.

ISBN : 2-86899-340-0

*Collection dirigée par Philippe Olivier
Edition réalisée par Frédérique Beaudonnet*

OMIKRON BASIC est une marque déposée de OMIKRON.

Atari est une marque déposée de Atari Corp.

GFA Basic est une marque déposée de GFA Systemtechnik GmbH.

Préface

Il y a peu de temps encore, il m'aurait paru impensable d'écrire un livre sur le Basic, moi qui écris la plupart de mes programmes en langage C, voire en assembleur lorsque cela s'avère indispensable.

Je gardais un souvenir indélébile de ma première rencontre avec le ST-BASIC fourni en standard avec toutes les machines de la gamme Atari ST. On ne pouvait entrer la moindre instruction sans recevoir immédiatement une ribambelle de petites bombes à l'écran ! Par prudence, je n'ai ensuite plus jamais infligé un tel traitement à mon ordinateur.

Les adeptes de l'Atari ST pensèrent un moment que la solution allait venir de l'éditeur Metacomco, qui travaillait alors sur un nouvel interpréteur Basic. Lorsque ce produit fut enfin mis sur le marché, il était effectivement débarrassé de ses défauts originels, mais il montrait bien que la rapidité n'était pas le souci majeur de ses concepteurs.

Il aura donc fallu attendre quelques années pour qu'Atari Corp. se décide à doter sa gamme complète de machines - dans les familles ST et STE - d'un langage digne de ce nom.

Le choix se fit en faveur de l'OMIKRON BASIC, que l'on peut tout simplement appeler Basic ST à partir de sa version 3.00. La firme Atari a ainsi refermé une page bien sombre de l'histoire des ST. Terminée, l'époque où le programmeur en Basic devait se procurer un interpréteur d'une autre firme uniquement à cause des défauts du Basic livré avec le matériel.

Les possibilités toujours plus nombreuses offertes par ce langage de programmation n'ont certes pas simplifié l'écriture des programmes. Il n'est plus indispensable de passer par un langage compilé pour écrire de bons programmes, ceci est tout à fait réalisable avec le Basic. Il n'en reste pas moins qu'il faut pour ce faire non seulement posséder le langage en question mais aussi avoir une bonne connaissance de base du fonctionnement interne de l'ordinateur.

Et nous en venons au point essentiel : voilà que le Basic, depuis longtemps décrié comme le langage de programmation le plus simple susceptible de n'intéresser que les novices, permet maintenant de réaliser des programmes de niveau professionnel. Evidemment, cette nouvelle orientation a un prix : elle implique la perte de cette simplicité qui a fait le succès du Basic. Soulignons toutefois qu'il est toujours beaucoup plus facile d'écrire un programme en Basic qu'en langage compilé ou même en assembleur.

Notre manuel s'adresse à tous ceux qui souhaitent explorer les possibilités offertes par ce nouveau Basic-ST. J'espère que vous profiterez des informations qu'il vous propose et que de cette lecture enrichissante naissent les programmes dont vous rêvez depuis longtemps.

Sommaire

1.	L'éditeur	11
1.1.	Chargement de l'Omikron Basic	12
1.2.	L'éditeur écran standard	12
1.3.	Le menu de l'éditeur pleine page	17
1.4.	Les commandes de l'éditeur pleine page	31
1.5.	Pour sortir du Basic Omikron	36
2.	Les principes fondamentaux de l'OMIKRON	37
2.1.	L'affichage écran par PRINT	37
2.2.	Les différents types de variables	41
2.3.	Votre premier programme	49
2.4.	Sauver, charger et détruire un programme	55
2.5.	Comment le dire à mon ordinateur ?	58
2.6.	Les fonctions mathématiques	62
2.7.	Manipulation des 'Strings'	68
2.8.	Les tableaux de variables	94
2.9.	Les aides à la programmation	96
2.10.	La programmation structurée	102
2.11.	Question de routine(s)	124
2.12.	READ, DATA et RESTORE	142
2.13.	Tout sur le Basic	146
3.	Gestion des fichiers	185
3.1.	Les fichiers sur disquette	186
3.2.	On ne peut rien faire sans les canaux	189
3.3.	Encore un PRINT, mais aussi WRITE	191
3.4.	Comment lire les fichiers séquentiels	193
3.5.	Comment recopier des fichiers	196
3.6.	Les boîtes de sélection d'objet : file-selector-box	200
3.7.	Parer aux erreurs de manipulation	210
3.8.	Les fichiers back-up	213
3.9.	Clarification du contenu de la disquette	217

3.10.	Les fichiers relatifs	219
3.11.	Un mini-fichier... relatif cette fois	223
3.12.	Pourquoi réinventer la roue ?	235
3.13.	Noir sur blanc : la sortie impression	240
3.14.	Equation de recherche complexe avec OR et AND	244
4.	Le système d'exploitation de l'Atari ST	245
4.1.	Les variables-système	245
4.2.	TOS, GEMDOS, BIOS et XBIOS	250
4.3.	GEMDOS	253
4.4.	BIOS	263
4.5.	XBIOS	268
4.6.	Programmation sous le système d'exploitation	281
5.	La programmation graphique	291
5.1.	Les commandes graphiques simples	292
5.2.	BITBLT	295
5.3.	L'affichage à l'écran	298
5.4.	Déplacer et retourner des graphiques	299
6.	GEM	303
6.1.	Comment travailler avec RCS	304
6.2.	La programmation GEM sous l'Omikron Basic	313
6.3.	Les 'ascenseurs' (sliders)	344
6.4.	Les menus déroulants	346
6.5.	Comment créer votre boîte de sélection ?	356
6.6.	La technique des fenêtres	368
7.	Multitasking	377

8.	Le compilateur	381
8.1.	Comment utiliser le compilateur	382
8.2.	Les commandes	383
8.3.	Les programmes en BASIC et le compilateur	386
8.4.	Comment optimiser un programme	388
8.5.	Les messages d'erreur envoyés par le compilateur	389
8.6.	Les types de variables retournés par les fonctions	390
8.7.	Les autres programmes utilitaires sur la disquette du compilateur	391

ANNEXES :

A	Tableau des codes ASCII	395
B	Tableau des codes SCAN de l'Atari-ST	405
C	Liste des fonctions du standard VT-52	407
D	Messages d'erreur envoyés par l'interpréteur	409
F	Les messages d'erreur envoyés par le TOS	417
G	Conversion des listings en Basic GFA	419
	Index	421

Chapitre 1

L'éditeur

Il est rigoureusement impossible d'écrire un programme sans maîtriser les outils de base indispensables. Dans le cas de notre Omikron Basic utilisé sur Atari-ST, les outils en question se présentent sous la forme de deux éditeurs vous permettant d'entrer les commandes, d'écrire des programmes entiers, de les sauvegarder, de les reprendre pour les modifier ou de les recharger. C'est pourquoi nous y consacrons tout ce premier chapitre.

Il est malheureusement impossible de décrire à fond ce genre de programme sans recourir à des termes techniques du jargon informatique. Je vous conseille de commencer par lire ce chapitre d'un bout à l'autre : toutes les fonctions de base, comme par exemple le chargement de l'Omikron Basic, y sont décrites aussi simplement que possible. Les fonctions plus compliquées sont entièrement reprises et réexpliquées plus en détail dans la suite de l'ouvrage.

Je vous recommande d'ailleurs de relire ce premier chapitre après avoir pris connaissance de l'ensemble du contenu de l'ouvrage : vous constaterez alors que vous en comprendrez toutes les fonctions.

1.1. Chargement de l'Omikron Basic

Avant même de pouvoir travailler avec l'éditeur, il vous faut commencer évidemment par charger le Basic depuis la disquette. Pour ce faire, insérez la disquette Omikron dans le lecteur de disquette et exécutez, à l'aide du bouton gauche de la souris, un double-clic sur l'icône de l'unité A. En langage clair placez la flèche de la souris en haut à gauche de l'écran sur l'icône appelé "DISQUE A" puis appuyez très rapidement deux fois sur le bouton gauche de la souris. Vous voyez apparaître sur votre écran une fenêtre vous montrant tous les fichiers contenus sur la disquette.

Amenez le curseur de la souris sur OM-BASIC.PRG et lancez le processus de chargement en effectuant de nouveau un double-clic toujours à l'aide de la touche gauche de la souris. Dès que le programme est chargé en mémoire vive, vous vous retrouvez dans l'éditeur standard.

Les explications qui suivent se rapportent toutes à l'éditeur du nouveau Basic ST (Basic Omikron version 3.01). Cependant, pour que tous nos lecteurs travaillant avec des versions antérieures à la version 3.00 puissent profiter de ce manuel, j'ai ajouté ci-après un chapitre entièrement consacré à l'éditeur pleine page des versions antérieures. Ce chapitre intéressera d'ailleurs tous les utilisateurs du Basic-ST car il contient une description des possibilités supplémentaires offertes par le recours à des touches de fonction simplifiant l'utilisation de l'éditeur.

1.2. L'éditeur écran standard

Une fois le chargement de l'Omikron Basic terminé, vous voyez apparaître en haut à gauche de votre écran un petit encadré comportant le numéro de la version du programme utilisé. Quelques lignes plus bas, vous apercevez un petit rectangle clignotant : c'est le curseur, que vous pouvez déplacer dans les quatre directions en vous servant des quatre <touches fléchées> qui se trouvent entre votre clavier et le pavé numérique.

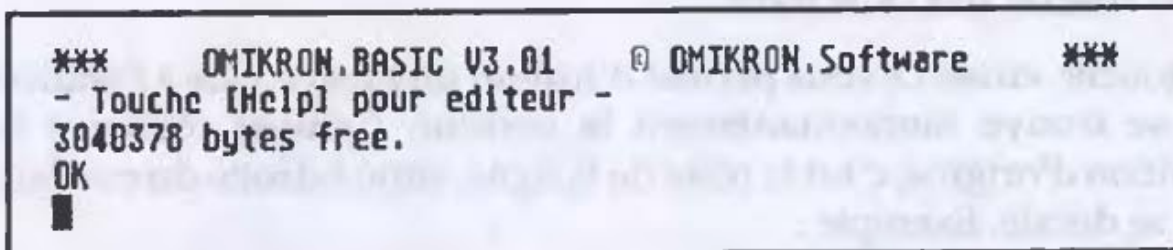


Figure 1.1 : L'écran après le chargement de l'éditeur standard

Vous amenez le curseur en haut à gauche de votre écran en appuyant sur la touche <home> ; si vous appuyez simultanément sur <control>+<home>, vous voyez que l'écran se vide et que le curseur va se positionner en haut à gauche. Toutes les commandes qui suivent, concernent l'éditeur standard, c'est-à-dire directement après le chargement du Basic sans appuyer sur <HELP>.

☐ Pour effacer des caractères

La touche <backspace> sert à effacer le caractère se trouvant immédiatement à gauche du curseur : le curseur se déplace d'un espace vers la gauche, tandis que le reste de la ligne ne change pas de position.

La touche <delete> sert aussi à effacer un caractère, mais le reste de la ligne se déplace cette fois d'un espace vers la gauche. Voici un petit exemple qui illustre ces propos. Tapez la ligne suivante :

Le livre de l'OMIKRON*BASIC

L'étoile représente ici la position du curseur ; si vous appuyez sur <backspace>, vous obtenez :

Le livre de l'OMIKRO* BASIC

Si par contre, vous appuyez sur <delete>, vous voyez que 'BASIC' se décale d'une position vers la gauche, ce qui évite l'apparition d'un espace vide :

Le livre de l'OMIKRO*BASIC

Pour insérer des caractères

La touche <insert> vous permet d'insérer un espace vide à l'endroit où se trouve momentanément le curseur. Celui-ci conserve sa position d'origine, c'est le reste de la ligne, situé à droite du curseur, qui se décale. Exemple :

Le livre de l'OMIKRON BASIC

Lorsque le curseur se trouve par exemple sur le 'v' et que vous appuyez sur <insert>, cela fait apparaître un espace vide qui vous permet ensuite d'insérer le caractère manquant. Exemple :

Le livre de l'OMIKRON BASIC

Insérer ou détruire des lignes

Lorsque vous souhaitez insérer ou détruire non pas seulement un caractère mais une ligne entière, vous appuyez sur les touches suivantes :

<control>+<touche fléchée vers le haut>

efface la ligne sur laquelle se trouve le curseur ;

<control>+<touche fléchée vers le bas>

insère une ligne ;

<control>+<delete>

efface tout ce qui se trouve à droite du curseur sur la ligne.

☐ Le mode 'insert' (insertion)

En mode normal, le caractère qui se trouve sous le curseur est remplacé par un autre signe lorsque vous appuyez sur la touche d'un autre caractère. Cela ne correspond pas toujours à vos besoins. Lorsque vous souhaitez insérer un nouveau caractère là où se trouve le curseur, sans pour autant écraser le caractère qui se trouve déjà à cet endroit, vous devez passer en mode 'insert' (insertion) en appuyant sur les touches

<control>+<insert>

pour ressortir de ce mode, il vous suffit de réappuyer sur la même combinaison de touches ; une fois en mode 'insert', vous constatez que les touches <delete> et <backspace> n'ont plus tout à fait le même effet.

<backspace>

en mode 'insert', cette touche a le même effet que la touche <delete> en mode normal ;

<delete>

en mode 'insert', cette touche sert à effacer le caractère se trouvant sous le curseur tout en provoquant le décalage du reste de la ligne ; attention, le curseur par contre ne se décale pas vers la gauche mais reste à la même position !

□ L'émulateur VT52

D'une certaine façon, on peut dire que le moniteur Atari est "intelligent". En effet, il comprend certaines instructions, comme par exemple effacer tout le contenu de l'écran ou mémoriser la position actuelle du curseur etc. Toutes ces commandes sont introduites en les faisant précéder d'un appui sur la touche <escape> suivi d'une (ou plusieurs) lettre(s). Vous trouverez la liste de toutes les commandes du standard VT52 dans les annexes de ce manuel.

Vous pouvez utiliser toutes ces fonctions lorsque vous travaillez sous l'éditeur standard Omikron. Il vous suffit d'appuyer tout d'abord sur la touche <escape> puis d'entrer les caractères correspondant à la commande souhaitée, qui est exécutée immédiatement.

□ Liaison des lignes (link)

Le moniteur de l'Atari accepte au maximum 80 caractères par ligne. Lorsque vous saisissez une ligne plus longue, l'éditeur continue à écrire tout simplement en passant à la ligne suivante ; toutefois, pour que vous puissiez repérer facilement ce genre de ligne dont le contenu n'est que la fin de la ligne précédente, l'éditeur vous place un repère en début de ligne qui est le trait vertical (|) appelé 'pipe' dans le jargon du métier. Il en va de même lors de l'affichage d'une ligne qui dépasserait les 80 caractères admis par le moniteur : la ligne se poursuit sur la ligne suivante, qui commence alors par le trait vertical (|).

□ Autres combinaisons de touches

<return>

lorsque la ligne commence par un nombre, la touche <return> sert à inscrire, dans la mémoire de programme, la ligne que vous venez d'écrire ; dans les autres cas, l'interpréteur identifie le contenu de la ligne comme étant une commande qu'il cherche à exécuter dès que vous appuyez sur <return> : c'est pourquoi on parle alors de 'mode direct'.

<control>+<curseur gauche>

sert à définir la position du coin supérieur gauche d'un cadre ;

<control>+<curseur droite>

sert à définir la position du coin inférieur droit d'un cadre.

Le cadre sert à délimiter un bloc.

L'affichage (par exemple à l'aide de PRINT) ne se fait plus sur toute la surface de l'écran mais uniquement dans les limites du cadre défini ; notez que le curseur ne se déplace plus qu'à l'intérieur de ce cadre.

- <home><home>** vous effacez le cadre en appuyant deux fois consécutivement sur la touche <home> : vous pouvez de nouveau déplacer votre curseur sur toute la surface de l'écran.
- <alternate>+<control>** cette combinaison de touches sert à vider le buffer des touches du clavier ; le Basic-ST mémorise dans ce buffer toutes les touches qui ont été appuyées mais sans avoir été immédiatement suivies d'effet (par exemple lorsque vous appuyez sur une touche pendant que le programme est en train d'exécuter une tâche quelconque). Quand ce buffer est plein et que l'on continue à appuyer sur une touche, on entend alors un "beep" sonore.
- <shift>+<shift>** lorsque vous appuyez simultanément sur les deux touches <shift> de l'Atari ST, le système revient à l'écran utilisé par le programme avant son retour au mode direct.

1.3. Le menu de l'éditeur pleine page

L'éditeur pleine page (full-screen-editor, appelé aussi super-éditeur dans la documentation Omikron) sert généralement à saisir des textes de programmes complets alors que l'éditeur standard sert surtout à travailler en mode direct (exécution immédiate des commandes entrées).

Le super-éditeur est doté d'une série de fonctions facilitant considérablement la saisie de vos programmes. Entre autres choses, vous disposez ainsi d'une barre d'accès à des menus déroulants à partir desquels vous pouvez lancer diverses fonctions.

Vous passez dans ce super-éditeur tout simplement en appuyant sur la touche <help> lorsque vous vous trouvez dans l'éditeur standard ; pour en ressortir, appuyez sur <control>+<c> ou cliquez sur l'option 'quit edit' qui se trouve dans le menu 'file'.

☐ La barre des titres des menus déroulants

La barre des menus s'affiche tout en haut de votre écran ; elle contient les titres des menus suivants :

File, Find, Block, Mode, Go et Run.

Pour 'ouvrir' un de ces menus 'déroulants', il vous suffit d'amener la flèche de la souris sur le titre de l'un des menus : vous voyez alors que le menu se 'déroule' en quelque sorte très rapidement du haut vers le bas, montrant ainsi les diverses fonctions auxquelles il donne accès. Amenez le curseur de la souris sur l'une des options offertes, et vous constatez qu'elle passe en inversion-vidéo ; on dit que vous 'sélectionnez' l'une des options contenues dans un menu lorsque vous cliquez sur son intitulé une fois qu'il est passé en inversion-vidéo.

La barre des titres des menus dissimule encore une autre fonction : vous pouvez cliquer sur l'un des titres des menus, ce qui lance alors la fonction figurant en tête de ce menu.

Vous constatez que les titres et les options de menu sont formulés en langue anglaise, car ils correspondent le plus souvent à des instructions du Basic : vous trouverez entre parenthèses une traduction en français.

☐ Le menu 'File' (fichiers)

Ce menu rassemble toutes les fonctions servant à charger ou enregistrer les textes des programmes. Il utilise fréquemment la "boîte de sélection d'objet" qui permet de choisir un nom de fichier parmi ceux existants sur la disquette en cliquant dessus ou de taper le nom directement au clavier.

Save *.* (sauvegarder, enregistrer) :

cette fonction vous sert à enregistrer le texte de programme se trouvant dans la mémoire vive dans un fichier dont vous indiquez le nom dans la boîte de sélection d'objet (file selector box).

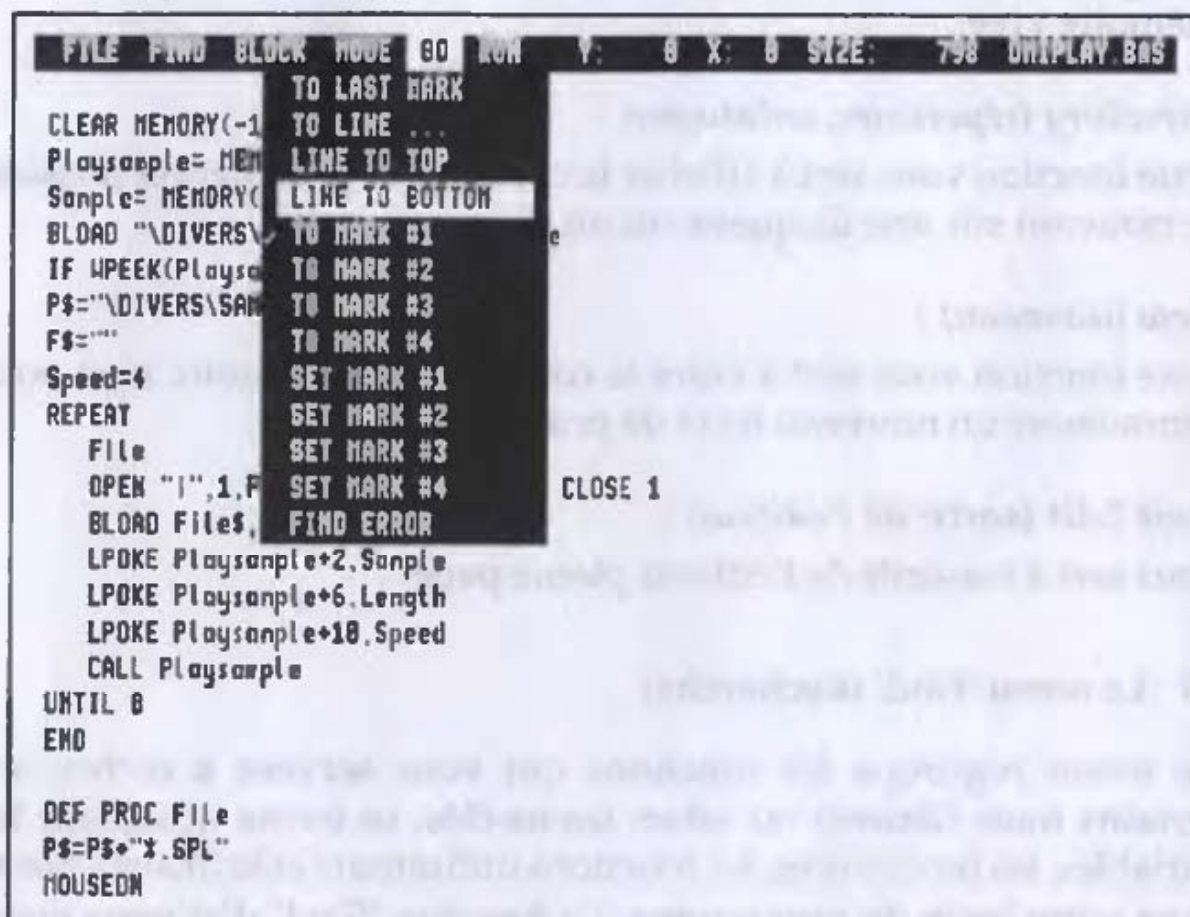


Figure 1.2 : L'éditeur pleine page (super-éditeur)

Load *.* (charger) :

cette fonction sert à recharger dans la mémoire vive le texte d'un programme dont vous indiquez le nom dans la boîte de sélection d'objet.

Save block *.* (enregistrer un bloc) :

cette fonction sert à enregistrer le contenu d'un bloc que vous avez auparavant délimité.

Load block *.* (charger un bloc) :

cette fonction sert à recharger un bloc (ou un texte en ASCII) sauvegardé auparavant par 'save block' sur un disque ou une disquette ; le bloc vient s'insérer à l'endroit désigné par les numéros de ligne dans le texte de programme présent éventuellement dans la mémoire vive.

Directory (répertoire, catalogue) :

cette fonction vous sert à afficher le catalogue des fichiers et dossiers se trouvant sur une disquette ou un disque.

New (nouveau) :

cette fonction vous sert à vider le contenu de la mémoire vive pour commencer un nouveau texte de programme.

Quit Edit (sortir de l'éditeur) :

vous sert à ressortir de l'éditeur pleine page.

□ Le menu 'Find' (Recherche)

Ce menu regroupe les fonctions qui vous servent à rechercher certains mots (lettres) ou token (mots-clés, ce terme désignant les variables, les procédures, les fonctions utilisateurs et les instructions) dans votre texte de programme. La fonction 'Find' distingue entre les mots ordinaires et les mots-clés (tokens). Lorsqu'il s'agit de mots ou de chaînes de caractères ordinaires, l'interpréteur vous signale l'endroit où il les a retrouvés dans le texte du programme.

Lorsqu'il s'agit de tokens, il en va un peu différemment. Pour épargner au maximum la place occupée en mémoire vive, l'interpréteur Basic ne mémorise pas les codes de commande en toutes lettres (print) mais sous la forme de codes abrégés que l'on appelle 'tokens'. La recherche d'un token dans le texte d'un programme ne peut donc se faire qu'en convertissant le code complet (tel que l'écrit le programmeur) en token. Notez qu'à côté des

commandes à proprement parler, les variables sont elles aussi consignées sous forme de token et ne sont redécrites en texte clair que peu avant leur affichage à l'écran.

Si vous utilisez l'option 'Find token' (recherche d'un token) pour rechercher un nom de variable, vous pouvez être certain que le système ne vous signalera bien que la variable arborant ce code et non pas les variables dont le nom pourrait contenir la même combinaison de lettres.

Pour rechercher une variable de tableau à l'aide de l'option 'Find token', vous formulez votre recherche de la façon suivante :

Array() :

Omikron recherche une variable accompagnée d'un seul indice ; cela ne permet pas par exemple de retrouver *Array(5,9)*

Array(,,) :

Omikron recherche une variable accompagnée de trois indices. En règle générale, lorsque vous souhaitez rechercher une procédure, une fonction ou une étiquette (label), vous devez observer les règles suivantes pour formuler votre recherche :

- le nom de la procédure à rechercher doit être précédé d'un 'P'; si vous entrez par exemple :

P Ligne()

Omikron recherche la procédure 'Ligne' accompagnée d'un seul paramètre. Lorsque votre programme contient plusieurs procédures arborant le même nom et ne se distinguant que par le nombre de paramètres, vous entrez (à partir du 2ème paramètre) une virgule par paramètre ; vous entrez par exemple :

P Ligne(,)

pour rechercher la procédure Ligne suivie de 2 paramètres.

- le nom de la fonction à rechercher doit être précédé de FN lorsque vous la recherchez à l'aide de l'option 'Find token'.
- la recherche des étiquettes se fait en faisant précéder leur nom d'un signe moins.

☐ Les options du menu 'Find'

Find next (recherche de l'occurrence suivante)

permet de poursuivre la recherche de la même chaîne de caractères, que vous avez formulée sous l'option se trouvant juste en-dessous :

Find ... (recherche)

vous permet d'entrer la chaîne de caractères que vous voulez rechercher ; lorsque l'ordinateur retrouve la chaîne demandée, il positionne le curseur à l'endroit où elle apparaît ; s'il ne la retrouve pas, vous entendez un signal sonore.

List ... (lister)

Omikron parcourt tout le texte de votre programme et vous envoie une liste des endroits où il a retrouvé la chaîne de caractères demandée.

Replace all ... (remplacer toutes les occurrences)

L'interpréteur parcourt l'ensemble du texte de programme en y recherchant la chaîne de caractères demandée, qu'il remplace immédiatement par la nouvelle chaîne de caractères indiquée.

Query replace ... (remplacement sélectif, à la demande)

L'interpréteur recherche une chaîne de caractères, à chaque fois qu'il la trouve, il vous envoie un message demandant s'il doit ou non la remplacer par la nouvelle chaîne indiquée.

Find token ... (rechercher un mot-clé)

L'interpréteur recherche des commandes (ou variables) ; il ignore les variables contenant la même combinaison de caractères.

List token ... (lister les mots-clés)

L'interpréteur recherche les tokens demandés, et vous les signale en provoquant l'inversion-vidéo du passage où ils apparaissent.

Rename Token (remplacer un mot-clé)

sert à renommer un mot-clé.

List to printer (sortir les listes vers l'imprimante)

Une fois cette option activée, l'interpréteur sortira toutes les listes demandées sur l'imprimante.

Find error (recherche d'une erreur)

Le curseur vient se placer devant ou derrière le mot dans lequel l'interpréteur a reconnu une erreur de syntaxe (utile car très courant !).

❑ Le menu 'Block'

Ce menu regroupe les fonctions permettant de manipuler des blocs dans l'éditeur pleine page. Avant de pouvoir manipuler un bloc, il vous faut le délimiter ; vous pouvez, pour ce faire, procéder de deux façons :

- ❶ cliquez avec la souris là où vous voulez positionner le coin supérieur gauche du bloc, puis 'tirez' la souris vers le bas, sans relâcher la touche gauche, pour l'amener jusqu'à la fin du bloc (coin inférieur droit) : relâchez alors la touche gauche de la souris, votre bloc est ainsi défini ;
- ❷ placez votre curseur sur le premier caractère du bloc, sélectionnez l'option 'Mark block start' (marquer le début du bloc) : vous venez de définir le début du bloc ; amenez ensuite votre curseur à la fin du bloc à marquer, puis sélectionnez l'option 'Mark block end' (marquer la fin du bloc) : voilà votre bloc défini.

Il convient de distinguer deux sortes de blocs :

- le bloc ne contenant qu'une seule ligne ou même un morceau de ligne
- le bloc contenant plusieurs lignes : il doit toujours commencer au début d'une ligne et se terminer à la fin d'une autre ligne.

□ Les options du menu 'Block'

Insert (insérer)

Cette fonction sert à recopier le bloc délimité là où se trouve le curseur.

Move (déplacer)

sert à déplacer le bloc pour le remettre là où se trouve le curseur : le bloc n'est pas recopié puisqu'il disparaît de son ancien emplacement (nuance !).

Kill (supprimer)

sert à détruire irrémédiablement le bloc.

Mark block start (marquer le début du bloc)

La ligne sur laquelle se trouve le curseur devient le début du bloc : cette fonction sert à délimiter le début du bloc à créer ou à déplacer la limite supérieure d'un bloc déjà défini.

Mark block end (marquer la fin du bloc)

La ligne sur laquelle se trouve le curseur est repérée comme étant celle de la fin du bloc.

*Save block *.* (enregistrer un bloc)*

sert à enregistrer sur disque(tte) le bloc défini, sous sa forme ASCII.

*Load block *.* (charger un bloc)*

sert à recharger un bloc et à l'insérer là où se trouve le curseur.

Print block (imprimer un bloc)

sert à sortir le contenu d'un bloc sur l'imprimante.

Hide (désactiver le marquage du bloc)

sert à effacer le marquage délimitant le bloc, sans pour autant effacer le bloc lui-même ; ses limites sont conservées.

☐ Le menu 'Mode' (mode d'affichage écran)

Ce menu regroupe toutes les fonctionnalités qui concernent l'affichage à l'écran, comme par exemple le passage en mode 'insertion' ou 'recouvrement', le passage d'un écran à l'autre ou le partage de l'écran en deux fenêtres.

Insert (insertion)

En cliquant sur cet item, vous entrez dans le mode insertion (l'item est alors précédé d'un crochet) ; vous en ressortez en re cliquant dessus.

Switch screen (passer d'une fenêtre à l'autre)

Le Basic-ST est capable de gérer deux demi-écrans après que vous ayez activé l'option suivante 'Split screen' ; l'option 'Switch screen' vous sert alors à passer d'une partie de l'écran à l'autre ; vous obtenez d'ailleurs le même effet en cliquant dans la partie de l'écran concernée à l'aide de la souris.

Split screen (partager l'écran en deux parties)

Cette option vous sert à partager l'écran en deux parties dans le sens horizontal : une barre noire assez large sépare alors les deux parties. Par défaut, l'écran se divise en deux parties égales lorsque vous lancez l'option ; par contre, vous pouvez cliquer sur la barre de séparation et la déplacer en maintenant la touche gauche de la souris enfoncée, ce qui vous sert à modifier la taille respective des deux parties de l'écran. Pour passer d'une fenêtre à l'autre, vous activez l'option 'switch screen' (voir ci-dessus). Attention : vous ne pouvez

afficher que deux parties différentes d'un même programme. Toute modification d'une partie du programme dans l'une des fenêtres entraîne les modifications correspondantes dans l'autre fenêtre.

Change size (modifier la taille des caractères)

Cette option sert à changer de taille de caractères, ce qui vous permet d'afficher à l'écran des passages plus importants de votre texte de programme ; il vous suffit ensuite de réactiver l'option pour revenir à la taille antérieure.

Line numbers (afficher la numérotation des lignes)

Le Basic-ST vous permet de programmer avec ou sans la numérotation des lignes en passant par cette option.

☛ **Attention :** Si vous sortez de l'éditeur pleine-page lorsque vous travaillez sans numérotation de ligne, l'interpréteur se met à renuméroter les lignes une par une, mais sans modifier pour autant les indications de sauts !

Show errors (afficher les erreurs)

L'interpréteur parcourt le texte du programme en recherchant les erreurs éventuelles.

Save settings (sauvegarder les paramètres)

Cette option sert à sauvegarder les paramètres dans un fichier nommé OMIKRON.INF, qui sera rechargé automatiquement lors du prochain rechargement de OMIKRON. Voici la liste des paramètres ainsi sauvegardés :

- les définitions des touches de fonction
- la résolution de l'écran du moniteur
- le mode de fonctionnement (insertion ou recouvrement)
- la numérotation ou non des lignes
- les variables entrées sous DEFXXX

Line to bottom (afficher en bas de l'écran la ligne actuelle)

Le texte affiché à l'écran défile vers le bas jusqu'à ce que la ligne sur laquelle se trouve le curseur se retrouve en bas de l'écran.

To mark #X (sauter jusqu'à la marque X)

Les quatre options 'To mark #1' à 'To mark #4' servent à amener le curseur sur l'une des quatre marques.

Set mark #X (placer la marque #X)

Les quatre options 'Set mark #1' à 'Set mark #4' servent à définir la position de l'une des quatre marques.

Find error (rechercher une erreur)

L'interpréteur parcourt le texte du programme à la recherche d'une erreur éventuelle.

☐ Le menu RUN (lancer un programme)

Cette option sert à appeler et lancer un programme dont vous venez d'écrire ou de charger le texte.

Run (lancer, exécuter un programme)

Cette option vous permet de sortir du super-éditeur tout en lançant l'exécution du programme se trouvant dans la mémoire vive ; vous pouvez obtenir exactement le même effet en appuyant sur les touches <control>+<R>.

Save & Run (sauvegarder puis lancer le programme)

Cette option vous permet de sauvegarder votre programme avant qu'il ne soit exécuté (chaudement recommandé !).

Tron & Run (activer le mode TRACE puis lancer le programme)

Cette option permet d'activer le mode TRACE (TRON) avant que le programme ne soit lancé.

Compile (compiler)

Cette option sert à appeler le compilateur OMIKRON (version 3.00 et au-delà) ; si vous possédez un compilateur portant un numéro de version inférieur à 3.00, vous devez tout d'abord enregistrer votre programme en Basic.

Save & compile (sauvegarder puis compiler)

Cette option permet de sauvegarder le programme avant qu'il ne soit compilé (concerne surtout les versions antérieures à 3.00). Attention : L'éditeur recherche le compilateur dans l'unité de disque C (si vous l'avez déclaré) ou sinon dans la disquette A.

Run *.Bas (lancer un programme *.Bas)

Cette option vous permet de charger et lancer immédiatement un programme (généralement en Basic, d'où l'extension 'Bas') depuis une unité de disque ;

⚠ **Attention :** Le programme se trouvant éventuellement dans la mémoire vive à ce moment sera écrasé, prenez soin de le sauvegarder auparavant !

Exec *.Prg (exécuter le programme *.prg)

Cette option permet de charger et exécuter immédiatement un programme, après quoi vous vous retrouvez de nouveau dans le super-éditeur du Basic-ST. Pour plus de sécurité, il est conseillé de sauvegarder le programme Basic qui se trouverait éventuellement dans la mémoire vive avant d'appeler un autre programme.

Accessory (accessoires)

Cette option permet d'activer un menu GEM qui vous donne accès aux accessoires chargés à l'allumage de l'unité centrale.

□ Les raccourcis par combinaison de touches du clavier

Voici les commandes que vous pouvez lancer en appuyant simultanément sur la touche <alternate> suivie de la touche indiquée :

Touches appuyées	Fonction exécutée
<alternate><A>	ASC(
<alternate>	BLOAD
<alternate><C>	CONT
<alternate><D>	DATA
<alternate><E>	ELSE
<alternate><F>	FOR
<alternate><G>	GOTO
<alternate><H>	HCOPY
<alternate><I>	INPUT
<alternate><K>	KEY
<alternate><L>	LPRINT
<alternate><M>	MID\$(
<alternate><N>	NEXT
<alternate><O>	OPEN
<alternate><P>	PRINT
<alternate><R>	RETURN
<alternate><S>	SYSTEM
<alternate><T>	THEN
<alternate><U>	USING
<alternate><V>	VARPTR
<alternate><W>	WHILE
<alternate><X>	MOUSEX
<alternate><Y>	MOUSEY

Nous en avons terminé maintenant avec le tour des possibilités offertes par le nouveau Basic-ST. Les fonctions décrites dans les chapitres qui vont suivre sont en principe valables aussi pour nos lecteurs possédant des versions plus anciennes du Basic-ST, sauf précision contraire.

1.4. Les commandes de l'éditeur pleine page

Dans toutes les versions du Basic-ST antérieures à 3.00, vous entrez dans l'éditeur pleine-page (full-screen-editor) à l'aide de la commande EDIT (suivie d'un appui sur la touche <return>) et vous en ressortez en appuyant sur <control>+<C>.

☐ Mode insertion ou recouvrement

L'éditeur pleine page est lui aussi doté des modes insertion et recouvrement : la forme du curseur vous indique lequel de ces deux modes est actuellement actif. Lorsque vous êtes en mode 'insertion', le curseur prend la forme d'un trait vertical se transformant en carré dans les intervalles pour redevenir ensuite un trait.

Lorsque vous êtes en mode 'recouvrement', le curseur prend la forme d'un carré clignotant. Vous pouvez passer d'un mode à l'autre en appuyant sur <control>+<insert>. Si vous préférez utiliser les menus, vous ouvrez 'Mode' en amenant le curseur de la souris sur cet intitulé, puis vous cliquez sur l'item 'insert' : lorsque le mode 'insert' est actif, l'item est précédé d'un crochet.

☐ Insérer, effacer, séparer et relier une ligne

Vous avez le choix entre deux manipulations pour insérer une ligne : en appuyant sur <return>, vous ajoutez une ligne vide juste en-dessous de celle sur laquelle se trouve le curseur alors qu'en appuyant sur <F9>, vous créez une ligne vide là où se trouve actuellement le curseur.

La manoeuvre est assez semblable pour effacer une ligne : appuyez sur <shift>+<F9> pour effacer la ligne actuelle du curseur. Pour couper une ligne en deux parties, appuyez d'abord sur <shift>+<F6> puis sur <F9> ; pour 'recoller' deux lignes successives, appuyez d'abord sur <shift>+<F6> puis sur <shift>+<F9>.

☐ Autres fonctions

- <Tab>** sert à faire avancer le curseur de huit caractères
- <Undo>** sert à restaurer le contenu d'une ligne dans la mesure où le curseur s'y trouve encore
- <Esc>** a le même effet que <Undo> mais peut aussi servir à mettre fin à certaines fonctions avant que leur exécution ne soit terminée (par exemple : chargement d'un programme).
- <Home>** renvoie le curseur au tout début du texte de programme, et s'il s'y trouve déjà, à l'envoyer tout à la fin du texte de programme ; en appuyant deux fois sur <Home> vous pouvez à tout moment envoyer le curseur à la fin du texte, lorsqu'il se trouve en plein milieu.
- <control>+<delete> :** sert à effacer la partie de la ligne se trouvant derrière le curseur.
- <alternate> :** sert à interrompre la fonction 'recherche' ou 'remplacement', ainsi que la fonction 'répétition'.

☐ Manipulations des blocs

Pour déterminer le début d'un bloc, positionnez le curseur sur la ligne souhaitée et appuyez deux fois sur la touche de fonction <F7> ; pour fixer la fin du bloc, positionnez le curseur à l'endroit voulu puis appuyez d'abord sur <F7> puis sur <shift>+<F7>.

- <F7>+<F8>** sert à sauvegarder un bloc dans une unité de disque(tte), sous forme de fichier ASCII ;
- <F7>,<shift>+<F8>** sert à charger un bloc se trouvant sous la forme d'un fichier ASCII dans une unité de disque(tte) ;
- <F7>+<F9>** sert à recopier un bloc là où se trouve actuellement le curseur ;
- <F7>,<shift>+<F9>** sert à détruire le contenu d'un bloc.

☐ Recherche et remplacement

- <F2>+<F2> "texte" <return>**
Sert à rechercher, à partir de la position actuelle du curseur, la chaîne de caractères entrée à la place de "texte" ; si vous omettez d'entrer un texte, OMIKRON répète la même recherche que la recherche antérieure.
- <F2>+<F3> "texte" <return>**
Sert à demander une liste de tous les endroits où apparaît la chaîne de caractères demandée.
- <F3>+<F2>**
Sert à rechercher une chaîne de caractères et à la remplacer par une autre chaîne après confirmation ; la recherche reprend aussitôt.
- <F3>+<F3>**
Recherche et remplacement d'une chaîne de caractères par une autre, sans demande de confirmation.

□ Fonction GO

La touche de fonction <F1> permet de sauter jusqu'à la ligne indiquée: voici les combinaisons possibles :

<F1>+<numéro de ligne>

saut jusqu'à la ligne demandée

<F1>+<+>+<Déplacement>

sert à sauter de <Déplacement> lignes vers le bas, exemple :

<F1>+20

<F1>+<->+<Déplacement>

sert à sauter de <Déplacement> lignes vers le haut.

<F1>+<curseur haut> l'écran défile jusqu'à ce que la ligne actuelle s'affiche sur la première ligne en haut de l'écran

<F1>+<curseur bas> l'écran défile jusqu'à ce que la ligne actuelle s'affiche sur la dernière ligne en bas de l'écran

<F10> avancer d'une page dans le texte du programme

<shift>+<F10> reculer d'une page dans le texte du programme.

□ Comment définir des touches de fonction

Appuyez sur <shift>+<F7>: vous pouvez alors attribuer le texte de votre choix aux deux touches de fonction <F4> et <F5> (y compris leur combinaison avec <shift>) ; après avoir saisi le texte souhaité, validez en réappuyant sur <shift>+<F7>.

☐ Repeat (répéter)

Appuyez sur la touche que vous voulez répéter puis sur <F6> : indiquez ensuite le nombre de répétitions souhaité ; validez en appuyant sur <return>, après quoi le caractère apparaît, répété autant de fois que demandé, sur l'écran.

☐ Charger et sauvegarder un programme

Appuyez sur la touche <F8> : une boîte de sélection d'objet vous demande d'entrer le nom du fichier dans lequel vous voulez sauvegarder votre programme. Le texte de programme est enregistré sur disque(tte), sous sa forme ASCII. Pour recharger un programme se trouvant sur disque(tte), appuyez sur <shift>+<F8>. Lorsque le fichier ASCII contient des lignes incompréhensibles pour l'interpréteur OMIKRON, celles-ci s'affichent en inversion vidéo.

☐ Activer/désactiver la numérotation des lignes

En appuyant sur <control>+<Clr>, vous pouvez saisir votre programme sans que les lignes soient numérotées : réappuyez sur la même combinaison de touches pour désactiver cette option.

⚠ Attention : Si vous quittez l'éditeur pleine-page lorsque vous travaillez sans numérotation, le Basic Omikron renumérote une à une toutes les lignes, sans pour autant modifier les ordres de saut. Cela peut facilement mener à des programmes strictement incompréhensibles si vous utilisez les numéros de ligne dans vos ordres de saut.

☐ Partage de l'écran

L'ancien éditeur pleine-page OMIKRON pouvait lui aussi partager l'écran en deux parties indépendantes : cette fonction est lancée en appuyant sur <shift>+<F1>, et vous faites passer le curseur d'une partie à l'autre en appuyant sur <shift>+<F2> (le partage de l'écran se fait ici dans le sens vertical). La combinaison <shift>+<F3> vous permet de modifier le découpage de l'écran (par exemple 44*108 caractères ou 57*128).

1.5. Pour sortir du Basic Omikron

Pour ressortir du Basic Omikron, repassez dans l'éditeur standard, entrez :

SYSTEM

et appuyez sur <return> ; répondez par 'Y' (pour Yes = oui) au message de confirmation qui apparaît à l'écran ou par 'N' (pour No = non) si vous ne voulez plus quitter l'interpréteur.

Chapitre 2

Les principes fondamentaux de l'OMIKRON

Dans le chapitre précédent, vous avez pris connaissance des principes théoriques d'utilisation de l'éditeur, c'est pourquoi je ne m'étendrai pas plus longtemps sur la théorie et vous amènerai le plus rapidement possible à la pratique.

2.1. L'affichage écran par PRINT

Si ce n'est déjà fait, chargez votre Basic OMIKRON et saisissez la ligne suivante :

```
print "Essai"
```

Appuyez sur la touche <return>, et vous voyez apparaître le mot 'Essai' dans la ligne suivante. Vous venez d'apprendre la première instruction permettant d'afficher un morceau de texte à l'écran (dans notre exemple, le mot 'Essai').

Vous voyez que le texte à afficher à l'écran doit être saisi entre guillemets, à la suite de l'instruction PRINT. Que se passe-t-il si vous oubliez les guillemets ? Saisissez :

```
print Essai
```

puis appuyez sur la touche <return>. Sur l'écran, vous ne voyez plus apparaître le mot 'Essai' mais un zéro (0). Qu'est-ce à dire ?

Votre ordinateur distingue les caractères et les variables ; ses créateurs lui ont appris que les caractères doivent se trouver entre guillemets. Dans notre deuxième exemple, l'ordinateur a vainement cherché après ces guillemets, il en a déduit qu'il ne pouvait pas s'agir d'un texte (d'une chaîne de caractères) et qu'il s'agissait donc d'une variable.

Au cours de votre scolarité, vous avez certainement déjà rencontré les variables : elles représentent généralement un nombre quelconque dans une équation. Ainsi par exemple, lorsque vous lisez :

$$x + 2 = 10$$

vous en déduisez que $x = 8$ pour que l'équation soit juste. 'x' est appelé variable car sa valeur peut varier en fonction des autres valeurs (ici 2 et 10). Dans le langage de programmation, il en va à peu près de même : les variables représentent une valeur (un nombre).

Imaginez que chaque variable soit semblable à une "petite boîte" se trouvant dans l'ordinateur. Chaque boîte porte un nom, dans notre exemple ci-dessus, il s'agirait donc d'une petite boîte nommée 'Essai'. Après avoir constaté l'absence de guillemets, l'ordinateur "pense" qu'il ne s'agit pas d'une chaîne de caractères mais d'une variable : il cherche donc s'il possède dans sa 'mémoire' une petite boîte nommée 'Essai'. S'il trouve une petite boîte portant le nom indiqué, il lit son contenu et inscrit immédiatement à l'écran le résultat des fonctions qu'il devait exécuter. S'il ne trouve pas de petite boîte portant ce nom, il vous le signale en affichant un zéro. Evidemment, dans le langage des informaticiens, on ne parle pas de 'petite boîte' mais de 'variable', et c'est bien le mot que nous continuerons à employer ci-dessous.

Reste à expliquer comment nous nous y prenons pour attribuer une valeur à une variable. Pour ce faire, en Basic, on utilise le signe d'égalité (=) qui fonctionne comme un opérateur d'attribution d'une valeur :

1et Essai - 3

Après avoir saisi cet exemple, vous n'oublierez pas d'appuyer sur <return>. C'est important car l'ordinateur ne commence à travailler qu'après avoir reçu un <return>. Son premier travail va consister à rechercher dans sa mémoire s'il possède déjà une variable portant ce nom ; s'il ne l'y trouve pas, il en crée une nouvelle portant ce nom. L'opérateur '=' lui dit qu'il doit attribuer la valeur 3 à cette variable. Par conséquent, nous devrions voir s'afficher le chiffre 3 lorsque nous entrerons :

```
print Essai
```

suivi d'un appui sur <return>. Effectivement, nous voyons bien un 3 s'afficher à l'écran. Vous pouvez d'ailleurs abandonner l'instruction LET, car elle ne change rien au résultat de l'opération : le Basic est assez "intelligent" pour reconnaître seul l'attribution d'une valeur à une variable. Vous pouvez désormais taper :

Essai - 3

Après ce petit aparté sur les variables, je voudrais revenir à l'instruction PRINT qui est finalement l'objet de ce chapitre. Vous savez déjà qu'il faut entrer le texte que vous voulez voir s'afficher à l'écran entre guillemets et précédé de l'instruction PRINT. Lorsque les guillemets manquent, l'ordinateur considère le mot suivant comme une variable et renvoie sa valeur à l'écran. Si vous souhaitez afficher simultanément du texte et des variables, vous écrivez par exemple (sachant que Essai = 3) :

```
print "nous sommes aujourd'hui le";Essai;" septembre"
```

vous voyez s'inscrire à l'écran :

```
nous sommes aujourd'hui le 3 septembre
```

L'ordinateur commence par écrire 'nous sommes aujourd'hui le' puisqu'il s'agit d'un texte entre guillemets. Le point-virgule lui indique qu'il doit inscrire le contenu de la variable 'Essai' non pas sur la ligne suivante mais juste après la première chaîne de caractères :

'nous sommes aujourd'hui le'.

Vient ensuite le contenu de la variable `Essai`, qui correspond à 3 dans notre exemple. L'ordinateur rencontre à nouveau un point-virgule, il sait qu'il doit donc inscrire le reste sur la même ligne.

Vous avez sans doute remarqué que l'ordinateur insère un espace blanc entre 'nous sommes aujourd'hui le' et le chiffre 3, alors qu'il n'en insère pas entre le 3 et le mot 'septembre'. L'explication est très simple : vous savez que 'Essai' est une variable représentant une certaine valeur (ici le chiffre 3). Un nombre peut être positif ou négatif : comme la mention d'un signe n'améliore pas la lisibilité, l'ordinateur n'inscrit un signe devant une valeur que s'il s'agit du signe négatif (-), alors qu'il remplace le signe '+' par un espace blanc. Cette petite astuce a donc été utilisée pour créer l'espace blanc nécessaire.

Résumons :

- l'instruction `PRINT` permet d'afficher du texte ou des variables à l'écran.
- les textes doivent figurer entre guillemets faute de quoi ils sont considérés comme des variables.
- le point-virgule (;) à l'intérieur d'une séquence `PRINT` a pour effet de faire continuer l'affichage sur la même ligne, à la suite du texte déjà affiché.
- lors de l'affichage d'une variable à l'aide de `PRINT`, l'ordinateur remplace le signe positif (+) par un espace vide alors qu'il affiche bien un signe moins (-) dans le cas d'une valeur négative.

2.2. Les différents types de variables

Nous n'avons encore utilisé jusqu'ici qu'une variable, nommée 'Essai', à laquelle nous avons attribué une valeur positive (3). Mais cela n'épuise pas, loin s'en faut, le chapitre des variables. Que se passe-t-il par exemple lorsque nous souhaitons attribuer à cette variable un nombre non-entier, comprenant une virgule ?

```
Essai - 234.18
```

Il suffit de l'entrer avec l'instruction PRINT pour constater que l'ordinateur supprime purement et simplement ce qui se trouve après la virgule. Les chiffres concernés sont par contre conservés lorsque nous écrivons :

```
Essai! - 234.18
```

La comparaison des deux formulations nous montre une seule différence : le nom de la variable est maintenant suivi d'un point d'exclamation. L'ordinateur tient compte alors des chiffres placés après la virgule, car nous lui avons indiqué non seulement le nom de la variable mais aussi ce que nous enregistrons sous ce nom de variable. Nous avons pour ce faire utilisé un 'suffixe' (ou 'postfix' en anglais), un signe placé juste après le nom (Le point d'exclamation).

□ Les variables 'integer' (nombres entiers)

Le terme anglais 'integer' désigne les nombres entiers, c'est-à-dire les nombres sans virgule. Notre première variable 'Essai' (= 3) était donc une variable integer : vous ne rencontrerez pas de gros problème aussi longtemps que vous attribuerez des nombres entiers à vos variables. Les problèmes surgissent par contre dès que vous essayez d'attribuer un nombre décimal (à virgule) à une variable 'integer'. L'ordinateur ne le refuse pas, loin de là ! il se contente d'ignorer tout ce qui se trouve après la virgule car il n'a pas de place pour les prendre dans la variable. Revenons un peu aux principes fondamentaux pour expliquer ce phénomène.

Prenons par exemple le nombre entier (integer) 3548 ; on peut le décomposer rang par rang en une suite d'opérations :

$$3 * 1000 + 5 * 100 + 4 * 10 + 8 * 1$$

On peut de cette façon représenter tous les nombres du système décimal reposant sur les chiffres 0 à 9. En partant de la droite, le premier rang (celui des unités) a toujours une valeur multiple de 1, et ceci s'accroît de dizaine en dizaine à chaque rang. Pour l'ordinateur, tout ce travail est un peu plus compliqué, car il n'utilise pas le système décimal : il ne connaît que deux valeurs, le 0 et le 1. Le rang le plus à droite est toujours là aussi un multiple de 1. Mais, alors que la valeur du multiple de chaque rang croît de dizaine en dizaine dans le système décimal, il ne fait que se doubler dans le système binaire utilisé par l'ordinateur. Ainsi par exemple, le nombre binaire 110101 se décompose de la façon suivante :

$$1 * 32 + 1 * 16 + 0 * 8 + 1 * 4 + 0 * 2 + 1 * 1$$

ce qui donne en système décimal le nombre 53. On a convenu que 8 rangs binaires de cette sorte (bit) seraient toujours réunis en un nombre binaire (octet octet ou 'byte' en anglais). Huit bits ou un octet nous permettent donc de représenter 256 valeurs différentes, c'est-à-dire en système décimal, les nombres 0 à 255.

Evidemment, ces 256 valeurs ne nous permettent pas de faire de grands calculs, c'est pourquoi on utilise deux octets pour les nombres supérieurs à 255. Le deuxième octet représente donc des multiples de 256.

Dans l'ordinateur, le nombre 3548 occupe deux octets :

220 (premier octet ou octet de poids faible)
13 (deuxième octet ou octet de poids fort)

$$3548 = 13 * 256 + 220$$

à l'intérieur de l'ordinateur, il prend l'aspect suivant :

00001101 (premier octet)
11011100 (deuxième octet)

ou 00001101 11011100

Lorsqu'on juxtapose ainsi deux octets pour représenter un seul nombre, on appelle cela un 'word' (terme anglais) ou un 'mot'. Dans notre cas, la variable sera appelée 'Entier Court'. A côté de l'octet et du mot, il existe encore le 'mot long' ('long word') qui réunit quatre octets. Un tel mot long permet de représenter les valeurs comprises entre - 2147483658 et + 2147483658. Nous appellerons notre variable "entier long".

Contrairement aux autres interpréteurs Basic, l'Omikron Basic considère toutes les variables comme des entiers longs, tant qu'ils ne sont pas flanqués d'un suffixe. Vous comprenez maintenant pourquoi nous ne pouvons attribuer un nombre décimal à 'Essai' : pour l'interpréteur, Essai est une variable 'integer' du type mot long entier. Toute tentative de lui attribuer un nombre décimal se voit sanctionnée par la disparition pure et simple de la partie du nombre située après la virgule.

Venons-en maintenant aux suffixes, qui nous permettent de distinguer les différents types de variables :

Type de variable	Valeurs représentées	Suffixe
octet entier (8 bits)	de 0 à 255	%B
entier court (16 bits)	de -32768 à +32767	%W ou %
entier long (32 bits)	de -2147483658 à +2147483657	%L ou rien

Attention : 'integer-byte' (octet entier) ne peut s'utiliser que dans les tableaux de variables (arrays ; sera expliqué plus loin).

Exemple d'utilisation des suffixes :

Variable	Type
Essai%B(X)	Integer-byte (octet entier)
Essai%W	Integer-Word (entier court)
Essai%	Integer-Word
Essai%L	Integer-Long (entier long)
Essai	Integer-Long

□ Les variables à virgule flottante (float)

Alors que les variables de type integer ne vous permettent de représenter que des nombres entiers, les variables à virgule flottante (réel simple ou double précision, float) vous permettent de représenter des nombres décimaux ou des nombres extrêmement grands dépassant largement les valeurs des entiers longs.

L'Omikron Basic distingue les nombres réels simple précision et les nombres réels double précision. L'ordinateur réserve 10 octets pour les réels double précision et 6 octets pour les réels simple précision.

En simple précision, vous pouvez aller jusqu'à environ 9 chiffres significatifs après la virgule (7 seulement sont affichés) alors qu'en double précision, vous pouvez aller jusqu'à 19 positions significatives après la virgule (pour 17 affichées). Le champ des valeurs ainsi représentables avec les réels s'étend jusqu'à 10 à la puissance 4931.

Type de variable	Valeurs représentées	Suffixe
réel simple précision	$\pm 5.11 \cdot 10^{\pm 4931}$!
réel double précision	$\pm 5.11 \cdot 10^{\pm 4931}$	#

Exemples :

Variable	Type
Essai!	nombre réel simple (6 octets)
Essai#	nombre réel double (10 octets)

Ainsi lorsque vous voulez attribuer une constante à une variable double précision, comme par exemple :

Essai# = 1.66#

vous devez la désigner comme une constante du type réel double précision par le suffixe adéquat (ici : #), faute de quoi vous n'attribuez à la variable Essai qu'une constante du type réel simple précision (1.66) ce qui ne correspond pas aux intentions du créateur de l'interpréteur Omikron Basic.

☐ Les variables 'string' (les chaînes de caractères)

Voilà que vous avez fait connaissance avec les variables integer et float, qui servent toutes deux à l'attribution de valeurs numériques. A côté des nombres, l'ordinateur sait aussi reconnaître et manipuler des lettres, plus précisément des caractères. On préfère parler de 'caractères' car les lettres de l'alphabet ne sont qu'une partie du stock de caractères utilisables : la lettre 'G' est un caractère, la parenthèse ouvrante '(' aussi, alors qu'elle n'est pas une lettre.

Dans le langage des informaticiens, on utilise souvent la forme anglaise 'character', abrégée en 'char'. Plusieurs caractères qui se suivent forment une chaîne de caractères ou 'string'. Pour les reconnaître, les variables string sont suivies d'un signe dollar '\$'. Exemple : la variable Test\$ (que l'on prononce 'test-string'), à laquelle nous pouvons attribuer une chaîne de caractères :

test\$ = "Essai"

Il vous suffit d'entrer `Print Test$` pour que le mot 'Essai' s'affiche à l'écran ; rappelons que la chaîne de caractères doit s'écrire entre guillemets pour que l'ordinateur puisse le distinguer d'une variable. si vous entrez :

```
Test$ - Essai  
Test% - "Essai"  
Test# - "Essai"
```

l'ordinateur vous renvoie un message d'erreur **TYPE MISMATCH ERROR** : vous tentez d'attribuer à une variable une valeur qu'elle ne peut prendre en raison du suffixe qui l'accompagne.

☐ Comment déclarer autrement vos variables

Nous venons de voir que vous pouvez déclarer le type de vos variables en utilisant des suffixes. Mais l'Omikron Basic vous offre une autre possibilité, qui consiste à entrer une instruction au début de votre texte de programme. En entrant par exemple

```
DEFINT "A,B,C"
```

vous indiquez à l'ordinateur que toutes les variables commençant dans votre programme par A, B ou C doivent être considérées comme des entiers courts, (deux octets) aussi longtemps qu'un autre suffixe ne vient pas infirmer cette règle.

Lorsque l'ordinateur rencontrera par exemple une variable `A$` dans le cours du programme, il l'interprétera bien comme une chaîne de caractères, indépendamment de l'instruction `DEFINT "A,B,C"` figurant en début de programme. Par contre une variable nommée tout simplement 'Auguste' sera bien traitée comme une variable du type mot entier (de deux octets) puisqu'elle ne comporte aucun suffixe.

Voici les instructions servant à définir les types de variables (nous laissons les formulations anglaises car elles permettent de mieux comprendre les abréviations servant d'instruction) :

Commande	Type de variable
DEFINT	Integer - word (entier court)
DEFINTL	Integer - long (entier long)
DEFSNG	Float - single precision (réel simple précision)
DEFDBL	Float - double precision (réel double précision)
DEFSTR	String (chaîne de caractères)

Exemples d'utilisation :

DEFSNG "A-Z" : toutes les variables commençant par une lettre comprise entre 'A' et 'Z' sont déclarées comme des réels simple précision.

DEFSTR "A,C,F-M" : toutes les variables commençant par les lettres 'A', 'C' et les lettres comprises entre 'F' et 'M' seront traitées comme des variables strings (chaînes de caractères).

Ces instructions doivent toujours se trouver tout au début du texte du programme, sur la première ligne, faute de quoi l'ordinateur les ignore. Lorsque vous avez oublié de faire une telle déclaration dès la première ligne, vous pouvez vous en tirer par une petite astuce : insérez votre déclaration sur la première ligne du programme puis sauvegardez votre texte de programme sous forme ASCII. Il vous suffira ensuite de recharger le texte du programme pour que les variables prennent bien la définition qui leur est conférée sur la première ligne et le tour est joué.

Autre précision : certains autres interpréteurs du Basic considèrent automatiquement les variables dépourvues de suffixe comme des réels simples. C'est pourquoi, s'il vous arrive de reprendre des programmes ou des routines provenant d'autres interpréteurs Basic, vous devez penser à déclarer toutes les variables dépourvues de suffixe comme étant des réels simples en entrant

```
DEFSNG "A-Z"
```


faute de quoi votre Omikron Basic les prendra pour des mots longs entiers et supprimera les parties des nombres se trouvant après la virgule.

☐ Les flags (drapeaux)

Dans les programmes, les flags servent à définir les variables booléennes, pouvant prendre deux états : vrai ou faux. Dans le premier cas (vrai) on attribue la valeur -1, dans le deuxième cas (faux) on attribue la valeur 0.

A quoi cela sert-il ?

Admettons que vous ayez écrit un petit programme de traitement de texte. Vous y insérez un flag qui doit se trouver sur 0 lorsque vous lisez le texte de ce programme. Admettons maintenant que vous ayez procédé à une petite rectification (correction d'une faute de frappe par exemple) : le flag prend alors la valeur -1.

Lorsque vous voulez ressortir de votre texte de programme, l'ordinateur questionne le flag. Lorsqu'il y rencontre la valeur -1, vous pouvez décider que l'ordinateur doit malgré cela enregistrer le texte sans autre formalité, y compris les modifications faites et sortir du texte. Mais vous pouvez aussi décider que, lorsque l'ordinateur rencontre sous ce flag la valeur -1, il doit d'abord envoyer un message attirant l'attention de l'utilisateur sur le fait qu'il va enregistrer le texte avec des modifications.

Pour remplir ce rôle, vous pourriez fort bien avoir recours à une variable du type réel double précision mais ceci revient à gaspiller purement et simplement 10 octets de la précieuse mémoire vive de votre ordinateur. Les concepteurs d'ordinateur en vinrent donc à l'idée astucieuse de créer un type de variable n'admettant que les deux valeurs 'vrai'(-1) ou 'faux'(0), ce qui économise de la place mémoire.

Dans l'Omikron Basic, les flags sont déclarés par le suffixe %F, avec cependant une restriction : ce type de variable, tout comme les variables integer-byte (octet entier), ne peut s'employer que dans des

tableaux (arrays). Vous trouverez un peu plus loin quelques mots d'explication au sujet de ces tableaux 'arrays', ce paragraphe ne traitant que des différents types de variables.

Voilà, nous venons de faire le tour de tous les types de variables existant dans l'Omikron Basic, un peu de repos est nécessaire avant de passer à l'écriture de votre premier programme.

2.3. Votre premier programme

Vous avez été jusqu'à présent abreuvé de théorie, mais si vous tenez à écrire un programme, autant qu'il soit intéressant :

Sur la Côte d'Azur, la pension de famille 'Dupont' loue des chambres d'hôte d'un rapport différent à des touristes. Le propriétaire de cet établissement, Monsieur Dupont, s'est acheté un ordinateur qui doit lui faciliter son travail. Comme il reçoit beaucoup de touristes allemands, il tient à afficher ses prix en Francs français mais aussi en D.Mark, et à calculer les sommes dues dans chacune de ces deux monnaies.

Voici votre premier travail de programmation : vous allez tenter de faire un petit programme capable de convertir immédiatement en D.Mark les sommes exprimées en Francs français.

Pour simplifier, nous admettrons dans un premier temps que 10FF valent environ 3DM ; le change s'établit donc aux alentours de 10 pour 3, ce qui nous donne (arrondi à une valeur à quatre décimales) : $10/3 = 3,3333$. En divisant le montant des factures exprimé en FF par ce taux de change (3.3333) Monsieur Dupont obtiendra donc l'équivalent en D-Mark.

Nous supposerons (provisoirement) que Monsieur Dupont n'émet que des factures donnant des sommes rondes en Francs français, sans recourir aux centimes. Le montant des factures peut donc prendre la forme d'une variable integer. Pour le taux de change, nous nous servons d'une variable de type réel simple précision (6 octets) qui suffit amplement pour nos calculs. Pensons au résultat de notre

division, qui doit prendre aussi la forme d'une variable, évidemment de type réel simple précision, puisque notre division va certainement engendrer des valeurs décimales.

```

10*****
20*                                     CHANGE.BAS *
30*-----*
40* AUTEUR: MICHAEL MAIER Version 1.00      Date: 22.07.1990 *
50* Programme joint au 'Livre de l'Omikron Basic' *
60* (c) 1990 Micro Application *
70*****
80*
90*
100 CoursI=3.3333: REM Taux de change FF/DM
110 Montant%=322: REM Montant de la facture en FF
120 Changel=Montant%/CoursI conversion des FF en DM
130 CLS : PRINT Montant%;" FF%;" correspondant à";Changel;" D-Mark"
140 END

```

Voici votre premier programme, qui vous permet de convertir (loin du taux officiel !) en DM des montants exprimés en FF. Au début de chaque ligne figure un numéro. Lorsque le Basic est apparu, ces numéros de ligne furent obligatoires : toute ligne devait porter un numéro. Comme aucun de ces numéros ne pouvait apparaître deux fois dans le programme, on numérotait les lignes de dizaine en dizaine : ainsi, lorsqu'on voulait insérer ensuite des lignes à tel ou tel endroit, il suffisait de recourir aux chiffres non encore utilisés dans la dizaine concernée, ce qui évitait d'avoir à renuméroter complètement les lignes du programme.

L'Omikron Basic est beaucoup plus souple puisqu'il n'est pas indispensable de numérotter les lignes. Si vous souhaitez éviter de numérotter les lignes, désactivez la numérotation en appuyant sur <control>+<clr-home> (réappuyez sur ces mêmes touches pour réactiver la numérotation).

En ce qui me concerne, je préfère programmer sans numéro de ligne (la force de l'habitude !) et je renoncerais donc à la numérotation dans les exemples qui suivent. Mais comme l'Omikron Basic se charge automatiquement d'attribuer un numéro de ligne à chaque ligne de votre programme (que la numérotation soit ou non activée), vous constaterez que chaque ligne porte bien son numéro dans les listings détaillés ci-après. Par contre, vous n'avez pas besoin de les ressaisir lors d'un recopiage éventuel. Assurez-vous par contre que vous avez

bien désactivé la numérotation de ligne pour ne pas avoir de mauvaises surprises lorsque vous ferez tourner le programme écrit. Revenons à notre texte de programme.

Nous avons vu que le taux de change s'inscrit sous la forme d'une variable de type nombre réel simple précision, variable que nous avons appelée 'cours' dans le texte du programme. Ce nom est suivi du suffixe indiquant à l'ordinateur qu'il s'agit d'une variable du type nombre réel simple précision. Le taux de change lui-même est inscrit juste après le signe égal =. Si vous aviez un taux commençant par un zéro (par exemple 0,4276), vous pourriez laisser tomber ce zéro et écrire directement à partir de la virgule. Attention : contrairement aux usages français, l'ordinateur n'identifie pas la virgule mais le point pour indiquer la coupure décimale d'un nombre réel, comme vous pouvez d'ailleurs le constater dans le texte de programme donné ci-dessus.

Une ligne de programme peut contenir plusieurs instructions, qui sont alors séparées par le double-point ':'. Vous voyez que c'est le cas par exemple à la ligne 100 du texte de programme ci-dessus. Le taux de change est suivi d'un double-point introduisant l'instruction consécutive REM.

Au sujet de REM

En basic, l'instruction REM (de l'anglais 'Remark') vous permet d'introduire, dans le texte même de vos programmes, des commentaires explicatifs, ce qui est important dans les programmes très longs. En effet, l'écriture du programme une fois terminée, il arrive souvent que l'on veuille le reprendre quelques mois plus tard pour l'améliorer. Il est alors fréquent que l'on ne s'y retrouve plus très bien en l'absence de tout commentaire. Moralité : ne soyez pas avare d'explications dans vos textes de programme.

L'ordinateur ignore purement et simplement tout ce qui se trouve sur la ligne derrière REM : il est donc impossible d'entrer encore une autre instruction sur la même ligne derrière REM, car l'ordinateur n'en tiendrait aucun compte. En Omikron Basic, vous pouvez remplacer 'REM' par une apostrophe (') pour introduire un

commentaire dans le listing de programme. C'est d'ailleurs ce que nous avons fait ci-dessus dans l'en-tête de notre programme, de la ligne 10 à 90.

Comme Mr Dupont n'utilise que des montants en FF arrondis (pas de centime), le montant de la facture est attribué à une variable integer (nombre entier). J'ai renoncé ici à utiliser l'instruction LET, puisqu'elle n'est pas indispensable. En résumé, la variable 'Cours' représente le taux de change, la variable Montant% (entier court) contient le montant de la facture.

Dans la ligne 120, le montant de la facture (Montant%) exprimé en FF est divisé par le taux de change (Cours!) à l'aide de la barre oblique ou barre de fraction (appelée plus communément 'slash') qui est l'opérateur de division. Le résultat de cette opération est attribué à la variable 'Change!' (nombre réel simple précision, mais je n'ai sans doute plus besoin de vous rappeler constamment la signification des suffixes).

Cette variable représente le montant de la facture, exprimé cette fois en D.Mark, il ne reste plus qu'à l'afficher à l'aide de l'instruction PRINT. Auparavant, songeons à vider l'écran en envoyant une instruction CLS (clear screen = vider le contenu de l'écran) pour une meilleure lisibilité. Nous y sommes, et vous constatez que sur la ligne 130, l'instruction CLS est bien suivie d'un double-point puis de l'instruction PRINT. On commence par rappeler le montant de la facture en FF, le point-virgule écrit juste après Montant% fait que l'ordinateur continue l'affichage sur la même ligne. Vient ensuite un texte (FF) qui est donc encadré par des guillemets.

J'aurais pu éviter d'écrire le point-virgule suivant la mention FF, mais vous n'auriez pas pu bénéficier d'un bel exemple montrant que l'on peut aussi juxtaposer deux séquences de texte. Comme l'ordinateur enchaîne sans discontinuer l'affichage des deux séquences de texte, la deuxième chaîne de caractères ('correspondant à') est précédée par un espace vide.

La dernière instruction END indique à l'ordinateur que son travail prend fin à cet endroit.

Ouf ! Reste à expliquer maintenant le mode de fonctionnement de ce programme ; faisons tout de suite un essai. Placez-vous dans votre super-éditeur pleine page en appuyant sur la touche <Help> si vous possédez l'Omikron Basic version 3.00 ou en entrant l'instruction EDIT suivie d'un appui sur <return> si vous possédez le Basic Omikron. Recopiez les lignes 100 à 140 et lancez le programme. Les possesseurs de l'Omikron Basic version 3.00 sont ici encore favorisés : il leur suffit d'appuyer sur <control>+<R> et le résultat s'affiche immédiatement à l'écran. Dans les autres éditeurs, il faut ressortir de l'éditeur en appuyant sur <control>+<C> puis entrer l'instruction RUN (sans oublier un appui sur <return>). C'est plus long, mais le résultat devrait être rigoureusement identique :

322 FF correspondant à 96.6009 D-Mark

Pour calculer le change d'autres factures, il vous faudrait revenir à chaque fois dans le super-éditeur, modifier la valeur de Montant% et relancer le programme.

□ Les opérateurs arithmétiques

Nous avons utilisé la division à la ligne 130 du petit programme ci-dessus ; il existe aussi d'autres opérateurs arithmétiques : l'addition, la soustraction, la multiplication et l'élévation à une puissance. Chacune de ces opérations est représentée par un opérateur, comme le montre le tableau ci-dessous :

Opérateur	Fonction	Exemple
+	addition	$A\% = 3 + 4$
-	soustraction	$A\% = 5 - 2$
*	multiplication	$A\% = 2 * 3$
/	division	$A\# = 1 / 3$
\	division entière	$A\# = 6 \setminus 4$
^	élévation à la puissance	$A\% = 2 ^ 3$

☛ **Remarques:** Vous pouvez effectuer ces opérations non seulement sur des constantes (des nombres) mais aussi sur des variables (dans la mesure où cela a un sens). En écrivant par exemple

$$A\% = H\% + 3$$

vous attribuez à la variable A la valeur de la variable H à laquelle vous additionnez 3. Autre possibilité, même si elle semble quelque peu bizarre :

$$A\% = A\% + 1$$

ce qui revient à augmenter de 1 la valeur de A.

Le signe '=' joue bien le rôle d'opérateur attribuant une valeur à une variable et ne doit pas être assimilé au signe 'égal' de l'arithmétique traditionnelle.

- La différence entre la division normale et la division entière (integer-division) réside dans le fait que la seconde ne fournit que des résultats entiers alors que la première engendre souvent des nombres décimaux, à virgule. Ainsi par exemple, si vous écrivez 'A! \ 2' cela revient à attribuer la valeur 3 à la variable A, même si cette dernière est déclarée comme étant un réel simple précision (suffixe '!'). Vous pouvez d'ailleurs retrouver le reste de la division grâce à la fonction MODULO, en écrivant :

$$A\% = 7 \text{ MOD } 2$$

ce qui revient à attribuer la valeur 1 à la variable A% : le résultat de la division entière $7 \setminus 2$ étant égal à 3, il est possible de calculer le reste de la division en utilisant la formule

$$7 - (3 * 2) = 1$$

L'intérêt des divisions entières (par rapport aux divisions normales) réside avant tout dans le gain énorme de rapidité, l'ordinateur n'ayant pas à s'encombrer des valeurs figurant après la virgule.

- Même si notre programme fonctionne parfaitement, Monsieur Dupont n'en sera pas très satisfait, car il lui faut rentrer dans le texte du programme pour modifier la valeur de Montant% à chaque nouvelle facture. Il préférerait sans aucun doute que le programme lui demande d'entrer le montant et se charge ensuite de le convertir dans l'autre monnaie.
- Le résultat exprimé en D-Mark peut aller jusqu'à quatre chiffres après la virgule, ce qui est inutile, car deux suffisent amplement (le D-Mark se divise en Pfennig, comme le Franc Français se divise en Centimes) : notre programme ne devrait donc afficher que deux positions après la virgule, le montant étant arrondi au Pfennig supérieur ou inférieur.

Nous allons nous efforcer d'écrire une deuxième version de notre programme qui tienne compte de ces remarques, mais nous allons d'abord nous soucier d'un problème encore plus important :

2.4. Sauver, charger et détruire un programme

Le texte de votre programme se trouve actuellement dans la mémoire vive de votre ordinateur : si vous éteignez votre configuration (surtout ne le faites pas !) ou si vous appuyez sur le bouton 'reset' (ne le faites pas non plus !), vous ferez disparaître le programme. Dans le cas de notre petit exemple, ce ne serait pas à vrai dire une tragédie catastrophique, puisqu'il vous suffirait de ressaisir les quelques lignes de programmation dont il se compose. Imaginez un peu la catastrophe que représenterait la perte d'un texte de programme de plusieurs pages qu'il vous faudrait retaper entièrement.

□ SAVE

Cette instruction sert à sauvegarder un texte de programme en l'enregistrant sur une disquette ou un disque dur ; voici sa syntaxe :

```
SAVE ["<Nom_du_programme>"]
```


ou encore :

```
SAVE ["<Nom_du_programme>"].A
```

Dans les deux cas, votre programme sera sauvegardé dans un fichier portant le nom que vous lui aurez donné, suivi de l'extension .BAS : <Nom_du_prg.BAS>. L'extension .BAS est ajoutée automatiquement par l'interpréteur derrière le nom de fichier, elle vous indique qu'il s'agit d'un programme écrit en BASIC.

Le nom du fichier ne doit pas dépasser huit caractères ; si vous souhaitez conférer une autre extension à votre fichier, vous devez la spécifier en toute lettre (exemple : PROGRAM.OMK). Je vous recommande cependant d'éviter les extensions dont la liste suit, car elles ont une signification à peu près constante dans le monde de l'informatique :

Extension	Signification
TOS	Programme exécutable
PRG	Programme exécutable
TTP	Programme exécutable ; vous pouvez entrer les paramètres lors du chargement du programme (TTP = TOS Take Parameter).
DOC	Fichier contenant de la documentation
BAS	Programme écrit en Basic
BAK	Fichier Back-Up (ancienne version ou sauvegarde).

Si vous omettez d'entrer un nom de fichier, l'interpréteur Omikron Basic reprend le dernier nom dont vous vous êtes servi, soit avec l'instruction LOAD soit avec NEW. D'où les crochets carrés dans notre exemple ci-dessus, puisque vous pouvez fort bien ne pas spécifier de nom de fichier.

Si vous ajoutez 'A' derrière le nom du fichier, l'ordinateur va également sauvegarder le texte de programme sur le disque ou la disquette, mais sous une forme non codée. Ceci vous permet de reprendre ce texte de programme sous un traitement de texte, ce qui

est impossible avec l'instruction SAVE tout court, qui provoque un enregistrement encodé (ce qui économise de la place et améliore les temps d'exécution lorsque le programme tourne). Vous devez également sauvegarder les textes de programme sous leur forme non-codée lorsque vous pensez ensuite avoir à les mettre bout-à-bout (merge).

□ NEW

L'instruction NEW efface le texte de programme ainsi que les contenus de toutes les variables qui se trouvent actuellement dans la mémoire vive de l'ordinateur. Il est de votre propre intérêt d'être très prudent lorsque vous en faites usage ! Vous ne devez vous en servir que pour commencer à écrire un nouveau programme alors qu'un ancien texte se trouve encore dans la mémoire vive de l'ordinateur (assurez-vous d'avoir sauvegarder tout d'abord cet ancien texte !).

Vous pouvez entrer directement avec l'instruction NEW le nouveau nom de fichier destiné à contenir votre programme :

```
NEW "<Nom_du_programme>"
```

ce qui vous permet ensuite d'utiliser l'instruction SAVE sans avoir à préciser le nom du fichier, puisqu'il est déjà indiqué après NEW.

□ LOAD

Cette instruction vous permet de recharger un programme se trouvant sur une disquette ou un disque dur :

```
LOAD ["<Nom_du_programme>"]
```

Cette instruction provoque le chargement en mémoire vive du fichier dont le nom est indiqué entre guillemets. Vous pouvez aussi vous servir de LOAD sans préciser le nom du fichier : dans ce cas, l'Omikron Basic se sert automatiquement du dernier nom de fichier que vous avez entré auparavant derrière une instruction SAVE, NEW ou même LOAD. Peu importe que le programme en question ait été enregistré sous forme codée ou non : l'Omikron Basic identifie l'une ou l'autre forme et encode lui-même le texte si nécessaire.

□ RUN

Nous avons déjà utilisé cette instruction dans les pages précédentes. Elle sert à lancer un programme qui se trouve actuellement en mémoire vive. Si vous faites suivre cette instruction d'un nom de programme entre guillemets, l'Omikron Basic charge le programme (dans la mesure où il existe !) et se met immédiatement à l'exécuter.

RUN "CHANGE"

servira par exemple à charger et lancer le programme CHANGE.BAS ; vous obtenez le même effet à l'aide des deux instructions :

```
LOAD "CHANGE"  
RUN
```

2.5. Comment le dire à mon ordinateur ?

Après ce bref intermède, reprenons notre programme pour tenter d'y apporter les améliorations promises.

Première amélioration : on doit pouvoir saisir le montant de la facture sans avoir à repasser par le texte même du programme. Le technicien appelle cela un 'programme interactif' : dans le cours du programme, l'ordinateur s'arrête et attend que l'utilisateur entre une donnée avant de poursuivre l'exécution du dit programme. L'Omikron Basic vous offre une foule d'instructions pour cela : depuis la simple instruction standard jusqu'à la saisie formatée répondant même aux besoins professionnels les plus poussés.

Pour l'instant, nous allons provisoirement nous satisfaire de l'instruction de saisie standard.

□ INPUT

Cette instruction INPUT indique à l'ordinateur qu'il doit offrir à l'utilisateur une possibilité d'entrer une valeur, et ce sous une forme quelconque. Mais que faire de cette valeur entrée par l'utilisateur ? Pour que l'ordinateur puisse en faire quelque chose, il faut qu'elle soit attribuée à une variable. Cela ne servirait à rien que l'ordinateur mémorise cette valeur dans un coin quelconque de sa mémoire sans que nous, programmeurs, nous puissions la reprendre pour la retraiter dans le cours du programme. Il nous faut donc une variable de plus, à laquelle nous allons attribuer la valeur entrée par l'utilisateur. Nous écrivons :

```
INPUT A%
```

ce qui attribue la valeur saisie à la variable A%, alors que

```
INPUT A!
```

servirait à attribuer cette même valeur à une variable du type nombre réel simple précision appelée 'A!'. Si vous attendez de l'utilisateur qu'il entre une chaîne de caractères, vous devez veiller à utiliser le suffixe adéquat :

```
INPUT A$
```

faute de quoi l'ordinateur ignorera le texte (la chaîne de caractères) et attribuera la valeur 0 à votre variable (par exemple A%, A!, A# etc.). Dans d'autres versions du Basic, il vous enverrait même un message d'erreur et vous planterait là.

Lorsqu'il rencontre une instruction INPUT dans un programme, l'ordinateur envoie un point d'interrogation à l'écran et attend patiemment que l'utilisateur veuille bien entrer la valeur souhaitée. Pour l'ordinateur, cette saisie est terminée (validée) lorsque l'utilisateur appuie sur <return> : il prend alors cette nouvelle valeur et l'attribue à la variable désignée.

L'instruction INPUT répond donc tout à fait à notre objectif, reste à voir comment l'utilisateur (qui n'a pas écrit le programme) saura qu'il doit à un moment donné entrer une valeur, que ce soit son nom,

une date, l'heure ou le montant d'une facture ? Pour rendre notre programme 'convivial' il faudrait qu'il envoie à l'écran le texte d'une question indiquant à l'utilisateur ce qu'il doit entrer comme valeur (il serait certes amusant de convertir une indication de date en D-Mark, mais cela reste sans grande utilité...). Nous connaissons déjà une des instructions utilisables : PRINT. Nous pouvons écrire :

```
PRINT " Montant de la facture en FF: "; : INPUT Montant%
```

Le point-virgule est là pour que le point d'interrogation (que l'ordinateur envoie dès qu'il reçoit une instruction INPUT) vienne s'afficher juste derrière la mention 'montant de la facture en FF: ' alors que le double-point sert à séparer l'instruction PRINT de l'instruction INPUT. Cette ligne de programme doit avoir pour résultat l'affichage à l'écran de :

```
Montant de la facture en FF : ?
```

Très bien, mais nous pouvons encore simplifier : l'instruction INPUT autorise en effet la saisie d'un texte venant s'afficher à l'écran. Nous pouvons écrire :

```
INPUT " Montant de la facture en FF: ";Montant%
```

ce qui aura le même effet que ci-dessus, à l'exception du point d'interrogation qui ne s'affiche pas :

```
Montant de la facture en FF :
```

Lorsque l'instruction INPUT sert à demander à l'utilisateur d'entrer plusieurs valeurs l'une à la suite de l'autre, vous devez séparer les différentes variables par une virgule ; lors de la saisie, les valeurs seront elles aussi séparées par une virgule. Exemple :

```
10 INPUT " Veuillez entrer deux valeurs : ";A%,B%  
20 PRINT " Voici la somme de ces deux nombres : ";A%+B%  
30 END
```

Ce qui nous donne à l'écran :

```
Veuillez entrer deux valeurs : 12,14 [saisie faite par  
l'utilisateur]  
Voici la somme de ces deux nombres : 26
```


Ce type de saisie des données est particulièrement intéressant lorsqu'il s'agit d'entrer des coordonnées (x,y) :

```
10 REM Calcul de la distance séparant deux points
20 INPUT " Coordonnées du point A (x1,y1) : ";x1,y1
30 INPUT " Coordonnées du point B (x2,y2) : ";x2,y2
40 Distance# = SQR((x1-x2)^2 + (y1-y2)^2)
50 PRINT " Voici la distance entre les deux points : ";Distance#
60 END
```

Les lignes 20 et 30 servent à demander la saisie des coordonnées des points A et B, séparées par une virgule, ce qui ne semble pas poser de problème particulier.

A la ligne 40, le programme calcule la distance séparant ces deux points, grâce à la formule :

Distance = racine de ((x1 - x2) au carré + (y1 - y2) au carré)

La fonction SQR(<valeur>) sert à calculer la racine carrée de <valeur> qui est ensuite attribuée à la variable 'Distance#'. Nous avons déjà vu l'opérateur ^ qui sert à élever un nombre à une puissance quelconque ; dans notre cas, il sert à élever au carré la différence x1-x2 et y1-y2.

La ligne 50 sert à afficher le résultat sur l'écran, grâce à l'instruction PRINT. Comme les virgules servent, avec l'instruction INPUT, à séparer les variables auxquelles doivent être attribuées les différentes valeurs, il est impossible d'utiliser ce signe de ponctuation lorsque vous attribuez un string à une variable : il faut pour cela utiliser une autre instruction :

LINE INPUT

qui respecte la même syntaxe que INPUT, à la différence qu'elle vous permet d'entrer une virgule dans un string ; par contre, lorsque vous souhaitez entrer plusieurs variables l'une à la suite de l'autre derrière la même instruction, il faut prévoir une ligne par variable. Exemple :

```
10 LINE INPUT " Veuillez entrer deux valeurs : ";A%,B%
20 PRINT " Voici la somme de ces deux nombres : ";A%+B%
30 END
```


ce qui donnera à l'affichage :

Veuillez entrer deux valeurs : 12 [saisie, validée par un <return>]

14 [saisie, validée par un <return>]

Voici la somme de ces deux nombres : 26

Nous en avons terminé ici avec la présentation des instructions INPUT et LINE INPUT, mais nous reviendrons dans un prochain chapitre sur les autres instructions de saisie.

2.6. Les fonctions mathématiques

Ne vous effrayez pas de cet intitulé qui vous ramène aux mathématiques, discipline exécrée évocant irrésistiblement vos "réussites" scolaires : tout est bien plus simple que vous ne le supposez. D'ailleurs, vous avez déjà fait connaissance avec une de ces fonctions mathématiques sans même vous en apercevoir : la fonction SQR(). Mon objectif n'est pas de vous assurer quelques heures de cours particulier en mathématiques mais de vous présenter quelques fonctions que nous utiliserons pour améliorer notre programme, comme par exemple d'arrondir le montant de la facture à deux chiffres après la virgule.

Rappelez-vous : nous avons une facture d'un montant de 300 FF qui correspondait à 90.0000 D-Mark. Les deux derniers zéros ne servent à rien du tout, il faut donc les éliminer. Nous avons déjà vu comment on risquait, involontairement, de retrancher toutes les positions après la virgule dans un nombre réel. Vous vous en souvenez ? Mais oui : en attribuant un nombre décimal à une variable du type integer (nombre entier), nous avons fait disparaître la partie du nombre située après la virgule.

Dans ce cas, le système ne se contente pas de retrancher les chiffres se trouvant après la virgule, il en profite aussi pour arrondir si nécessaire le nombre entier, c'est-à-dire si le nombre après la virgule dépasse la valeur 0,5.

Ainsi par exemple, si vous attribuez la valeur 1,75 à une variable du type integer A%, cette variable prendra la valeur 2. Je vous entends déjà penser 'Eurêka, j'ai deviné'. Si nous attribuons à une variable du type integer la valeur de notre variable Montant%, non seulement le programme va faire disparaître les chiffres après la virgule, mais il va de plus arrondir ce chiffre à la valeur entière la plus proche. Malheureusement cela fera aussi disparaître les deux chiffres juste après la virgule, correspondant aux 'centimes' allemands qui s'appellent des 'pfennig'. Mince, ce n'est donc pas si simple ! Essayons autre chose.

Admettons que nous puissions, par un tour de passe-passe, sauver les deux chiffres après la virgule (les pfennig), puis faire disparaître tous les chiffres après la virgule et enfin récupérer les deux chiffres des 'pfennig' pour les réaccrocher au montant de la facture. Voilà qui réglerait notre problème, non ? ça vaut certainement la peine d'essayer. Tentons d'abord de sauver les deux premiers chiffres juste après la virgule.

Dans le système décimal, chaque chiffre composant un nombre lu à partir de la droite, possède dix fois la valeur de son voisin de droite. Je vous ai assez rasé avec ça dans le chapitre précédent ! En multipliant un nombre par 10, nous décalons d'un rang l'ensemble des chiffres composant le nombre, en le multipliant par 100, nous le décalons de deux rangs. Ainsi par exemple :

$$2345 * 10 = 23450$$

$$2345 * 100 = 234500$$

Il en va de même lorsque vous multipliez un nombre décimal :

$$2099,3702 * 10 = 20993,702$$

$$2099,3702 * 100 = 209937,02$$

Dans notre exemple en D-Mark :

$$96,6009 * 10 = 966,009$$

$$96,6009 * 100 = 9660,09$$

Nous pouvons maintenant retrancher les chiffres restant après la virgule puisque les deux chiffres des pfennig sont passés à gauche de la virgule grâce à la multiplication par 100.

Après quoi il faudra refaire passer ces deux chiffres de l'autre côté de la virgule, mais cela ne nous pose plus de problème. Si la multiplication par 10 suffit à décaler un nombre d'un rang vers la gauche de la virgule, une division par 10 devrait redécaler le tout d'une position vers la droite :

$$\begin{aligned} 9660 / 10 &= 966,0 \\ 9660 / 100 &= 96,60 \end{aligned}$$

Il ne reste qu'une source d'erreur à éviter : la division d'un entier ne donne qu'un nombre entier, dépourvu de décimales après la virgule ; à part ça, plus de problème. Nous écrivons :

120	Change-(Montant%/Cours!)*100*	décalage deux rangs vers la gauche
125	Change!-Change/100*	puis de nouveau vers la droite

Insérons ces deux lignes dans le texte de notre programme : nous obtenons bien un nombre à deux chiffres seulement après la virgule, et qui (ô miracle) a même été arrondi si nécessaire à la plus proche valeur entière de pfennig. Voilà une des solutions possibles à notre problème, et je ne m'y étendrai pas, car on peut faire encore plus simplement, en n'écrivant qu'une seule ligne !

L'Omikron Basic vous offre trois fonctions mathématiques capables de traiter les chiffres d'un nombre réel se trouvant après ou avant la virgule :

- ❶ INT
- ❷ FIX
- ❸ FRAC

□ INT

Cette fonction fait disparaître les chiffres se trouvant après la virgule ; plus exactement, elle arrondit le nombre décimal au nombre entier le plus proche :

$$A = \text{INT}(B)$$

La partie après la virgule du nombre B disparaît purement et simplement, le nombre entier ainsi obtenu est attribué à la variable A. Dans le cas d'un nombre négatif, vous obtenez le nombre entier le plus grand inférieur à B :

Nombre B	Résultat fourni par Int(B)
+ 3.1	3
+ 3.8	3
+ 1.9	1
- 2.1	-3
- 2.9	-3

□ FIX

Cette fonction a le même effet que INT, sauf que la partie après la virgule d'un nombre négatif disparaît elle aussi purement et simplement, sans que le nombre soit arrondi :

Nombre B	Int(B)	Fix(B)
+ 3.1	3	3
+ 3.8	3	3
+ 1.9	1	1
- 2.1	-3	-2
- 2.9	-3	-2

❑ FRAC

Cette fonction sert à faire disparaître la partie à gauche de la virgule dans les nombres entiers positifs ; pour les valeurs négatives, la fonction respecte la formule : $\text{FRAC}(X) = X - \text{FIX}(X)$

Nombre B	Frac(B)
3.1	.1
3.61	.61
3	0
-2.5	-.5
-6.9	-.9

Pour supprimer la partie d'un nombre après la virgule, nous utilisons soit INT soit FIX. Dans notre exemple de conversion d'une monnaie dans l'autre, il serait fort étonnant que nous rencontrions des nombres négatifs (!), nous pouvons donc nous servir indifféremment de INT ou FIX. Nous nous servons de INT.

Cette fonction INT va faire disparaître la partie du nombre après la virgule, nous devons donc le multiplier par 100 pour sauvegarder les deux chiffres immédiatement à droite de la virgule, après quoi nous faisons intervenir notre fonction puis une division par 100 :

120 Change! = INT(Montant%/Cours!*100)/100

Il nous manque encore un petit détail : la fonction INT recoupe les chiffres se trouvant après la virgule sans arrondir le nombre si nécessaire. Il ne nous reste rien d'autre à faire que de nous soucier d'arrondir ce nombre. Examinons de plus près la fonction INT, en prenant un exemple chiffré :

1	2	3
4	5	6
7	8	9

B	INT(B)
2.0	2
2.1	2
2.2	2
2.3	2
2.4	2
2.5	2
2.6	2
2.7	2
2.8	2
2.9	2
3.0	3
3.1	3
3.2	3

Vous constatez que seul le chiffre se trouvant à gauche de la virgule est retenu, mais nous allons encore nous en tirer par une petite astuce. Comme il suffit que la partie du nombre à droite de la virgule dépasse la valeur 0,5 pour que le nombre soit arrondi, nous allons purement et simplement additionner cette valeur à notre nombre. Lorsque la valeur de la partie située après la virgule est inférieure à 0,5 nous avons vu que la partie avant la virgule n'est pas modifiée et que la fonction INT retranche ce qui se trouve à gauche. Alors pourquoi tout ce travail ?

A partir d'une valeur de 0,5 après la virgule, cette addition va provoquer aussi le changement du chiffre juste à gauche de la virgule. Exemple :

$$2,6 + 0,5 = 3,1$$

En faisant alors intervenir la fonction INT, il suffit de jeter un coup d'oeil sur le tableau ci-dessus pour voir que la valeur passe alors à 3, ce qui est exactement ce que nous recherchions.

Vous trouvez ci-dessous le texte complet et réécrit de notre programme. Toutes les explications précédentes devraient vous suffire pour comprendre toutes les fonctions et opérations utilisées. Nous n'avons apporté qu'une toute petite modification : la variable Montant ! est devenu un nombre réel simple précision, ce qui nous permettra d'indiquer des montants de facture non seulement en FF entiers mais aussi avec des centimes. Qui plus est, très logiquement, nous ajoutons une instruction vidant l'écran juste avant le premier 'INPUT' (ligne 110).

```

10'*****
20'*          CHANG_V2.BAS
30'*-----*
40'* AUTEUR: MICHAEL MAIER Version 1.00      Date: 22.07.1990 *
50'* Programme joint au 'Livre de l'Omikron Basic'
60'* (c) 1990 Micro Application
70'*****
80'
90'
100 Cours! = 3.3333:REM Taux de change FF/DM
110 CLS : INPUT " Montant de la facture en FF: ";Montant!
120 Changel = INT((Montant!/Cours!*100)+.5)/100
130 PRINT : PRINT Montant!;" FF";" correspondant à";Changel;" D-Mark"
140 END

```

2.7. Manipulation des 'Strings'

Quelques explications vous ont déjà été fournies concernant les chaînes de caractères. Avant d'en venir au détail de toutes les manipulations réalisables sur ces chaînes de caractères, nous allons faire un petit voyage à l'intérieur de votre ordinateur.

☐ Des lettres qui ne sont finalement que des chiffres ou des nombres.

Qu'est-ce encore à dire ? Les lettres sont des lettres et les chiffres des chiffres, ne mélangeons pas les torchons et les serviettes ! Vous connaissez la différence entre un nombre et une lettre ou une chaîne de caractères (qui n'est qu'une combinaison de lettres). En ce domaine, l'ordinateur est bien moins avancé que vous ! avant qu'il n'affiche une lettre à l'écran, il lui faut réaliser une foule d'opérations.

Le clavier de votre Atari est géré par un petit processeur qui lui est dédié, et que nous nommerons tout simplement le processeur-clavier. Dès que vous appuyez sur une touche, le processeur-clavier avertit le système d'exploitation et lui envoie un 'paquet de données' permettant d'identifier avec précision la touche qui a été appuyée. Il serait faux d'en déduire que, lorsque vous appuyez sur la touche <A> le processeur-clavier envoie la lettre 'A' vers le système d'exploitation. Ce dernier reçoit en fait cette information sous la forme d'un nombre. Vous savez bien que l'ordinateur apprécie surtout les nombres.

C'est le système d'exploitation qui va ensuite décider de ce qu'il doit advenir de ce caractère. S'il doit l'afficher à l'écran, il va d'abord faire appel à un tableau contenant tous les signes et leur identification sous forme numérique. Ce tableau indique l'aspect que doit prendre le caractère lors de son affichage à l'écran. L'ordinateur réunit toutes ces données et peut enfin afficher le caractère correspondant à la touche sur laquelle vous aviez appuyé.

Pour l'ordinateur, chaque lettre, chaque caractère n'est donc finalement qu'un code numérique représentant le numéro du caractère en question dans le tableau mentionné ci-dessus. Comme ces codes numériques servent non seulement lors de l'affichage écran mais aussi lors de la sortie des données vers l'imprimante ou pour la communication entre deux ordinateurs, on en est vite venu à les normaliser. Il fallait fixer une fois pour toutes par exemple le code numérique de la lettre 'a'. Faute de normalisation entre les appareils, vous ne pourriez guère faire communiquer votre ordinateur avec une imprimante quelconque.

Malheureusement, tous les caractères ne sont pas normalisés : seuls les lettres 'de base' et les chiffres le sont. Les caractères propres à des alphabets particuliers (en français par exemple les caractères accentués 'é,è,à,ù et le 'ç') ne sont pas normalisés : chaque fabricant d'ordinateur décide en conséquence de remplacer des signes très peu utilisés dans le tableau par ces caractères dits 'nationaux'. En laissant votre imprimante sur le jeu de base standard, vous avez de fortes

chances de voir vos 'é' remplacés par des '!' et vos 'è' par des '}'. La majorité des lettres seront correctement restituées puisqu'elles sont normalisées.

Ce tableau des codes numériques normalisés attribués aux caractères et signes les plus courants s'appelle le tableau ASCII (American Standard Code for Information Interchange). Il contient 256 caractères, dont 128 sont normalisés, parmi lesquels on trouve la plupart des lettres (majuscules et minuscules), les chiffres, les signes de ponctuation et certains codes de commande.

Bien, me direz-vous, mais comment puis-je me servir de ces codes de caractère ? En vous servant de la fonction Basic s'écrivant :

ASC

Vous devez écrire entre parenthèses et derrière ASC le caractère dont vous souhaitez connaître le code numérique, la valeur :

ASC("a")

Naturellement, la valeur ainsi trouvée doit être attribuée à une variable pour ne pas se perdre dans un recoin de l'ordinateur. En écrivant :

A% = ASC("a")

vous attribuez à la variable A% la valeur ASCII du caractère 'a'. Vous pouvez aussi faire l'inverse, ce qui vous permet de retrouver le caractère se trouvant sous un code numérique :

PRINT CHR\$(97)

provoquera l'affichage de la lettre 'a' sur l'écran.

Ceci vous permet de transformer sans peine des lettres en nombres et inversement. Voici quelques exemples :

```
10 A$ = "A"  
20 PRINT ASC(A$)  
30 END
```

Ce programme permet de retrouver le code du caractère 'A' et de l'afficher à l'écran : c'est toujours le nombre 65.


```
10 A$ = "Au début, tout va bien"  
20 PRINT ASC(A$)  
30 END
```

Vous constatez que les trois lignes ci-dessus provoquent aussi l'affichage du nombre 65 et rien de plus, alors que nous avons transmis comme argument (la valeur entre guillemets) à la fonction ASC non pas un seul caractère mais toute une chaîne. Lorsque vous transmettez ainsi une chaîne de caractères, l'ordinateur ne calcule que la valeur ASCII du premier caractère rencontré et ignore les autres.

```
10 A% = 65  
20 PRINT CHR$(A%)  
30 END
```

Vous voyez apparaître la lettre 'A' sur votre écran. Attention : le code ASCII n'est défini que pour des valeurs comprises entre 0 et 255. Si vous entrez des valeurs supérieures ou négatives comme argument, l'ordinateur vous envoie un message d'erreur et vous plante là... après avoir planté le programme. La fonction CHR\$ permet d'afficher aussi des signes particuliers. Ainsi par exemple

```
PRINT CHR$(14);CHR$(15)
```

fait apparaître sur votre écran l'emblème commercial de la firme Atari. Le tableau contient aussi des codes de commande :

```
PRINT CHR$(7)
```

provoque l'émission d'un signal sonore. La fonction inverse devrait permettre de retrouver le code ASCII du signal sonore mais elle n'existe pas : si vous y arrivez, surtout prévenez-moi ! Un des codes de commande les plus usités se trouve sous la valeur 13 de la table ASCII : il s'agit du 'retour-chariot' (carriage return). C'est le code ASCII qui est envoyé lorsque vous appuyez sur la touche <return>.

Vous trouverez le tableau complet de ces codes ASCII dans les annexes à la fin de ce livre, et nous aurons encore l'occasion d'en reparler.

Pour en revenir à ce que nous affirmions au début, voilà bien la preuve faite que l'ordinateur ne distingue pas entre les lettres et les chiffres. Ceci peut sembler surprenant et gênant au premier abord, mais finit par s'avérer avantageux. Lorsque vous souhaitez trier des noms par ordre alphabétique, ce tableau ASCII est très efficace : la lettre 'A' prend la valeur '65', la lettre 'B' la valeur '66' etc. jusqu'à 'Z' qui prend la valeur 90. Tout naturellement, la lettre 'B' semble 'supérieure' à la lettre 'A' pour l'ordinateur, puisqu'il ne 'réfléchit' qu'en termes numériques. Pour trier les noms par ordre alphabétique, l'ordinateur fait appel à son tableau des codes ASCII qui lui permet de redonner des valeurs numériques aux caractères. Il ne lui reste plus qu'à classer ces valeurs numériques en ordre croissant, chose qu'il ne pourrait absolument pas faire avec des lettres.

Ce tableau des codes ASCII sert encore à bien d'autres choses, surtout pour le programmeur. Vous utilisez ce tableau ASCII par exemple pour dissimuler certaines données à des regards indiscrets. Reprenez les valeurs ASCII des caractères qui sont codées d'une certaine façon. Avant que vos données ne viennent s'afficher à l'écran, demandez à l'ordinateur de les décoder et faites en un string en utilisant la fonction CHR\$, dans lequel les nombres décodés sont utilisés comme argument. C'est aussi simple que cela !

□ Addition de chaînes de caractères

Nous savons que nous pouvons additionner des nombres. Mais le Basic vous permet aussi d'additionner des chaînes de caractères, en vous servant également de l'opérateur '+' (concaténation). Cela a pour effet d'accrocher deux chaînes de caractères l'une à la suite de l'autre.

⚡ **Attention :** la chaîne engendrée ne doit cependant jamais dépasser 32766 caractères.


```

10 A$ = "Le livre de"
20 B$ = "l'Omikron Basic"
30 C$ = A$+" "+B$
40 PRINT " Contenu de A$: ";A$
50 PRINT " Contenu de B$: ";B$
60 PRINT " Contenu de C$: ";C$
70 END

```

Faites tourner ce petit programme, et vous verrez apparaître sur votre écran :

```

Contenu de A$: Le livre de
Contenu de B$: l'Omikron Basic
Contenu de C$: Le livre de l'Omikron Basic

```

Dans la ligne 10, nous avons attribué à la variable A\$ la chaîne de caractères 'Le livre de' et dans la ligne 20, nous avons attribué à la variable B\$ la chaîne de caractères 'l'Omikron Basic'.

La ligne 30 nous sert à ajouter un espace vide (space) derrière la chaîne de caractères de A\$ avant que ne vienne s'afficher la chaîne de caractères attribuée à B\$. Cette addition des deux strings est attribuée à la variable C\$. Le programme vous affiche d'abord le contenu de chacune des variables. Vous voyez que je me suis bien servi du signe '+' pour juxtaposer les deux variables, en intercalant un espace vide pour que les mots 'de' et 'l'Omikron Basic' ne soient pas collés. L'espace vide possède aussi un code ASCII, la valeur 32. J'aurais donc aussi pu écrire sur la ligne 30 :

```
C$=A$+CHR$(32)+B$
```

☐ Multiplication de (chaînes de) caractères

Voilà une curiosité qui n'existe qu'en Omikron Basic : vous pouvez 'multiplier' les strings ! Ecrivez par exemple :

```
A$ = "+" * 5
```

pour attribuer la chaîne de caractères '+++++' à la variable A\$, soit exactement cinq fois le caractère '+'. Il s'agit donc d'une concaténation multiple.

En écrivant :

```
A$ = "-"*10+CHR$(32)*2+"TITRE"+CHR$(32)*2+"-"*10
```

vous attribuez à la variable A\$ une chaîne de caractères semblable à celle-ci :

```
----- TITRE -----
```

On suit donc 'à la lettre' si je puis dire, les règles de calcul élémentaires, et vous pouvez même vous servir des parenthèses :

```
PRINT "+" + "-" *5
PRINT ("+" + "-")*5
```

Dans le premier cas, vous obtiendrez : +--- alors que dans le deuxième, vous obtiendrez : +-+--+-. L'opérateur de multiplication doit se trouver en dehors des parenthèses qui contiennent le string que vous souhaitez 'multiplier' (il serait plus juste d'employer le verbe 'répéter'). En cas d'erreur, l'ordinateur vous envoie un message de 'type mismatch' (confusion dans les types de variable).

Si vous avez le souci de rendre vos programmes 'portables' (anglicisme signifiant que vous souhaitez pouvoir faire tourner vos programmes sur d'autres ordinateurs) vous devriez éviter de recourir à ce type de 'multiplication' des strings qui est propre à l'Omikron Basic, et tenter d'obtenir le même résultat à l'aide de la fonction STRING\$ qui, elle, existe dans tous les autres interpréteurs. Seule différence : l'instruction STRING\$ ne permet de multiplier qu'une seule fois un caractère alors que l'opérateur (*) permet de multiplier des strings. Vous pouvez vous en tirer en prenant exemple sur les quelques lignes ci-dessous, dans lesquelles nous juxtaposons x-fois un même string (A\$). Le résultat de la multiplication se trouve dans la variable Multi\$:

```
Multi$=""
FOR I% = 1 TO N' remplacez 'N' par un nombre quelconque
Multi$=Multi$+A$
NEXT I%
```


Ne paniquez pas, je ne vous présente ces quelques lignes que pour vous montrer qu'il faut vous y prendre autrement pour multiplier vos strings dans d'autres variantes du Basic : il faut alors additionner le string à lui-même n-fois.

□ Conversion d'un string à l'aide de Val et Str\$

Le fait de parler d'addition et de multiplication des strings ne signifie pas pour autant que l'on puisse vraiment exécuter des calculs sur les strings : il s'agit plutôt de 'juxtaposer' ou 'répéter' des chaînes de caractères. Avec des nombres, nous pouvons par contre effectuer tous les calculs imaginables. Si vous tentez d'additionner le caractère 134 avec le nombre 3, vous recevrez un message d'erreur. Si vous prenez la précaution de convertir les caractères composant le nombre 134 en valeur numérique 134 avant l'addition, celle-ci pourra se faire sans problème : c'est précisément l'utilité de l'instruction VAL, qui convertit une valeur de caractère en un nombre (expression numérique) :

VAL (chaîne de caractères)

Le système parcourt la chaîne de caractères de gauche à droite et convertit systématiquement les chiffres en valeurs numériques, jusqu'à ce que l'interpréteur rencontre un signe qu'il ne peut pas convertir, ou qu'il arrive à la fin de la chaîne de caractères. L'ordinateur considère alors le travail comme terminé : même s'il rencontre encore une valeur représentant un nombre, il ne la convertit pas. Exemple :

Chaîne de caractères	VAL(chaîne de caractères)
354	354
21.septembre	21
Basic	0
32.5_blabla	32.5
43 mille 432	43

La colonne de gauche vous montre la chaîne de caractères donnée comme argument à la fonction VAL ; lorsque la chaîne ne peut être convertie (exemple 'Basic') la fonction renvoie la valeur zéro. La saisie de :

```
VAL("FF 42.17")
```

donnera aussi un résultat 0 puisque le premier caractère du string ne peut être converti tandis que

```
VAL("42.17 FF")
```

donnera bien 42,17 comme résultat de l'opération.

Procédure inverse : essayons de transformer une expression numérique en codes caractères représentant un nombre. On utilise alors la fonction STR\$:

```
STR$( <expression numérique> )
```

La fonction fournit alors une chaîne de caractères que l'on peut attribuer à une variable de type string :

Valeur numérique	Chaîne de caractères
12	"12"
48.53	"48.53"
-14	"-14"
STR\$(43+7)	"50"
STR\$(A%)	[selon le contenu de la variable]

Lorsque la valeur numérique est positive, la première place du string est occupée par un espace blanc, qui sert à réserver la place d'un signe négatif éventuellement nécessaire.

☐ Manipulation sur les chaînes de caractères

Nous avons vu que nous pouvons juxtaposer des chaînes de caractères en utilisant l'opérateur '+' ; vous pouvez aussi dissocier un string pour en faire deux.

□ LEFT\$

L'instruction LEFT\$ extrait du string d'origine <string> les n premiers caractères comptés à partir de la gauche :

A\$=LEFT\$(<string>,n)

Admettons une variable B\$ contenant le mot 'anticonstitutionnellement'; voilà les chaînes de caractères que l'on pourrait en extraire à l'aide de l'instruction LEFT\$ pour les attribuer à la variable A\$ en écrivant A\$=LEFT\$(B\$,n)) et en faisant varier la valeur le 'n' :

n	chaîne de caractères extraite avec LEFT\$
5	antic
10	anticonsti
15	anticonstitutio
25	anticonstitutionnellement

□ RIGHT\$

Alors que LEFT\$ permet d'extraire n caractères en partant de la gauche du string, RIGHT\$ permet la même chose mais en partant cette fois de la droite de la chaîne de caractères ; la syntaxe est identique :

A\$=RIGHT\$(B\$,n))

n	chaîne de caractères extraite avec RIGHT\$
9	nellement
21	constitutionnellement
25	anticonstitutionnellement

□ MID\$

Il arrive souvent que l'on ait besoin de reprendre une partie de la chaîne de caractères ; il faut alors utiliser MID\$; deux syntaxes sont possibles :

A\$ - MID\$(<chaîne de caractères>,n)

qui fournit tous les caractères à partir de la nième position jusqu'à la fin du string :

n	chaîne de caractères extraite avec MID\$
5	constitutionnellement
13	tionnellement
22	ment

Dans la deuxième syntaxe, vous indiquez de plus combien de caractères (quantité) vous voulez extraire après la nième position :

A\$ - MID\$(<chaîne de caractères>,Quantité,n)

n	quantité	chaîne de caractères extraite
5	7	constit
5	12	constitution
11	2	tu
1	4	anti
22	4	ment

Une fois ces instructions assimilées, vous ne devriez plus avoir de problème pour 'exploser' vos chaînes de caractères en mille morceaux. Vous pouvez procéder de diverses façons pour modifier certaines parties dans une chaîne de caractères.

Exemple :

A\$ - "Le petit livre de l'Omikron Basic"

Pour remplacer le mot 'petit' par 'grand', vous pouvez par exemple vous servir de LEFT\$ et de MID\$:

```
A$ = LEFT$(A$,3)+"grand"+MID$(A$,9)
```

qui vous permet tout d'abord de reprendre l'article 'Le' avec son espace blanc (3 caractères) que vous faites suivre du 'grand' souhaité, puis vous ajoutez le restant de la chaîne de caractères à partir du 9ème caractère jusqu'à la fin. La nouvelle chaîne de caractères est attribuée de nouveau à A\$, qui représente dorénavant : "Le grand livre de l'Omikron Basic". Vous auriez pu obtenir le même résultat en vous servant de LEFT\$ et RIGHT\$:

```
A$ = LEFT$(A$,3)+"grand"+RIGHT$(A$,18)
```

Une troisième possibilité peut vous être offerte :

```
MID$(A$,4,5) = "grand"
```

Dans ce dernier exemple, une formulation un peu spéciale de MID\$ a été utilisée :

```
MID$(<string>,n,quantité) = X$
```

qui sert à remplacer la <quantité> de caractères à partir de la position n (4 dans notre exemple) dans le string par X\$.

Voici les erreurs les plus fréquentes survenant lors de la manipulation de chaînes de caractères à l'aide de LEFT\$, MID\$ et RIGHT\$:

- le programmeur tente d'extraire plus de caractères que la chaîne n'en contient. Si vous écrivez :

```
A$="Essai"  
B$=LEFT$(A$,10)
```

vous recevrez un message d'erreur puisque vous tentez d'extraire 10 caractères d'une chaîne qui en compte seulement 5. Autre erreur (toujours par rapport à A\$="Essai") :

B\$=MID\$(A\$,6) ou
 B\$=RIGHT\$(A\$,10) ou
 B\$=MID\$(A\$.2,8) etc...

- le résultat d'une de ces trois fonctions est une chaîne de caractères qui doit donc être attribuée à une variable string :

B\$=LEFT\$(A\$,2) est correct alors que
 B%-LEFT\$(A\$,2) est faux.

- LEFT\$, MID\$ et RIGHT\$ ne peuvent s'employer que sur des chaînes de caractères ; il est faux d'écrire :

A\$=MID\$(B,2,3)

puisque B est alors une variable integer (entier long, car absence de tout suffixe).

- le premier caractère d'un string occupe la position 1 ; dans l'exemple ci-dessus, A\$=MID\$(B\$,1,5) équivaut à A\$=LEFT\$(B\$,5); par contre A\$=MID\$(A\$,0,5) est faux.

□ LEN

La fonction LEN sert à vous indiquer la longueur (anglais LENgth) d'une chaîne de caractères, c'est-à-dire le nombre de caractères qu'elle contient :

Chaîne de caractères	LEN(<string>)
Gironde	7
Basic-ST	8
Atari ST	8
g	1


```
10 CLS : INPUT " veuillez entrer un mot quelconque :  
":A$  
20 Longueur%=LEN(A$)  
30 PRINT " Le mot ";A$;" contient";Longueur%;" lettres"  
40 END
```

Le petit bout de programme ci-dessus vous montre comment vous servir concrètement de la fonction LEN :

sur la ligne 10, une première instruction sert à vider le contenu de l'écran (CLS), après quoi vous demandez, par l'instruction INPUT, à l'utilisateur de bien vouloir entrer un mot quelconque, qui est attribué à la variable A\$.

La fonction LEN intervient dans la ligne 20 pour calculer le nombre de caractères contenus dans la chaîne de caractères représentée par A\$. Le résultat est attribué à la variable de type nombre entier (integer) que nous appelons Longueur%.

La ligne 30 sert à provoquer l'affichage à l'écran d'un message donnant le résultat de l'opération.

☐ SPC ou SPACE\$

Ces deux fonctions servent à confectionner un string comprenant la <quantité> n d'espaces vides :

```
A$=SPC(10)  
A$=SPACE(10)
```

ces deux formulations reviennent à attribuer un string de 10 espaces vides à la variable A\$.

Ces deux fonctions sont surtout utiles lorsqu'il s'agit de présenter des données de façon structurée à l'écran ou sur l'imprimante, c'est-à-dire lorsque des lignes de caractères variant en longueur doivent être complétées par des espaces vides pour que les données soient bien présentées et lisibles.

☛ **Attention :** Dans d'autres variantes du Basic, la fonction SPC est limitée à l'instruction PRINT ; pour confectionner un string d'espaces vides, vous devez alors absolument recourir à SPACE\$, alors que ces deux écritures sont absolument équivalentes dans l'Omikron Basic. Mais attention : si vous exportez vos programmes vers d'autres interpréteurs Basic, vous rencontrerez des problèmes si vous utilisez l'instruction SPC autrement qu'en liaison avec PRINT.

□ STRING\$

Cette fonction correspond à la 'multiplication' (en fait répétition) par l'opérateur '*', mais se limite à la répétition d'un seul caractère :

A\$=STRING\$(*<quantité>*,*<caractère>*)

cette formule revient à attribuer à la variable A\$ une *<quantité>* de fois le caractère *<caractère>*. Quelques exemples :

Quantité	Caractère	STRING\$(quantité,caractère)
10	"+"	+++++
5	"A"	AAAAA
5	CHR\$(65)	AAAAA
5	CHR\$(32)	" " [soit 5 espaces vides]

□ MIRROR\$

La fonction **A\$=MIRROR\$(*<string>*)** sert à attribuer à la variable A\$ le contenu de *<string>*, mais en commençant de la fin pour remonter vers le début (Les ordinateurs aussi parlent le verlan !)

<string>	MIRROR\$(<string>)
essai	iasse
martine	enitram
roma	amor [tiens donc, comme par hasard]
opéra	arépo

□ UPPER\$

La fonction `A$=UPPER$(<string>)` sert à attribuer le contenu de `<string>` à la variable `A$`, mais en transformant en majuscules toutes les lettres minuscules ; les caractères non alphabétiques (signes de ponctuation) et les chiffres restent inchangés :

<string>	UPPER\$(<string>)
atari	ATARI
ordinateur	ORDINATEUR
Microapplication	MICROAPPLICATION
FranCiNE	FRANCINE

Petite astuce : Les caractères spécifiques à un alphabet national ne sont pas affectés par cette fonction (en français les caractères accentués) car ils figurent parmi les caractères spéciaux dans le tableau ASCII ; il faut penser à activer le mode 'alphabet français' en entrant l'instruction `MODE"F` au début du programme. Dans ce cas, les caractères accentués français sont aussi transformés en majuscules par l'instruction `UPPER$`.

□ LOWER\$

C'est la fonction inverse de UPPER\$: elle sert à transformer en minuscules les caractères contenus dans <string>. Les caractères déjà en minuscule, les signes spéciaux et les chiffres ne sont pas affectés par cette transformation. Activez le MODE"F" pour que les caractères spécifiques à l'alphabet français soient aussi transformés.

<string>	LOWER\$(<string>)
ATARI	atari
ORDINATEUR	ordinateur
MICROAPPLICATION	microapplication
AGneS	agnes

□ MODE

A proprement parler, cette instruction n'a rien à voir avec la manipulation des chaînes de caractères, mais elle y joue un rôle déterminant, et c'est pourquoi j'en profite pour vous la présenter ici. Vous avez déjà vu son utilité dans les instructions UPPER\$ et LOWER\$.

MODE"F" active l'écriture avec les caractères français, MODE"GB" les caractères anglo-saxons, MODE"USA" les caractères Etats-Unis et MODE"D" pour les caractères allemands.

Je mentionnerai cette instruction à chaque fois qu'elle peut influencer sur d'autres instructions.

□ String-matching

Dans le jargon des informaticiens, le 'string-matching' désigne le fait de rechercher un morceau de string dans un string 'père' plus étendu : vous pourriez par exemple rechercher ainsi le mot 'string matching' dans le présent paragraphe, 'string-matching' serait alors le morceau de string recherché et l'ensemble du paragraphe constituerait le string 'père'. Nous n'allons pas commencer par des

choses si compliquées et nous nous contenterons de rechercher un morceau de string de taille 'raisonnable' dans un string plus vaste mais aussi de taille 'raisonnable'. Par chance, le Basic nous offre une fonction faite exprès pour cela, alors que les programmeurs en langage C doivent écrire eux-mêmes cette fonction, ce qui s'avère assez compliqué :

`Pos=INSTR(<string père>,<string recherché>)`

La fonction INSTR sert à rechercher si le string-cible est contenu dans le string-père. Lorsque tel est bien le cas, la variable 'Pos' reçoit une valeur indiquant la position du string recherché dans le string père ; si tel n'est pas le cas, la fonction vous renvoie la valeur 0. Cela vous semble compliqué ? Alors vite un exemple concret :

<string père>	string recherché	Position
voici un petit test	petit	10
voici un petit test	grand	0
voici un petit test	oi	2

Vous pouvez même préciser l'endroit à partir duquel vous souhaitez que la recherche commence dans le string-père en écrivant :

`Pos=INSTR(<debut>,<string père>,<string recherché>)`

L'instruction joue toujours le même rôle, mais la recherche commence à <debut>. Lorsque la fonction trouve le string recherché dans le string-père, elle vous renvoie la position correspondante ; lorsqu'elle ne le trouve pas, elle renvoie la valeur 0 :

String père	début	string recherché	Position
Cette tarte est bonne...	1	te	4
Cette tarte est bonne...	5	te	10
Cette tarte est bonne...	11	te	0

Vous savez tout sur le 'string matching' et je vous joins un petit programme qui illustre son utilisation dans un contexte plus vaste. Je ne m'y suis guère soucié de l'élégance (je ne laisserai jamais un

programme dans cet état !), mais bien plutôt d'utiliser un peu toutes les fonctions que nous avons étudiées jusqu'ici. Je vous prie de m'excuser de vous avoir ainsi assommé avec la théorie, c'est promis, je ne recommencerai plus dans les pages suivantes. Venons en à notre petit programme.

Il s'agit d'entrer le nom et le prénom de l'utilisateur sans une variable du type 'string' ; en premier viendra le prénom, suivi d'un espace vide puis du nom. Le prénom commencera par une majuscule suivie de minuscules, alors que le nom de famille sera entièrement en majuscules. Pour corser un peu les choses, disons que le tout doit entrer dans un affichage formaté. Voici ce programme :

```

0 *****
1 *                               *
2 *                               *
3 * Auteur: Michael Maier      Version 1.00   Date: 30.07.1990   *
4 * Programme joint au livre de l'Omikron Basic                 *
5 * (c) 1990 Microapplication                                   *
6 *****
7 *
8 *
9 CLS : PRINT " vider l'écran et donner une ligne vide
10 INPUT "veuillez entrer votre prénom suivi de votre nom: ":"Identite$
11 " recherche de l'espace vide ...
12 Pos%=INSTR(Identite$ , CHR$(32))
13 *
14 * nous vérifions ici si l'utilisateur a bien entré un
15 * espace vide, faute de quoi la variable Pos% recevra
16 * la valeur 0, ce qui provoquera l'interruption du programme ...
17 *
18 Identite$=UPPER$( LEFT$(Identite$,1)+MID$(Identite$,2))
19 MID$( Identite$,2,Pos%-2)=LOWER$( MID$(Identite$,2,Pos%-2))
20 Identite$= LEFT$(Identite$,Pos%)+ UPPER$( RIGHT$(Identite$,
LEN(Identite$)-Pos%))
21 *
22 Prenom$=LEFT$(Identite$,Pos%-1)
23 Nom$=MID$(Identite$,Pos%+1)
24 *
25 PRINT
26 PRINT STRING$(60,"*")
27 PRINT SPC(2);"Identité": SPACE$(10);"Prenom":SPACE$(10):
28 PRINT "Caractères Nom de famille      Prenom"
29 PRINT"*****29+*****"
30 PRINT
31 PRINT "      ":Nom$: SPACE$(14-LEN(Nom$));
32 PRINT Prenom$: SPACE$(17- LEN(Prenom$));
33 PRINT SPC( LEN("Caractères"));
34 PRINT LEN(Nom$); SPACE$(12- LEN( STR$( LEN(Nom$)-1)));
35 PRINT LEN(Prenom$)
36 *
37 END

```


- Ligne 9 :** le programme vide l'écran ; l'instruction PRINT sert à ce que la première ligne de l'écran reste vide.
- La ligne 10 :** sert à demander à l'utilisateur d'entrer son prénom puis son nom de famille, le tout étant attribué à la variable 'Identité'.
- Ligne 12 :** le programme recherche, dans la variable Identité\$, la position de l'espace vide (CHR\$(32)) et attribue cette position à la variable Pos%. Comme nous nous servons par la suite de cette variable, nous devrions nous soucier du confort de l'utilisateur et lui demander si Identité\$ contient bien un espace vide, faute de quoi la variable Pos% reçoit la valeur 0, ce qui empêche ensuite le programme de se dérouler normalement. Nous ne le faisons pas pour l'instant, car je ne vous ai pas encore appris à programmer des choses si complexes.
- Ligne 18 :** l'instruction LEFT\$(Identité\$,1) sert à retrancher la première lettre du string ; LEFT\$ étant ici un argument de UPPER\$, la première lettre de Identité\$ est transformée en majuscule si nécessaire. Le restant des caractères composant l'identité de l'utilisateur est recomposé par une simple addition de string, et le résultat est réaffecté à la variable Identité\$. On y retrouve alors la chaîne de caractères d'origine, la seule différence étant que nous sommes maintenant certains qu'elle commence bien par une majuscule.

L'étape suivante consiste à transformer toutes les lettres restantes dans le prénom en minuscules : vous savez que l'Omikron Basic vous offre pour ce faire l'instruction LOWER\$. La variable Pos% contient, rappelons-le, la position de l'espace vide qui sépare en principe le prénom du nom de famille. Comme nous avons déjà transformé en majuscule la première lettre du string Identité (première lettre du prénom), il nous reste à transformer en minuscules tous les caractères à partir de la deuxième position du string identité et jusque Pos%-1.

Nous écrivons Pos%-1 car Pos% désigne la position de l'espace vide que nous ne tenons pas à transformer. Rappelons que la première lettre du prénom doit être une majuscule, si bien que les paramètres dans MID\$ doivent être 2 et Pos%-2. Nous avons ainsi extrait une partie du string commençant par la deuxième lettre et se terminant juste avant l'espace vide. En même temps, nous lui appliquons deux possibilités liées à l'instruction MID\$: à droite de l'opérateur d'attribution (=) nous découpons le morceau du string auquel nous appliquons la fonction LOWER\$. Nous remplaçons ensuite cette partie de la chaîne de caractères dans le string complet <Identite>, par l'instruction MID\$ se trouvant à gauche de l'opérateur égal.

Nous en avons terminé ainsi avec le traitement du prénom, et nous passons au nom de famille, qui commence juste après l'espace vide et se termine à la fin du string <Identite>. Nous pourrions réutiliser MID\$, mais il est aussi possible d'utiliser RIGHT\$, même si cela s'avère un peu plus compliqué. Après MID\$, les paramètres doivent préciser combien de caractères nous décomptons à la fin du string. C'est ce que nous calculons en soustrayant la position de l'espace vide (Pos%) de la longueur totale LEN du string : il nous reste alors le nombre de caractères pris par le nom de famille. Nous utilisons ce paramètre avec RIGHT\$ pour isoler le nom de famille. La fonction UPPER\$ nous sert alors à transformer toutes les lettres en majuscules. Mais l'utilisation de RIGHT\$ ne doit pas nous faire oublier qu'après avoir isolé la fin du string, il nous faut la replacer derrière. Cela n'est plus un problème pour nous : LEFT\$ nous donne la chaîne de caractères jusqu'à l'espace vide inclus, il suffit alors de juxtaposer la fin du string. Terminé !

Lignes 22 et 23 : nous redécoupons notre string Identite que nous avons eu tant de peine à recoller, en deux parties : le prénom et le nom de famille.

Ligne 25 : cette ligne sert de nouveau à créer une ligne vide à l'écran, avant que la ligne 26 ne fasse apparaître un string se composant de 60 fois le signe étoile (*).

Ligne 27 : sert à faire apparaître à l'écran les en-tête destinés à l'utilisateur. Remarquez bien le point-virgule (;) à la fin de la ligne 27 : il est là pour que l'instruction PRINT de la ligne 28 ne provoque pas un saut de ligne mais que les données viennent s'inscrire à la suite, sur la même ligne.

Ligne 29 : vous y constatez l'usage de la multiplication et de l'addition du string. Nous y demandons aussi l'affichage de 60 étoiles, mais pas à l'aide de l'instruction STRING\$. Certes, il eut été plus logique de multiplier 30 fois le string "***" plutôt que de le multiplier 29 fois pour y ajouter un "***" : mais ce programme sert avant tout d'exemple, et je tenais à vous y montrer toutes les virtualités de l'Omikron Basic en matière de manipulation de string.

Ligne 30 : il s'agit ici aussi de créer une ligne vide à l'écran, après laquelle les données viennent s'afficher. Nous avons parlé ci-dessus d'un affichage 'formaté' car notre programme doit en principe veiller à ce que le prénom et le nom viennent bien s'inscrire en colonnes, sous les intitulés de colonnes. Le nom de famille doit s'inscrire à partir de la colonne 3 et le prénom à partir de la colonne 17.

Pour amener le nom à la colonne 3, je l'ai tout simplement fait précéder de la chaîne " " (deux espaces vides). Selon les cas, le nom de famille peut être plus ou moins long ; alors que le prénom doit

commencer à la colonne 17. Il faut donc sauter les espaces séparant la fin du nom du début du prénom. La fonction `SPACE$` est ici toute indiquée : nous demandons l'affichage de 14 espaces vides (colonne 17 moins 3 espaces) desquels nous soustrayons la longueur du nom de famille. La position d'affichage du prénom est ainsi indépendante de la longueur du nom de famille et commence bien à la colonne 17. Nous réitérons ce petit jeu pour l'affichage du prénom.

Ligne 33 : nous demandons l'affichage de 7 espaces vides, ce qui amène le curseur juste au-dessous de l'intitulé 'Nom de famille'. Le programme nous indique le nombre de caractères contenus dans le nom de famille, lequel peut dans certains cas occuper deux rangs. Il nous faut de nouveau sauter les colonnes non utilisées, mais c'est encore plus compliqué. Nous savons que `LEN` nous donne une variable du type 'nombre entier' (integer) que nous venons d'afficher. Mais ce nombre n'est pas une variable de type string, et `LEN` ne peut nous servir à calculer la longueur d'un nombre, puisque cette fonction exige d'avoir un string pour argument. Alors, comment faire ?

Nous transformons cette longueur en un string à l'aide de la fonction `STR$`, et nous calculons la longueur de ce string cette fois sans problème grâce à `LEN`. La fonction `STR$` a une particularité : nous savons qu'en cas de nombre positif, le string commence par un espace vide (réservé pour le + éventuel), il est donc d'un espace plus grand que le string réel : nous retranchons une unité pour que le résultat soit correct. Cette ligne n'est pas aussi complexe qu'elle en a l'air. Nous commençons par calculer la longueur du nom de famille, nous transformons ce nombre en un string dont nous calculons la longueur, dont nous retirons 1 pour l'espace réservé au signe '+'. Si tout cela vous semble bien compliqué, sachez qu'il existe une autre méthode beaucoup plus simple, en utilisant la fonction logarithmique. Mais je voudrais réserver ce chapitre pour plus tard.

□ Les systèmes numériques

Il existe une foule de manuels traitant des langages de programmation qui commencent par expliquer les systèmes numériques en long et en large avant d'en venir à présenter la première instruction au lecteur harassé par tant de théorie. On y montre comment écrire et transformer des nombres d'un système à l'autre : je souhaiterais quant à moi vous épargner ces acrobaties intellectuelles, car l'Omikron Basic prend tout cela en charge.

Je pense que vous connaissez le système décimal ainsi que le système binaire, c'est pourquoi je me bornerai à vous présenter les deux autres systèmes très utilisés en informatique : le système hexadécimal et le système octal.

Le système hexadécimal

L'être humain a pour habitude de compter avec le système décimal (peut-être parce qu'il a dix doigts), alors que l'ordinateur ne peut guère s'en servir : il préfère nettement le système binaire qui, lui, est quasi incompréhensible pour les humains (bien que l'on ai au moins deux doigts). D'où l'idée d'introduire le système hexadécimal, incompréhensible tant par l'humain que par l'ordinateur.

Trêve de plaisanterie : le système hexadécimal recourt à 16 nombres différents. De 0 à 9, on utilise les 9 premiers chiffres du système décimal, puis on fait appel aux six premières lettres de l'alphabet (de A à F). En système hexadécimal, le nombre A représente le nombre 10 du système décimal et le nombre F en hexadécimal représente le nombre 15 en décimal. Pour bien noter que ces lettres doivent être considérées comme des nombres hexadécimaux et non comme des lettres ordinaires, on les fait précéder d'un signe dollar '\$': exemple \$A.

Le système octal

C'est le système usuel des programmeurs en langage C ; il recourt aux chiffres 0 à 7, qui sont précédés du signe &.

Le système binaire

Vous savez que ce système est basé sur les deux chiffres 0 et 1. Pourquoi le répéter ? pour vous rappeler que ces deux chiffres sont alors précédés du signe %. Il existe une autre possibilité de préciser le système numérique utilisé :

Préfixe	base correspondante
&d	décimal
&h	hexadécimal
&b	binaire
&o	octal

Venons-en au fait, c'est-à-dire à la transformation d'un nombre pour le faire passer d'un système à l'autre :

□ HEX\$

Cette fonction sert à transformer une valeur numérique quelconque donnée comme argument en un nombre hexadécimal (plus exactement d'ailleurs en une chaîne de caractères) :

A\$=HEX\$(174)

va attribuer à la variable A\$ la chaîne de caractères " AE"; vous remarquez que cette chaîne de caractères commence toujours par un espace vide.

□ BIN\$

Cette fonction sert à transformer une expression numérique en un nombre binaire (chaîne de caractères) :

A\$=BIN\$(160)

va attribuer la chaîne de caractères " 10100000" à la variable A\$.

□ OCT\$

Devinez un peu l'utilité de la fonction OCT\$? Tout juste, elle sert à transformer une valeur numérique en un nombre octal (chaîne de caractères) :

```
A$=OCT$(63)
```

va attribuer la chaîne de caractères " 77" à la variable A\$.

Pour retransformer ces chaînes de caractères en une expression numérique décimale, vous utilisez la fonction VAL, à laquelle vous donnez comme argument la chaîne de caractères voulue (attention : précédée du préfixe indiquant le système numérique concerné) :

Instruction	Contenu de A
A= VAL("&77")	63
A= VAL("\$AE")	174
A= VAL("%10100000")	160

Lorsque la variable A\$ contient un string binaire, il suffit d'ajouter le préfixe par une simple addition de string :

```
Nombre= VAL("%"+A$)
```

ce qui vaut naturellement aussi pour le système hexadécimal et octal, du moment que vous n'oubliez pas d'utiliser le préfixe adéquat : en hexadécimal, le dollar (Nombre= VAL("\$"+A\$)) et la perluette & en octal (Nombre= VAL("&"+B\$)).

Voilà tout ce que je tenais à vous dire concernant les systèmes numériques pour vous montrer que la transformation d'un nombre d'un système à l'autre ne pose pas de problème particulier en Omikron Basic. Dans la suite de ce manuel, j'utiliserai le plus possible le système décimal, pour que les exemples de programmes restent lisibles et facilement compréhensibles. Cette courte incursion sur les autres systèmes numériques était cependant indispensable pour que le manuel soit complet.

2.8. Les tableaux de variables

Voilà une autre espèce de variable : les tableaux de variables ou 'arrays'. Plusieurs variables du même type (integer, float ou string) sont réunies sous un même nom. Le nom de la variable est suivi d'un indice qui indique l'élément mis en cause dans le tableau à un moment donné.

`A$(3) = "voici un exemple de tableau"`

Pour décrire très clairement ce qu'est un tableau de variables, on peut dire qu'il s'agit d'une sorte de commode à plusieurs tiroirs (placés les uns au-dessus des autres) ; ces tiroirs sont numérotés en continu. Ces numéros permettent ensuite d'ouvrir un tiroir à la fois. Dans l'exemple ci-dessus nous avons rangé la chaîne de caractères "voici un exemple de tableau" dans le tiroir numéro 3. Lorsque vous souhaitez afficher le contenu de ce tiroir, vous entrez l'instruction :

`PRINT A$(3)`

Pour pouvoir travailler avec un tableau de variables dans un programme, il faut auparavant le dimensionner, ce qui consiste à indiquer à l'ordinateur combien de place il doit réserver pour ce tableau (nombre de tiroirs) : en écrivant par exemple

`DIM A$(5)`

vous installez un tableau A\$ à six dimensions, les indices allant de 0 à 5, donc de A\$(0) à A\$(5).

Si vous oubliez de dimensionner votre tableau en Omikron Basic, l'interpréteur s'en charge automatiquement, l'indice allant au maximum jusqu'à 10. Attention : ceci n'est valable qu'avec l'Omikron Basic ! En clair, cela signifie que toutes les opérations d'attribution et d'affichage seront réalisées sans problème tant que vous ne dépasserez pas l'indice 10. Si vous tentez d'accéder à un indice supérieur à 10 alors que vous n'avez pas dimensionné votre tableau, l'ordinateur vous renverra un message d'erreur.

Je vous conseille cependant de ne pas prendre la mauvaise habitude de ne pas dimensionner vos tableaux de variables. En effet, les autres versions du Basic ne vous le pardonneront pas, et un tableau non-dimensionné fera planter le programme, avec un bel avis d'erreur.

Vous pouvez aussi recourir à des tableaux à plusieurs dimensions (plusieurs colonnes de tiroirs) : par exemple, en écrivant

`DIM A$(5,5)`

vous créez un tableau à 36 champs. Reprenons notre exemple de la commodé pour éclaircir cela. Elle a maintenant 36 tiroirs : vous accédez au deuxième tiroir de la troisième colonne en écrivant `A$(3,2)`. Si vous écrivez `A$(4,1)`, vous accédez logiquement au premier tiroir de la quatrième colonne.

Vous êtes maintenant assez savant pour créer des tableaux de variables à trois ou quatre dimensions, et même plus. Attention, dans ces tableaux, la première dimension ne doit pas dépasser la valeur 65535, et toutes les dimensions suivantes réunies ne doivent pas non plus dépasser cette valeur. La dimension globale du tableau se calcule en multipliant les dimensions les unes par les autres. Vous n'avez pas à vous faire de cheveux blancs : si vous n'utilisez que des tableaux raisonnables, il est quasiment impossible de dépasser la taille maximale autorisée.

Type de variable	Place à réserver, en octet par élément
boolean	1 octet par élément
octet	1 octet par élément
entier court	2 octets par élément
entier long	4 octets par élément
reel simple précision	6 octets par élément
reel double précision	10 octets par élément
chaîne de caractères	6 (+ longueur + 10, une fois installé)

2.9. Les aides à la programmation

Je voudrais maintenant vous présenter certaines instructions prévues par l'Omikron Basic pour alléger le travail du programmeur (les produits allégés étant à la mode).

□ Utilisation des touches de fonction

L'Omikron Basic vous permet d'attribuer une chaîne de caractères quelconque à chacune des dix touches de fonction, sans que la chaîne ne dépasse cependant 32 caractères.

Quelle utilité ? En écrivant un programme, vous constaterez que certaines chaînes de caractères reviennent très fréquemment. Naturellement, vous pouvez fort bien décider de les entrer toujours 'à la main', par le clavier, mais vous finirez par trouver cela bien fastidieux. Vous pouvez attribuer chacune de ces chaînes à l'une des touches de fonction : il vous suffira ensuite d'appuyer sur la touche en question pour voir apparaître la chaîne de caractères souhaitée à l'écran.

Pour attribuer ainsi une chaîne de caractères à une touche de fonction, l'Omikron Basic vous offre l'instruction

KEY

En écrivant par exemple

```
KEY 2-"le livre de l'Omikron Basic"
```

Attribue à la touche de fonction <F2> le chaîne de caractères "le livre de l'Omikron Basic".

Dès que vous avez ainsi attribué une chaîne de caractères à une touche de fonction, il vous suffit d'appuyer sur la dite touche pour voir apparaître la chaîne de caractères à l'écran.

Par contre, si vous souhaitez attribuer à la touche F1 l'instruction RUN pour qu'elle soit effectivement immédiatement exécutée dès que vous appuyez sur F1, vous devez rajouter le code ASCII de la

touche <RETURN>. En effet, comme la touche <return> sert à valider l'attribution d'une chaîne de caractères à une touche de fonction, il est impossible de l'attribuer à une de ces touches en appuyant sur cette même touche <return>. Le code ASCII de RETURN est normalisé, il s'agit du code 13 ; nous écrivons donc :

```
KEY 1="run"+chr$(13)
```

pour attribuer à la touche de fonction F1 l'instruction RUN. La touche de fonction F1 servira donc à lancer le programme se trouvant dans la mémoire vive. Il en va de même pour les guillemets, que vous devez aussi entrer sous la forme de leur code ASCII (34) : si vous écrivez

```
KEY 3="FILES"+CHR$(34)+"A:\OMIKRON\*.BAS"+CHR$(34)+CHR$(13)
```

il vous suffira ensuite d'appuyer sur la touche de fonction F3 pour lister à l'écran tous les textes de programme en BASIC se trouvant dans le dossier OMIKRON sur la disquette A.

Au-delà de 10 chaînes de caractères, vous pouvez recourir à la combinaison de la touche de fonction avec la touche <shift>.

```
KEY 13= "CLS"+CHR$(13)
```

Cette suite d'instructions vous permet ensuite d'appuyer sur <shift>+<F3> (équivalent de la touche de fonction numéro 13) pour vider le contenu de l'écran, alors que la touche <F3> seule vous permettra toujours (exemple ci-dessus) de lister les textes de programme.

Lorsque vous souhaitez visualiser toutes les chaînes de caractères ou instructions attribuées à vos touches de fonction, vous entrez l'instruction :

```
KEY LIST
```

☐ Commandes destinées à lister les textes de programme

L'instruction LIST sert à visualiser entièrement ou partiellement un texte de programme sur l'écran ; vous pouvez aussi demander une sortie sur imprimante.

- LIST** sert à visualiser l'ensemble du texte de programme
- LIST -200** sert à visualiser toutes les lignes jusqu'à la ligne 200
- LIST 220** sert à visualiser la ligne 220
- LIST 100-200** sert à visualiser les lignes 100 à 200
- LIST 200-** sert à visualiser toutes les lignes à partir de 200.

Le tiret de liaison peut être remplacé par une virgule : vous pouvez écrire :

LIST 100,200 pour visualiser les lignes 100 à 200

Les nombres peuvent aussi être remplacés par des variables :

LIST (ERL),(ERL+50) sert à visualiser les 50 lignes suivant celle dont le numéro se trouve dans la variable ERL.

✱ **Remarque :** Lorsque le texte contient une erreur, la variable ERL (ERror Line) contient le numéro de la ligne contenant l'erreur détectée. Vous pouvez relier cela avec une routine d'erreur, nous y reviendrons plus tard.

Vous pouvez encore remplacer les numéros de ligne par des étiquettes (labels) :

LIST -marque sert à visualiser le texte du programme jusqu'à la ligne contenant l'étiquette 'marque'.

Vous pouvez faire défiler le texte vers le haut ou le bas (scrolling) à l'aide des <touches fléchées>.

Pour ce faire, commencez par charger un texte de programme en mémoire vive et provoquez par exemple l'affichage des 100 premières lignes en envoyant l'instruction

LIST -100

(vous pouvez naturellement entrer n'importe quelle autre valeur). Une fois à la ligne 100, vous pouvez 'descendre' jusqu'à la dernière ligne de l'écran à l'aide de la <touche curseur> fléchée vers le bas. Une fois parvenu sur cette dernière ligne, vous constatez qu'en maintenant la touche enfoncée, vous faites défiler le texte qui se décale vers le haut. Vous pouvez faire de même avec la <touche curseur> fléchée vers le haut, pour 'remonter' dans le texte.

Vous pouvez aussi sortir tout le texte ou une partie seulement sur votre imprimante à l'aide de l'instruction

LLIST

qui suit exactement la même syntaxe que LIST. Vous pouvez encore détourner la sortie de votre listing pour le sauvegarder dans un fichier (sur un disque ou une disquette) en entrant :

```
OPEN "O".1."<nom_du_fichier>": CMD 1: LLIST: CLOSE 1
```

ce qui revient à enregistrer le listing sur la disquette, dans un fichier portant le <nom_du_fichier> indiqué. Pour envoyer votre texte, via un modem (port de sortie RS232), à un de vos amis vous écrivez :

```
OPEN "V".1: CMD 1: LLIST: CLOSE 1
```

☐ Comment afficher le contenu des variables

L'instruction DUMP sert à afficher à l'écran toutes les variables présentes dans le texte d'un programme, avec leur contenu. Dans le cas des tableaux (arrays), vous ne voyez s'afficher que le dimensionnement et non le contenu :

```
10 A$="Essai"
20 DIM Toto$(10,2,5),Tata(25)
30 Foul= 2.18
40 END
```

```
DUMP
```

```
A$="Essai"  
DIM TOTO$(10,2,5)  
DIM TATA(25)  
FOU1= 2.18
```

OK

Si vous préférez sortir un listing des variables sur l'imprimante, vous écrivez :

```
LDUMP
```

et vous pouvez également détourner cette instruction pour obtenir un fichier sur disquette :

```
OPEN "0".1."<nom_du_fichier>": CMD 1: LDUMP: CLOSE 1
```

Vous pouvez détruire le contenu de toutes les variables grâce à :

```
CLEAR
```

☐ Trace (contrôle du déroulement du programme)

Cette instruction vous permet de savoir dans quel ordre l'ordinateur exécute votre programme. Vous entrez l'instruction :

```
TRON (abréviation condensée de 'trace on')
```

Après avoir lancé le programme, vous constatez que l'ordinateur vous affiche le numéro de ligne de l'instruction qu'il est en train d'exécuter ainsi que son intitulé entre crochets. Ceci vous permet de surveiller le déroulement de l'exécution de votre programme. Voici un exemple :

```
10 A$="que sais-je ?"  
20 B%=30  
30 X%=X%+1  
40 PRINT A$  
50 END
```

```
TRON: RUN
```

```
[10 LET] [20 LET] [30 LET] [40 PRINT] que sais-je ?  
[50 END]
```

Pour ressortir de cette instruction, vous utilisez

TROFF (abréviation condensée de 'trace off')

Lorsque vous utilisez TRON et TROFF non pas en mode direct mais à l'intérieur d'un programme, l'Omikron Basic gère ces deux instructions 'par niveau'. Si vous avez par exemple activé quatre fois TRACE, il vous faut quatre fois TROFF pour désactiver le mode 'trace'. Dans le mode direct par contre, il suffit d'entrer une seule fois TROFF pour désactiver le mode trace, quel que soit le nombre de TRON entrés auparavant.

Vous pouvez surveiller encore plus étroitement le déroulement de votre programme en entrant

ON TRON GOSUB <Cible>

Après chaque instruction, l'ordinateur retourne au numéro de ligne indiqué dans <cible> pour y exécuter les indications qui y figurent, jusqu'à ce que vous lui demandiez de reprendre le traitement en appuyant sur <return>. Il exécute alors l'instruction suivante et retourne à <cible>. Ceci vous permet de surveiller de près l'évolution de certaines variables. Plus vous exercez un contrôle serré, plus vous ralentissez l'exécution du programme : je vous recommande donc d'être bref. La variable ERL contient le numéro de la ligne à partir de laquelle l'interpréteur est passé dans la routine de contrôle, alors que la variable ERR\$ contient la dernière instruction exécutée.

Je me dois de vous apporter une petite consolation si vous n'avez pas bien compris tout ce qui précède : en effet, pour comprendre l'utilité de ON TRON GOSUB, il faut posséder certaines connaissances que vous n'avez pas encore. J'espère cependant que vous avez un peu compris la façon dont fonctionne ON TRON GOSUB.

❑ RENUMBER (renuméroter les lignes)

Si vous écrivez vos programmes en numérotant les lignes, vous constaterez qu'il arrive souvent que l'on doive intercaler une ou deux lignes supplémentaires ou en supprimer quelques-unes. Il peut vous arriver d'avoir à intercaler une ligne entre deux lignes portant déjà des numéros consécutifs.

L'Omikron Basic dispose d'une fonction permettant alors de renuméroter complètement les lignes du texte de programme : RENUM. Vous pouvez indiquer derrière RENUM le numéro de la ligne à partir de laquelle vous faites commencer votre nouveau programme, ou le numéro de la ligne à partir de laquelle vous souhaitez lancer la renumérotation. Dernier paramètre : vous pouvez préciser le pas de la renumérotation, sachant qu'on numérote généralement de dix en dix.

Exemples :

RENUM 100,50,10 signifie que vous souhaitez renuméroter le texte du programme à partir de la ligne 50 ; la ligne 50 prend le numéro 100, après quoi la numérotation se fait de 10 en 10 (100, 110, 120, etc).

RENUM 10 signifie que vous souhaitez renuméroter le programme par pas de dix ; la première ligne du programme porte le numéro 10.

RENUM faute de paramètre, le programme renumérote l'ensemble des lignes du texte par pas de dix, la première ligne portant le numéro 100 ; RENUM tout court est donc équivalent de RENUM 100.

2.10. La programmation structurée

Disons-le tout de suite : le vocable 'programmation structurée' ne désigne rien d'autre que l'écriture propre d'un programme logiquement articulé. L'utilisateur ne saura généralement pas si le programme qu'il utilise est ou n'est pas structuré proprement, et cela lui est bien égal : il veut surtout un programme qui tourne correctement par rapport au travail à exécuter.

Le programmeur pense un peu différemment, car il sait qu'il devra éventuellement reprendre le texte de son programme pour y apporter des améliorations ou le compléter. Les anciens interpréteurs du Basic favorisaient la 'programmation spaghetti' ce qui rendait très difficile la reprise d'un texte de programme : on passait son temps à sauter d'un bout à l'autre d'un programme déjà bien complexe de par le nombre d'instructions contenues sur chaque ligne. Au bout d'un moment, on perdait forcément le fil directeur, et le programme devenait un véritable sac de noeuds impossible à démêler.

Pour éviter ces désagréments, les langages de programmation modernes offrent la possibilité de structurer le programme, ce que fait aussi l'Omikron Basic. L'idée directrice en matière de structuration des programmes est aussi simple que géniale : chaque étape du travail est décomposée en opérations de plus en plus simples, qui sont affinées jusqu'à devenir une suite d'instructions de base. Voici un exemple.

Tous nos lecteurs connaissent sans doute le jeu appelé 'mastermind', qui consiste à deviner une combinaison de couleurs ou de chiffres. L'un des joueurs (A) crée une combinaison que l'autre (B) doit retrouver aussi rapidement que possible, en formulant une suite d'hypothèses. Lorsque B place deux couleurs (ou deux chiffres) correctement, A le lui indique ; A doit d'ailleurs aussi indiquer le fait qu'une des couleurs a bien été employée dans la combinaison, même si B ne l'a pas mise à la bonne place. B peut alors examiner la réponse de A à sa première hypothèse, et en tenir compte pour formuler sa deuxième hypothèse. Le processus se répète jusqu'à ce que la planche de jeu soit remplie (la combinaison de départ n'a pu être reconstituée) ou jusqu'à ce que B ait correctement reconstitué la combinaison créée par A.

Voici comment nous pourrions décomposer ce processus en opérations simples :

- ❶ il faut tout d'abord créer la combinaison à retrouver (rôle de A)
- ❷ formuler une première hypothèse (rôle de B)

- ③ comparer cette première hypothèse avec la combinaison réelle
- ④ si l'hypothèse formulée ne correspond pas, reprendre à l'étape 2.

Nous pouvons décomposer encore plus finement l'étape numéro 3 :

- 3.1 comparer la première position des deux combinaisons
- 3.2 si les deux couleurs correspondent
- 3.3 indiquer que la couleur et la position sont correctes
- 3.4 sinon, vérifier si la couleur proposée existe au moins dans la combinaison à une autre place
- 3.5 si tel est le cas
- 3.6 montrer que la couleur est valable, mais pas à cette position
- 3.7 après avoir comparé ainsi toutes les positions,
- 3.8 reprendre à l'étape 4
- 3.9 sinon, comparer la position suivante et reprendre à 3.2

voilà comment nous pourrions décomposer la problématique du 'mastermind' en une suite d'opérations élémentaires. Evidemment, nous pourrions encore affiner ce découpage, mais ce n'est pas l'objet principal de notre ouvrage.

☐ IF...THEN...ELSE (si... alors... sinon...)

En décomposant notre problème 'mastermind', nous avons rencontré une des structures élémentaires d'articulation d'un programme : la condition IF..THEN..ELSE.

En effet :

si les deux premiers rangs de chacune des combinaisons correspondent
alors montrer que la couleur et la position sont justes
sinon, vérifier si au moins la couleur existe

Cette condition s'exprime en Basic à l'aide des instructions

```
IF (si, conditionnel)
  (la condition est remplie)
  THEN (alors, conséquence)
    (indiquer la conséquence que cela aura)
  ELSE (sinon, faute de quoi : la condition n'est pas
  remplie)
    (indiquer la conséquence que cela aura)
```

Détail de cette construction : l'instruction IF est suivie d'une condition (pour reprendre notre exemple : si les deux couleurs correspondent) ; lorsque cette condition est remplie (les deux couleurs correspondent), cela a une conséquence introduite par THEN (afficher que les deux couleurs correspondent). Lorsque la condition n'est pas remplie (les deux couleurs sont différentes) cela a une autre conséquence qui, elle, est introduite par ELSE (contrôler si...).

Comment l'ordinateur s'y prend-t-il pour vérifier si une condition est bien remplie ? tout simplement en comparant deux termes entre eux. Dans notre exemple du 'mastermind', nous pourrions formuler cela de la façon suivante :

```
IF <couleur_1 comme couleur_2>
THEN ...
```

ou plus exactement, pour employer la bonne syntaxe :

```
IF couleur_1 = couleur_2
THEN
```

Nous connaissons déjà le rôle du signe '=' en tant qu'opérateur d'attribution ; nous découvrons maintenant sa deuxième utilité, en tant qu'opérateur de comparaison. Admettons que couleur_1 et couleur_2 soient deux variables représentant les couleurs choisies

dans les deux combinaisons. Si le contenu de ces deux variables est identique, les deux couleurs sont identiques : le résultat est alors la valeur de vérité 'vrai', si ce n'est pas le cas, la valeur est 'faux'.

Nous utilisons cette valeur de vérité avec l'opérateur IF : si la condition est vérifiée (vrai), le programme passe à l'instruction introduite par THEN ; si la condition n'est pas vérifiée (faux), le programme passe à l'instruction introduite par ELSE. Notre opérateur de comparaison = nous fournit donc une 'valeur de vérité' : les deux possibilités ouvertes par la condition (vrai ou faux) peuvent prendre un aspect numérique pour l'ordinateur.

Le nombre -1 représente la valeur 'vrai', le nombre 0 représente la valeur 'faux'. Et nous voici revenus aux flags qui servent précisément à mémoriser ce genre de valeurs. Vous souvenez-vous du chapitre 2 ? nous y avons justement vu que nous pouvions attribuer aux flags les deux valeurs 0 (faux) et -1 (vrai). Il est donc tout à fait pertinent d'attribuer à une variable du type 'flag' le résultat d'une comparaison pour ensuite pouvoir soumettre la valeur obtenue à une condition IF. Le résultat reste le même :

```
100 DIM Flag%F(7)' évitons de gaspiller de la place
    mémoire
110 Nombre_1%- 10
120 Nombre_2%- 20
130 Nombre_3%- 10
140 Flag%F(0)- Nombre_1%- Nombre_2%
150 Flag%F(1)- Nombre_3%- Nombre_1%
160 '
170 IF Flag%F(0) THEN PRINT "Nombre 1 semblable à
    nombre 2"
180 IF Flag%F(1) THEN PRINT "Nombre 1 semblable à
    nombre 3"
190 '
200 END
```

Les lignes 140 et 150 servent à comparer le contenu de deux variables. Le résultat de l'opération est attribué à une variable déclarée comme étant du type 'flag' (Flag%F()). Les deux conditions IF des lignes 170 et 180 utilisent le contenu de Flag%F() comme valeur de vérité pour

savoir s'il faut exécuter l'instruction introduite ensuite par THEN. On aurait pu tout aussi bien placer la comparaison après la condition IF :

```
170 IF Nombre_1%- Nombre_2% THEN PRINT "Nombre 1
semblable à nombre 2"
180 IF Nombre_3%- Nombre_1% THEN PRINT "Nombre 1
semblable à nombre 3"
190 ...
```

Comme flags ne peut contenir que 0 ou -1, l'ordinateur n'a besoin que d'un bit pour mémoriser ces deux valeurs. Nous savons que chaque bit ne se promène pas dans la 'nature' mais que l'ensemble des bits est divisé en jolis petits paquets de huit, formant un octet. Il est aisé d'en conclure qu'un octet peut contenir huit flags. C'est la raison pour laquelle l'Omikron Basic range les flags dans des tableaux (arrays) et que 8 flags correspondent exactement à un octet.

Revenons à notre condition IF. Nous savons que la (ou les) instruction(s) introduite(s) par THEN ne sera (seront) exécutée(s) que si la condition est remplie, le résultat de l'opération étant alors la valeur 'vrai'. L'Omikron Basic permet de présenter les instructions à exécuter d'une façon encore plus lisible, en répartissant les différentes instructions sur plusieurs lignes.

L'expérience prouve que cette écriture sur plusieurs lignes augmente considérablement la clarté du texte, surtout dans les programmes longs. Comme nous répartissons les instructions introduites par THEN sur plusieurs lignes, il nous faut bien indiquer à l'ordinateur à quel moment cette instruction prend fin : c'est le rôle de l'instruction ENDIF.

```
IF <condition 1>
  THEN <instruction 1>
      <instruction 2>
      <instruction 3>
  ELSE
    IF <condition 2>
      THEN <instruction 4>
          <instruction 5>
    ENDIF
  ENDIF
```

```
ENDIF
```


Dans l'exemple ci-dessus, nous avons inséré deux conditions IF l'une dans l'autre, ce qui ressort bien de la présentation du texte : la façon d'écrire permet de constater immédiatement qu'une des conditions dépend de l'autre.

En effet, lorsque la <condition 1> est remplie, l'ordinateur exécute les instructions 1 à 3, faute de quoi il passe aux instructions 4 et 5 introduites par ELSE. Dans ELSE, nous trouvons une deuxième condition IF, qui ne vaut que lorsque la <condition 1> n'est pas remplie. Lorsque la <condition_1> n'est pas remplie mais que la <condition_2> est remplie, l'ordinateur exécute les instructions 4 et 5.

La deuxième condition IF n'est pas une alternative semblable à celle introduite par ELSE, cette dernière instruction n'étant d'ailleurs pas toujours indispensable. Remarquez encore la position des 'ENDIF' : vous devez en effet refermer la deuxième condition avant la première, faute de quoi l'imbrication des deux conditions ne fonctionnerait pas.

Pour plus de lisibilité, il est recommandé d'écrire ENDIF à la même hauteur que le IF auquel il se rapporte. Vous devez maintenant avoir compris tout l'intérêt de l'écriture structurée des programmes : en effet, la séquence IF...THEN...ELSE...ENDIF peut former l'ossature d'un programme. Il vous suffit de comparer ce style d'écriture avec un programme absolument non-structuré, en prenant par exemple le même texte que ci-dessus (Condition est abrégé en 'C' et instruction en 'I') :

```
IF <C1> THEN <I1>:<I2>:<I3> ELSE IF <C2> THEN <I4>:<I5>
```

Le résultat sera absolument le même, mais la lecture est bien plus difficile dès que l'on dépasse le premier THEN. Imaginez que ces conditions et instructions soient en fait d'autres instructions : il est probable que vous ne parviendriez plus à réaliser la signification exacte de la ligne écrite. D'où la réputation de l'ancien BASIC comme langage de programmation-spaghetti.

Au fur et à mesure que vous imbriquez des conditions IF l'une dans l'autre, vous décalez de plus en plus vers la droite. Vous pouvez économiser un peu de place en écrivant le THEN sur la même ligne que la condition IF :

```

IF <condition 1> THEN
  <instruction 1>
  <instruction 2>
  <instruction 3>
ELSE
  IF <condition 2> THEN
    <instruction 4>
    <instruction 5>
  ENDIF
ENDIF

```

Le texte du programme n'a rien perdu en lisibilité, mais le décalage vers la droite est moins important qu'avec le premier style d'écriture. C'est avantageux surtout lorsque l'on écrit des structures assez complexes, car on utilise ainsi mieux la largeur de l'écran : la structure ne se décale pas vers l'extrême-droite, ce qui permet de voir les lignes complètes.

Jusqu'ici, nous avons utilisé le signe '=' comme opérateur de comparaison, mais ceci ne s'avère pas toujours suffisant. On est souvent amené à comparer deux nombres entre eux pour savoir si l'un est supérieur ou inférieur à l'autre, ou encore s'ils sont bien différents l'un de l'autre : c'est pourquoi il existe d'autres opérateurs de comparaison. Ils ont tous un point commun : le résultat de la comparaison est 0 pour 'faux' et -1 pour 'vrai' :

Opérateur de comparaison	Signification
=	semblable
<	inférieur
>	supérieur
<=	inférieur ou égal
>=	supérieur ou égal
<>	différent

Notez que les opérateurs \leq , \geq et \neq peuvent s'écrire en sens inverse :

Opérateur de comparaison	Signification
$=<$	inférieur ou égal
$=>$	supérieur ou égal
$><$	différent

Pour plus de simplicité, il est recommandé d'écrire ces signes dans l'ordre où on les prononce lorsqu'on lit à voix haute (on dit plutôt 'supérieur ou égal' que l'inverse!).

Quelques exemples :

```
IF A% > B% THEN ... si A% est supérieur à B% ...  
IF A% <> B% THEN ... si A% est différent de B% ...  
IF A% <= B% THEN ... si A% est inférieur ou égal à B% ...
```

Naturellement, vous pouvez comparer deux chaînes de caractères entre elles, car l'ordinateur recourt alors au tableau des codes ASCII, dans lequel les lettres sont représentées par des chiffres. (j'espère que vous vous souvenez de ce que je vous ai expliqué ci-dessus).

En conséquence, le string 'Essai' est 'inférieur' au string 'Vanillepudding'! Normalement, de telles opérations de comparaison (en dehors de $=$) ne sont mises en oeuvre que pour des opérations de tri. L'Omikron Basic brille encore à cette occasion par le fait qu'il vous offre pour ce faire une routine intégrée dans l'interpréteur. Le programmeur n'a plus à se torturer avec des opérations de tri, c'est l'opérateur qui s'en charge en un tour de main.

□ GOTO

L'instruction GOTO semble être finalement le pire ennemi de la programmation structurée : elle interrompt le déroulement du programme (du début vers la fin) pour demander à l'ordinateur de sauter jusqu'à un endroit quelconque (un embranchement) du programme et d'y reprendre le traitement : on dit alors que le

programme 'se branche' à tel ou tel 'branchement'. Malgré tout - ou peut-être justement à cause de cela - il m'a semblé judicieux de la faire figurer dans ce chapitre.

La syntaxe de GOTO est très simple :

```
GOTO <cible>
```

<cible> désignant tout simplement un numéro de ligne :

```
10 PRINT "Essai"  
20 GOTO 10  
30 END
```

Si vous recopiez ce programme et que vous le lancez (abstenez-vous en !), vous constatez que l'ordinateur se met à écrire sans arrêt le mot "Essai" à l'écran, et qu'il n'y a plus moyen de l'arrêter ! pourquoi ?

La ligne 10 contient une instruction PRINT demandant à l'ordinateur d'afficher le mot "Essai" à l'écran. L'ordinateur est un "animal" très très obéissant qui s'empresse d'exécuter l'ordre donné. Dans la deuxième ligne, il tombe sur l'instruction GOTO, lui enjoignant d'exécuter pour ainsi dire un pas en arrière et de revenir à la ligne 10. Là, on lui demande d'écrire le mot "Essai" à l'écran, il s'empresse de s'exécuter et de passer à la ligne suivante, où on lui demande de revenir à la ligne précédente... Ce petit jeu risque de se répéter jusqu'à ce que vous y mettiez une fin en appuyant sur <control>+<C> (ce qui vous vaudra la reconnaissance éperdue de votre ordinateur) ou en coupant tout simplement le courant (avec l'interrupteur uniquement !).

L'Omikron Basic ne mériterait pas sa réputation s'il ne tenait pas ici aussi quelques atouts en réserve ; il vous offre la possibilité de calculer le numéro de ligne, en écrivant

```
GOTO 100+10*ligne
```

vous verrez l'ordinateur se brancher sur une ligne donnée, en fonction du contenu de la variable 'ligne'. Si vous n'indiquez qu'une seule variable, vous devez l'écrire entre parenthèses (vous allez tout de suite savoir pourquoi) :

GOTO (ligne)

l'instruction ci-dessus fait que l'ordinateur continue d'exécuter le programme à partir de la ligne '(ligne)' à laquelle vous lui demandez de se rendre. Les numéros de ligne sont certes très très utiles, mais que faire lorsque l'on écrit ses programmes sans numéroter les lignes ? Il suffit alors de déterminer des 'étiquettes' (labels) désignant les 'embranchements' ou 'branchements'. Pour que l'ordinateur ne confonde pas ce 'label' avec une variable (ce qui le ferait se planter car les variables n'admettent pas de tels sauts), il faut faire précéder l'étiquette par un signe '-' qui est le signe caractérisant le statut de l'étiquette. Il va de soi qu'il faut éviter de placer deux labels identiques dans le même programme, ce qui ferait perdre les pédales à votre ordinateur (même si vous, vous les avez déjà perdues). Vous écrivez

-label

pour définir un embranchement (ou branchement) dans le programme et

GOTO label

pour que l'exécution du programme se poursuive à partir de l'embranchement indiqué. Il est en principe conseillé de recourir plutôt à des étiquettes (labels) qu'à des numéros de lignes pour rejoindre un embranchement : cela contribue à rendre le programme plus lisible et facilite une modification ultérieure éventuelle (la rénumérotation se fait sans douleur) . En utilisant comme étiquettes des mots significatifs et faciles à mémoriser, vous ne pouvez quasiment plus vous tromper..

Vous pouvez fort bien placer une étiquette en plein milieu d'une ligne de programme, mais il faut alors la séparer des autres instructions par un double-point :

```
100 PRINT "Bonjour" : -Boucle: INPUT "entrez votre mot  
de passe : ";A$  
110 IF A$ <> "Omikron Basic"  
120 THEN GOTO Boucle  
130 ENDIF  
140 END
```


Après un amical 'bonjour', l'ordinateur demande à l'utilisateur d'entrer son mot de passe, qui est attribué à la variable A\$. Dans la ligne 110, le mot de passe entré par l'utilisateur est comparé à celui qui est mémorisé dans le programme : s'il s'avère différent, le programme revient à l'embranchement appelé 'boucle', et l'utilisateur doit ressaisir son mot de passe. L'ordinateur ne poursuivra l'exécution du programme qu'après avoir reçu le mot de passe correct. L'étiquette entrée derrière GOTO peut aussi prendre la forme d'une variable :

```
A$="Boucle": GOTO A$
```

cette variante n'est certes pas très rapide (c'est même la moins rapide des possibilités de branchement) mais elle peut toujours servir.

□ ON...GOTO...

Cette suite d'instructions est étroitement apparentée à l'instruction GOTO ; voici sa syntaxe :

```
ON <valeur> GOTO <cible 1>,<cible 2>,<cible 3>, ...
```

<valeur> représente une variable : lorsqu'elle prend la valeur 1, le programme saute à la cible 1, lorsqu'elle prend la valeur 2, il saute à la cible 2 etc... Si <valeur> est inférieure à zéro, l'ordinateur n'effectue aucun saut, et il en va de même si <valeur> est supérieure au nombre des cibles entrées après GOTO. Vous devinez déjà qu'il sera possible de préciser des <cibles> non seulement sous forme de numéro de ligne mais aussi d'étiquettes, de calculs et même de variable-string représentant un nom d'étiquette, le tout pouvant d'ailleurs se mélanger :

```
ON nimportequoi GOTO 100.Fin.Dieu_sait_ou$
```

L'instruction ON... GOTO... sert également à accroître la lisibilité d'un texte de programme, surtout lorsqu'elle est utilisée avec des étiquettes. C'est bien la raison pour laquelle les programmeurs s'en servent beaucoup dans les programmes comprenant des menus.

□ Variations sur le thème de la boucle

L'un des plus gros avantages des ordinateurs réside justement dans le fait qu'ils peuvent se charger sans peine de répétitions fastidieuses. En terme d'informatique, on appelle ces répétitions des 'boucles'. La plus simple des boucles est créée à l'aide de l'instruction GOTO :

```

100 PRINT "Bonjour"
110 -Boucle: INPUT "entrez votre mot de passe : ";Pa$
120 IF Pa$ <> "Omikron Basic" THEN
130     GOTO Boucle
140 ELSE
150     CLS
160     PRINT "mot de passe correct"
170 ENDIF
180 '
190 ' le programme à proprement parler commence ici
200 .....
```

Si l'utilisateur entre un mot de passe erroné, l'ordinateur revient de la ligne 130 à la ligne 110. Ces lignes représentent une boucle, même si en informatique les boucles sont habituellement écrites avec une structure propre.

□ FOR..NEXT

Vous ne pouviez pas y couper, je suis bien obligé de commencer par la boucle la plus fréquente dans tous les interpréteurs Basic (elle existait déjà à l'époque de la préhistoire des ordinateurs !). Je ne vais pas vous assommer de déclarations théoriques mais vous présenter immédiatement cette boucle dans le contexte d'un programme :

```

0 *****
1 *                               *
2 *                               *
3 * Auteur: Michael Maier   Version 1.00   Date: 29.07.1990   *
4 * Programme joint au 'Livre de l'Omikron Basic'             *
5 * (c) MICROAPPLICATION                                     *
6 *****
7 .
8 . demandons le nombre de valeurs admises
9 .
10 CLS
11 INPUT " Nombre de valeurs à saisir: ";Nbre%
12 '
13 ' valeur correcte?
```



```

14 *
15 IF Nbre% <= 0 THEN
16 PRINT "quelle est cette farce?"
17 END
18 ELSE
19     DIM X1 1(Nbre%-1) ' valeurs à partir de 0 jusque ...
20 ENDIF
21 *
22 * saisie des différentes valeurs
23 *
24 FOR TX=1 TO Nbre%
25     INPUT " Valeur saisie, numéro "; STR$(TX); ": "; X1 1(TX-1)
26 NEXT TX
27 *
28 * addition des différentes valeurs
29 *
30 Somme 1=0
31 FOR TX=0 TO Nbre%-1
32     Somme 1=Somme 1+X1 1(TX)
33 NEXT TX
34 *
35 * puis division par le nombre de valeurs saisies
36 *
37 Somme1=Somme1/Nbre%
38 *
39 PRINT " voici la moyenne arithmétique de ces valeurs: "; Somme1
40 END

```

Ce programme sert à calculer la valeur moyenne (moyenne arithmétique) d'une suite de valeurs, selon la formule :

$$\text{Moyenne} = \frac{1}{n} * \sum_{i=1}^n x(i)$$

Les valeurs saisies sont additionnées, et le résultat de cette addition est ensuite divisé par le nombre de valeurs saisies. Nous sommes évidemment tentés d'entrer les valeurs x dans un tableau 'array': pour lui conférer des dimensions adaptées, le programme demande (ligne 11) le nombre de valeurs qui vont être entrées, puis attribue cette valeur à la variable Nbre% (ce qui correspond au 'n' de la formule).

Avant d'écrire (ligne 19) l'instruction DIM, nous contrôlons, à l'aide d'une condition IF, si le nombre de valeurs demandées n'est pas inférieur ou égal à zéro (il est interdit de demander un dimensionnement négatif !) : si tel est le cas, l'ordinateur demande s'il s'agit d'une farce. Sinon (ELSE), il dimensionne un tableau array pouvant recevoir des nombres réels simples (Xi!). Nous savons que

l'indice des variables dans un tableau part de zéro et va jusqu'à ... C'est pourquoi il nous faut diminuer les dimensions du tableau d'une unité, pour ne pas gaspiller de la place en mémoire.

Venons-en à la seule véritable nouveauté contenue dans ce texte de programme : pour ne pas avoir à écrire une ligne INPUT pour chacune des valeurs saisies par l'utilisateur, nous utilisons une boucle à cet endroit : nous n'avons ainsi besoin que d'une seule instruction INPUT (ligne 25).

Voici en détail le mode de fonctionnement de la boucle FOR NEXT : nous commençons par créer un compteur de répétitions (la variable T%) que nous plaçons sur la valeur de départ T%=1. La boucle est ensuite exécutée autant de fois qu'il le faut, jusqu'à ce que l'interpréteur rencontre l'instruction NEXT. Il revient alors à sa position de départ (FOR T% = 1 TO Nbre%), augmente le compteur de répétitions de une unité et contrôle si la valeur terminale de la boucle (figurant derrière TO, ici : Nbre%) est déjà atteinte. Une fois cette dernière valeur atteinte, le programme sort de la boucle et reprend le cours de son déroulement après NEXT ; sinon, il réexécute encore la boucle :

```
FOR <compteur de répétitions> = <valeur de départ> TO  
  <valeur d'arrivée>
```

```
  ...  
NEXT <compteur de répétitions>
```

Comme le programme prévoit de n'entrer dans le tableau que le nombre exact de Nbre%-valeurs, notre boucle FOR..NEXT est logiquement parcourue un nombre de fois allant de 1 à Nbre%. Nous utilisons le compteur de répétitions comme variable indiciaire par Xi ! (ligne 25), servant au rangement des différentes valeurs dans le tableau (array). Attention cependant : le compteur de répétitions part de 1 et va jusque Nbre%, alors que les indices des valeurs dans le tableau partent eux de 0 et vont jusque Nbre%-1 : c'est pourquoi nous diminuons de 1 la valeur indiciaire (T%-1). Nous aurions pu aussi faire partir le compteur de répétitions de zéro et le laisser aller jusqu'à Nbre%-1, en écrivant :


```
FOR T%-0 TO Nbre%-1  
<Input>  
NEXT T%
```

Pourquoi je ne l'ai pas fait ? à cause d'un petit détail : je tiens à ce que l'utilisateur sache à tout moment combien de valeurs il a déjà entré, et le programme se sert pour cela du compteur de répétition T%, qui compte donc à partir de 1.

Si le compteur partait de zéro, il me faudrait à chaque fois augmenter de 1 le numéro des valeurs saisies, ce qui ferait un double calcul pour l'ordinateur, travail que nous évitons d'une façon assez élégante.

J'attire encore votre attention sur le STR\$(T%) de la ligne 25. T% contient la valeur de la répétition de la boucle momentanément en cours. J'utilise STR\$() pour transformer T% en string, pour éviter que la valeur saisie ne soit attribuée par erreur à la variable T% (qui doit être affichée en même temps puisqu'elle sert de compteur) au lieu d'être rangée dans le tableau sous Xi!(). L'Omikron Basic fait d'ailleurs cela automatiquement, si bien que STR\$(T%) apparaît dès que l'on saisit T%.

Une fois que l'utilisateur a terminé d'entrer les valeurs souhaitées, le programme doit les additionner : il le fait encore à l'aide d'une boucle, dont le compteur de répétitions part cette fois de zéro pour aller jusqu'à Nbre%-1. Pour plus de simplicité, ce compteur de répétitions s'appelle aussi T%. En ressortant de la boucle, nous obtenons une valeur correspondant au résultat de cette addition, valeur attribuée à la variable Somme !. Il ne reste plus qu'à diviser Somme ! par le nombre de valeurs saisies pour obtenir la moyenne arithmétique. Terminé !

Lancez ce programme, et vous voyez apparaître à l'écran :

Nombre de valeurs à saisir : 3 [selon l'indication de l'utilisateur]

Valeur saisie, numéro 1 : 2

Valeur saisie, numéro 2 : 6

Valeur saisie, numéro 3 : 4

voici la moyenne arithmétique de ces valeurs : 4

OK

Pour que la boucle soit exécutée par pas différent de 1, utilisez l'instruction STEP juste après la formule de condition :

```
FOR T% = 0 TO 20 STEP 2
```

```
NEXT T%
```

le compteur de répétition augmentera de 2 unités à chaque parcours de la boucle ; vous pouvez aussi demander un pas inférieur à 1 : en écrivant par exemple

```
FOR T% = 0 TO 20 STEP .5
```

```
....
```

```
NEXT T%
```

le compteur de répétition n'augmentera que 0,5 à chaque parcours de la boucle. Vous pouvez au contraire demander à ce que le compteur diminue à chaque parcours de la boucle en entrant un pas négatif à la suite de STEP :

```
FOR T% = 20 TO 0 STEP -1
```

```
....
```

```
NEXT T%
```

Pour plus de clarté, vous avez constaté que j'ai réparti la structure FOR...NEXT sur plusieurs lignes, en décalant le contenu de la boucle vers la droite. En théorie, vous obtiendriez exactement le même résultat en écrivant tout sur une seule ligne :

```
FOR T% = 0 TO Nbre% : PRINT T% : NEXT T%
```

☛ **Attention :** si la condition de départ est supérieure à celle d'arrivée, et que vous n'indiquez pas un pas de comptage négatif, le programme n'exécutera pas du tout les instructions contenues dans la boucle.

On parle alors de 'boucle vide' qui n'est exécutée que lorsque la condition de départ est vérifiée. D'autres interpréteurs Basic (parmi lesquels le fameux GfA-Basic) possèdent une boucle non-vide FOR..NEXT, dont la caractéristique est d'être en tout cas exécutée au

moins une fois. Vous devez en tenir compte si vous reprenez en Omikron Basic des textes de programme écrits sous d'autres interpréteurs, faute de quoi vous vous exposez à des désagréments certains !

□ REPEAT...UNTIL

Voici une autre structure de boucle : REPEAT-UNTIL, ce qui signifie à peu près : 'répéter... jusqu'à ce que...'. Le contenu de cette boucle est en tout cas parcouru au moins une fois avant que ne soit testée la validité de la condition introduite par UNTIL.

```
REPEAT
  <instruction 1>
  <instruction 2>
  ...
  <instruction n>
UNTIL <condition remplie>
```

Il est possible de remplacer une boucle FOR...NEXT par une boucle REPEAT UNTIL :

```
FOR T%-1 TO 30
  <instruction 1>
  <instruction 2>
  ....
  <instruction n>
NEXT T%
```

correspond à :

```
T% = 0
REPEAT
  T% = T% + 1 'compteur de répétition de pas STEP 1
  <instruction 1>
  <instruction 2>
  ....
  <instruction n>
UNTIL T% = 30
```

Cette équivalence des deux formulations n'est cependant exacte que si la condition de la boucle FOR...NEXT est vérifiée ; si tel n'est pas le cas, les deux boucles sont différentes. La boucle FOR...NEXT n'est

pas du tout exécutée lorsque sa condition de départ n'est pas vérifiée, alors que la boucle REPEAT...UNTIL est au moins parcourue une fois. Si nécessaire, on peut la faire précéder d'une condition IF...THEN, pour que REPEAT...UNTIL ne soit exécutée que si telle ou telle condition est vérifiée ; si tel n'est pas le cas, on ressort de la condition par EXIT.

Voici un petit exemple qui rendra cela plus clair : la saisie d'un mot de passe. Le programme ci-dessous répète l'instruction INPUT jusqu'à ce que l'utilisateur ait entré le mot de passe convenable :

```
100 REPEAT
110   CLS
120   INPUT " veuillez entrer votre mot de passe : ";Pa$
130 UNTIL Pa$="Omikron Basic"
140 '
150 ' le programme à proprement parler commencera ici
160 '
170 END
```

□ Boucle sans fin

La plupart des variantes du Basic sont dotées d'une structure particulière permettant de créer une boucle sans fin : il est impossible de ressortir d'une telle boucle par une procédure normale. Cela n'existe pas dans l'Omikron Basic, mais vous pouvez facilement programmer vous même une boucle sans fin, en utilisant REPEAT...UNTIL. Il suffit d'introduire par UNTIL une condition qui ne sera jamais vérifiée, comme par exemple la valeur 'faux' représentée en Basic par 0 :

```
REPEAT
  <instruction 1>
  <instruction 2>
  ....
  instruction n>
UNTIL 0
```

ou mieux encore :

```
REPEAT
  <instruction 1>
  <instruction 2>
```



```

....
<instruction n>
UNTIL toujours_et_eternellement

```

Tout cela est bien beau, mais comment ressortir d'une telle boucle ? en principe, c'est impossible tant que le programme ne rencontre pas une instruction EXIT.

❑ EXIT

Cette instruction vous permet de ressortir à tout moment d'une boucle, même si la condition d'interruption n'est pas encore remplie, qu'il s'agisse d'une boucle sans fin ou d'un autre type de boucle :

```

100 REPEAT
110   CLS
120   INPUT " veuillez entrer votre mot de passe : ";Pa$
130   IF Pa$="OMIKRON BASIC"
140     THEN EXIT
150   ENDIF
160 UNTIL toujours_et_eternellement
170 ...
180 ...

```

Grâce à l'instruction EXIT, le programme ressort de la boucle dès que l'utilisateur entre le mot de passe convenu et passe à l'exécution du programme, à partir de la ligne 170. Vous pouvez encore ajouter après EXIT une marque (numéro de ligne ou étiquette) indiquant à l'ordinateur l'embranchement où il doit se rendre pour reprendre l'exécution du programme ; on utilise pour cela une sorte de mélange de EXIT et de GOTO, puisque l'on peut écrire :

```
EXIT TO <cible>
```

<cible> doit avoir ici exactement les mêmes caractéristiques qu'avec GOTO.

☛ **Attention :** le GfA-Basic est doté d'une boucle sans fin particulière :

```
DO
  <instruction 1>
  ....
  <instruction n>
LOOP
```

que vous remplacerez éventuellement en Omikron Basic par :

```
REPEAT
  <instruction 1>
  ...
  <instruction n>
UNTIL 0
```

Il en va de même pour l'instruction EXIT IF <condition> du GfA-Basic que vous réécrirez en Omikron Basic de la manière suivante :

```
IF <condition> THEN EXIT
```

lorsqu'il s'agira de ressortir d'une boucle avant qu'elle ne soit entièrement parcourue.

□ WHILE...WEND

Rassurez-vous, WHILE...WEND (tant que... recommencer) est le dernier type de boucle : il s'agit d'une boucle qui n'est pas toujours exécutée, puisque l'ordinateur vérifie, avant son exécution, si la condition figurant derrière WHILE est vérifiée ou non. Lorsque cette condition n'est pas vérifiée, l'ordinateur ne se donne pas la peine de lire la boucle. Si la condition est vérifiée, l'ordinateur exécute les instructions programmées jusqu'à ce qu'il rencontre WEND, ce qui le ramène au début de la boucle, où il recontrôle si la condition de départ est toujours vérifiée.

WHILE...WEND vous permet donc aussi d'écrire une boucle sans fin : il suffit pour cela que la condition introduite par WHILE soit toujours vérifiée, ce qui revient à lui attribuer la valeur -1 ('vrai') :


```

WHILE -1
<instruction 1>
<instruction 2>
...
WEND

100 *****
110 **                CRIBLE.BAS                **
120 **-----**
130 ** Auteur : Michael Maier   Version 1.00   Date : 02.07.1990 **
140 ** Programme joint au 'livre de l'Omikron Basic                **
150 ** (c) MICROAPPLICATION                **
160 *****
170 '
180 ' crible d'Eratosthene
190 '
200 True%=-1 : False%=0
210 REPEAT
220     CLS
230     INPUT " veuillez indiquer la limite supérieure/2 (> 50) ";N%
240 UNTIL N%>50
250 DIM Liste%(N%)
260 FOR TX=0 TO N% attribuer -1 aux valeurs contenues dans liste
270     Liste%(TX)=True%
280 NEXT TX
290 FOR TX=0 TO N%
300     IF Liste%(TX) THEN
310         'affichage des nombres premiers
320         Prem%=2*TX+3
330         PRINT Prem%,
340         IX=TX+Prem%
350         'faire disparaître tous les multiples
360         WHILE IX<N%
370             Liste%(IX)=False%
380             IX=IX+Prem%
390         WEND
400     ENDIF
410 NEXT TX
420 END

```

Pour conclure en beauté notre chapitre sur les boucles, voici le célèbre crible d'Eratosthène, qui permet de retrouver tous les nombres premiers inférieurs à un nombre entier naturel N.

Le procédé est aussi simple qu'efficace : on commence par installer une liste dans laquelle toutes les valeurs sont placées sur -1 ('vrai') (lorsque $Liste\%(X) = -1$ alors X est un nombre premier). En commençant par le commencement (qui est toujours 'vrai' donc -1 !), le programme contrôle si chaque élément de la liste est ou n'est pas un nombre entier. Il suffit de faire disparaître de la liste les multiples de chacun de ces nombres premiers en leur attribuant dans la liste la

valeur 'faux' soit 0. Cette méthode permet de contrôler la liste valeur par valeur et d'en faire disparaître à chaque fois les multiples de la valeur examinée.

L'algorithme ci-dessus permet de lister tous les nombres premiers jusqu'à la limite supérieure indiquée $N * 2 + 3$. Seul petit problème : la valeur 2 n'y apparaît pas, c'est pourquoi il faudrait rajouter une ligne

```
285 PRINT 2
```

Ce petit programme contient une nouveauté pour vous : vous constatez qu'à la fin de la ligne 330, la ligne introduite par un PRINT se termine par une virgule. Cela a pour effet de faire sauter le curseur, dès la fin de la ligne, jusqu'à la marque de tabulation la plus proche (une largeur d'écran est découpée en sauts de tabulation faisant chacun 8 colonnes), où il reprend l'affichage lorsque l'ordinateur reçoit de nouveau un PRINT.

2.11. Question de routine(s)

Oui, vous avez bien lu, tout n'est finalement qu'une question de routine(s) ! mais sachez que le mot 'routine' prend ici un sens précis.

Dans le cours d'un programme, il arrive en effet souvent qu'il faille répéter certaines procédures plusieurs fois, et ce, à des moments très différents. Citons par exemple le tri d'un tableau de variables. Naturellement, on peut fort bien décider de réécrire à chaque fois que nécessaire toute la procédure dans le texte du programme, mais cela nous fera consommer inutilement une grande place en mémoire vive. Vous me répondrez peut-être que le ST en a plus que suffisamment. Qu'en est-il alors du temps de travail passé à recopier des lignes et des lignes rigoureusement semblables ? Admettez qu'il serait beaucoup plus pratique d'écrire une fois pour toute la procédure, pour pouvoir l'appeler ensuite à chaque fois qu'elle sera nécessaire. C'est justement ce que les informaticiens appellent une 'routine' (ou sous-programme). Comment fonctionne ces routines ?

Une routine n'est donc rien d'autre qu'une partie de programme qui sert à exécuter telle ou telle tâche répétitive (par exemple le tri d'un tableau de variables, sa sauvegarde sur disquette ou disque dur, la saisie de la date et de l'heure etc.), que l'on peut ensuite appeler depuis n'importe quelle ligne de programme, et après l'exécution de laquelle l'ordinateur reprend le cours normal du programme principal. En Basic, on écrit :

```
GOSUB <cible>
```

le retour au programme principal se faisant toujours par

```
RETURN
```

```
<instruction 1>
```

```
<instruction 2>
```

```
<instruction 3>
```

```
...
```

```
...
```

```
...
```

```
GOSUB Sous-programme
```

```
<instruction n>
```

```
...
```

```
...
```

```
END
```

```
-Sous-programme
```

```
<instruction S1>
```

```
<instruction S2>
```

```
...
```

```
...
```

```
<instruction Sn>
```

```
RETURN
```

Le programme principal commence à l' <instruction 1> et se termine par END. A l'intérieur de ce programme principal, le programmeur demande à l'ordinateur de se brancher à la routine baptisée 'sous-programme' ; l'ordinateur exécute alors les instructions <instruction S1> à <instruction Sn> qui y sont indiquées ; lorsqu'il rencontre RETURN, il retourne au programme principal, pour y reprendre le cours normal, ce qui revient ici à exécuter l'instruction <instruction n> écrite juste après 'GOSUB Sous-programme'

Il y a donc une grande différence entre GOTO et GOSUB : avec GOSUB, l'ordinateur revient à l'endroit où il se trouvait avant de se brancher sur la routine et y reprend le cours normal du programme principal. Ceci est pratiquement impossible avec GOTO, car l'interpréteur ne sait plus à partir de quel endroit il a sauté pour se rendre à l'embranchement <cible>. Par contre, avec GOSUB (vous avez sans doute deviné qu'il s'agit de la contraction de GOSUBroutine : sauter jusqu'à la routine) l'interpréteur pose une marque à l'endroit qu'il quitte pour aller exécuter la routine, ce qui lui permet ensuite d'y revenir après RETURN, et de reprendre l'exécution du programme.

La <cible> indiquée après GOSUB peut être un numéro de ligne, une étiquette ou tout autre paramètre accepté par l'instruction GOTO.

Le recours aux routines a plusieurs avantages : tout d'abord, cela rend le texte du programme beaucoup plus lisible ; ensuite, cela vous permet ultérieurement de reprendre et modifier plus facilement le programme ou la routine. Imaginez que vous écriviez un programme dans lequel vous souhaitez trier des nombres par ordre croissant à une dizaine d'endroits différents. Pour une raison quelconque, vous changez d'avis et vous voulez un tri en ordre décroissant : si vous avez pris soin d'écrire une routine, il vous suffit de la reprendre et de modifier l'ordre de tri, terminé ! sinon, vous devez corriger le texte du programme à une dizaine d'endroits. Non seulement vous vous infligez ainsi un travail superflu, mais vous y ajoutez un risque d'erreur, car l'expérience prouve que l'on oublie vite de corriger telle ou telle partie du programme lorsque celui-ci devient un peu long.

L'Omikron Basic dispose également d'une variante :

ON...GOSUB

qui répond à la même syntaxe que ON..GOTO. Seule différence : ON..GOSUB permet de sauter jusqu'à une routine, après l'exécution de laquelle l'interpréteur reprend le cours normal du programme.

□ Les procédures

Les procédures sont elles-aussi des sous-programmes, mais elles sont plus complètes que les routines, car elles sont apparues plus récemment.

Une procédure vous permet de définir une nouvelle instruction que vous pouvez ensuite appeler à l'intérieur du programme principal ; une fois définie, elle est disponible même dans l'éditeur standard. Mais une procédure permet encore bien d'autres choses :

- vous pouvez lui transmettre des paramètres
- vous pouvez y définir des variables locales
- elle peut renvoyer les paramètres
- une procédure peut s'appeler elle-même (récursivité)

Rappelons que toutes ces choses étaient, il y a encore peu de temps, irréalisables avec la plupart des interpréteurs Basic, et qu'elles le sont devenues grâce à l'Omikron Basic. Assez de grands mots, passons à des choses plus concrètes.

Pour que l'interpréteur puisse reconnaître une procédure, il faut qu'elle soit créée à l'aide de l'instruction

```
DEF PROC <nom>
```

la fin de la procédure étant signalée par un RETURN. Voici comment définir une procédure <nom> :

```
DEF PROC <nom>  
  <instruction 1>  
  <instruction 2>  
  <instruction 3>  
  ....  
  ....  
  <instruction n>  
RETURN
```

Pour appeler cette procédure depuis le programme principal, il vous suffit d'entrer son nom <nom> comme s'il s'agissait d'une instruction ordinaire. En effet, lorsque l'interpréteur rencontre, dans un texte de programme, une instruction qu'il ne connaît pas, il contrôle automatiquement s'il ne s'agit pas d'une procédure. Si tel est le cas, il exécute les instructions contenues entre DEF PROC et le plus proche RETURN, puis il repasse dans le programme principal. Prenons un exemple simple.

L'Omikron Basic se sert de l'instruction CLS pour vider le contenu de l'écran. Ceci étant fait, le curseur vient se placer dans le coin supérieur gauche. Il arrive souvent que l'on ne veuille pas vider le contenu de l'écran mais simplement amener le curseur tout en haut à gauche, ce que provoque la touche <home> sur le clavier : l'Omikron Basic ne vous offre pas cette possibilité à l'intérieur d'un programme. Nous allons donc confectionner une procédure qui se chargera de cette tâche :

```
DEF PROC Home  
PRINT CHR$(27);"H"  
RETURN
```

après quoi il vous suffira d'écrire l'instruction "Home" dans le programme principal pour que le curseur vienne se placer en haut à gauche de votre écran. L'instruction "Home" sera d'ailleurs également utilisable sous l'éditeur standard. La deuxième ligne de notre définition ci-dessus vous paraît sans doute obscure : il s'agit en fait de l'instruction servant à repositionner le curseur à la position voulue.

Il convient en effet de préciser que le moniteur de l'Atari possède une certaine 'intelligence' : il comprend certaines instructions, comme par exemple

- vider l'écran
- placer le curseur à tel ou tel endroit
- effacer ou insérer une ligne
- activer/désactiver le curseur.

Ces instructions sont imitées sur celles d'un terminal (c'est pourquoi on parle d'émulateur VT-52) et commencent toujours par CHR\$27 (Escape, ESC) suivi d'une ou plusieurs lettres. Nous y reviendrons plus tard.

❑ Comment passer des paramètres

Contrairement à ce qui se passe avec GOSUB, une procédure peut reprendre un ou plusieurs paramètres du programme principal : il faut pour cela qu'ils soient écrits entre parenthèses juste après le nom de la procédure :

```
...
<instruction 1>
Setcursor(10,10)
....
....
<instruction n>

DEF PROC Setcursor(colonne, ligne)
    PRINT CHR$(27);CHR$(colonne+32);CHR$(ligne+32)
RETURN
```

Dans ce fragment de programme, la procédure 'setcursor' sert à amener le curseur à un endroit bien précis de l'écran, défini par les deux paramètres 'colonne' et 'ligne'. Ici aussi, vous faites appel à l'émulateur VT-52 de l'Atari. Les deux paramètres cités, colonne et ligne, sont définis localement : qu'est-ce que cela signifie ?

❑ Les variables globales et locales

Normalement, les variables ont une portée globale en Omikron Basic : cela signifie que la variable représente une même valeur dans tout le programme. Les variables locales ne représentent par contre une valeur qu'à l'intérieur des limites de la procédure concernée. Hors de la procédure, la même variable globale peut représenter une toute autre valeur, même si elle porte exactement le même nom. Les variables globales n'ont absolument aucun rapport avec leurs collègues locales. Prenons un exemple :

```
100 Colonne=20 : Ligne=10
110 PRINT "Colonne :";Colonne;" Ligne :";Ligne
120 Setcursor(10,5)
130 PRINT "Colonne :";Colonne;" Ligne :";Ligne
140 END
150 'voici la procédure setcursor
160 DEF PROC Setcursor(Colonne,Ligne)
170 PRINT CHR$(27);CHR$(Colonne+32);CHR$(Ligne+32)
180 PRINT "Dans la procédure : Colonne :";Colonne;"
    Ligne :";Ligne
190 RETURN
```

Ce petit exemple vous montre bien que les deux variables 'colonne' et 'ligne' sont définies localement (ligne 170), à l'intérieur de la procédure. Elles apparaissent aussi dans le programme principal (ligne 100) où elles ont un sens tout différent. Elles reprennent d'ailleurs cette valeur globale dès que l'on ressort de la procédure. Il en irait tout autrement si les deux paramètres n'étaient pas transmis lors de l'appel de la fonction :

```
100 Colonne=20 : Ligne=10
110 PRINT "Colonne :";Colonne;" Ligne :";Ligne
120 Setcursor
130 PRINT "Colonne :";Colonne;" Ligne :";Ligne
140 END
150 'voici la procédure setcursor
160 DEF PROC Setcursor
170 PRINT CHR$(27);CHR$(Colonne+32);CHR$(Ligne+32)
180 PRINT "Dans la procédure : Colonne :";Colonne;"
    Ligne :";Ligne
190 RETURN
```

Dans ce cas, les variables 'colonne' et 'ligne' ont une portée globale, et prennent donc chacune la même valeur dans la procédure et dans le programme principal. Si vous souhaitez que ces variables prennent dans la procédure une valeur purement locale, vous devez alors le préciser en écrivant :

```
LOCAL <variable>,<variable>,...
```

la variable ainsi déclarée représente dans la procédure une valeur, qui sera purement et simplement ignorée par l'interpréteur lorsqu'il sera revenu dans le programme principal. Vous pouvez même y créer une variable de même nom qui pourra prendre une toute autre

valeur. En résumé, une variable représente une valeur valable dans l'ensemble du programme tant que vous ne l'avez pas écrite entre parenthèses juste après l'appel d'une procédure ou que vous ne l'avez pas définie comme étant de portée uniquement locale dans la procédure.

Vous passez des valeurs à une procédure en les inscrivant entre parenthèses juste après l'instruction servant à appeler cette fonction. Evidemment, le nombre des paramètres ainsi passés doit correspondre avec le nombre de paramètres indiqué dans la définition de la procédure, et les variables doivent aussi être de même type. Faute de quoi l'interpréteur vous enverra un message d'erreur, et le déroulement du programme sera interrompu.

Il arrive souvent que l'on appelle une procédure avec certaines valeurs qui sont traitées et modifiées pour être ensuite repassées dans le programme principal. Vous indiquez dans ce cas à l'ordinateur que la valeur en question sera retournée en faisant précéder la variable d'un 'R' dans la définition de la procédure :

```
DEF PROC Setcursor(R Colonne, R Ligne)
```

```
...  
RETURN
```

Si vous intégrez ce type de définition dans la procédure de notre programme de démonstration, les variables 'Colonne' et 'Ligne' prennent les valeurs 10 et 5 à partir du moment où vous appelez la procédure, car les valeurs figurant après l'appel de la fonction sont repassées au programme principal dès la fin de l'exécution de la procédure.

☐ La récursivité

Les anciennes variantes du Basic ne permettaient pas à une procédure de s'appeler elle-même, contrairement à ce qui est aujourd'hui faisable avec l'Omikron Basic. C'est ce qu'on appelle la 'récursivité'.


```
....  
...  
Dessin(10,20,100,250)  
....  
....  
DEF PROC Dessin(x1,y1,x2,y2)  
....  
...  
Dessin(x1,y1,x2,y2)  
....  
....  
RETURN
```

Ce fragment de programme illustre le mode de fonctionnement d'une procédure prévue pour être récursive. On commence par appeler la routine comme d'habitude, et les paramètres transmis sont traités selon les indications entrées. Ce qui est par contre nouveau ici, c'est que cette fonction s'appelle elle-même à l'intérieur de la routine. Elle reprend les paramètres qu'elle vient de traiter.

Ce nouvel appel fait que les paramètres transmis subissent de nouveau le traitement indiqué dans la procédure, et il en va de même jusqu'à ce que le programme en ressorte. L'interpréteur ne revient dans le programme principal qu'après avoir traité les paramètres comme il se doit selon la fonction utilisée. Cette récursivité permet de traiter certains problèmes d'une façon très élégante. Nous dépasserions certainement les limites fixées à cet ouvrage en présentant toutes les possibilités que la récursivité offre aux programmeurs (il existe d'ailleurs une foule d'ouvrages consacrés à ce sujet) mais nous allons tout de même nous arrêter sur quelques exemples concrets.

J'espère que vous avez maintenant acquis une bonne connaissance des tableaux de variables. Pour trier ces tableaux, il faut recourir à un algorithme spécifique, et les informaticiens firent preuve de beaucoup d'ingéniosité en ce domaine. Il existe un nombre quasi illimité d'algorithmes de tri, du plus simple 'bubble sort' (tri par permutation) en passant par le 'shell-sort', le 'heap sort' (tri vertical) pour finir par le 'bucket-sort' (tri par groupe de blocs). Chaque algorithme de tri a ses avantages et inconvénients. L'un des plus

rapides fut inventé en 1962 par C.A.R. Hoare : c'est le 'quicksort' (tri rapide), qui -vous l'aviez deviné- se sert de la récursivité pour trier les données :

```

100 *****
110 *                                     QUICK.BAS                                     *
120 *-----*
130 * Auteur : Michael Maier      Version 1.00   Date :    16.07.1990   *
140 *      Programme joint au 'Livre de l'Omkron Basic'                 *
150 *      (c) 1990 MICROAPPLICATION                                     *
160 *****
170 *
180 *
190 DIM Tableau%(9)
200 RESTORE
210 FOR T%=0 TO 9
220   READ Tableau%(T%)
230 NEXT T%
240 *
250 Quicksort(0,9)
260 *
270 FOR T%=0 TO 9
280   PRINT Tableau%(T%); " ";
290 NEXT T%
300 DATA 3,9,7,2,0,6,1,5,4,8
310 END
320 *
330 DEF PROC Quicksort(debut%,Fin%)
340   ' dans la ligne ci-dessous, nous procédons a un test
350   ' qui n'a rien a voir avec la routine a proprement
360   ' parler !
370   PRINT "#"; : FOR T%=0 TO 9 : PRINT Tableau%(T%); " "; :NEXT % : PRINT
380   ' nous entrons dans le vif du sujet
390   LOCAL A%,Z%
400   A%=Debut%
410   Z%=Fin%
420   X%=Tableau%((Debut%+Fin%)/2)
430   REPEAT
440     WHILE Tableau%(A%)<X%
450       A%=A%+1
460     WEND
470     WHILE Tableau%(Z%)>X%
480       Z%=Z%-1
490     WEND
500     IF A%<=Z% THEN
510       SWAP Tableau%(A%),Tableau%(Z%)
520       ' les deux lignes ci-dessous ne servent aussi qu'à
530       ' faire un test et vous pouvez les supprimer
540       FOR T%=0 TO 9 :PRINT Tableau%(T%); " "; :NEXT T%
550       PRINT ,A%; " ";Z%; " ";X%
560       ' le programme normal reprend ici
570       A%=A%+1
580       Z%=Z%-1
590     ENDIF
600   UNTIL A%>Z%
610   IF Debut%<Z% THEN
620     Quicksort(Debut%,Z%)

```

```
630     ENDIF
640     IF A% < Fin% THEN
650         Quicksort(A%, Fin%)
660     ENDIF
670 RETURN
```

Ce programme sert à trier un integer-array (tableau de nombres entiers) que nous appelons 'tableau%'. Avant de trier le tableau en appelant la routine 'quicksort' à la ligne 250, il faut attribuer un indice particulier à chacune des différentes valeurs : d'où l'utilité de la boucle FOR..NEXT accompagnée de l'instruction READ-DATA. Les valeurs elles-mêmes se trouvent à la ligne 300, derrière une instruction DATA. Une fois cette tâche terminée, nous pouvons appeler la routine de tri 'quicksort'. Les paramètres à transmettre sont le premier et le dernier indice.

A chaque branchement dans la routine, le programme commence par afficher le contenu complet du tableau grâce à une instruction PRINT. Après quoi, les variables locales A% et Z% se voient attribuer les paramètres (locaux) Debut% et Fin% entrés juste après DEFPROC (ligne 330).

L'étape suivante consiste à diviser le tableau en deux parties, ce que l'on fait en attribuant à la variable X% l'élément se trouvant au milieu du tableau qui est dès lors utilisé comme variable de comparaison. La première boucle WHILE recherche le premier élément à partir du bas (Debut) qui ne soit pas inférieur à la variable servant de point de comparaison. Parallèlement, la boucle suivante recherche le premier élément à partir du haut (Fin du tableau) qui ne soit pas supérieur à la variable X%. Les deux éléments ainsi retrouvés sont échangés l'un contre l'autre puis ignorés puisqu'ils peuvent théoriquement être équivalents.

A propos d'échanger un élément contre un autre, sachez qu'il existe, en Omikron Basic, une instruction particulière : en écrivant

```
SWAP <Tableau(X)>, <Tableau(Y)>
```

vous échangez entre eux les éléments X et Y du tableau 'tableau'.

La boucle REPEAT veille à ce que ce petit jeu se poursuive jusqu'à ce que la limite inférieure dépasse la valeur du début de tableau. Il n'y a plus alors dans les deux parties du tableau d'élément qui puisse appartenir à l'autre partie.

Après quoi le programme trie les deux parties de tableau. Pour cela, la fonction fait appel à la récursivité et s'appelle elle-même : les limites de tri sont fixées dans chacune des deux parties de tableau. Pour que cela vous semble encore plus évident, j'ai ajouté deux lignes de test dans le texte du programme, lignes qui vous montrent le contenu du tableau. Cela se produit donc à chaque branchement dans la routine 'quicksort' et à chaque permutation de deux variables entre elles. Pour que vous puissiez distinguer facilement chacune des deux lignes, je fais toujours commencer la première par un dièse (#). Après chaque permutation, vous voyez de plus s'afficher le contenu du bas ou du haut du tableau, ainsi que l'élément servant de point de comparaison.

Lancez maintenant ce programme, et vous voyez tout d'abord s'afficher sur votre écran le tableau de départ :

```
# 3 9 7 2 0 6 1 5 4 8
```

Le programme prend le 6 comme élément de référence. Le premier élément à partir du bas (ici : à partir de la gauche) qui ne soit pas inférieur à 6 est le 9, et le premier élément à partir du haut qui ne soit pas supérieur à 6 est le 4. Le programme les échange l'un contre l'autre, ce qui donne :

```
3 4 7 2 0 6 1 5 9 8
```

```
1 8 6
A% Z% X%
```

Les deux variables A% et Z% ne dépassant pas encore les valeurs de début et de fin du tableau, la procédure est donc relancée ; l'interpréteur rencontre le 7 à partir du bas, et le 5 à partir du haut ; il les permute et nous obtenons :

```
3 4 5 2 0 6 1 7 9 8
```

```
2 7 6
A% Z% X%
```

Les deux variables A% et Z% n'ont pas encore respectivement dépassé les valeurs de début et de fin du tableau, ceci signifie que la procédure est relancée. les deux variables suivantes que l'interpréteur va échanger l'une contre l'autre sont le 6 et le 1, ce qui nous donne :

3 4 5 2 0 1 6 7 9 8	5	6	6
	A%	Z%	X%

Nous y sommes : la limite inférieure dépasse la valeur de début du tableau : le programme doit donc trier les deux parties du tableau en appelant récursivement la procédure. Auparavant, il réaffiche à l'écran les éléments contenus dans le tableau :

3 4 5 2 0 1 6 7 9 8

Lors de son appel récursif, la fonction prend 2 comme valeur de comparaison ; elle recherche à nouveau à partir du bas le premier élément qui ne soit pas plus petit que la valeur de comparaison, ce qui donne le 3 dans notre exemple. Cet élément est échangé contre le premier élément à partir du haut qui ne soit pas supérieur à 2, ici le 1 ; ce qui nous donne :

1 4 5 2 0 3 6 7 9 8	0	5	2
	A%	Z%	X

Le processus se poursuit tant qu'il le faut :

1 0 5 2 4 3 6 7 9 8	1	4	2
	A%	Z%	X%

1 0 2 5 4 3 6 7 9 8	2	3	2
	A%	Z%	X%

La procédure s'appelle elle-même de nouveau, en prenant cette fois le zéro comme point de comparaison, qu'elle échange contre le 1 :

# 1 0 2 5 4 3 6 7 9 8			
0 1 2 5 4 3 6 7 9 8	0	1	0
	A%	Z%	X%

La prochaine comparaison se fait autour du 2, qui est échangé avec lui-même :

0 1 2 5 4 3 6 7 9 8

0 1 2 5 4 3 6 7 9 8

2	2	2
A%	Z%	X%

La procédure se répète jusqu'à ce que les deux parties du tableau soient correctement triées. Cet algorithme (qui est d'ailleurs implémenté dans la routine de tri de l'Omikron Basic) montre bien le fonctionnement et l'élégance d'écriture des routines récursives. Ne cherchons pas à en dissimuler les inconvénients : il est souvent difficile de créer une routine récursive, tout simplement parce qu'il est difficile de trouver une utilisation appropriée de la récursivité. D'autre part, il faut savoir que l'exécution d'une routine récursive coûte plus de place mémoire qu'une routine itérative (contraire de récursive). Pourquoi ?

L'interpréteur doit identifier l'endroit qu'il quitte dans le programme principal pour sauter jusqu'à la routine : il a besoin pour cela de mémoriser les données correspondantes dans un espace de la mémoire vive qui est aussi utilisé par le processeur : le 'stack' (la pile). Lorsqu'une procédure s'appelle elle-même, il est évident que cela fait croître la pile. Il existe certainement des problèmes de programmation qu'il serait impossible de résoudre à l'aide de procédures récursives, car même la mémoire d'un Méga-ST 4 ne suffirait plus pour mémoriser la pile.

□ Les fonctions

L'Omikron Basic vous permet de définir vous-même des fonctions, tout comme vous le faites pour les procédures. Contrairement à une procédure, une fonction retourne toujours une valeur (une procédure pouvant quand à elle servir pour effectuer une impression, produire un son,...).

[Paramètres affectés à la fonction] ---Fonction---> Valeur

Cette valeur peut être ensuite attribuée à une variable ou affichée immédiatement à l'écran ; une fonction peut fort bien ne pas être affectée de paramètres, mais elle fournira toujours une valeur en retour. Il existe deux types de fonctions en Omikron Basic :

- celles qui tiennent sur une seule ligne
- celles qui tiennent sur plusieurs lignes.

□ Les fonctions sur une seule ligne

Avant d'appeler une fonction, vous devez la définir à l'aide de :

```
DEF FN <Nom(paramètres)> = <formule>
```

L'instruction DEF FN indique à l'interpréteur qu'il va devoir créer une fonction appelée <Nom> à laquelle il devra affecter les arguments (paramètres) alors que <formule> contient la véritable formulation de la fonction. Je voudrais ici reprendre l'exemple de la conversion des Francs français en D-Mark. (Si si, j'insiste !)

Commençons par écrire la définition de la fonction :

```
DEF FN D-Mark(Montant!) =  
INT((Montant!/3.3333*100)+.5)/100
```

Vous pouvez écrire cette définition de la fonction n'importe où dans le texte du programme, car l'interpréteur recherche et exécute automatiquement toutes les DEF FN dès que vous lancez un programme. Lorsque vous souhaitez appeler une de ces fonctions, vous écrivez FN suivi du nom de la fonction voulue, suivi des paramètres nécessaires comme argument :

```
Change! = FN D-Mark(100)
```

Le montant de 100 FF est converti en D-Mark puis passé à la variable Montant!, et vous pouvez ensuite demander son affichage par une instruction PRINT. En entrant directement :

```
PRINT FN D-Mark(100)
```


on évite de passer le montant à la variable pour demander sa visualisation immédiate : 30.00.

Lorsque la fonction doit retourner un string (chaîne de caractères), il faut que son nom soit suivi d'un signe '\$' : en programmant

```
DEF FN Screen$(X$) = CHR$(27)+X$
```

nous obtenons un string long de deux caractères, commençant par un 'escape', ce qui permet d'interpeller l'émulateur VT-52 de l'Atari-ST (vous vous en souvenez ?).

```
PRINT Screen$("H")
```

sert à repositionner le curseur tout en haut à gauche de l'écran (HOME) sans pour autant vider son contenu. Une fonction peut aussi appeler d'autres fonctions ; prenons par exemple la fonction de repositionnement du curseur :

```
DEF FN Crsr$(Col,Lig) = FN  
Screen$("Y")+CHR$(32+Col)+CHR$(32+Lig)
```

Pour ensuite amener le curseur à la position 10,10, il vous suffira d'écrire :

```
PRINT FN Crsr$(10,10)
```

Il va de soi que le nombre et le type des paramètres passés doivent respecter la définition de la fonction. Mais il est possible d'écrire une fonction dépourvue de paramètres :

```
DEF FN Hasard = -RND(-6)
```

Cette fonction fournira des nombres tirés au hasard et compris entre 1 et 6 ; pour ce faire, nous nous servons du générateur de nombres aléatoires de l'Atari, qui est lancé par la fonction RND (randomize). En lui affectant un argument sous forme de nombre négatif, il fournit des nombres entiers compris entre -1 et la valeur indiquée. Le signe négatif placé devant la fonction RND fait que la valeur retournée par la fonction devient positive :

```
-(-4) = +4
```

□ Les fonctions sur plusieurs lignes

L'Omikron Basic vous permet aussi d'écrire des fonctions nécessitant plusieurs lignes. Reprenons notre conversion des FF en DM pour bien illustrer la différence entre ces deux types de fonctions :

```
100 DEF FN D-Mark(Montant!)  
110 RETURN INT((Montant!)*3.3333*100)+.5)/100
```

Cette fonction se distingue sur deux points de sa variante écrite sur une seule ligne :

- l'opérateur d'attribution '=' a disparu
- on sort de la fonction par un RETURN, mais attention : la valeur à retourner doit se trouver après RETURN.

On se doute immédiatement qu'une fonction écrite sur plusieurs lignes coûte plus de travail ; on en est récompensé du fait qu'elle offre plus de possibilités et qu'elle est plus souple d'emploi :

- on peut intégrer dans sa définition des structures de contrôle conditionnel du genre IF..THEN..ELSE ou des boucles,
- selon les conditions fixées, on peut lancer l'exécution de la telle ou telle formulation de la fonction,
- on peut programmer en jouant sur sa récursivité.

Prenons comme exemple le calcul d'une factorielle, fonction mathématique et plus précisément statistique définie par :

- la factorielle de 0 est 1
- la factorielle de X est X fois la factorielle de (X-1).

La factorielle de 5 (que l'on écrit 5 !) se calcule selon la formule :

$$5*4! = 5*4*3! = 5*4*3*2! = 5*4*3*2*1! = 5*4*3*2*1 = 120$$

Un problème très classique de récursivité, que l'on solutionne de la façon suivante :


```

100 DEF FN Factorielle(X!) ' Float, car valeurs
    extrêmement grandes
110 IF FRAC(X!) > 0 THEN ' y-a-t-il des chiffres après
    la virgule?
120     PRINT "la factorielle ne s'applique qu'à des
    nombres entiers!"
130     'EXIT TO <cible> revenir par EXIT
140 ENDIF
150 IF X!=0 THEN
160     RETURN(1) ' 0! = 1
170 ENDIF
180 RETURN FN Factorielle(X!-1)*X!

```

Nous sommes obligés d'utiliser un nombre à virgule flottante comme argument de la fonction, car on atteint rapidement des nombres extrêmement élevés lors du calcul d'une factorielle ($69! = 1,71 * 10^{98}!!!$), nombres qu'il est impossible d'attribuer à une variable du type integer.

Il est possible de sortir d'une fonction par EXIT TO <cible>. C'est ici nécessaire lorsque l'utilisateur entre un nombre décimal, car cela n'est pas (mathématiquement) acceptable : le calcul d'une factorielle ne s'applique qu'à des nombres naturels entiers positifs, y compris zéro. Vous indiquez sous <cible> un numéro de ligne ou une étiquette auquel l'interpréteur doit retourner dans ce cas. Je sais que je me répète, mais je tiens à redire que la fonction FRAC(X!) retourne les chiffres d'un nombre se trouvant après la virgule, et donc une valeur 'vrai' lorsqu'il y en a.

L'utilisation de la récursivité ne devrait plus vous poser de problème; appelez la fonction par Factorielle(5) et vous constaterez que :

- la fonction se rappelle elle-même aussi longtemps que sa valeur n'a pas atteint 0, en diminuant à chaque fois sa valeur de 1
- un fois atteint la valeur 0, les différentes valeurs sont multipliées entre elles, puis retournées ; terminé !

Sachez d'ailleurs que l'Omikron Basic est doté d'une fonction permettant de calculer la factorielle d'un nombre entier, ce qui vous épargne de l'écrire vous-même : FACT(X). Contrairement aux

procédures, il est impossible de donner comme argument à une fonction des paramètres à retourner : les fonctions ne peuvent retourner qu'une seule valeur !

Je ne voudrais pas terminer ce chapitre sans vous donner une fonction très utile pour manipuler les chaînes de caractères et qui n'existe malheureusement pas en Omikron Basic :

Insert\$.

```
100 DEF FN Insert$(Master$,Inserer$,Position)
110 Master$= LEFT$(Master$,Position-1)+Inserer$+
MID$(Master$,Position)
120 RETURN Master$
```

Cette fonction vous permet d'insérer le string 'Inserer\$' dans la chaîne de caractères 'Master\$' à partir de la position 'Position'.

Nous voilà maintenant équipé de connaissances suffisantes pour passer à des projets plus ambitieux ! mais je vous dois encore des explications au sujet des instructions READ, DATA et RESTORE dont je me suis servi dans le programme Quicksort sans vous les expliquer.

2.12. READ, DATA et RESTORE

Il existe plusieurs façons d'entrer des valeurs dans un tableau :

- vous pouvez attribuer une valeur à chaque élément

A(0) = 10

A(1) = 3

A(2) = 41

....

....

- vous pouvez saisir manuellement les valeurs à chaque lancement du programme, ce qui n'est pas particulièrement pratique :


```
FOR T% = 0 TO Quantite
INPUT A(T%)
NEXT T%
```

- vous pouvez enfin et surtout inscrire les valeurs après une instruction DATA et les rappeler ensuite par READ.

```
100 REM entrer des valeurs dans le tableau Array A()
110 DIM A(10)
120 FOR T%=0 TO 10
130   READ A(T%)
140 NEXT T%
150 '
160 '
170 '
180 DATA 10,20,30,50,60,20,65,0,4,521,87
190 END
```

La ligne 180 contient exactement 11 valeurs différentes, très proprement séparées l'une de l'autre par une virgule. Lorsque l'interpréteur rencontre une instruction

READ

il prend dans la ligne DATA l'élément désigné par le pointeur DATA. Ce pointeur indique à l'ordinateur l'élément qu'il doit prendre dans la ligne DATA. Au lancement du programme, ce pointeur désigne toujours le premier élément de la première ligne DATA. Comme notre programme ne comprend qu'une ligne de DATA, l'interpréteur prend donc la valeur 10.

Après quoi le pointeur DATA avance d'une position et désigne l'élément suivant (dans notre exemple : 20). A la prochaine instruction READ, l'interpréteur se saisira de cette valeur tout en faisant avancer le pointeur d'une position. Ce processus se répète à chaque instruction READ, jusqu'à ce que toute la ligne de DATA ait été lue.

Si vous entrez plus d'instructions READ qu'il n'y a d'éléments sur la ou les ligne(s) de DATA, l'ordinateur vous renvoie un message d'erreur OUT OF DATA. Il existe une instruction permettant de déplacer le pointeur de DATA pour qu'il désigne une valeur voulue :

RESTORE

l'instruction **RESTORE** remplace le pointeur **DATA** sur le premier élément de la première ligne **DATA** ; vous pouvez aussi lui faire désigner une valeur quelconque en faisant suivre **RESTORE** d'un numéro de ligne ou d'une étiquette (label).

RESTORE 500

amène le pointeur **DATA** sur le plus proche élément identifiable à partir de la ligne 500 du programme.

RESTORE Marque

amène le pointeur **DATA** sur l'élément le plus proche identifiable après l'étiquette 'Marque'.

ON...RESTORE

fonctionne comme **ON...GOTO** et amène le curseur **DATA** sur la ligne **DATA** indiquée après **RESTORE**.

```

100 INPUT " Mouvements du mois ... (1 - 12) ";Mois%
110 ON Mois RESTORE Janvier, Février, Mars, Avril, Mai, Juin ...
120 '
130 FOR Tx=1 TO 31
140   READ Mouvement
160   IF Mouvement=-1 THEN
170     EXIT
180   ENDIF
190   PRINT Tx;" " ;Mois%;" " ;Mouvement
200 NEXT Tx
210 END
220 '
230 -Janvier
240 DATA 5436,4986,45673,12345,23987,23769,2976
250 DATA 7659,3276,4598,5456,5986,3456,-1
260 -Février
270 DATA 65786,4987,3096,2386,2387,30975,23765,2386
280 DATA 54675,23987,-1
290 -Mars
300 'etc. etc.
```

Ce programme est très incomplet ; il indique les mouvements effectués chaque mois sur un compte quelconque. L'instruction **ON...RESTORE** fait que le pointeur **DATA** désigne la première valeur, le premier mouvement, de chaque mois. Les valeurs seront ensuite lues grâce à une instruction **READ**.

Ce qui est nouveau, c'est que nous contrôlons la lecture de la ligne DATA : lorsque le pointeur arrive sur -1, cela signifie que toute la ligne a été lue et qu'il convient donc de ressortir de la boucle. Cela nous garantit que l'interpréteur ne peut lire qu'autant de valeurs qu'il en existe effectivement : c'est absolument indispensable lorsque le nombre d'éléments varie d'une ligne DATA à l'autre (comme c'est le cas dans notre exemple puisque nous traitons des valeurs mensuelles), faute de quoi nous recevions un message OUT OF DATA.

Naturellement, les lignes DATA peuvent aussi contenir des chaînes de caractères, que vous devez écrire entre guillemets :

```

90 DIM A$(12) : TX=0
100 REPEAT
110   TX=TX+1
120   READ A$(TX) ' affectation d'une variable string
130 UNTIL A$(TX)="Décembre"
140 '
150 END
160 '
170 DATA "Janvier","Février","Mars","Avril","Mai","Juin"
180 DATA "Juillet","Août","Septembre","Octobre","Novembre"
190 DATA "Décembre"

```

La variable A\$(0) contiendra les différents noms des douze mois de l'année, qui se trouvent dans les lignes DATA. La boucle REPEAT...UNTIL provoquera la lecture des douze noms différents, jusqu'à 'Décembre' inclus : ceci étant fait, l'interpréteur ressort de la boucle. Ceci évite d'avoir à entrer les différentes valeurs, ce qui non seulement coûte beaucoup de temps mais représente aussi une source d'erreurs lorsqu'on doit entrer de longues colonnes de chiffres.

2.13. Tout sur le Basic

Nous approchons à grand pas du prochain chapitre, dans lequel nous verrons comment gérer les fichiers et les sauvegarder sur une disquette. Et comme cela annonce la fin de votre initiation aux bases de l'Omikron Basic, nous allons tenter quelque chose de plus risqué.

Avant de sauvegarder des données sur une disquette, il faut commencer par les saisir. Cela nous fait tout de suite penser à écrire un petit programme simple qui vous servira à gérer votre carnet d'adresses. Nous en profiterons pour éclaircir quelques principes de base que nous n'avons pas encore eu l'occasion d'étudier.

Jusqu'à présent nous n'avons écrit que des fragments de programme très courts, dont nous pouvions écrire le texte sans réflexion préalable. Plus les programmes deviennent compliqués, plus il convient de réfléchir avant de passer à la programmation proprement dite. Ce délai de réflexion nous épargnera ensuite beaucoup de temps de travail et de test.

Le Basic pousse le programmeur à bricoler un texte de programme sans trop réfléchir puis à procéder par essais successifs jusqu'à obtenir l'effet voulu. Les langages compilés n'induisent pas les programmeurs dans ce genre de tentation, car le programme n'est pas exécutable avant d'être complètement écrit. Imaginez que vous ayez écrit un bout de programme et que vous deviez à chaque fois attendre quelques minutes (délai nécessaire pour la compilation) avant de voir le résultat de votre oeuvre : il vous suffira de quelques essais pour vous persuader qu'il est indispensable et plus économique de bien réfléchir votre programmation avant de procéder à des essais. Vous devriez, dans votre propre intérêt, adopter la même démarche lorsque vous programmez en Basic.

☐ Examiner le problème à résoudre

Avant toute programmation, il convient d'examiner attentivement le problème à résoudre. Notre objectif est de gérer un carnet d'adresses, cela paraît facile. Quelles tâches devra assumer le programme ?

Les professionnels sont astreints à respecter un 'cahier des charges', dans lequel le client décrit exactement ce que le programme doit pouvoir faire. Pour notre carnet d'adresses, nous pourrions mentionner les points suivants :

- ① saisie des adresses
- ② correction, modification et éventuellement suppression d'une adresse
- ③ recherche d'une adresse précise
- ④ sauvegarde des adresses sur une unité de disque(tte)
- ⑤ rechargement des adresses depuis une unité de disque(tte)
- ⑥ sortir du programme

Quelles données allons-nous retenir sous chaque adresse ?

- ① le nom propre et le prénom
- ② la rue, avec le numéro
- ③ le code postal et la ville
- ④ le numéro de téléphone
- ⑤ la date de naissance

Après avoir ainsi déterminé les objectifs poursuivis et les données à saisir, nous devons nous soucier de rendre notre programme 'convivial', c'est-à-dire facile à utiliser pour des gens non-avertis. Il faut que nous offrions toutes les possibilités (les fonctionnalités) du programme à l'utilisateur, qui pourra choisir quelle fonction il souhaite utiliser : c'est ce qu'on appelle un 'MENU'. Et pour qu'il ait bien l'aspect d'un véritable menu, nous allons construire la première page d'écran de la façon suivante :

- d'abord un en-tête (HEADER)
- ensuite les fonctions disponibles
- enfin un message demandant à l'utilisateur de faire son choix.

Ce menu est affiché à l'écran juste après le lancement du programme, et l'ordinateur attend que l'utilisateur ait appuyé sur une touche adéquate ; à l'étape suivante, il contrôle si la touche appuyée correspond bien à une des fonctionnalités du menu ou s'il s'agit d'une touche erronée, auquel cas il l'ignore sans que le programme se plante. Lorsque la touche appuyée correspond bien à une des options offertes par le menu, le programme lance la fonction qui s'y rapporte (voir figure 2.1).

Les fonctions elles-mêmes sont écrites dans des sous-programmes (modules) accessibles par `ON..GOSUB`.

Nous nous sommes déjà mis d'accord sur le type de données que nous entendions enregistrer dans notre fichier d'adresses. Mais comment faire pour organiser cela au mieux dans l'ordinateur ? Evidemment, me répondrez-vous, il nous faut un tableau array. La dimension de ce tableau sera affectée à une variable, de façon à pouvoir la modifier rapidement par la suite si cela s'avère nécessaire. Dans un premier temps, une dimension de 100 devrait largement nous suffire.


```

*****
*                                     *
*          FICHER - Menu principal          *
*          -----                      *
*          Un programme tiré du livre de l'Omikron Basic      *
*                                     *
*****

          1. Saisie d'une adresse
          2. Correction d'une adresse
          3. Recherche d'une adresse
          4. Sauvegarde du fichier
          5. Chargement du fichier
          6. Sortir du programme

          Veuillez préciser votre choix

```

Figure 2.1 : le menu principal du fichier d'adresses.

Une fois le terrain ainsi déblayé, nous ne devrions plus guère avoir de problèmes pour écrire notre programme. Le programme commence généralement par le dimensionnement du tableau de strings, puis vient la construction du menu. On utilise pour cela l'instruction PRINT, ou mieux encore son cousin PRINT AT (en tant qu'informaticien quasiment chevronné, vous prononcez maintenant tout cela à l'anglaise : Print èit).

□ PRINT AT

Cette instruction permet d'afficher des données à l'écran en précisant leur position selon un système de coordonnées désignant la ligne et la colonne ; syntaxe :

```
PRINT @(Ligne.Colonne);<texte>
```

Nous n'avons rien à ajouter au sujet de l'instruction PRINT ; pour écrire le signe '@' (en bon français : arrobasse), vous appuyez simultanément sur <alternate>et<#>. Derrière PRINT @, vous indiquez la ligne et la colonne où vous souhaitez que commence l'affichage de votre texte ; remarquez que le contenu du texte est séparé de l'indication du positionnement par un point-virgule.

Sachez que la première colonne de la première ligne de l'écran (position tout en haut à gauche) est désignée par les coordonnées 0,0 alors que la dernière colonne de la dernière ligne (position la plus à droite tout en bas de l'écran) correspond aux coordonnées 24,79. En entrant

```
PRINT @(0,1): "*" * 78
```

vous provoquez l'affichage d'un string se composant de 78 fois l'étoile '*' sur la première ligne à partir de la colonne 2. PRINT AT nous permet de structurer sans problème l'affichage à l'écran. Pour faire plus beau, nous décidons que l'affichage de 'Veuillez préciser votre choix' se fera en inversion-vidéo (caractères blancs sur fond noir). L'émulateur VT-52 se charge de l'inversion-vidéo, que l'on active par

```
PRINT CHR$(27): "p"
```

alors que

```
PRINT CHR$(27): "q"
```

sert à désactiver ce mode d'affichage. Tout cela est rendu plus lisible grâce à la fonction SCREEN\$() figurant sur la ligne 9, à laquelle nous affectons le caractère correspondant comme argument.

Le premier écran étant ainsi créé, nous demandons à l'ordinateur d'attendre que l'utilisateur appuie sur une touche. On pourrait pour cela penser à utiliser une instruction INPUT, mais il existe une façon plus élégante et plus efficace de régler ce problème :

□ INKEY\$

INKEY\$ retourne une chaîne de caractères de quatre octets, contenant les données relatives à la dernière touche appuyée. Contrairement à INPUT, la fonction INKEY\$ n'attend pas qu'une touche soit appuyée : tant qu'aucune touche n'est appuyée, elle retourne un string vide. Il faut donc l'intégrer à une boucle pour qu'elle se répète jusqu'à ce qu'une touche soit appuyée :

```
REPEAT
  ' affecte à A$ le résultat fourni par INKEY$
  A$ = INKEY$
UNTIL A$ <> "" 'jusqu'à ce qu'une touche soit appuyée.
```

Dès qu'une touche est effectivement appuyée, A\$ contient une chaîne de caractères de quatre octets, qui se décompose comme suit :

1er octet :

- Bit 0 : contrôle de la touche <shift> droite
- Bit 1 : contrôle de la touche <shift> gauche
- Bit 2 : contrôle de la touche <control>
- Bit 3 : contrôle de la touche <alternate>
- Bit 4 : contrôle de la touche <capslock>

2ème octet : scancode de la touche

3ème octet : non utilisé

4ème octet : code ASCII de la touche appuyée.

Seul le quatrième octet nous intéresse pour l'instant, et nous pouvons l'isoler en écrivant

```
RIGHT$(A$.1)
```

Attention : Aussi longtemps que l'utilisateur n'appuie pas sur une touche, la variable A\$ ne contient qu'un string vide : si nous tentons d'en isoler le dernier octet, l'ordinateur nous renverra un message d'erreur. A part ça, nous pouvons immédiatement concilier l'utile et l'agréable : l'interpréteur ne ressortira de la boucle que si l'utilisateur appuie sur une touche valable, c'est-à-dire comprise entre 1 et 6. Voici la version améliorée de cette boucle :

```
REPEAT
  A$= INKEY$
  IF A$ <> "" THEN
    A$= RIGHT$(A$,1)
  ENDIF
UNTIL A$ > "0" AND A$ <"7"
```

L'opérateur AND indique à l'ordinateur qu'il ne doit considérer la condition comme vraie que si ses deux parties sont vérifiées : $A\$ > 0$ et $A\$ < 7$: la variable doit être à la fois supérieure à zéro et inférieure à 7, nous y reviendrons plus loin. La variable A\$ se voit affecter les données correspondant à la dernière touche appuyée, elle représente un string vide tant qu'aucune touche n'est appuyée. La condition IF sert à vérifier si la variable A\$ contient ou non une chaîne de caractères, et si tel est le cas, elle isole le dernier octet, qui nous sert ensuite de critère de référence pour décider si le programme continue ou non.

L'interpréteur sort de la boucle lorsqu'une touche comprise entre 1 et 6 est appuyée. Vous vous rappelez bien sûr que les lettres ont une valeur ASCII et qu'elles peuvent donc aussi être comparée avec des touches numériques.

Dès que l'utilisateur appuie sur une des touches correctes, la variable A\$ identifie cette touche grâce à l'octet qui la caractérise. Cela nous embête un peu pour la suite du programme, car vous savez que ON...GOSUB attend de reprendre un nombre compris entre 1 et 6 et non un caractère : il ne nous reste plus qu'à transformer la variable en chiffre, mais comment ?

Certes, nous pourrions recourir à VAL() et écrire :

```
A = VAL(A$())
```

mais c'est trop simple pour nous. Connaissez-vous une autre solution ? Très juste ! nous allons rechercher le code ASCII de la touche en question à l'aide de la fonction ASC(). Prenez deux petites minutes pour vous reporter au tableau des codes ASCII qui se trouve dans les annexes à la fin de ce manuel : vous constatez que 1 porte le code 49, 2 le code 50 etc.

Il nous suffit donc de reprendre le code ASCII de A\$ grâce à ASC(A\$), puis d'en soustraire la valeur 48, et nous obtiendrons bien, selon la touche appuyée, des valeurs comprises entre 1 et 6, utilisables avec ON...GOSUB. Certes, nous serions parvenus plus rapidement au même résultat en utilisant VAL(), mais vous n'auriez alors rien appris sur la transformation des codes ASCII en valeurs numériques.

Il ne faut surtout pas oublier d'effacer le contenu de la variable A\$ avant que l'interpréteur n'entre dans la boucle, faute de quoi nous assisterions à un vrai miracle : l'interpréteur ressortirait de la boucle sans même que l'utilisateur n'ait appuyé sur une touche ! Essayez, histoire de rire un peu.

Le choix entre les différentes options du menu est intégré dans une boucle dont l'interpréteur ne peut ressortir que lorsque l'utilisateur appuie sur la touche 6. Voici la syntaxe de cette structure :

```
REPEAT
  <écran affiché>
  REPEAT
    <appui sur une touche?>
  UNTIL <appui sur une des touches correctes>
  ON <touche> GOSUB ---->
UNTIL <sortir du programme>
```

J'oubliais de vous dire qu'avant d'entrer dans la boucle INKEY\$, nous désactivons l'affichage du curseur d'écran pour qu'il ne nous gêne pas en clignotant n'importe où. Nous le réactivons après être sorti de la boucle, car nous en aurons bien besoin pour l'instruction INPUT qui va suivre.

Nous en avons terminé avec l'écriture de la structure du menu, venons-en aux différentes fonctions assumées par le programme.

Commençons par la saisie d'une nouvelle adresse : pour l'instant notre écran contient toujours le menu principal, que nous effaçons par CLS. Après quoi nous souhaitons qu'apparaisse à l'écran un formulaire de saisie dans lequel l'utilisateur entrera les données correspondant à l'adresse voulue. Une fois l'adresse complètement saisie, il faut demander à l'utilisateur ce qu'il veut faire :

- ❶ entrer une autre adresse
- ❷ modifier ou corriger une adresse existante
- ❸ revenir au menu principal.

Comment formuler au mieux les procédures assurant ces diverses fonctions ? Le masque affiché à l'écran sera le même pour les deux premières options, puisque la création et la correction d'une fiche-adresse se fait sur la base du même modèle d'affichage : nous pouvons formuler la présentation de l'écran (le masque de saisie) dans une procédure particulière.

Nous avons aussi intérêt à formuler une seule procédure pour la saisie de données, procédure qui pourra être utilisée pour saisir de nouvelles données ou modifier des fiches déjà existantes. L'utilisateur pourra choisir entre ces deux dernières possibilités, mais la façon de saisir les données restant la même, elle est gérée par un sous-programme particulier.

Toutes ces réflexions préalables sont indispensables, car elles vont nous permettre de répartir judicieusement entre diverses procédures le travail à fournir, pour écrire un programme aussi court que possible. Résumons les fonctionnalités dont nous devons disposer pour saisir une nouvelle adresse :

```
**** sous-programme : saisie d'une nouvelle fiche-adresse ****  
<construire le masque de saisie>  
<permettre la saisie d'une nouvelle adresse>  
<demander à l'utilisateur ce qu'il souhaite faire ensuite>
```


Nous allons maintenant affiner progressivement cette grille d'analyse encore grossière, avant de passer à l'écriture du programme. Nous ne devrions guère rencontrer de gros obstacles pour confectionner un masque de saisie : il faudra penser à intégrer un en-tête dans l'affichage. Pour ce faire, l'appel de la procédure sera accompagné d'un string (local) contenant l'en-tête, et nous faisons en sorte que cette mention vienne s'afficher en plein milieu de la largeur d'écran :

```
PRINT @(Ligne, (Longueur_de_ligne - Longueur_du_string)/2):"....."
```

L'algorithme ci-dessus suffit pour que l'affichage du string soit centré à l'écran : on commence par soustraire la longueur de la chaîne de caractères de la longueur totale d'une ligne d'écran. Il nous reste le nombre d'espaces non-occupés sur la ligne, que nous divisons par deux pour les répartir à gauche et à droite du string, qui est alors centré.

La procédure de saisie de nouvelles adresses sera plus difficile à écrire, car il va falloir recourir à un grand nombre de notions nouvelles. Procédons par ordre.

□ La saisie formatée de données

Vous connaissez déjà les instructions INPUT et LINE INPUT, et nous venons de voir INKEY\$ dans ce chapitre. Je vais maintenant vous présenter l'instruction utilisée le plus fréquemment par les professionnels : INPUT USING.

Vous avez bien lu : INPUT USING est une instruction de niveau professionnel, conçue avant tout pour les logiciels commercialisés auprès du grand public. Certes, son utilisation est un peu plus compliquée que celle de INPUT ou INKEY\$, mais il n'y a pas de quoi s'effrayer outre mesure : c'est en forgeant que l'on devient forgeron. D'ailleurs, vous avez maintenant acquis suffisamment de connaissances pour aborder cette instruction. Si j'écris :

```
INPUT @(Ligne,Colonne):<variable>
```


vous allez me répondre tout heureux que ça, vous connaissez déjà ! c'est la syntaxe de notre vieille connaissance, l'instruction INPUT. Effectivement, les deux instructions se ressemblent jusqu'ici, mais ce qui suit est assez nouveau :

USING <chaîne de commandes>

Arrêtons-nous un peu sur cette chaîne de commandes : elle nous permet de limiter la saisie à un certain nombre de caractères autorisés, l'ordinateur ignorant purement et simplement les caractères non autorisés par la <chaîne de commandes>. Par exemple, dans le champ réservé pour le numéro de téléphone, nous pouvons interdire la saisie de lettres ; inversement, nous pouvons interdire la saisie de chiffres dans la zone du nom propre ou du prénom. Voici les signes pouvant entrer dans la chaîne de commandes :

Signe	caractères autorisés
0	autorise tous les chiffres de 0 à 9
a	autorise toutes les lettres de l'alphabet
%	autorise les caractères spéciaux
^	autorise la combinaison <control>++caractère
+<caractère>	autorise le caractère signalé entre crochets
-<caractère>	exclut le caractère signalé entre crochets
c<caractère1><caractère2>	remplace le <caractère1> par le <caractère2>
u	toutes les lettres passent en majuscules
l	toutes les lettres passent en minuscules

Vous pouvez combiner ces 9 signes dans la chaîne de commandes :

Chaîne de commande	Formatage obtenu
"0"	autorise la saisie de tous les chiffres de 0 à 9
"0a"	autorise la saisie de tous les chiffres et de toutes les lettres
"0au"	autorise la saisie des chiffres et lettres, celles-ci étant passées en majuscules
"0a+.u"	comme ci-dessus, mais en ajoutant l'autorisation de saisir le point
"0a+.-au"	comme ci-dessus, mais en interdisant la saisie de 'a'
"a+b+c"	seules les lettres a,b,c peuvent être saisies
"a+b+c+"	comme ci-dessus, mais en ajoutant l'espace vide (space)
"a+b+c+++.c.-"	comme ci-dessus, en autorisant de plus les signes '-' et '.' mais l'appui sur la touche '.' fera apparaître un signe '-' puisque la chaîne se termine par 'c.-' (remplacement du . par le -).

Vous pouvez écrire la chaîne de commandes indifféremment en majuscules ou minuscules. Nous ressortons de INPUT USING comme de INPUT : par un <return>. Vous pouvez introduire dans la chaîne de commandes des conditions menant à l'interruption de la saisie lorsqu'elles sont vérifiées :

Signe	Sortir de la saisie si ...
x<ASCII>	l'utilisateur appuie sur la touche dont le code ASCII est indiqué
s<Scan>	l'utilisateur appuie sur la touche dont le code SCAN est indiqué
<	dépassement de la marge gauche
>	dépassement de la marge droite

Par exemple, si vous souhaitez que la saisie soit interrompue dès que l'utilisateur appuie sur la touche <E>, vous pouvez procéder de deux façons différentes :

- ① en utilisant le code ASCII de cette touche : 101 ; vous écrirez alors

```
"ax"+CHR$(101)
```

ou encore :

```
"axe"
```

La représentation du code ASCII par CHR\$() est toujours nécessaire lorsque vous indiquez comme caractère d'interruption de la saisie un signe qu'on ne peut saisir en tant que tel, comme par exemple la touche <escape> qui porte le code ASCII 27 :

```
"x"+CHR$(27)
```

- ② en utilisant le code SCAN de cette touche, ce qui nécessite sans doute une petite explication ; outre son code ASCII, chaque touche du clavier possède un code SCAN qui permet de la désigner individuellement. c'est important, surtout pour les touches n'ayant pas de code ASCII, comme par exemple les <touches fléchées> du curseur. Nous y reviendrons plus tard ; INKEY\$ fournit aussi, dans son deuxième octet, le Scancode de la touche appuyée.

Notez bien que les touches du pavé numérique ne portent pas les mêmes codes SCAN que leurs homologues du clavier alphanumérique : ceci permet de distinguer les deux jeux de touches numériques. Vous trouverez un tableau de codes Scan en annexe, à la fin de ce manuel.

Revenons à notre petit problème initial : la touche <E> porte le code SCAN \$12, ce qui donne 18 en notation décimale. Pour qu'un appui sur cette touche provoque la sortie de la saisie, nous écrivons :

```
"s"+CHR$(18)
```


Maintenant que nous savons comment construire la chaîne de commandes, nous pouvons revenir à notre fichier d'adresses.

❑ NOM

Pour saisir le nom propre, il nous faut en tout cas pouvoir entrer toutes les lettres ; nous autorisons l'utilisation du trait d'union pour les noms composés (Dupont-Durand) et de l'espace vide pour séparer les titres éventuels (!!!) ; nous convenons enfin de n'utiliser ici que des majuscules ; cela nous donne la chaîne de commandes suivante :

"a+ +-u"

que nous pouvons réutiliser pour le prénom, en supprimant toutefois la conversion systématique en majuscules :

"a+ -+"

❑ NUMERO et RUE

Il nous faut disposer des lettres, des chiffres, de l'espace vide et du point :

"0a+ +-+."

❑ CODE POSTAL

Il se compose en France de cinq chiffres (attention aux adresses étrangères !) une fois le code postal saisi, l'ordinateur ressort de l'instruction INPUT USING par le dépassement de la limite droite du champ '>'

"0>"

☐ VILLE et lieu-dit

Nous devons autoriser ici la saisie de lettres, éventuellement des chiffres (arrondissements des grandes villes), du trait d'union (Châlons-sur-Marne) et de la barre oblique (Châlons/Marne), ce qui nous donne :

"a0+/-"

☐ NUMERO de TELEPHONE

La saisie du numéro de téléphone réclame l'autorisation de saisir des chiffres ainsi que le trait d'union (indicatif) ; à titre d'exercice, demandons à l'ordinateur de transformer le trait d'union en barre oblique :

"0+--/c-/"

☐ DATE de NAISSANCE

Ce sera la dernière zone de notre fiche ; il nous suffit d'autoriser la saisie des chiffres et du point :

"0+."

L'instruction INPUT USING, suivie de sa chaîne de commandes, doit se terminer par une variable <return> conditionnant la sortie de la saisie et précédée d'une virgule. L'interpréteur consigne dans cette variable la touche qui a servi à sortir du mode saisie : elle contient la valeur 0 pour la sortie par <return>, -1 pour la sortie par la droite du champ et -2 pour la sortie par la gauche du champ.


```

***** Saisie d'une fiche *****

*****
*
*   Nom: ERNOUX                Prenom: Pierrette
*
*   No et rue: 214 avenue St Paul
*
*   Code postal: 75010 Ville: PARIS
*
*   Tel: 47425221             Date de naiss.: 12.05.1939
*
*****

[ ] Fiche suivante  [ ] Correction  [ ] Retour au menu principal

```

Figure 2.2 : Masque de saisie dans le fichier d'adresses

Comme avec INKEY\$, la variable contient de plus l'état de la touche <shift>, le code scan, la dernière position du curseur et la valeur ASCII de la touche appuyée ayant provoquée la sortie de la saisie.

Vous pouvez aussi préciser la longueur maximale admise dans chacun des champs de saisie ; en l'absence de toute autre indication, l'interpréteur admet une longueur par défaut de 255 caractères. C'est beaucoup trop pour notre fichier d'adresses, dans lequel nous limiterons la capacité des champs de la façon suivante :

Champ	Longueur maximale autorisée
NOM	15 caractères
Prénom	15 caractères
No et rue	31 caractères
Code postal	5 caractères
Ville	23 caractères
Téléphone	17 caractères
Date de naissance	10 caractères

Nous souhaitons de plus que la longueur du champ soit immédiatement visible par l'utilisateur grâce à un caractère de remplissage. Si vous tenez à recourir à un caractère différent du tiret bas de soulignement '_' vous devez indiquer le code ASCII du caractère de remplissage choisi en le faisant figurer, séparé par une virgule, derrière l'indication de la longueur maximale du champ. Dans notre programme, nous nous satisfaisons du tiret bas.

Dernier paramètre à préciser, la variable contenant la position du curseur ; faute de précision contraire, l'appel de INPUT USING suffit à placer le curseur automatiquement sur la première position d'entrée des données.

Voilà, nous avons fait le tour de INPUT USING, dont nous rappelons la syntaxe :

```
INPUT [@(x,y);]["texte";]<chaîne> USING  
[<chaîne de commandes>].[<Longueur maxi>].[<code du  
caractère de remplissage>].[<position du curseur>]
```

□ Les opérateurs logiques

L'Omikron Basic met à votre disposition tous les opérateurs logiques usuels pour les opérations booléennes (partie du domaine des mathématiques).

□ AND

Cet opérateur sert à vérifier que les deux parties de la condition sont vérifiées avant d'exécuter l'instruction résultante :

```
IF      X<3      AND      X>0      THEN  
      condition1      condition2
```

La condition introduite par IF ne sera déclarée 'vraie' que si les deux parties qui la composent sont 'vraies'. Il suffit qu'une de ces parties soit fausse pour que l'ensemble de la condition soit déclarée non-remplie. Nous avons déjà utilisé l'opérateur AND, dans la boucle de choix du menu. Mais cela n'épuise pas toutes les possibilités de l'opérateur AND : il remplit une autre fonction, la

comparaison AND, bit par bit, de deux nombres. Admettons que les deux variables A et B contiennent l'une le nombre 31 et l'autre 43, ce qui s'écrit en binaire :

A : %00011111

B : %00101011

La comparaison se fait ensuite bit par bit, selon la table de vérité suivante :

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

La comparaison ne fournit une valeur 1 (vraie) que lorsque les deux bits contiennent tous deux un 1, dans tous les autres cas, le résultat est 0. Appliqué à nos deux nombres 31 et 43, cela donne :

A : %00011111

B : %00101011

A AND B : %00001011

En notation décimale, cette comparaison logique entre 31 et 43 nous donne le nombre 11. A quoi peut nous servir cette comparaison AND ? A masquer certains bits d'un nombre, c'est-à-dire à les ramener à la valeur 0. Voici un exemple pour illustrer cela :

Nous savons qu'un long-integer (entier long) se compose de quatre octets ; admettons que seul l'un de ces octets nous intéresse, à savoir le deuxième, alors que les autres octets sont dépourvus de signification pour nous. Nous allons donc comparer cette variable avec une constante dont le deuxième octet ne contiendra que des bits de valeur 1. L'opérateur logique AND donnera aux bits du premier,

troisième et quatrième octets la valeur 0, afin de les masquer. Cette opération n'aura cependant pas modifié le contenu du deuxième octet. Exemple :

```
A AND 111111110000000000000000
```

ou en notation hexadécimale, plus lisible (n'est-ce pas ?) :

```
A AND $FF0000
```

Nous nous souviendrons de cette particularité de l'opérateur AND et nous en ferons usage pour vérifier la variable <return> avec INPUT USING.

□ OR

Voici le deuxième opérateur, le OU logique. Il suffit que l'une des deux parties de la condition soit exacte pour que l'ensemble de la condition soit vérifiée :

```
IF A > 2 OR B < 30 THEN ....
```

Avec l'opérateur AND, il fallait que les deux parties soient vraies pour que la condition soit vérifiée, alors qu'avec OR, il suffit que l'une des deux parties soit vraie. On peut aussi comparer bit par bit deux nombres en les soumettant à l'opérateur OR, ce qui sert à rendre positifs certains bits, selon la table de vérité :

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

Le bit résultant de cette comparaison contient toujours la valeur 1 si l'un des deux bits comparés contient 1. Reprenons notre exemple, et soumettons les nombres 31 et 43 à l'opérateur OR :

A : %00011111

B : %00101011

A OR B : %00111111

ce qui donne, en notation décimale, le nombre 63.

□ NOT

Troisième opérateur, le NON logique, qui se charge d'inverser le contenu des bits : l'affirmation vraie est niée (elle devient fausse) tandis que l'affirmation fausse devient vraie :

IF NOT (A=B) THEN

que l'on peut noter aussi :

IF A <> B THEN

La condition n'est vérifiée que si A n'est pas égal à B, donc si les variables A et B représentent bien des valeurs différentes. Il est possible de soumettre un nombre à cet opérateur :

A	NOT A
0	1
1	0

Si B représente la valeur 43, la comparaison

A = NOT B

donne à A la valeur

B = %00101011

NOT B = %11010100

ce qui correspond à 212 en notation décimale. C'est ce qu'on appelle aussi le 'complément à deux' d'un nombre. Le nombre 255 étant représentable par 8 bits au maximum, nous en soustrayons 43 pour obtenir aussi le résultat 212 :

255 - A = NOT A (complément à deux)

☐ XOR

Cet opérateur représente le OU exclusif : pour que l'ensemble de l'affirmation soit vraie, il faut absolument que l'une des deux parties soit fausse. Si les deux parties de la condition sont toutes deux vraies ou fausses, la condition n'est pas vérifiée. XOR peut aussi servir à une comparaison bit par bit, selon la table de vérité suivante :

A	B	A XOR B
0	0	0
1	0	1
0	1	1
1	1	0

Avec ces quatre opérateurs logiques, vous voilà équipé pour résoudre la plupart des problèmes rencontrés. Sachez cependant qu'il existe encore d'autres opérateurs logiques en Omikron Basic, sur lesquels je ne m'étendrai pas car il s'agit en fait de combinaisons des opérateurs que nous venons de voir. Pour en savoir plus, veuillez vous reporter au manuel fourni avec l'Omikron Basic.

☐ "Décalage des bits"

Les deux instructions de ce paragraphe sont relativement simples à expliquer. Je n'en ai pas l'usage pour l'instant, mais je tiens à les présenter pour que le panorama soit complet.

SHR

Que se passe-t-il lorsque je déplace d'une position vers la droite les bits composant le nombre 56 (%00111000)?

%00111000 ---décalage d'un bit vers la droite---> %00011100

La valeur s'est divisée par deux !

La fonction SHR (SHift Right) provoque le décalage des bits composant le nombre A de N positions vers la droite :

Résultat = A SHR N

ce qui équivaut à diviser n-fois ce nombre par deux (opération non signée). L'opération se déroule à une vitesse très supérieure à la division ordinaire, car l'ordinateur travaille alors dans son domaine de prédilection, le traitement bit par bit.

SHL

Alors que SHR sert à diviser un nombre par deux, SHL (SHift Left) sert à le multiplier par deux en décalant les bits qui le composent de N positions vers la gauche :

Résultat = A SHL N

qui équivaut à la multiplication (non signée)

Résultat = A * 2^N

□ L'affichage formaté à l'aide de PRINT

De même que INPUT USING vous permet de saisir les données de façon formatée, PRINT USING permet aussi de les restituer en formatant l'affichage ; syntaxe :

```
PRINT USING <"masque">.<chaîne à afficher(variable)>
```

Attention : Cette instruction ne s'utilise que pour les chiffres, et jamais avec des variables représentant des chaînes de caractères. PRINT USING est particulièrement adapté pour afficher proprement des tableaux de chiffres, alignés par exemple sur la droite, ou sur la virgule. Voyons le <masque> de plus près :

```
PRINT USING "###.##".2.8
```

donnera à l'affichage écran :

2.80

Dans la définition du masque, le dièse '#' sert à 'réserver' la place d'un chiffre. Le masque doit comprendre autant d'emplacements réservés qu'il y aura de rangs numériques dans les nombres saisis. Les nombres décimaux seront automatiquement rangés par rapport à la virgule (ou au point). Les positions non affichables à droite de la virgule sont remplacées par des 0.

Voici les signes utilisables pour définir le masque :

Signe	Signification
#	réserve de l'emplacement d'un chiffre
.	position du point décimal lorsque la virgule sert de séparateur décimal, le point peut servir à séparer les milliers
,	position de la virgule décimale lorsque le point sert de séparateur décimal, la virgule peut servir à séparer les milliers
'''	séparer les milliers par une virgule, supprimer la partie décimale
...	séparer les milliers par un point, supprimer la partie décimale
-	afficher à cet endroit le signe 'moins' en cas de nombre négatif
+	afficher le signe + à cet endroit
Attention : un '+' ou un '-' placé directement avant le premier '#', '','' ou ',' provoquera l'affichage du signe en question juste devant le premier chiffre du nombre	
*<caractère>	le caractère venant après l'étoile sert de caractère de remplissage dans la mesure où la place est vide ; en l'absence de cette indication, l'interpréteur représente les positions (#) non occupées par des espaces vides ; ne jamais utiliser le tiret bas '_' comme caractère de remplissage.

Signe	Signification
_<caractère>	le tiret bas sert à forcer l'affichage du caractère qui le suit même s'il s'agit d'un signe spécial chargé d'une signification (code de commande) dans le string de formatage
.^^^	affichage en notation scientifique.

Les autres caractères non-mentionnés ici sont affichés tels qu'ils se présentent. En écrivant

```
PRINT USING ".####.## FF",25875.57
```

vous verrez s'afficher :

```
25.875.57 FF
```

le point servant à séparer les milliers et la virgule la partie décimale.

Si vous écrivez :

```
PRINT USING "*0.####.## FF",25875.57
```

cela donnera à l'affichage :

```
000025.875. 57 FF
```

les positions du début du nombre n'étant pas toutes occupées, elles sont remplies par des zéros, conformément à *0 placé au début de la définition. Vous constatez que la définition du caractère de remplissage ainsi que du séparateur sert à réserver des emplacements. Le signe (+ ou -) prend également une place lorsque la définition ne prévoit de lui attribuer une autre position dans le string, et ce même si aucun signe ne s'affiche lorsque le nombre est positif.

Ainsi la définition suivante

```
PRINT USING "####.##".A1
```

ne permet que l'affichage de nombres à trois positions (< 1000) puisque le premier '#' réserve l'emplacement du signe + ou -. Il est possible de provoquer l'affichage du signe à un autre endroit du string.

PRINT USING permet aussi l'affichage des nombres en notation scientifique, avec un exposant. Lorsque le string de formatage ne comprend pas le signe exposant '^', l'Omikron Basic affiche en tout cas le nombre sous sa forme habituelle.

Le string de formatage est soumis à quelques autres limitations : vous ne pouvez pas utiliser plus de 30 emplacements (#), le tiret bas de soulignement (_) ne peut pas servir de caractère de remplissage et la longueur du string ne peut excéder 253 caractères. En cas d'erreur dans la formulation du string, vous recevez le message 'syntax error'.

□ USING

Vous pouvez aussi utiliser USING<string de formatage> sans l'instruction PRINT : le masque ainsi défini sera pris en compte à chaque appel ultérieur d'une instruction PRINT, LPRINT, PRINT# ainsi qu'après la fonction STR\$.

Ces quelques explications étant données, revenons à la création de notre fichier d'adresses. Il faut maintenant écrire la procédure permettant la saisie des données. Pour faciliter le travail de l'utilisateur, prévoyons la possibilité d'utiliser les touches de curseur pour passer d'un champ de saisie à l'autre. Là où nous avons défini les chaînes de commandes pour chacune des zones de saisie, ajoutons la possibilité de sortir de la saisie dans un champ par un simple appui sur la touche <curseur haut> ou <curseur bas>. Le tableau des codes ASCII ne nous est d'aucune utilité, puisque ces touches n'ont pas de code ASCII ; elles ont par contre un code SCAN :

Code SCAN	Touche
72	curseur haut
80	curseur bas

Nous savons que l'instruction servant à conditionner la sortie de la saisie après identification d'un code Scan doit commencer par 's', suivi du code correspondant, qui doit tenir en un octet. Nous allons recourir de nouveau à la fonction CHR\$(). Comme ce string pourra nous resservir pour tous les INPUT USING, nous en faisons une chaîne de caractères à part, que nous ajouterons par addition de string à chacune des chaînes de commandes :

```
BACK$="s"+CHR$(72)+"s"+CHR$(80)
```

Le simple appui sur les des deux touches curseur nous permet de ressortir de la saisie ; la variable return contiendra, entre autre, le code scan de la touche ayant servi à sortir du mode saisie. Reste à identifier la touche ayant servi à quitter INPUT.

En lisant de la gauche vers la droite, c'est le deuxième octet de la variable return qui nous intéresse. Il faut faire disparaître les autres pour pouvoir demander le code SCAN par une condition IF. Nous allons nous servir de l'opérateur AND, permettant de masquer (mettre à 0) les trois octets qui ne nous servent à rien. Rappelons le contenu de la variable :

1er octet contrôle de l'état des touches shift

2ème octet code SCAN de la touche

3ème octet position du curseur

4ème octet code ASCII

Nous allons ramener sur zéro le premier, troisième et quatrième octets en utilisant l'opérateur AND, alors que le deuxième octet sera confronté à huit bits positifs (ce qui correspond à la valeur 255 en notation décimale ou \$FF en hexadécimal)

	1er octet	2ème octet	3ème octet	4ème octet
AND	00	255	00	00
ou				
AND	\$00	\$FF	\$00	\$00

La routine elle-même sera exploitée du haut vers le bas. L'utilisateur n'aura qu'à appuyer sur <return> ou <curseur bas> pour passer automatiquement dans le champ suivant. A l'aide de la touche <curseur haut>, il doit pouvoir revenir au champ de saisie précédent. Le code SCAN de la touche <curseur haut> est 72 ou \$48 en notation hexadécimale. Nous préférons ici la notation hexadécimale, car elle nous donne accès directement au deuxième octet, alors qu'en notation décimale, il faut lire tout le nombre. Le contrôle de la touche <curseur haut> se fera en écrivant :

```
IF (<returnvariable> AND $FF) = $480000 THEN  
    ... revenir dans le champ précédent  
ENDIF
```

Nous plaçons un label (une étiquette) devant chaque INPUT USING de façon à ce que l'interpréteur puisse s'y brancher par une simple instruction GOTO. Il nous reste à mentionner un dernier petit problème.

Si l'utilisateur appuie sur la touche <curseur haut> lorsqu'il se trouve précisément dans le premier champ de saisie, il faudrait que le curseur saute jusqu'au dernier champ (date de naissance) ; inversement, si l'utilisateur appuie sur <curseur bas> lorsqu'il se trouve dans le dernier champ, il faudrait que le curseur aille se positionner au début du premier champ. Une fois ce détail réglé, nous pouvons considérer que nous en avons fini avec la procédure de saisie des données. Pour faire d'une pierre deux coups, nous lui passons une variable locale comme paramètre, variable déterminant sous quel indice du tableau les données seront classées. Lorsque l'utilisateur passera dans la routine de saisie à partir de l'option 'correction', ce paramètre indiquera le numéro de l'enregistrement à modifier.

Sous quel élément du tableau array sera rangé un nouvel ensemble de données ? Tout simplement, sous le plus proche qui soit libre, c'est-à-dire pas encore occupé par une adresse. Une boucle se charge de faire croître une variable numérique jusqu'à ce que l'interpréteur trouve un élément libre dans le tableau. Stop ! que se passe-t-il lorsque le tableau est plein et que l'interpréteur ne trouve plus un seul élément libre ?

L'ordinateur vous salue bien bas. Il convient donc d'éviter d'en arriver là ! et de n'autoriser la saisie de données qu'aussi longtemps que le tableau compte des éléments libres, et que sa taille globale ne dépasse pas le maximum convenu (résultant du dimensionnement et consigné dans la variable 'taille') :

```
<Compteur> = 1
WHILE <Array(<Compteur>)> <> "" AND <Compteur> < <Taille>
  <Compteur> = <Compteur> + 1
WEND
```

Avant l'entrée dans la boucle, nous installons une variable de comptage que nous appelons <Compteur> et à laquelle nous attribuons la valeur 1, ce qui correspond au deuxième élément du tableau. Nous évitons en effet de nous servir du premier élément arborant l'indice zéro : il servira ultérieurement à consigner les données à rechercher dans le tableau. La boucle WHILE sert à contrôler si un élément est déjà occupé ou non ; simultanément, on ne doit pas encore avoir atteint la fin du tableau (<Taille>). Lorsque ces deux conditions sont remplies, le compteur est incrémenté de 1 (il augmente d'une unité) et le programme vérifie l'élément suivant. Ce processus se répète jusqu'à ce que l'interpréteur ait trouvé un élément libre ou jusqu'à ce qu'il ait parcouru tout le tableau. Lorsqu'il ne trouve aucun élément libre, le programme réaffiche à l'écran le dernier élément - qui est certes déjà occupé, mais que faire d'autre ?

La deuxième option du menu offre la possibilité de corriger une fiche ; les deux erreurs les plus fréquentes sont généralement :

- une faute de frappe dactylographique qu'il faut rectifier
- une modification complète exigeant la suppression d'un enregistrement entier.

Le programme serait certes très confortable s'il permettait de rechercher un enregistrement pour le modifier, mais je me suis décidé en faveur d'une autre procédure de correction.

L'utilisateur voit s'afficher son premier enregistrement, après quoi il peut feuilleter l'ensemble du tableau. La touche <C> lui permet de procéder à une correction et donc de passer dans la routine de saisie.

La touche <E> lui permet d'effacer les données affichées à l'écran. Pour faciliter le choix de l'utilisateur, j'ai intégré deux facilités dans le programme :

- un sous-menu s'affiche dans la partie inférieure de l'écran, ce qui permet de mieux contrôler le déroulement du programme ;
- la première lettre de chaque option s'affiche en inversion-vidéo, ce qui rappelle à l'utilisateur qu'il lui suffit d'appuyer sur la touche correspondante pour lancer la fonction souhaitée ;
- pour que l'utilisateur puisse lancer la fonction en appuyant sur la bonne touche en majuscule ou en minuscule, le programme transformera 'character' (pour l'appeler par son nom) en majuscule grâce à UPPER\$().

La lecture des enregistrements d'un tableau ne devrait pas vous poser de problème particulier. Il suffit de veiller à ce que la limite inférieure (premier enregistrement portant l'indice 1) et la limite supérieure (enregistrement portant un indice égal à <Taille> ou portant le dernier indice occupé si le tableau dispose encore d'éléments libres) ne soient pas franchies.

Attardons-nous sur l'élimination d'un enregistrement dans le tableau. Deux situations peuvent théoriquement survenir :

- l'utilisateur souhaite effacer le dernier enregistrement du tableau ; c'est le cas le plus simple, il suffit de remplacer les éléments concernés par une chaîne d'espaces vides ;
- cas plus compliqué mais sans doute le plus fréquent, l'utilisateur souhaite éliminer un enregistrement se trouvant quelque part dans le tableau. Il faut éviter de créer des espaces vides en plein milieu du tableau des variables, car cela ne ferait pas de bien à notre fonction permettant de feuilleter le tableau ! il faut donc diminuer d'une unité l'indice de tous les éléments de tableau se trouvant après l'élément à éliminer. Le dernier enregistrement d'indice 'n'

existe alors deux fois, une fois sous 'n' et une fois sous 'n-1', et il faut éliminer l'une des deux mentions : évidemment, nous préférons éliminer celle qui porte l'indice 'n', car si nous éliminions l'enregistrement portant l'indice 'n-1' nous ferions apparaître une chaîne d'espaces vides avant la fin du tableau. Si tout cela vous semble un peu compliqué, reportez-vous à la figure 2.3, ci-après.

Lorsque l'élément à effacer se trouve dans une liste ne contenant qu'un seul élément, nous nous trouvons dans la variante numéro 1, car le premier élément est en même temps le dernier, et nous savons comment effacer le dernier élément d'une liste.

La dernière option du menu (pour l'instant) doit permettre à l'utilisateur de retrouver un mot dans une liste. Nous utilisons pour cela une instruction INPUT USING à part, qui permet de saisir le mot à rechercher, de lui conférer l'indice 0 dans le tableau et enfin de parcourir tout le tableau grâce à une boucle.

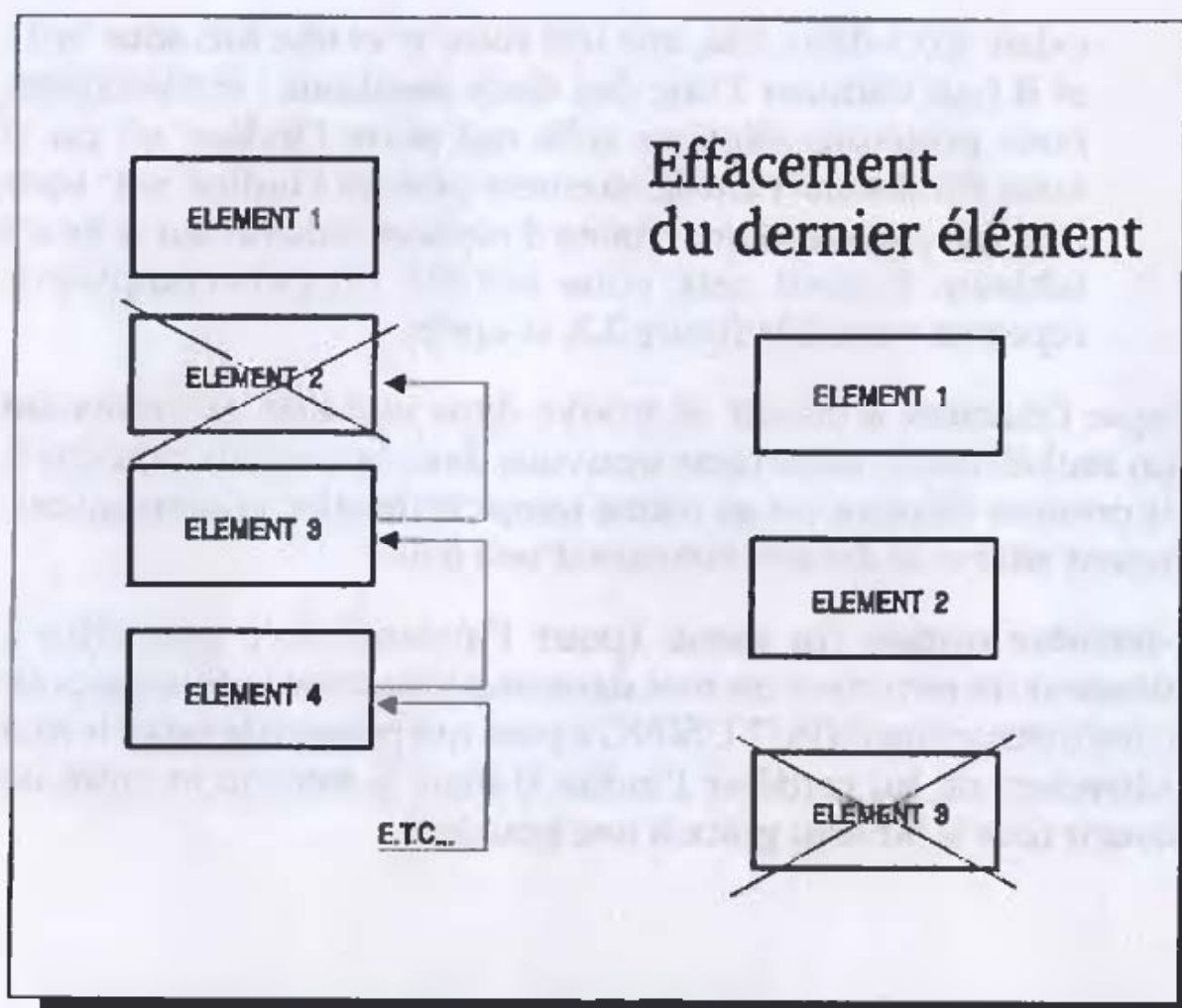


Figure 2.3 : Comment effacer un élément dans une liste

La condition d'interruption de la boucle sera :

`<mot recherché> = <contenu du tableau>`

ce que l'on peut aussi formuler par une condition négative :

```

<compteur> = 1
WHILE    <mot_recherché> <> <contenu_du_tableau>
        <compteur> = <compteur + 1>
WEND

```

☐ Un petit avant-goût du GEM

Vous les avez certainement déjà rencontrés : vous savez, ces petits panneaux ressemblant à ceux du code de la route, que l'Atari vous envoie soit pour vous signaler une erreur de manipulation soit pour vous demander prudemment de confirmer une manoeuvre. C'est ce

que l'informaticien appelle une 'Alert-Box' (message d'erreur ou demande de confirmation, souvent traduit textuellement par 'boîte d'alerte', nous utiliserons l'expression 'panneau d'avertissement'). Ces fonctions GEM sont directement implantées dans l'Omikron Basic, et vous pouvez donc les appeler directement.

□ FORM_ALERT

Cette instruction fait apparaître un panneau d'avertissement à l'écran; deux syntaxes possibles :

```
FORM_ALERT(<Default>, <Texte>)
ou
FORM_ALERT(<Default>, <Texte>, <Button>)
```

<Default> représente le numéro du 'Button' (1-3) activé par défaut après un simple appui sur la touche <return> : le 'button' est représenté par un cadre au trait épais. Lorsque vous donnez la valeur 0 à <Default>, l'appui sur <return> ne sélectionne aucun 'Button'. <Texte> représente une chaîne de caractères structurée de la façon suivante :

```
"[<Numéro>][<Ligne 1>|<Ligne 2>|<Ligne 3>]
[<Button 1>|<Button 2>|<Button 3>]"
```

<Numéro> indique le pictogramme, le symbole qui doit venir s'inscrire dans le panneau d'avertissement (Alert-Box) :

- 0 : pas de symbole
- 1 : Attention, point d'exclamation '!'
 - 1 : Attention, point d'exclamation '!'
 - 2 : point d'interrogation '?'
 - 3 : Stop

Notez que vous devez entrer les crochets carrés ; après le numéro viennent les lignes de texte qui s'inscriront dans le panneau d'avertissement. Vous pouvez entrer au maximum 5 lignes séparées par une barre verticale '|'. Rappelons que vous entrez la barre

verticale en appuyant sur <shift> et la touche portant le dièse '#', immédiatement à droite de la grosse touche <return>. Chaque ligne comprendra au maximum 40 caractères.

En dernier lieu, vous entrez le texte qui suivra chacun des 'button' offert au choix de l'utilisateur. Chacun de ces textes est aussi séparé du précédent par une barre verticale '|'. Un panneau d'alerte contiendra au maximum trois 'buttons' ; le button <numéro> pouvant aussi être sélectionné par un simple appui sur <return>, ce que l'on remarque car le trait de son cadre est plus épais. Si vous ajoutez encore <button>, cette variable contient le numéro du bouton qui sert à ressortir du panneau d'avertissement :

1.	premier bouton
2.	deuxième bouton
3.	troisième bouton

Lorsqu'il y a plusieurs possibilités pour ressortir du panneau, il est possible de mémoriser le bouton qui a servi à ressortir du panneau.

La fonction FORM_ALERT est un moyen très performant pour envoyer à l'écran des messages attirant l'attention de l'utilisateur sur une erreur de manipulation. Voici quelques exemples :

```
FORM_ALERT(1,[2][quelle est cette plaisanterie?][Annuler])
```

```
FORM_ALERT(1,[3][Ne recommencez jamais|une chose pareille !][Fin])
```

```
FORM_ALERT(2,[1][Souhaitez-vous vraiment|effacer cet  
enregistrement?][Oui | Non | Bof ]". exit)
```

on trouvera dans 'exit' le numéro du 'button' sélectionné pour ressortir de ce panneau d'avertissement.

Mais attention ! il y a encore quelque chose qui cloche. L'utilisateur ne peut pour l'instant sélectionner que le bouton activé par défaut par un simple appui sur <return>, puisque le malheureux n'a plus le curseur de la souris à sa disposition. Prenez soin de le faire apparaître avant d'appeler la fonction FORM_ALERT, en écrivant :

MOUSEON

pour le faire redisp paraître après l'exécution de la fonction, en écrivant

MOUSEOFF

tout est maintenant réglé, voilà une affaire qui tourne correctement.

Attention : L'Atari mémorise le nombre de fois qu'apparaît soit MOUSEON soit MOUSEOFF ; si vous écrivez deux fois MOUSEOFF, il faut que vous entriez deux fois MOUSEON pour faire réapparaître le curseur.

Nous examinerons dans le chapitre suivant les deux dernières options 'sauvegarder le fichier' et 'charger le fichier'. Dans le texte du programme ci-dessous, nous leur réservons deux sous-programmes vides, pour que le programme ne se plante pas si vous appelez une de ces deux options. Voilà maintenant le texte promis :

```

0  *****
1  *                               FICHIER.BAS                               *
2  *-----*
3  * Auteur: Michael Maier   Version 1.00   Date: 15.07.1990   *
4  *   Programme joint au Livre de l'Omikron Basic   *
5  *                               (c)MICROAPPLICATION   *
6  *-----*
7  *
8  *
9  MODE "F"
10 DEF FN Screen$(X$)= CHR$(27)+X$
11 *
12 Taille%L=100' modifiez si nécessaire
13 DIM Nom$(Taille%L),Prenom$(Taille%L),Rue$(Taille%L)
14 DIM Postal$(Taille%L),Ville$(Taille%L),Tel$(Taille%L),Nele$(Taille%L)
15 *
16 Erreur$="[3][Ceci est malheureusement]   IMPOSSIBLE!!![Désolé]"
17 *
18 REPEAT
19   CLS
20   PRINT @ (0.1):""*78

```

```

21 FOR Y%=1 TO 5: PRINT @ (Y%,1); "": @ (Y%,78); "": NEXT Y%
22 PRINT @ (6,1); "****78
23 PRINT @ (2,28); "FICHIER - Menu principal"
24 PRINT @ (3,28); "-----"
25 PRINT @ (4,16); "Un programme tiré du livre de l'Omikron Basic"
26 PRINT @ (9,28); "1. Saisie d'une adresse"
27 PRINT @ (11,28); "2. Correction d'une adresse"
28 PRINT @ (13,28); "3. Recherche d'une adresse"
29 PRINT @ (15,28); "4. Sauvegarde du fichier"
30 PRINT @ (17,28); "5. Chargement du fichier"
31 PRINT @ (19,28); "6. Sortir du programme"
32 PRINT @ (22,29); FN Screen$ ("p"); " Veuillez préciser votre choix"; FN
Screen$ ("q")
33 PRINT FN Screen$ ("f") ' pour désactiver le curseur
34 '
35 A%L=0
36 REPEAT
37   A$= INKEY$
38   IF A$ <> "" THEN
39     A%L= ASC ( RIGHT$ (A$,1)) - 48
40   ENDIF
41   UNTIL A%L > 0 AND A%L < 7
42   PRINT FN Screen$ ("e") ' pour réactiver le curseur
43   ON A%L GOSUB Saisir, Corriger, Rechercher, Sauvegarder, Charger
44 UNTIL A%L = 6 ' répéter la boucle jusqu'à ce que touche 6 appuyée
45 CLS
46 END
47 '
48 Saisir
49 CLS
50 Header$ = "***** Saisie d'une fiche *****"
51 ' d'abord rechercher le premier élément libre
52 T%=1
53 WHILE Nom$ (T%) <> "" AND T% < Taille%L
54   T%=T%+1
55 WEND
56 Masque (Header$)
57 REPEAT
58   Saisie (T%)
59   PRINT FN Screen$ ("f") ' le curseur ne ferait que nous gêner
60   PRINT @ (19,15); FN Screen$ ("p"); "F"; FN Screen$ ("q");
61   PRINT "fiche suivante "; FN Screen$ ("p"); "C"; FN Screen$ ("q");
62   PRINT "orrection "; FN Screen$ ("p"); "R"; FN Screen$ ("q");
63   PRINT "etour au menu principal"
64   A$="": INPUT " "; A$ USING "+f+r+cu>".Ret%L,1,32
65   PRINT FN Screen$ ("e") ' réactiver le curseur
66   IF A$="C" THEN
67     Masque (Header$)
68     Affichage (T%)
69   ELSE
70     IF A$="F" AND T% < Taille%L THEN
71       T%=T%+1
72       Masque (Header$)
73       Affichage (T%)
74     ENDIF
75   ENDIF
76 UNTIL A$="R"
77 RETURN
78 '

```



```

79-Corriger
80 Header$="***** Correction d'une fiche *****"
81 PRINT FN Screen$("f")
82 TX=1
83 Masque(Header$)
84 REPEAT
85   Affichage(TX)
86   PRINT @(19,4);FN Screen$("p");"F";FN Screen$("q");"fiche suivante ";
87   PRINT FN Screen$("p");"D";FN Screen$("q");"derniere fiche ";
88   PRINT FN Screen$("p");"C";FN Screen$("q");"correction ";
89   PRINT FN Screen$("p");"E";FN Screen$("q");"effacer ";
90   PRINT FN Screen$("p");"R";FN Screen$("q");"retour au menu principal"
91   '
92   AS=""
93   REPEAT
94     AS= INKEY$
95     IF AS<>"" THEN
96       AS= UPPER$( RIGHT$(AS,1))
97     ENDIF
98     UNTIL AS="F" OR AS="D" OR AS="C" OR AS="E" OR AS="R"
99     '
100    IF (AS="F") AND (TX<Taille%L) AND (Nom$(TX+1)<> "") THEN
101      TX=TX+1
102    ELSE
103      IF AS="F" THEN
104        FORM_ALERT (1,Erreurs$)
105      ENDIF
106    ENDIF
107    IF AS="D" AND TX>1 THEN
108      TX=TX-1
109    ELSE
110      IF AS="D" THEN
111        FORM_ALERT (1,Erreurs$)
112      ENDIF
113    ENDIF
114    IF AS="C" THEN
115      PRINT FN Screen$("e")
116      PRINT @(19,1);" *78
117      Saisie(TX)
118      PRINT FN Screen$("f")
119    ENDIF
120    IF AS="E" THEN
121      MOUSEON
122      FORM_ALERT (2,"[2][faut-il vraiment effacer?][oui | non]".But%)
123      MOUSEOFF
124      IF But%-1 THEN
125        Delete(TX)
126      ENDIF
127    ENDIF
128  UNTIL AS="R"
129  RETURN
130  '
131-Rechercher
132 Header$="***** Recherche d'une fiche *****"
133 Nom$(0)="" : Prenom$(0)="" : Rue$(0)="" : Postal$(0)=""
134 Ville$(0)="" : Tel$(0)="" : Nele$(0)=""
135 TX=0
136 Masque(Header$)
137 PRINT FN Screen$("e")

```

```

138 INPUT @(7,20):Nom$(0) USING "a+ +-u",Ret%L,15
139 PRINT FN Screen$("f")
140 TX=1
141 REPEAT
142   WHILE Nom$(TX)<>Nom$(0) AND TX<Taille%L
143     TX=TX+1
144   WEND
145   IF Nom$(TX)=Nom$(0) THEN
146     Affichage(TX)
147   ELSE
148     FORM_ALERT (1,"[1][Fiche non trouvée][Que faire?]")
149   ENDIF
150   PRINT @(19,15):FN Screen$("p");"C":FN Screen$("q");
151   PRINT "ontinuer ";FN Screen$("p");"N":FN Screen$("q");
152   PRINT "ouvelle recherche ";FN Screen$("p");"R":FN Screen$("q");
153   PRINT "etour au menu principal"
154   AS=""
155   REPEAT
156     AS= INKEY$
157     IF AS<>"" THEN
158       AS= UPERS( RIGHT$(AS,1))
159     ENDIF
160     UNTIL AS="C" OR AS="N" OR AS="R"
161     IF AS="C" AND TX<Taille%L THEN
162       TX=TX+1
163     ENDIF
164     IF AS="N" THEN
165       EXIT TO Rechercher
166     ENDIF
167   UNTIL AS="R"
168   RETURN
169 *
170 -Sauvegarder
171 * nous écrirons cette routine plus tard
172 RETURN
173 *
174 -Charger
175 * nous écrirons cette routine plus tard
176 RETURN
177 *
178 *
179 DEF PROC Masque(Texte$)
180   LOCAL TX
181   CLS
182   PRINT @(2,(78- LEN(Texte$))/2):Texte$
183   PRINT @(5,10);"***60
184   FOR TX=1 TO 9: PRINT @(5+TX,10);"***";@(5+TX,69);"***": NEXT TX
185   PRINT @(15,10);"***60
186   PRINT @(7,15);"Nom: _____ Prénom: _____"
187   PRINT @(9,15);"No et rue: _____"
188   PRINT @(11,15);"Code postal: _____ Ville: _____"
189   PRINT @(13,15);"Tel: _____ Date de naiss.: _____"
190 RETURN
191 *
192 DEF PROC Affichage(Numero%)
193   PRINT @(7,20):Nom$(Numero%);
194   PRINT STRING$(15- LEN(Nom$(Numero%)),"_")
195   PRINT @(7,52):Prenom$(Numero%);
196   PRINT STRING$(15- LEN(Prenom$(Numero%)),"_")

```



```

197 PRINT @(9,26);Rue$(Numero%);
198 PRINT STRING$(31- LEN(Rue$(Numero%)),"_")
199 PRINT @(11,28);Postal$(Numero%);
200 PRINT STRING$(5- LEN(Postal$(Numero%)),"_")
201 PRINT @(11,41);Ville$(Numero%);
202 PRINT STRING$(23- LEN(Ville$(Numero%)),"_")
203 PRINT @(13,20);Tel$(Numero%);
204 PRINT STRING$(17- LEN(Tel$(Numero%)),"_")
205 PRINT @(13,54);Nele$(Numero%);
206 PRINT STRING$(10- LEN(Nele$(Numero%)),"_")
207 RETURN
208 '
209 DEF PROC Saisie(Numerp%)
210 LOCAL Back$="s"+ CHR$(72)+"s"+ CHR$(80)
211 -Nom
212 INPUT @(7,20);Nom$(Numero%) USING "a+ +-u"+Back$.Ret%L,15
213 IF (Ret%L AND $FF0000)=$480000 THEN
214 GOTO Nele
215 ENDIF
216 -Prenom
217 INPUT @(7,52);Prenom$(Numero%) USING "a+ +-" +Back$.Ret%L,15
218 IF (Ret%L AND $FF0000)=$480000 THEN
219 GOTO Nom
220 ENDIF
221 -Street
222 INPUT @(9,26);Rue$(Numero%) USING "0a+ +-+." +Back$.Ret%L,31
223 IF (Ret%L AND $FF0000)=$480000 THEN
224 GOTO Prenom
225 ENDIF
226 -Postal
227 INPUT @(11,28);Postal$(Numero%) USING "0>"+Back$.Ret%L,5
228 IF (Ret%L AND $FF0000)=$480000 THEN
229 GOTO Street
230 ENDIF
231 -Ville
232 INPUT @(11,41);Ville$(Numero%) USING "a0+/-" +Back$.Ret%L,23
233 IF (Ret%L AND $FF0000)=$480000 THEN
234 GOTO Postal
235 ENDIF
236 -Telephone
237 INPUT @(13,20);Tel$(Numero%) USING "0+--+/c-/" +Back$.Ret%L,17
238 IF (Ret%L AND $FF0000)=$480000 THEN
239 GOTO Ville
240 ENDIF
241 -Nele
242 INPUT @(13,54);Nele$(Numero%) USING "0+." +Back$.Ret%L,10
243 IF (Ret%L AND $FF0000)=$480000 THEN
244 GOTO Telephone
245 ELSE
246 IF (Ret%L AND $FF0000)=$500000 THEN
247 GOTO Nom
248 ELSE
249 ENDIF
250 ENDIF
251 RETURN
252 '
253 DEF PROC Delete(Numero%)
254 LOCAL Quantite%-1
255 'rechercher le nombre de données existantes

```

```
256 WHILE (Nom$(Quantite%+1))<>"" AND Quantite%<Taille%L
257   Quantite%=Quantite%+1
258 WEND
259 IF Numero%=Quantite% THEN
260   Efface(Numero%)
261 ELSE
262   WHILE (Numero%<>Quantite%)
263     Nom$(Numero%)=Nom$(Numero%+1)
264     Prenom$(Numero%)=Prenom$(Numero%+1)
265     Rue$(Numero%)=Rue$(Numero%+1)
266     Villes$(Numero%)=Villes$(Numero%+1)
267     Postal$(Numero%)=Postal$(Numero%+1)
268     Tel$(Numero%)=Tel$(Numero%+1)
269     Neles$(Numero%)=Neles$(Numero%+1)
270     Numero%=Numero%+1
271   WEND
272   Efface(Numero%)
273 ENDIF
274 RETURN
275 '
276 DEF PROC Efface(Numero%)
277   Nom$(Numero%)="":Prenom$(Numero%)="":Rue$(Numero%)=""
278   Postal$(Numero%)="":Villes$(Numero%)="":Tel$(Numero%)=""
279   Neles$(Numero%)=""
280 RETURN
```


Chapitre 3

Gestion des fichiers

S'il est certes important de pouvoir saisir des données sur son ordinateur, il est tout aussi important de pouvoir les retrouver ultérieurement. Vous connaissez déjà les instructions READ et DATA qui permettent d'alimenter un programme en données, mais elles ne sont pas appropriées à la gestion des adresses de fichiers.

Il est plus judicieux de sauvegarder les données sur un disque dur ou une disquette, dans un ou plusieurs 'fichier(s)', en anglais : 'file'. L'utilisateur pourra ensuite recharger ces données pour les relire, les modifier et les sauvegarder à nouveau. Il existe deux façons de sauvegarder les données :

- dans des fichiers séquentiels
- dans des fichiers à accès direct

et il existe une troisième formule, qui réunit en quelque sorte les deux méthodes ci-dessus :

- les fichiers séquentiels indexés (ISAM).

3.1. Les fichiers sur disquette

Nous avons déjà parlé des fichiers séquentiels, mais je ne vous ai pas encore expliqué ce que cela représente exactement.

☐ Les fichiers séquentiels

Dans un fichier séquentiel, les données sont tout simplement enregistrées les unes à la suite des autres. Pour que l'ordinateur puisse par la suite distinguer entre eux les divers éléments, ceux-ci sont séparés par un 'carriage return' (retour chariot) représenté par le code ASCII 13. Les divers éléments peuvent être de longueurs différentes ; il est même possible d'enregistrer dans un tel fichier des variables de types différents dans n'importe quel ordre.

Ce procédé d'enregistrement des données dans un fichier a aussi des inconvénients. Comme les éléments peuvent être de longueurs différentes, il faut constamment recharger tout le fichier dans la mémoire vive de l'ordinateur pour pouvoir le relire. Il faut de plus bien faire attention au genre de données qu'on y a mis, faute de quoi on pourrait se heurter à de sérieux problèmes : imaginez qu'un fichier contienne 50 adresses, et que le prochain enregistrement commence par la date de saisie de la nouvelle fiche.

Notons enfin qu'on ne peut pas modifier un élément particulier sans recharger tout le fichier, pour le sauvegarder ensuite de nouveau entièrement.

☐ Les fichiers à accès direct (Random-access-files)

Dans ce type de fichier (appelés aussi 'fichiers relatifs'), la sauvegarde des données se fait en distinguant entre les divers enregistrements (records) contenus dans le fichier. Un enregistrement peut à son tour se composer de plusieurs éléments (dans notre exemple de fichier d'adresses : le nom, le prénom, la rue, la ville etc.), mais tout enregistrement respectera une longueur fixée à l'avance. Le gros

avantage étant alors que l'on peut accéder directement à un enregistrement particulier dans le fichier, ce qui était impossible avec le fichier séquentiel.

Mais cette souplesse se paie par un inconvénient : chaque enregistrement occupe une longueur donnée qui est fixe, indépendamment de la longueur réellement occupée par les données saisies. Le fichier à accès direct prend donc plus de place que le fichier séquentiel. Qui plus est, il faut définir ce genre de fichier avant d'y enregistrer quoi que ce soit : tous les enregistrements sont ainsi 'créés' sur la disquette, alors qu'ils ne contiennent encore que des espaces vides (spaces).

☛ **Attention :** La création d'un fichier à accès direct exige une réflexion préalable approfondie, une programmation plus complexe, et consomme plus de place sur une disquette. Par contre, l'ordinateur accède directement à un enregistrement, sans avoir à recharger tout le fichier.

□ Les fichiers ISAM

Les fichiers séquentiels indexés combinent astucieusement les avantages des deux autres procédés selon un principe de base que je vais vous expliquer.

Nous avons vu qu'il faut recharger dans la mémoire vive de l'ordinateur tout le fichier séquentiel avant de pouvoir le lire, alors que le fichier indexé nous permet d'accéder à tel ou tel enregistrement particulier. Que se passe-t-il lorsque nous souhaitons retrouver une donnée dans l'un de ces deux types de fichiers ?

La recherche se fait très rapidement dans le fichier séquentiel, puisqu'il est entièrement rechargé dans la mémoire vive de l'ordinateur. Elle est beaucoup plus longue dans le fichier à accès direct, puisque l'ordinateur lit chaque enregistrement jusqu'à ce qu'il ait retrouvé la séquence recherchée ; il lui faut de plus passer par la disquette, ce qui allonge encore le délai de recherche.

Naturellement, on pourrait aussi recharger tout le fichier à accès direct dans la mémoire vive, mais où serait alors le bénéfice ? autant faire un fichier séquentiel, non ?

On en est venu à une petite astuce : le fichier est sauvegardé sous la forme permettant l'accès direct, mais les concepts qui feront par la suite l'objet de recherches dans le fichier sont, eux, sauvegardés dans un fichier séquentiel, que l'on appelle un fichier d'index (ou fichier index). Tout enregistrement présent dans ce fichier index se compose de deux parties :

<mot clé> + <numéro d'enregistrement>

Lors du lancement du programme, le fichier d'index est intégralement chargé dans la mémoire vive. Lorsque l'utilisateur recherche un mot clé déterminé, il lance sa recherche qui se limite à parcourir dans un premier temps le fichier index. Une fois le concept retrouvé, l'ordinateur recharge l'enregistrement correspondant, qu'il identifie grâce à son numéro d'enregistrement. Ceci évite de perdre du temps à attendre que l'ordinateur lise chaque enregistrement en entier jusqu'à ce qu'il retrouve la bonne fiche.

Lorsque l'utilisateur ajoute de nouvelles fiches dans son fichier, il faut naturellement augmenter le fichier d'index, qui reçoit de nouveaux mots-clés et de nouveaux numéros d'enregistrement, lesquels devront permettre par la suite de retrouver aussi les nouveaux enregistrements ajoutés. Inversement, lorsque l'utilisateur détruit une fiche dans le fichier, le programme doit mettre à jour la liste de mots-clés du fichier index ainsi que les numéros d'enregistrement.

Il est enfin possible de créer plusieurs index pour un même fichier à accès direct (un index des noms, un autre des dates de naissance, des villes etc.) : ceci permet de rechercher des données à partir de critères différents.

3.2. On ne peut rien faire sans les canaux

Avant même de sauvegarder le moindre octet sur une disquette ou sur un disque dur, il vous faut ouvrir un 'canal'. Le canal est le chemin que suivront les données lors de leur(s) aller(s)-retour(s) entre l'ordinateur (unité centrale) et le périphérique de stockage (unité de disquette ou disque dur) que vous possédez. Le canal est tout à fait comparable à une ligne téléphonique qui ferait la jonction entre l'unité centrale et l'unité de disquette, la liaison étant ouverte grâce à l'instruction

OPEN

en respectant la syntaxe suivante :

OPEN <"type de fichier">,<numéro du canal>,<"Nom_du_fichier">

Le premier paramètre suivant OPEN indique le type de fichier concerné ; vous avez le choix entre les possibilités suivantes :

Type de fichier	Explication
"I" (Input)	lire un fichier séquentiel
"O" (Output)	ouvrir un fichier séquentiel
"A" (Append)	ajouter à la fin d'un fichier séquentiel
"F" (Files)	répertoire (directory)
"P" (Printer)	vers l'imprimante
"R"	fichier à accès direct (fichier relatif)
"C"	vers la console (écran et clavier)
"K" (Keyboard)	instruction transmise au processeur du clavier
"M" (Midi)	vers le port MIDI
"V"	vers l'interface RS-232

N'ayez crainte, les trois premiers types suffisent amplement pour remplir les objectifs que nous nous sommes fixés, nous verrons les autres plus tard.

Pour ouvrir un canal dans le but d'inscrire des données dans un fichier séquentiel, il faut utiliser "O" comme paramètre indiquant le type de fichier. Le canal ainsi ouvert ne peut servir qu'à écrire des données dans un fichier séquentiel, il ne peut absolument pas servir à reprendre les données pour les relire.

Dès que vous ouvrez un canal, vous lui attribuez une tâche spécifique : lecture des données, enregistrement des données dans le fichier, sortie de données vers le port MIDI etc. Comme il est possible d'ouvrir plusieurs canaux dans un même programme, il faut un critère permettant de les distinguer : c'est le rôle du deuxième paramètre accompagnant OPEN, dans lequel on précise le numéro du canal. Ce numéro doit être compris entre 1 et 16, il n'est pas nécessaire que les numéros se suivent, mais il faut éviter d'attribuer deux fois le même !

En écrivant par exemple :

```
OPEN "O",1
```

vous ouvrez un canal portant le numéro 1, dont le rôle consistera à acheminer les données sous forme séquentielle vers un fichier. Vous pouvez en même temps ouvrir un canal

```
OPEN "I",2
```

qui rendra possible la relecture des données contenues dans un fichier. Dans le cours de votre programme, vous utiliserez le canal 1 pour sauvegarder vos données en les acheminant vers un fichier et vous utiliserez le canal 2 pour relire des données du fichier.

Le troisième paramètre est le nom du fichier : il est indispensable pour pouvoir retrouver ultérieurement les données archivées dans un fichier. Rappelons que le nom du fichier doit comporter au maximum huit caractères, qui peuvent être suivis d'une extension de trois lettres ; le nom et l'extension sont séparés par un point.

Attention : Ici, l'ordinateur n'attribue pas automatiquement une extension, contrairement à ce qui se passe avec LOAD, SAVE et NEW ; vous devez préciser cette extension à la suite du nom du fichier. Résumons-nous :

OPEN sert à ouvrir un canal d'acheminement des données ; cette instruction exige d'être accompagnée de trois paramètres :

- | | |
|--------------------------------|---|
| <type de fichier> | pour préciser la tâche assignée à ce canal (lecture, écriture etc.) |
| <numéro du canal> | pour identifier et utiliser le canal ; ce numéro est un nombre compris entre 1 et 16 |
| <nom du fichier> | pour indiquer le nom du fichier qui contiendra les données en question sur la disquette ; ce nom se compose de huit caractères, suivis éventuellement d'une extension de trois lettres. |

3.3. Encore un PRINT, mais aussi WRITE

Nous venons certes d'ouvrir un canal, mais nous ne savons pas encore comment y expédier nos données. Nous savons que l'instruction PRINT répercute à l'écran les informations qui lui sont confiées, mais tel n'est pas notre objectif. Nous allons nous servir d'un cousin de PRINT qui répond à notre attente :

PRINT#<numéro du canal>.<données>

L'instruction PRINT# fonctionne exactement comme PRINT, à la différence qu'elle ne répercute pas automatiquement les données en les affichant à l'écran (ce qu'elle pourrait d'ailleurs fort bien faire) mais en les envoyant vers le canal dont le numéro est indiqué juste après. Après avoir ouvert un fichier, PRINT# nous sert à y envoyer des données ; il ne faut jamais oublier d'indiquer le numéro du canal :

```
OPEN "0",1,"EXEMPLE.DAT"  
PRINT#1, "n'importe quoi"
```


qui équivaut exactement à :

```
OPEN "0".5,"EXEMPLE.DAT"  
PRINT#5. "n'importe quoi"
```

Quel que soit le numéro attribué au canal ouvert par OPEN, vous devez ensuite le reprendre pour expédier vos données. Si vous tentez de faire transiter vos données par un canal non-ouvert, l'ordinateur vous renvoie un message d'erreur.

WRITE#

L'instruction WRITE# requiert la même syntaxe que PRINT#, et sert aussi à expédier les données vers un canal quelconque. Contrairement à ce qui se fait avec PRINT#, les données sont ici entrées entre guillemets, ce qui nous apporte un avantage appréciable. Nous avons vu qu'avec INPUT, la virgule sert de séparateur de champs, ce qui signifie que l'instruction INPUT prend fin dès que survient une virgule (voir INPUT A,B !). Si vous cherchez à lire une chaîne de caractères contenant des virgules, l'instruction INPUT n'ira donc que jusqu'à la première virgule. Il suffit par contre d'entrer la même chaîne de caractères en la plaçant entre des guillemets pour que la virgule ne soit plus considérée comme un séparateur : INPUT pourra donc reprendre toute la chaîne de caractères, même si elle contient des virgules.

Lorsque des données sont séparées par une virgule, WRITE# transmet aussi la virgule sur le canal indiqué (ce qu'INPUT ne fait pas) : ceci vous permet de sauvegarder plusieurs variables derrière un seul WRITE#, une instruction INPUT pouvant ultérieurement les reprendre sans problème, puisque la virgule vaudra alors comme séparateur. En écrivant par exemple :

```
OPEN "0". 1. "NOM.BAS"  
A=14:B=20:C=77:D=14  
WRITE #1. A,B,C,D  
...  
...
```

nous aurons dans le fichier NOM.BAS

```
14.20.77.14
```


fichier dans lequel une instruction INPUT pourra les reprendre.

Fort bien, voilà que nous savons maintenant sauvegarder des données dans un fichier. Mais avant de ressortir du programme, il nous faut refermer ce fichier, faute de quoi les données seraient perdues. Pour refermer des fichiers, nous utilisons l'instruction

```
CLOSE <numéro de fichier>,<numéro de fichier>.....
```

ce qui referme les fichiers portant les numéros indiqués. Lorsque CLOSE n'est suivi d'aucun numéro, l'Omikron Basic referme automatiquement tous les fichiers ouverts à cet instant.

3.4. Comment lire les fichiers séquentiels

Voilà nos fichiers bien à l'abri sur notre disquette, mais nous devons certainement un jour les relire : pour ce faire, nous utiliserons un proche parent de l'instruction INPUT :

```
INPUT #<numéro du canal>,<(liste de) variable(s)>
```

Cette instruction fonctionne exactement comme INPUT, la seule différence étant qu'elle prend les données non pas en provenance du clavier mais en provenance du canal indiqué dans <numéro du canal>. L'interpréteur met fin à la lecture dès qu'il rencontre un carriage return (retour charriot CHR\$(13)) ou une virgule. En écrivant par exemple :

```
OPEN "I", 1, "NOM.DAT"  
INPUT #1,A,B,C,D  
CLOSE 1
```

vous provoquez le rechargement dans la mémoire vive de l'ordinateur d'un fichier précédemment sauvegardé sur la disquette à l'aide de WRITE#. Sachez qu'il existe de plus une instruction LINE INPUT# qui vous permet de recharger des textes contenant des virgules car elle ne met fin au rechargement des données qu'en présence d'un carriage return.

Jusqu'ici, tout semble très simple. La situation est un peu plus compliquée lorsqu'on ne sait plus combien de données on a inscrit dans un fichier. En effet, si l'on souhaite écrire une boucle à l'aide de INPUT# pour recharger des données depuis la disquette jusqu'à ce que le fichier soit complètement rechargé dans la mémoire vive de l'Atari ST (c'est-à-dire jusqu'à ce que INPUT# retourne une chaîne vide ("") lorsqu'il ne trouve plus de données) l'interpréteur se manifestera dès qu'il dépassera la fin du fichier.

Cette méthode n'est pas appropriée pour recharger un fichier dont on ne sait plus combien d'éléments il contient. Comme il est impossible de savoir combien un fichier contient d'enregistrements (il faudrait faire passer les données dans une boucle FOR...NEXT) l'Omikron Basic possède une fonction spéciale qui permet de repérer la fin d'un fichier :

EOF(<numéro du fichier>)

La fonction EOF() (abréviation de End Of File = fin de fichier) retourne une valeur 'faux' tant que l'on n'a pas atteint la fin du fichier désigné par son <numéro de fichier> et la valeur 'vrai' quand elle l'atteint.

Cette fonction ne peut pas encore nous servir à écrire une boucle : en effet, elle retourne constamment la valeur 'faux' (0) ce qui mène à une interruption de la boucle alors qu'il faudrait précisément la réexécuter pour atteindre la fin du fichier. Il nous faut donc prendre la précaution d'inverser la valeur retournée par cette fonction avant de la prendre comme critère d'interruption de la boucle. Pour cela, nous utilisons la fonction NOT :

```
OPEN "I",1,"FICHER.NAM"  
WHILE NOT EOF(1) 'répéter jusqu'à la fin du fichier  
  INPUT #1, Nom$  
WEND  
CLOSE 1
```

Ce que nous venons de dire vaut surtout pour les fichiers séquentiels, le procédé est un peu différent lorsqu'il s'agit de gérer des fichiers relatifs, mais nous y reviendrons plus loin.

Après ces quelques explications théoriques, venons-en à la pratique : notre petit programme de gestion d'un fichier d'adresses n'est pas encore au point ! nous savons qu'il y manque les deux routines de chargement et sauvegarde des adresses. Voyons d'abord la sauvegarde des données : une boucle va permettre au programme de parcourir toute la liste des enregistrements, du premier au dernier. Ce faisant, il enregistre les données dans le fichier séquentiel grâce à l'instruction PRINT# (que nous pouvons utiliser de préférence à WRITE#, puisque nous n'avons pas inscrit de virgule dans les fiches). Voyons ensuite le chargement : il faut que les données viennent se replacer dans le champ adéquat (logiquement, dans l'ordre où elles ont été sauvegardées dans le fichier) jusqu'à ce que le programme identifie le critère d'interruption de cette boucle : la fin du fichier. Après quoi la routine doit revenir au menu.

Sauvegarde :

```
OPEN "0",1,"FICHIER.DAT"  
T%=1  
WHILE Nom$(T%)<>""  
PRINT #1,Nom$(T%)  
PRINT #1,Prenom$(T%)  
PRINT #1,Rue$(T%)  
PRINT #1,Postal$(T%)  
PRINT #1,Ville$(T%)  
PRINT #1,Tel$(T%)  
PRINT #1,Nele$(T%)  
T%=T%+1  
WEND  
CLOSE 1  
RETURN  
.
```

Chargement :

```
OPEN "I".5."FICHER.DAT"  
T%-1  
WHILE NOT EOF(5)  
  INPUT #5,Nom$(T%)  
  INPUT #5,Prenom$(T%)  
  INPUT #5,Rue$(T%)  
  INPUT #5,Postal$(T%)  
  INPUT #5,Ville$(T%)  
  INPUT #5,Tel$(T%)  
  INPUT #5,Nele$(T%)  
  T%-T%+1  
WEND  
CLOSE 5  
RETURN  
.
```

3.5. Comment recopier des fichiers

Il arrive souvent que l'on souhaite recopier tel ou tel fichier, que ce soit pour des raisons de sécurité ou pour recharger un fichier dans un disque RAM.

Vous disposez pour ce faire, en Omikron Basic, de l'instruction COPY dont voici la syntaxe :

```
COPY <fichier source> TO <fichier cible>
```

Nous allons tout de même écrire une petite procédure supplémentaire. Pourquoi ? Pour que je puisse vous présenter deux autres instructions.

L'algorithme utilisé est très simple ; nous admettons que le fichier source et le fichier cible sont tous deux des fichiers séquentiels. Le programme doit lire les données dans le fichier source et les recopier octet par octet dans le fichier cible, jusqu'à ce qu'il atteigne la fin du fichier source. La procédure se termine par un retour dans le programme principal.

INPUT# ne vous permet absolument pas de lire quelqu'octet que ce soit dans un fichier : seul un CR (carriage return) ou une virgule peuvent lui signaler d'arrêter d'entrer des données : que se passera-t-il si le fichier ne contient ni virgule ni retour-charriot ? L'interpréteur finira par vous planter là, puisqu'il refuse une chaîne de caractères supérieure à 32000 caractères ; et si votre fichier contient un nombre supérieur de caractères,...

L'instruction INPUT# ne nous est donc ici d'aucun secours, il nous en faut une autre que voici :

```
INPUT$( <nombre de caractères> , <numéro du fichier> )
```

Cette instruction va recharger exactement la quantité de caractères indiquée sous <nombre de caractères> à partir du fichier accessible par le canal <numéro du fichier>. Ainsi par exemple, si vous entrez la valeur 1 comme <nombre de caractères>, vous rechargerez exactement un octet du fichier concerné.

Voici la procédure indiquée dans le manuel de l'Omikron Basic (page 132) pour afficher le contenu complet du fichier 'Nom\$' à l'écran :

```
1000 DEF PROC Dump_Fich(Nom$)
1010 OPEN "I",1,Nom$
1020 WHILE NOT EOF(1)
1030   PRINT INPUT$(1,1):
1040 WEND
1050 CLOSE 1
1060 RETURN
```

On ouvre (ligne 1010) en lecture un fichier séquentiel, numéro de canal 1. Ligne 1030 : l'ordinateur lit un octet (INPUT\$) et l'affiche immédiatement (PRINT) à l'écran. La boucle fait que l'affichage se poursuit, octet par octet, jusqu'à la fin du fichier, après quoi le canal est refermé et on sort de cette procédure. Nous allons construire de façon similaire notre procédure de recopie d'un fichier : il suffit en effet que la sortie des données ne soit pas dirigée vers l'écran mais directement vers un autre fichier ; ce dernier doit être ouvert et porter un autre numéro pour que l'on puisse y enregistrer (séquentiellement) les données :


```
DEF PROC Filecopy(Depuis$,Vers$)
OPEN "I",1,Depuis$
OPEN "O",2,Vers$
WHILE NOT EOF(1)
    PRINT #2,INPUT$(1.1):
WEND
CLOSE 2
CLOSE 1
RETURN
```

Mais cette routine est bien trop lente ! cela irait beaucoup plus vite si on pouvait lire des masses de données plus importantes d'un seul coup. Cela vous semble paradoxal ? l'explication est pourtant simple : un lecteur de disquette fonctionne grâce à une mécanique complexe. Il lui faut amener une tête de lecture (ou deux têtes pour un lecteur de disquettes double-face) sur la piste contenant l'octet recherché. Comme la disquette tourne, la tête de lecture attend que l'octet en question vienne se positionner juste sous elle de façon à pouvoir le lire. Il serait donc judicieux de lui demander de lire plusieurs octets une fois qu'elle se trouve à la bonne place, puisqu'il lui suffit alors de poursuivre sa lecture sans l'interrompre. J'admets que c'est là une explication très simpliste du processus, mais elle a le mérite d'être claire.

L'ordinateur peut recharger en une seule fois au maximum 32000 octets puisque c'est la taille maximale admise pour une chaîne de caractères en Omikron Basic. Mais nous voilà de nouveau confrontés à une nouvelle difficulté, car la fonction EOF() ne nous sert plus à rien. En effet, en lisant plusieurs octets sans interruption, on peut dépasser le code de fin de fichier sans que la fonction EOF() le remarque. Imaginons par exemple qu'il nous reste 2000 octets à recharger depuis un fichier : la fonction EOF() ou plus exactement NOT EOF() nous renvoie la valeur 'vrai' puisque nous n'avons pas encore dépassé la fin du fichier, alors qu'il n'y a plus assez d'octets à recharger par rapport à l'instruction INPUT\$(32000,1). Que faire ?

Une des solutions consisterait à prendre en compte la longueur, la taille du fichier, c'est-à-dire le nombre d'octets qu'il contient. En connaissant la taille du fichier, nous pourrions écrire une boucle demandant de charger les données par 'tranches' de 32000 octets

jusqu'à ce qu'il reste un nombre d'octets à recharger qui soit inférieur à 32000. Une autre instruction servirait alors à recharger ce reste d'un seul coup, et voilà notre fichier entièrement recopié. Il existe en effet une fonction retournant la taille d'un fichier déterminé :

`LOF(<numéro de fichier>)`

Appliquée à un fichier séquentiel ouvert, la fonction `LOF()` (Length Of File) retourne le nombre d'octets qu'il contient, en incluant le code de fin de fichier EOF. C'est pourquoi il convient de retrancher un octet de ce nombre, puisque nous n'avons pas besoin de recopier ici le code EOF.

Lorsqu'un fichier séquentiel est activé en écriture, la fonction `LOF()` retourne la longueur des données déjà inscrites dans ce fichier.

⚠ Attention : Avant d'inscrire les données sur la disquette, l'ordinateur les stocke dans une mémoire intermédiaire (buffer), ceci afin d'inscrire des 'paquets' de données aussi importants que possible à chaque accès sur la disquette, et la fonction `LOF()` ne décompte pas les données se trouvant encore dans le buffer. Appliquée à un fichier relatif (à accès direct) la fonction `LOF()` retourne le nombre d'enregistrements déjà inscrits dans le fichier.

Nous savons maintenant que la fonction `LOF()` nous renvoie la taille d'un fichier ; reste à préciser que ce nombre est affecté à une variable (L). Lorsque l'on recopie un fichier d'une taille supérieure à 32000 octets, un parcours de la boucle provoque la lecture de 32000 octets qui sont immédiatement recopiés dans le fichier cible (le fichier est donc recopié 'par tranches'), après quoi la variable (L) se voit diminuée du nombre d'octets déjà lus ; l'interpréteur vérifie ensuite si la condition préalable à la boucle est toujours remplie ; il ressort de la boucle (ou n'entre même pas dedans) lorsque le fichier ne contient plus qu'un nombre d'octets à recopier inférieur à 32000. Le reste des octets (valeur de la variable L) est alors recopié. Voici le texte de cette petite procédure :

```

0  *****
1  **                                FILECOPY.BAS                                **
2  **-----**
3  ** Auteur: Michael Maier   Version 1.00           Date: 10.07.1990 **
4  **   Programme joint au Livre de l'Omikron Basic **
5  **   (c) MICROAPPLICATION **
6  *****
7  '
8  '
9  'la routine ci-dessous sert à recopier un fichier
10 '
11 'pour la lancer: Filecopy("A:\FICHIER_1.BAS", "D:\FICHIER_2.BAS)
12 '                   fichier source      fichier cible
13 '
14 DEF PROC Filecopy(Depuis$.Vers$)
15   LOCAL L
16   OPEN "I",1,Depuis$
17   OPEN "O",2,Vers$
18   L=LOF(1)
19   WHILE L>32000 'valeur maxi d'un string
20     PRINT #2, INPUT$(32000,1);
21     L=L-32000
22   WEND
23   ' inutile de recopier code EOF => L-1
24   PRINT #2, INPUT$(L-1,1);
25   CLOSE 2
26   CLOSE 1
27 RETURN

```

3.6. Les boîtes de sélection d'objet : file-selector-box

Nous sommes maintenant en mesure de sauvegarder sur disquette notre petit fichier d'adresses, et nous saurons même le recharger lorsque nous en aurons besoin. Nous avons décidé de le baptiser MINIADR.DAT.

Admettons que nous souhaitions ainsi créer plusieurs fichiers d'adresses (l'un contenant celles des amis, l'autre celles des relations d'affaires etc.) qui seront tous gérés par notre programme. Il faut alors leur attribuer des noms différents. La méthode la plus simple consiste à demander à l'utilisateur d'entrer un nom de fichier (à l'aide de INPUT) puis d'ouvrir ce fichier tout à fait normalement grâce à OPEN :


```
INPUT "quel fichier souhaitez-vous consulter ? ":Nom$
OPEN "I",1,Nom$
```

```
...
...
```

Voilà probablement la méthode qui serait utilisée sur la plupart des ordinateurs, mais pas sur l'Atari ST, et encore moins avec l'Omikron Basic. Sinon, à quoi bon avoir le GEM à sa disposition ? Le GEM nous offre en effet un moyen aussi simple que confortable de créer un nouveau fichier pour y enregistrer des données ou d'activer un fichier existant pour le recharger, et ce moyen, c'est la boîte de sélection d'objet (file selector box).

Dans cette boîte, vous trouvez une première ligne juste au-dessous de la mention 'INDEX' : cette première ligne est destinée à recevoir le 'chemin d'accès' du fichier. Qu'est-ce au juste que le chemin d'accès ? comme tous les chemins, il vous mène à un but : dans notre cas, il mène au fichier souhaité. En effet, les ordinateurs modernes permettent de subdiviser l'ensemble de votre disquette (ou disque dur) en dossiers. Ces dossiers peuvent à leur tour contenir soit des sous-dossiers soit des fichiers. Pour que l'ordinateur - ou plus précisément son système d'exploitation - puisse retrouver votre fichier, il faut que vous lui indiquiez dans quel dossier vous l'avez rangé. D'où la nécessité des chemins d'accès. Mais ce n'est pas tout : le chemin d'accès se compose de plusieurs éléments :

```
<identificateur du lecteur>:\<dossier 1>\<sous-dossier 2>\*.BAS
```

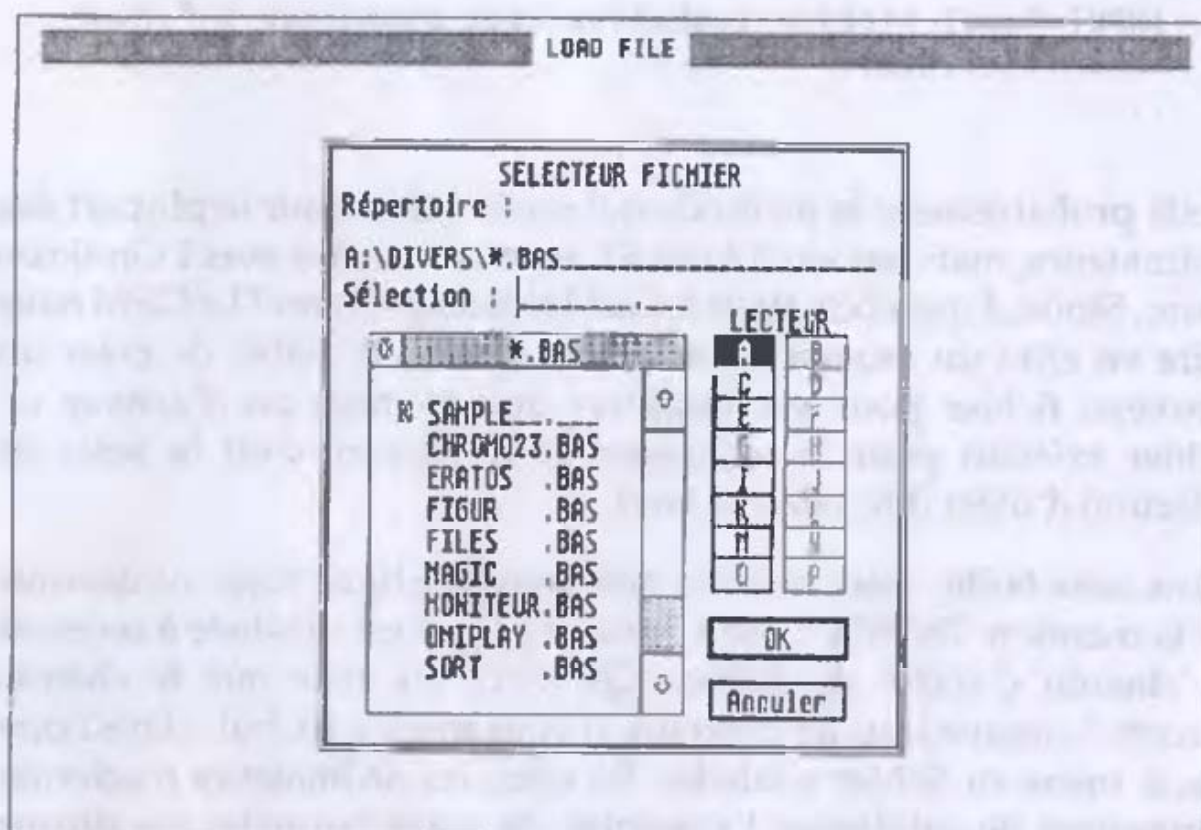


Figure 3.1 : la boîte de sélection d'objet 'file selector box'

Le chemin commence par une première lettre majuscule identifiant l'unité de disque(tte) activée ; dans la plupart des cas, il s'agira du 'A' (identificateur de l'unité de disquette incorporée dans l'Atari ST), mais vous pourriez aussi entrer (si vous possédez ces périphériques) 'B' (deuxième unité de disquette ajoutée) ou C, D etc (partitions du disque dur) ou 'd' (disque RAM).

L'identificateur de l'unité de disque(tte) est suivi de deux-points, puis du chemin à proprement parler, qui rend compte des dossiers et sous-dossiers s'imbriquant hiérarchiquement les uns dans les autres. Dans le cas le plus simple, le fichier se trouve au niveau du répertoire principal, c'est-à-dire directement au niveau de l'unité de disque(tte), lorsque vous n'avez pas créé de dossier :

<identificateur du lecteur> : *.*

soit dans le cas le plus répandu :

A:*.*

Si vous souhaitez charger un fichier se trouvant sur la disquette insérée dans l'unité A et dans un dossier nommé 'DOCUMENT.SDO' vous entrerez le chemin d'accès :

A:\DOCUMENT.SDO*.*

La mention *.* vous permet de voir s'afficher tous les noms des fichiers et sous-dossiers rangés dans le dossier DOCUMENT.SDO: on dit alors que vous prenez connaissance du "répertoire" de DOCUMENT.SDO. Bien entendu, vous pouvez affiner le rangement des fichiers en créant des sous-dossiers, des sous-sous-dossiers etc.: vous devez alors indiquer le chemin d'accès depuis le répertoire principal (niveau global de la disquette) jusqu'au sous-dossier le plus précis, hiérarchiquement juste au-dessus du fichier. Les noms de dossiers et sous-dossiers sont séparés par une barre oblique inversée, nommée 'backslash' :

A:\DOCUMENT.SDO\FORMUL.SDO*.*

Le chemin ci-dessus vous permettra de voir les fichiers et sous-sous-dossiers rangés dans le sous-dossier FORMUL.SDO lui-même rangé dans le dossier DOCUMENT.SDO, se trouvant sur la disquette insérée dans le lecteur 'A'. La dernière partie du chemin d'accès précise l'intitulé du fichier, se composant de deux parties séparées par un point : le nom proprement dit (huit caractères maximum) et l'extension (trois lettres maxi).

En règle générale, on se sert de l'extension pour distinguer entre différents types de fichiers :

Extension	Type de fichiers
BAS	programmes écrits en Basic
PRG	fichier contenant un programme exécutable sous GEM
TXT	fichier contenant du texte (traitement de texte)
DAT	fichier contenant des fiches (fichier d'adresses)
...	
...	

Mais je pense que vous connaissiez déjà tout cela. Vous ne savez peut-être pas que vous pouvez remplacer le nom du fichier (au niveau de la ligne d'édition juste au-dessous de INDEX dans la boîte de sélection) par un masque '?' ou une troncature '*', ce qu'on appelle en anglais 'wildcards' ou 'joker'. Ces deux caractères servent à réserver la place d'un caractère ou d'un intitulé complet, ce qui permet de sélectionner un groupe de fichiers.

Le point d'interrogation '?' masque un caractère quelconque alors que l'étoile '*' remplace une chaîne de caractères ; voici quelques exemples concrets.

CR?ER.PRG provoque l'affichage de tous les noms des fichiers (contenant en principe un programme) se composant des lettres indiquées, la troisième place représentée par '?' pouvant être occupée par n'importe quel caractère :

CRAER.PRG ou **CREER.PR**G ou **CRIER.PR**G ou **CR_ER.PR**G etc. etc.

C*PRG provoque l'affichage de tous les noms de fichier commençant par un C et arborant l'extension PRG ; par exemple : CREER.PRG ou **CALCUL.PR**G ou **COPIER.PR**G etc etc.

***.BAS** provoque l'affichage de tous les noms de fichier arborant l'extension BAS.

. provoque l'affichage de tous les noms des fichiers contenus dans un répertoire donné, quel que soit le nom et quelle que soit l'extension.

L'Omikron Basic est doté d'une instruction spécifique servant à appeler la boîte de sélection d'objet pour qu'elle s'affiche à l'écran :

FILESELECT(<chemin>,<nom_du_fichier>,<Flag>)

La variable-string <chemin> doit contenir le début du chemin d'accès, tel qu'il apparaîtra dans la boîte de sélection, sur la ligne d'édition INDEX. Il faut absolument que ce chemin d'accès soit correct. Le minimum sera :

A:*.*

Si vous souhaitez que la boîte de sélection affiche d'emblée, dès son affichage, tous les fichiers contenant (en principe !) des textes de programme en Basic, vous écrirez :

A:*.BAS

Vous pouvez aussi (mais ce n'est pas indispensable) spécifier le nom et l'extension d'un fichier, intitulé qui viendra s'afficher juste au-dessous de 'Sélection' : cela vous permet par exemple de proposer à l'utilisateur de reprendre plutôt tel nom par défaut. Lorsque l'utilisateur active un intitulé (nom + extension) de fichier, celui-ci vient s'inscrire sur la ligne d'édition juste au-dessous de 'Sélection' et est affecté à la variable-string <nom_du_fichier>.

Une fois la sélection terminée, la variable <flag> indique à l'ordinateur si l'utilisateur a cliqué sur 'CONFIRMER' ou 'ANNULER' : dans le premier cas, <flag> prend la valeur 1 et dans le deuxième cas il prend la valeur 0. Il faut penser à activer la souris (avant l'appel de la fonction) puis à la désactiver (après l'appel de la fonction) pour que l'utilisateur puisse s'en servir pour sélectionner un intitulé de fichier. Voici un petit programme illustrant l'appel de la boîte de sélection :

```
' on se contente d'un chemin d'accès minimum
Chemin$="A:\*.*"
MOUSEON 'activer la souris
FILESELECT (Chemin$,Nom$,Button)
MOUSEOFF
IF Button THEN
    PRINT "Fichier ";Nom$;" sélectionné"
ELSE
    PRINT "vous avez cliqué sur ANNULER !"
ENDIF
....
....
```

Une fois que l'utilisateur a confirmé sa sélection, le programme peut ouvrir le fichier Nom\$ grâce à OPEN, après quoi les données correspondent sont chargées ou sauvegardées.

Voici une procédure qui favorise une utilisation plus souple de la boîte de sélection :

```

0  *****
1  *                               DO_FILE.BAS                               *
2  *-----*
3  * Auteur: Michael Maier      Version 1.00      Date: 16.07.1990      *
4  * Programme joint au livre de l'Omikron Basic                               *
5  * (c) MICROAPPLICATION                               *
6  *****
7  .
8  .
9  'vous pouvez intégrer cette procédure DO_FILE dans vos programmes: elle
10 'facilite l'utilisation de la boîte de sélection d'objet.
11 .
12 F_Nom$="":F_Chemin$="":Ext$="*.*)"
13 Do_File(F_Nom$,F_Chemin$,Ext$,F_Nom$,F_Chemin$,Ext$,Touche%)
14 ' Just for info ... (pour celui qui tient vraiment à tout préciser)
15 IF Touche%=0 THEN
16   FORM_ALERT (1."[1][vous avez cliqué sur ANNULER !][ Annuler ]")
17 ELSE
18   FORM_ALERT (1."[1][vous avez cliqué sur OK!][ Suite ]")
19 ENDIF
20 .
21 END
22 .
23 'voici maintenant la procédure elle-même
24 .
25 DEF PROC Do_File(Nom$,Path$,Post$,R_Nom$,R_Path$,R_Post$,R_Tou%)
26 'utiliser des variables locales afin de pouvoir intégrer cette
27 'procédure dans d'autres programmes.
28 LOCAL R_Path$=Path$,R_Nom$=Nom$.TX,Drive%,Pointer%L,Ret%L
29 IF Path$="" THEN 'si aucun chemin indiqué, en élaborer un
30   Path$=" "64 'en respectant les conventions GEMDOS
31   Pointer%L= LPEEK( VARPTR(Path$))+ LPEEK( SEGPTR +28)
32   ' après quoi chercher le chemin actuel
33   GEMDOS (.71, HIGH(Pointer%L), LOW(Pointer%L),0)
34   ' puis découper correctement
35   Path$= LEFT$(Path$, INSTR(Path$+ CHR$(0), CHR$(0))-1)
36   GEMDOS (Drive%.25) ' identificateur de l'unité de disque activée
37   Path$= CHR$(65+Drive%)+": "+Path$+"\\"
38 ENDIF
39 IF Post$="" THEN 'on se passe difficilement d'une extension
40   Path$=Path$+"*.*)" ' ajouter une extension
41 ELSE
42   Path$=Path$+Post$
43 ENDIF
44 PRINT CHR$(27):"f" 'désactiver le curseur et activer la souris
45 MOUSEON
46 FILESELECT (Path$,Nom$,Ret%L)
47 MOUSEDFF
48 PRINT CHR$(27):"e" 'réactiver le curseur
49 IF Ret%L=0 THEN ' sélection de ANNULER
50   Path$=R_Path$ ' reprendre l'ancien chemin
51   Nom$=R_Nom$
52   Tou%=0
53 ELSE
54   ' rechercher le dernier backslash '\'
```



```
55     TX= LEN(Path$)
56     WHILE TX>0 AND MID$(Path$,TX,1)<>"\"
57         TX=TX-1
58     WEND
59     Path$= LEFT$(Path$,TX)
60     Tou%=1
61     ENDIF
62     RETURN
```

Je vais vous fournir quelques explications sur le fonctionnement de ce programme, bien que cela exigerait que vous possédiez des connaissances bien plus étendues qu'elles ne le sont à ce stade de la lecture de cet ouvrage. Ne vous découragez pas et lisez bien ce qui suit, car vous y apprendrez de plus comment vous servir au mieux de cette procédure.

Lorsqu'on l'appelle, on passe à la procédure trois variables-strings, le reste des paramètres étant constitué de valeurs en retour. Voici ces trois variables strings, dans l'ordre correct :

- ❶ le chemin
- ❷ le nom du fichier (sans extension)
- ❸ l'extension accompagnant le nom du fichier.

Après le lancement de la procédure, nous affectons le chemin et le nom du fichier à deux variables locales. L'Omikron Basic vous permet de déclarer une variable locale en lui affectant immédiatement une valeur, ce dont nous faisons usage ici (ligne 28). Après nous contrôlons si un chemin d'accès a été transmis au moment de l'appel de la procédure. Si tel n'est pas le cas, la procédure peut en bricoler un, en recourant au système d'exploitation de l'Atari ST (lignes 30 et 31).

Accompagnée du paramètre 71, la fonction GEMDOS() nous fournit le chemin actuel : elle ne se contente pas de l'affecter à une variable, mais attend de plus qu'on lui fournisse un pointeur lui indiquant l'endroit où elle doit inscrire ce chemin d'accès dans la mémoire vive. Le GEMDOS admet des chemins d'accès de 64 caractères maximum, c'est pourquoi nous commençons par remplir un string (Path\$) avec 64 espaces vides (ligne 30). Après quoi il faut préciser où le string va

se trouver dans la mémoire : l'adresse (le pointeur) ainsi obtenu est transmis à la routine GEMDOS, et notre variable string Path\$ contient alors le chemin actuel (attention, il manque encore l'identificateur de l'unité de disque ou disquette).

Ce chemin d'accès se termine par un octet nul CHR\$(0). Alors qu'il est tout à fait usuel en langage C de terminer une chaîne de caractères par un octet nul, l'Omikron Basic quant à lui ne peut rien en faire, si bien qu'il ne nous reste plus qu'à le retrancher du string (ligne 35). Le mieux pour ce faire est de rechercher sa position grâce à INSTR() puis de retrancher tout ce qui se trouve à gauche en utilisant LEFT\$ (d'où le -1 de la ligne 35). Ligne 36 : la fonction 25 du GEMDOS affecte à la variable DRIVE% le numéro de l'unité de disque(tte) actuelle en respectant le tableau suivant :

Valeur de Drive%	Désignant l'unité de disque
0	A
1	B
2	C

En tant qu'integer, cette valeur identifiant l'unité de disque(tte) ne nous sert à rien ! nous avons besoin d'un caractère dans le chemin d'accès. Encore une fois, nous allons nous en tirer en utilisant le code ASCII :

```
CHR$(65)
```

nous fournit la lettre 'A' et

```
CHR$(66)
```

nous fournit la lettre 'B'. Il suffit d'additionner l'identificateur de l'unité de disque(tte) (Drive%) à l'argument de cette fonction pour qu'il en résulte un identificateur d'unité allant de A jusqu'à ..., selon le contenu de la variable Drive%. Ceci nous permet d'obtenir un chemin d'accès (presque) complet et vous pouvez lire à la ligne 37 :

```
CHR$(65+Drive%)+":"+Path$+"\\"
```


Vous voyez qu'il nous faut même penser au dernier backslash, ce qui ne pose aucun problème en passant par une "addition" de strings. Il nous reste encore à ajouter l'extension à la fin de ce chemin. Si vous décidez de ne pas passer d'extension dans les paramètres, la procédure juxtapose par défaut ".*" à la fin du chemin.

Avant d'appeler la boîte de sélection, n'oublions pas de désactiver le curseur et d'activer la souris. Lorsque l'utilisateur clique sur 'ANNULER', la procédure restaure les anciens paramètres et les repasse au programme principal, tout en remettant le flag sur (0). Lorsque l'utilisateur clique par contre sur 'OK' il convient de retrancher la dernière partie du chemin (c'est-à-dire l'extension qui lui avait été ajoutée) : nous utilisons une boucle servant à contrôler chaque caractère du chemin d'accès, en commençant par la fin, jusqu'à ce que l'on trouve le backslash. Après quoi on conserve tout ce qui se trouve à sa gauche (ligne 59).

Le manuel de programmation livré avec l'Omikron Basic contient d'ailleurs un algorithme (qui m'a beaucoup plu) pour retrancher l'extension. La fonction INSTR sert à rechercher un caractère quelconque dans une chaîne de caractères et retourne ensuite la position de ce caractère ; la recherche commence au premier caractère. Dans notre cas, nous préférierions commencer par la fin de la chaîne : il suffit d'utiliser MIRROR\$ pour 'retourner' la chaîne de caractères, après quoi nous recherchons le backslash tout simplement à l'aide de INSTR :

```
Path$= LEFT$(Path$,LEN(Path$)-INSTR(MIRROR$(Path$)+"\\","\\"))
```

Nous obtenons ainsi le chemin d'accès, le nom du fichier y compris son extension ainsi que la valeur 1 du bouton 'OK'. Terminé ! si vous souhaitez ensuite ouvrir le fichier sélectionné (par exemple pour le recharger), vous écrivez :

```
OPEN "I",1, F_Chemin$+F_Nom$
```

Que se passe-t-il lorsque le fichier ainsi désigné n'existe pas sur la disquette ? l'ordinateur renvoie un message d'erreur, ce qui n'est pas bien grave ; plus grave par contre : l'interpréteur en profite pour vous

sortir ! Les bons programmes ne tolèrent pas une telle grossièreté : il faut repérer cette erreur de manipulation et l'Omikron Basic offre dans ce domaine des possibilités que nous allons examiner.

3.7. Parer aux erreurs de manipulation

Lorsqu'une erreur survient dans le cours de l'exécution d'un programme, le système envoie un message d'erreur puis sort du programme en cours : il est possible d'éviter cela par le biais d'une routine d'exploitation des erreurs. Il sera alors possible de vérifier quelle est l'erreur commise pour envoyer un message explicatif approprié. En Omikron Basic, vous pouvez demander au programme de se brancher sur une routine particulière (désignée par un label ou un numéro de ligne) lorsque survient une erreur, grâce à l'instruction :

```
ON ERROR GOTO <cible>
```

que l'on écrit habituellement au début du programme. Pour annuler l'effet de ON ERROR GOTO, il suffit d'écrire

```
ON ERROR GOTO 0
```

ce qui revient à interrompre le programme comme s'il n'y avait pas de procédure ; CLEAR (effacer le contenu de toutes les variables) ou RUN ou NEW ont d'ailleurs le même effet ; vous pouvez aussi modifier le code du programme.

Il existe une variable-système (variable gérée directement par l'ordinateur) qui permet de vérifier, dans la routine d'exploitation de l'erreur, à quel type celle-ci appartient :

ERR

cette variable-système contient le numéro de l'erreur commise tandis que

ERR\$

contient le texte du message qui serait envoyé s'il n'y avait pas de ON ERROR GOTO ; enfin,

ERL

contient le numéro de la ligne dans laquelle est survenue l'erreur. Vous trouverez en annexe à ce livre la liste des messages d'erreur ainsi que leur numéro.

Le message 'file not found' (fichier non trouvé sur la disquette) porte le numéro 53. Lorsque la variable ERR contient 53, cela signifie que le système ne trouve pas le fichier demandé sur la disquette présente dans l'unité de disquette :

```
ON ERROR GOTO Erreur
...
...
Do_File(Chemin$,Nom$,Ext$,Chemin$,Nom$,Ext$,Flag%)
IF Flag% THEN
OPEN "I",1,Chemin$+Nom$
WHILE NOT EOF(1)
...
WEND
CLOSE
ENDIF
...
-Erreur
IF ERR=53 THEN
' fichier non trouvé
FORM_ALERT(1"[3][Je ne trouve pas ce fichier
1][Annuler]")
ELSE
' s'il s'agit d'une autre erreur
ENDIF
RESUME NEXT
...
...
```

Lorsqu'une erreur surgit, l'ordinateur se branche sur la routine 'Erreur' : il vérifie d'abord s'il s'agit d'un fichier non trouvé (numéro 53 dans la variable-système ERR). Si tel est le cas, il envoie un message dans un panneau d'avertissement (alertbox). Si ce n'est pas le cas, il

passer sous ELSE où il devrait rencontrer un autre type d'erreur (il y en a beaucoup !). Enfin, l'instruction RESUME incite l'ordinateur à reprendre l'exécution du programme.

Il existe trois variantes d'utilisation de l'instruction RESUME :

1. **RESUME <cible>** le programme sort de la routine de traitement d'erreur et reprend l'exécution du programme à l'endroit désigné par <cible> ;
2. **RESUME NEXT** le programme sort de la routine de traitement d'erreur et reprend l'exécution du programme à partir de la ligne où l'erreur s'est produite ;
3. **RESUME** le programme sort de la routine de traitement d'erreur et tente de réexécuter l'instruction à l'origine du branchement dans la routine d'erreur.

Pour tester la routine d'erreur, vous pouvez vous-même provoquer l'appel d'une erreur en précisant son numéro :

ERROR <numéro d'erreur>

- ☛ **Attention :** ERROR doit se trouver dans le cours du programme pour que celui-ci puisse se brancher sur la routine d'erreur ; faute de quoi, en mode direct, l'ordinateur se borne à afficher l'erreur à l'écran.

3.8. Les fichiers back-up

Les instructions étudiées jusqu'ici vous permettent de gérer très simplement plusieurs fichiers d'adresses à l'aide de notre petit programme FICHIER. Il vous suffit de sélectionner un fichier dans la boîte de sélection pour que l'ordinateur le recharge en mémoire vive et que vous puissiez ainsi le modifier si nécessaire. Avant de terminer votre session de travail, il convient de resauvegarder le fichier dans son nouvel état sur la disquette.

Cette manoeuvre détruit le fichier dans son ancien état, car le programme écrase l'ancien pour sauvegarder le nouveau : en effet, il ne peut y avoir deux fichiers de même nom à l'intérieur d'un même niveau de répertoire. Il est cependant possible de conserver l'ancien état du fichier en modifiant son nom. La plupart des programmes ou logiciels d'aujourd'hui prennent ainsi la précaution de transformer l'ancien fichier en fichier-Back-up avant qu'il ne soit écrasé par le nouveau fichier. En fait, le fichier garde son nom, on ne modifie que son extension, qui devient généralement 'BAK' (abréviation de Back-up) : l'ancien état du fichier ne sera plus détruit, puisqu'il ne porte plus le même nom, lors de la sauvegarde du nouvel état du fichier. Ces deux versions (ancienne et nouvelle) ne se distinguent donc que par leur extension respective, mais ceci suffit amplement.

Comme il se peut fort bien qu'il existe déjà un fichier de même nom arborant l'extension BAK (résultat d'une sauvegarde antérieure), il faut veiller à le détruire ; nous utilisons pour ce faire l'instruction

`KILL <nom>`

qui fait disparaître le fichier <nom> : soyez donc très prudent lorsque vous utilisez cette instruction, dans votre propre intérêt, car elle détruit irrémédiablement le fichier désigné, que vous ne pourrez généralement plus retrouver ensuite. Si le fichier est rangé dans un dossier, indiquez le chemin d'accès complet.

L'instruction `NAME ... AS ...` sert à renommer un fichier ; voici sa syntaxe :

`NAME <ancien nom> AS <nouveau nom>`

Par exemple, en écrivant

```
NAME "OM_BASIC.PRG" AS "ST_BASIC.PRG"
```

vous changez le nom de OM_BASIC.PRG qui devient ST_BASIC.PRG. Ici aussi, vous devez préciser le chemin d'accès du fichier à renommer lorsqu'il est rangé dans un dossier.

L'instruction COPY sert à recopier un fichier, en respectant la syntaxe :

```
COPY <fichier source> TO <fichier cible>
```

Par exemple, en écrivant :

```
COPY "A:\ST_BASIC.PRG" TO "D:\ST_BASIC.PRG"
```

vous recopiez le fichier 'ST_BASIC.PRG' de l'unité de disquette A dans le disque 'D' ; cette instruction admet l'usage du masque '?' et de la troncature '*' dans le nom de <fichier source> mais pas dans le nom du <fichier cible>, ce qui paraît évident !

Dernière instruction de recopiage de fichier, BACKUP sert à confectionner une copie d'un fichier en lui donnant l'extension .BAK ; en écrivant :

```
BACKUP "FICHIER.DAT"
```

vous confectionnez une copie du fichier FICHIER.DAT qui porte le nom FICHIER.BAK.

Avant de confectionner un fichier back-up, on doit contrôler si le fichier concerné existe bien sur la disquette, faute de quoi on peut s'épargner ce travail, car il n'est guère possible de recopier un fichier inexistant ! Ce contrôle étant fait, il faut encore vérifier si le fichier en question n'est pas déjà flanqué d'un fichier back-up, auquel cas il faudrait le supprimer et le remplacer par le nouveau back-up.

La procédure ci-dessous sert à confectionner une copie back-up d'un fichier ; elle utilise une fonction qui vérifie d'abord si le fichier existe bien. Comme l'Omikron Basic ne possède pas cette fonction, je l'ai écrite moi-même en lui donnant le nom FN EXIST(Filename\$) : elle retourne la valeur -1 (vrai) lorsque le fichier existe sur la disquette et

la valeur 0 (faux) lorsqu'il n'existe pas. Je reviendrai plus loin sur cette fonction car elle exige que vous possédiez quelques connaissances supplémentaires.

Venons-en d'abord à la procédure RENAME servant éventuellement à confectionner un back-up :

```

0  *****
1  *                                     RENAME.BAS                               *
2  *-----*
3  * Auteur: Michael Maier           Version 1.00       Date: 16.07.1990 *
4  * Programme joint au livre de l'Omikron Basic
5  * (c) MICROAPPLICATION
6  *****
7  *
8  *
9  * Cette procédure doit être accompagnée de la fonction EXIST.BAS; elle
10 * confectionne une copie back-up lorsque le fichier existe bien
11 * sur l'unité de disque(tte), tout en détruisant auparavant un
12 * back-up éventuellement déjà existant.
13 *
14 * pour lancer cette procédure: Rename(<filename>)
15 *
16 *
17 DEF PROC Rename(Filename$)
18   IF FN Exist%L(Filename$) THEN
19     ' le fichier existe bien => est-il déjà flanqué d'un back-up ?
20     IF FN Exist%L( LEFT$(Filename$, INSTR(Filename$, ".")+ "BAK")
21       ' effacer le back-up existant ...
22       THEN KILL LEFT$(Filename$, INSTR(Filename$, ".")+ "BAK"
23     ENDIF
24     ' confectionner un back-up
25     BACKUP (Filename$)
26   ENDIF
27 RETURN

```

Pour respecter une certaine cohérence, je vous donne immédiatement le texte de la fonction EXIST, sans laquelle le programme ci-dessus ne peut fonctionner : je vous donnerai les explications nécessaires plus loin.

```

0  *****
1  *                                     EXIST.BAS                               *
2  *-----*
3  * Auteur: Michael Maier           Version 1.00       Date: 16.07.1990 *
4  * Programme joint au livre de l'Omikron Basic
5  * (c) MICROAPPLICATION
6  *****
7  *
8  *
9  * la fonction ci-après contrôle si la disquette contient déjà un
10 * fichier 'Nom$'
11 * si le fichier existe           => valeur en retour '-1' ('vrai')

```

```

12 'si le fichier n'existe pas -> valeur en retour '0' ('faux')
13 '
14 IF NOT FN Exist%L("EXIST.BAS") THEN
15     FORM_ALERT (1."[3][le fichier n'existe pas][ Annuler ]")
16 ELSE
17     FORM_ALERT (1."[1][tout est en ordre][ OK ]")
18 ENDIF
19 END
20 '
21 DEF FN Exist%L(Nom$)
22     LOCAL TX
23     '
24     OPEN "F".1,Nom$.55
25     IF EOF(1) THEN
26         'fichier non trouvé sur la disquette
27         '-> envoyer le message d'erreur (0)
28         CLOSE 1
29         RETURN (0)
30     ELSE
31         ' préparer un buffer DTA
32         FIELD 1,30 AS Buffer$.14 AS Files
33         GET 1,0
34         'fichier bien retrouvé -> retourner valeur '-1'
35         IF Nom$= LEFT$(Files$, INSTR(Files$, CHR$(0))-1) THEN
36             CLOSE 1
37             RETURN (-1)
38         ENDIF
39     ENDIF
40 CLOSE 1
41 'le fichier n'existe pas
42 RETURN (0)

```


3.9. Clarification du contenu de la disquette

Nous les avons déjà rencontrés dans le chapitre traitant de la boîte de sélection d'objet, et nous allons les examiner de plus près : les dossiers. Ils sont bien utiles pour mettre un peu d'ordre dans la multitude des fichiers qui finissent par s'amonceler sur la disquette. Nous ne savions pas encore comment créer un dossier, et c'est précisément ce que nous allons étudier maintenant.

☐ Comment créer un dossier

Pour créer un dossier en Omikron Basic, on utilise l'instruction

```
MKDIR <nom du dossier>
```

ce qui installe un dossier intitulé <nom du dossier> dans l'unité de disque(tte) actuelle.

Le dossier représente d'abord un intitulé de plus à inscrire dans le répertoire de la disquette, le 'directory'. Lorsque vous ouvrez un dossier, ou que vous allongez le chemin d'accès d'un nom de dossier supplémentaire, vous engendrez un sous-dossier (sub-directory).

☐ Comment effacer un dossier

Il est naturellement possible d'effacer un dossier, à l'aide de l'instruction

```
RMDIR <nom du dossier>
```

ce qui fait disparaître le sous-répertoire du répertoire principal.

☐ Toujours suivre le bon chemin

Lorsqu'il existe plusieurs dossiers sur une disquette, celle-ci contient logiquement plusieurs sous-répertoires (subdirectories) ; il faut alors veiller à bien spécifier le chemin d'accès complet pour accéder à un fichier.

Par exemple, si vous avez rangé un fichier LETTRE.SDO dans un dossier DOCUMENT.SDO, vous ne pourrez pas l'ouvrir en écrivant simplement

```
OPEN "I".1,"LETTRE.SDO"
```

puisqu'il ne se trouve pas directement au niveau du répertoire principal ; il vous faudra entrer le chemin d'accès complet pour pouvoir accéder à ce fichier :

```
OPEN "I".1,"A:\DOCUMENT.SDO\LETTRE.SDO"
```

Imaginez-vous ce qu'il faudrait écrire pour accéder au fichier LETTRE.SDO s'il se trouvait dans un sous-dossier AFFAIRE.SDO, lui-même rangé dans le dossier DOCUMENT.SDO ! et s'il vous fallait réécrire tout ce chemin d'accès à chaque fois que vous souhaiteriez lire un fichier différent rangé dans AFFAIRE.SDO.

C'est pourquoi il existe un moyen d'abrégé cette procédure, en déclarant un des sous-dossiers comme étant le 'répertoire actuel' ou le 'chemin d'accès actif par défaut', après quoi toutes les manipulations de fichiers se dérouleront au niveau de ce sous-répertoire, sans qu'il soit besoin de repréciser tout le chemin d'accès. Vous utilisez pour ce faire l'instruction

```
CHDIR <nom du dossier>
```

En écrivant par exemple

```
CHDIR "DOCUMENT.SDO"
```

vous déclarez le contenu du dossier DOCUMENT.SDO comme étant le répertoire actuel, auquel se rapporteront toutes les instructions de manipulation des fichiers. Une instruction SAVE enregistrera ainsi le fichier qui lui a été affecté automatiquement dans ce dossier, tandis que FILES (affichage du répertoire) ne provoquera plus que l'affichage du répertoire de ce dossier. Pour revenir au niveau du répertoire principal, il vous suffit de remplacer <nom du dossier> par deux points suivis d'un backslash :

```
CHDIR "..\"
```


L'instruction CHDIR vous permet également de changer d'unité de disque en faisant commencer le chemin d'accès par l'identificateur concerné :

```
CHDIR "A:\DOCUMENT.SDO"
```

ou même en n'indiquant que l'identificateur de l'unité de disque : il suffit d'écrire

```
CHDIR "D:"
```

pour que toutes les opérations de manipulation des fichiers se déroulent dorénavant dans l'unité de disque D.

3.10. Les fichiers relatifs

Comme vous le savez déjà, un fichier à accès direct se compose de plusieurs enregistrements : il ressemble fort à un véritable fichier se composant par exemple de fiches en bristol, chaque fiche portant divers renseignements répartis dans des champs. Dans notre fichier d'adresses, un enregistrement contient des données réparties entre les champs suivants :

Nom + Prénom + N et rue + Code postal + Ville + Téléphone + Date de naissance. Pour pouvoir recharger chacun de ces enregistrements (records) séparément, il faut qu'ils aient tous la même longueur. Cette longueur de l'enregistrement est le quatrième paramètre entré derrière l'instruction OPEN et le mode-fichier n'est plus "I" ou "O" mais "R" : tout ceci vaut tant en lecture qu'en enregistrement du fichier :

```
OPEN "R",1,"FICHIER.REL",<longueur d'un enregistrement>
```

☐ Comment structurer un enregistrement

Pour ce faire, vous utilisez l'instruction

```
FIELD <numéro du fichier>,<Longueur> AS <variable  
buffer>....
```

Le premier paramètre suivant FIELD est le numéro de fichier déjà mentionné derrière l'instruction OPEN. Après quoi il convient de préciser la longueur de chacune des variables-buffer (mémoire tampon) correspondant aux champs composant l'enregistrement, ainsi que le nombre de caractères que chaque champ pourra contenir. La longueur totale de tous les champs ne peut être supérieure à la longueur de l'enregistrement indiquée derrière OPEN. Prenons un exemple concret.

Admettons que chaque enregistrement d'un fichier doive contenir les renseignements suivants :

Champ	Longueur
Nom	15
Prénom	15
Rue	20
Ville	20

La longueur totale de chaque enregistrement sera de 70, il faudra donc ouvrir le fichier en écrivant :

```
OPEN "R".1,"FICHIER.REL",70
```

après quoi chaque enregistrement (fiche) sera découpé en champs grâce à FIELD, en écrivant :

```
FIELD 1,15 AS Nom$, 15 AS Prenom$, 20 AS Rue$, 20 AS Ville$
```

Après avoir chargé un enregistrement, il est possible de tester le contenu de chaque variable-buffer comme on le fait avec toute autre variable ordinaire (IF, boucle etc), ou de l'éditer avec un PRINT : exemples

```
IF Nom$ = "Arthur" THEN ....
```

```
WHILE Nom$ <> ""
```

```
...
```

```
WEND
```

```
PRINT Nom$,Prenom$,Rue$,Ville$
```


On ne peut passer une valeur à une variable-buffer qu'en utilisant MID\$(), LSET ou RSET. Vous savez en principe déjà comment passer une valeur avec MID\$()= et je vais vous donner quelques explications au sujet des deux autres instructions.

□ LSET et RSET

Ces deux instructions réclament la même syntaxe que LET. Contrairement à LET cependant, LSET ou RSET sont absolument indispensables lorsqu'il s'agit d'affecter une valeur à une variable-buffer : en écrivant

```
LSET Nom$ = "DUPONT"
```

vous affectez à la variable Nom\$ le contenu DUPONT ; la particularité de cette instruction étant que la chaîne de caractères est placée dans la variable à partir de la gauche (Lset = Left-set). Les espaces non occupés (différence entre la longueur de la variable et la longueur de la chaîne de caractères) sont remplis par des espaces vides

```
Nom$ = "DUPONT" + CHR$(32)*(15-LEN("DUPONT"))
```

Lorsque la chaîne de caractères contient plus de caractères que le champ ne peut en contenir, la chaîne est tronquée.

Contrairement à LSET, l'instruction RSET (Rset = right-set) place la chaîne de caractères dans la variable en partant de la droite ; les espaces non occupés sont remplis par des espaces vides et la chaîne est tronquée si elle est trop longue par rapport à la longueur du champ. Dès que vous jugez que les champs sont correctement remplis, vous pouvez sauvegarder l'enregistrement sur la disquette. Il existe deux instructions permettant de lire et écrire le contenu de la fiche qui se trouve dans un buffer.

□ Enregistrement et chargement d'une fiche sur disquette

PUT <numéro de fichier>,<numéro de l'enregistrement>

permet d'enregistrer dans le fichier ouvert, les données se trouvant dans le buffer, à la position désignée par <numéro d'enregistrement>.

GET <numéro de fichier>, <numéro d'enregistrement>

sert à recharger dans le buffer les données se trouvant à la position désignée par <numéro d'enregistrement> dans le fichier accessible par le canal <numéro du fichier>. Une fois dans le buffer, ces données sont reprises des différentes variables-buffer.

Avant d'illustrer ces sombres théories par un exemple concret, je voudrais encore éclaircir un point particulier. Lorsque vous souhaitez entrer dans un fichier relatif non seulement des lettres mais aussi des chiffres, il faut d'abord les amener à respecter une longueur définie (ceci pour la variable-buffer). Qui plus est, le buffer prend des variables du type string (chaîne de caractères) auxquelles il est impossible d'affecter des valeurs numériques. C'est pourquoi l'Omikron Basic est doté de fonctions transformant les nombres en chaînes de caractères de 2, 4, 6 ou 10 octets. Il est ensuite possible de passer ces nombres à une variable en recourant à LSET ou RSET.

Fonction	conversion d'un	en une chaîne de
MKI\$(Nbre)	integer 16 bits	2 octets
MKIL\$(Nbre)	integer de 32 bits	4 octets
MKS\$(Nbre)	single-float	6 octets
MKD\$(Nbre)	double-float	10 octets

La conversion en sens inverse se fait à l'aide de CVx :

Fonction	Conversion d'une chaîne de	En un nombre du type
CVI(Nbre)	2 octets	integer 16 bits
CVIL(Nbre)	4 octets	integer de 32 bits
CVS(Nbre)	6 octets	single-float
CVD(Nbre)	10 octets	double-float

Les deux fonctions

$X\$ - MKI\$(Nbre)$

et

$Nbre - CVI(X\$)$

s'annulent donc l'une l'autre.

3.11. Un mini-fichier... relatif cette fois

J'ai repris le fichier d'adresses que nous avons confectionné dans la première partie de cet ouvrage et j'en ai fait cette fois un fichier relatif.

Chaque enregistrement se compose des champs suivants :

Champ	Longueur
Nom	15
Prénom	15
Rue et n	32
Code postal	5
Ville	30
Téléphone	11
Né le	10

ce qui nous donne une longueur totale de 117 pour chaque enregistrement. Avant de commencer à saisir des adresses, il faut installer le fichier (vide), ce que l'on fait à l'aide du point 4 du menu. La sélection ou la création d'un nouveau fichier se fait en passant par

une boîte de sélection (file-selector-box) gérée dans la routine Do_File. La procédure RENAME se charge de confectionner une copie back-up du fichier dans le cas où l'on sauvegarde un fichier sous un nom déjà existant. La sécurité avant tout !

Après quoi on détermine le chemin d'accès (dans le cas où l'utilisateur souhaite ranger son fichier dans un dossier) et le programme reprend le nom de fichier renvoyé par la procédure Do_File (Nom\$). Le fichier peut contenir autant d'enregistrements que vous l'indiquez sous la variable 'Taille', qui sert également à dimensionner les tableaux.

Autre nouveauté : ce programme dispose d'une procédure "Sauvegarder(Numero%)" qui enregistre le contenu de la variable de l'indice numero% sur la disquette. Une fois le fichier 'Nom\$' ouvert, le programme structure la fiche à l'aide de l'instruction FIELD. Comme il faut réunir sous un même intitulé toutes les variables contenues dans le buffer, mais que les impératifs techniques font que chaque ligne ne peut accepter que 72 caractères, il faut bien que les deux dernières variables-buffer contiennent chacune deux champs différents de l'enregistrement. Cela ne fait rien, j'ai ainsi l'occasion de vous montrer les instructions permettant de remplir une variable-buffer.

Une fois la structure de notre buffer définie, il reste à l'alimenter en données, enregistrables sur disquette. Pour ce faire, nous utilisons LSET et MID\$()=, après quoi nous sauvegardons l'enregistrement entier sur disquette puis nous refermons le canal pour éviter que ne surgissent des conflits avec d'autres canaux ouverts. Comme les fichiers relatifs permettent justement d'écrire, lire et sauvegarder chaque enregistrement un par un, nous en profitons pour faire tout cela à chaque saisie ou correction d'une fiche-adresse.

Tant que nous en sommes à parler de la saisie des adresses : il est indispensable de disposer d'un nom de fichier avant d'appeler la procédure Sauvegarder(Numero%). C'est pourquoi le programme passe d'abord par la sub-routine 'Baptême' qui a pour objet de vérifier si NOM\$ contient déjà un nom de fichier (provenant de l'installation ou du rechargement du fichier). Si tel n'est pas le cas,

elle provoque l'affichage d'une boîte de sélection permettant d'entrer un nom de fichier. Il nous a aussi fallu modifier la routine de rechargement du fichier. Il faut d'abord effacer tous les contenus des variables pour éviter que deux fichiers ne se mélangent. On pourrait faire cela avec une boucle FOR...NEXT :

```
FOR T%=1 TO Taille 'dimensionnement  
  Efface(T%)  
NEXT T%
```

mais elle engendrerait beaucoup de travail inutile. En effet, lorsqu'une variable ne contient plus rien, c'est qu'on a atteint la fin du fichier et que l'on peut sortir de la boucle. C'est pourquoi nous avons préféré utiliser une boucle WHILE...WEND. Une boucle REPEAT...UNTIL nous sert ensuite à recharger tous les enregistrements (à partir du premier) jusqu'à ce que le buffer contienne un enregistrement vide, ce qui démontre que l'ensemble du fichier a été relu complètement. Nous pouvons alors sortir de la boucle et effacer le tout dernier enregistrement (vide). Nous choisissons de recharger entièrement le fichier car, dans un fichier relatif, la recherche d'une fiche dure assez longtemps lorsqu'on laisse le fichier sur la disquette.

Le fichier une fois rechargé, toutes les variables ont la taille maximum autorisée, puisque les espaces non-utilisés sont occupés par des espaces vides. Pour que la recherche d'un nom ne tombe pas dans un cercle vicieux (DUPOND <> DUPOND), la chaîne servant de critère de recherche est également comblée par des espaces vides jusqu'à prendre sa taille maximale (15 caractères). Il serait naturellement beaucoup plus élégant de faire en sorte que les caractères vides figurant à la fin d'une variable soient retranchés juste après le chargement. Vous pouvez tenter de le faire vous-même, en lisant bien les deux observations suivantes :

- en principe, l'instruction INSTR() vous permet de retrouver la première occurrence d'un espace vide, et LEFT\$ de tronçonner alors la chaîne de caractères ; mais cette méthode n'est applicable que si l'enregistrement était effectivement rempli par des données. Lorsqu'un enregistrement ne contient que des espaces vides, la fonction INSTR() renverra la valeur 1 ; mais comme vous souhaitez retrancher les espaces vides (LEFT\$(...,INSTR()-1), il en résultera une valeur 0, ce qui provoque l'affichage d'un message d'erreur.
- cas inverse : si l'enregistrement ne comporte par contre absolument aucun espace vide (tous les champs sont complètement remplis), vous voilà mal parti ! INSTR() retourne alors la valeur 0 ce qui provoque immédiatement l'affichage d'un message d'erreur !
- qui plus est, n'oubliez pas que certains enregistrements peuvent fort bien contenir des espaces vides utiles servant à séparer des données : prenez par exemple le champ du numéro et de la rue (19 rue des Roses) : il contient trois espaces vides indispensables, et vous feriez disparaître des données importantes en écrivant :

```
Rue$ = LEFT$(Rue$, INSTR(Rue$,CHR$(32))-1)
```

puisqu'il ne resterait plus que le numéro '19', et le facteur ne peut pas en faire grand chose. Pour sortir de ce problème, il serait judicieux - si vous tenez absolument à utiliser INSTR() - de retourner la chaîne de caractères à l'aide de MIRROR\$. Mais n'oubliez pas alors de soustraire la position calculée à l'aide de INSTR() de la longueur maximale de la variable pour pouvoir ensuite l'utiliser en tant que paramètre de LEFT().

Vous constatez qu'il n'est pas si simple de retrancher les caractères vides contenus dans une variable : il est beaucoup plus rapide, lors d'une comparaison, de compléter la variable par des espaces vides. Et d'ailleurs, pourquoi vouloir retrancher ces espaces vides s'ils

n'engendrent aucune gêne ni erreur ? Mais ne reculez pas devant la difficulté et faites quelques essais : c'est en forgeant que l'on devient forgeron !

Voici enfin le texte du programme servant à gérer un fichier relatif d'adresses, mais je ne puis vous garantir qu'il soit totalement dépourvu de sources d'erreur ; il ne s'agit que d'un programme de démonstration ne visant pas à faire concurrence aux logiciels de base de données !

```

0 *****
1 *                               FICHIER.REL                               *
2 *-----*
3 * Auteur: Michael Maier      Version 1.00      Date: 15.07.1990      *
4 *   Programme joint au livre de l'Omkron Basic                      *
5 *   (c) MICROAPPLICATION                                           *
6 *****
7 *
8 *
9 MODE "F"
10 DEF FN Screen$(X$)= CHR$(27)+X$
11 *
12 Taille%L=100'modifiez si nécessaire
13 DIM Nom$(Taille%L),Prénom$(Taille%L),Rue$(Taille%L)
14 DIM Postal$(Taille%L),Ville$(Taille%L),Tel$(Taille%L),Nels$(Taille%L)
15 *
16 Erreur$="[3][Ceci est malheureusement] IMPOSSIBLE!!![Désolé]"
17 Avis$="[1][Ce fichier contient"+STR$(Taille%L)+"|  fiches][OK]"
18 Erreur_2$="[3][Fichier impossible à retrouver][Désolé]"
19 *
20 REPEAT
21   CLS
22   PRINT @(0,1):"***78
23   FOR Y%=1 TO 5: PRINT @(Y%,1):"*":@(Y%,78):"*": NEXT Y%
24   PRINT @(6,1):"***78
25   PRINT @(2,28):"FICH_REL - Menu principal"
26   PRINT @(3,28):"-----"
27   PRINT @(4,16):"Un programme tiré du livre de l'Omkron Basic"
28   PRINT @(9,28):"1. Saisie d'une adresse"
29   PRINT @(11,28):"2. Correction d'une adresse"
30   PRINT @(13,28):"3. Recherche d'une adresse"
31   PRINT @(15,28):"4. Sauvegarde du fichier"
32   PRINT @(17,28):"5. Chargement du fichier"
33   PRINT @(19,28):"6. Sortir du programme"
34   PRINT @(22,29):FN Screen$("p");" Veuillez préciser votre choix";FN
Screen$("q")
35   PRINT FN Screen$("f");" pour désactiver le curseur
36   *
37   A%L=0
38   REPEAT
39     A$= INKEY$
40     IF A$<>" " THEN
41       A%L= ASC( RIGHT$(A$,1))-48

```

```

42     ENDIF
43     UNTIL A%L>0 AND A%L<7
44     PRINT FN Screen$("e")'pour réactiver le curseur
45     ON A%L GOSUB Saisir,Corriger,Rechercher,Sauvegarder,Charger
46 UNTIL A%L=6' répéter la boucle jusqu'à ce que touche 6 appuyée
47 CLS
48 END
49 '
50-Saisir
51 CLS
52 'il faut encore donner un nom à notre fichier
53 Baptiser(Button%)
54 'retour au menu principal si 'ANNULER' ou absence de nom
55 IF Button%=0 OR Nom$="" THEN
56     RETURN
57 ENDIF
58 Header$="***** Saisie d'une fiche *****"
59 ' d'abord rechercher le premier élément libre
60 T%=1
61 WHILE Nom$(T%)<>"" AND T%<Taille%L
62     T%=T%+1
63 WEND
64 Masque(Header$)
65 REPEAT
66     Saisie(T%)
67     PRINT FN Screen$("f")'le curseur ne ferait que nous gêner
68     PRINT @(19,15):FN Screen$("p");"F";FN Screen$("q");
69     PRINT "iche suivante ";FN Screen$("p");"C";FN Screen$("q");
70     PRINT "orrection ";FN Screen$("p");"R";FN Screen$("q");
71     PRINT "etour au menu principal"
72     A$="": INPUT " ";A$ USING "+f+r+cu>".Ret%L,1.32
73     PRINT FN Screen$("e")'réactiver le curseur
74     IF A$="C" THEN
75         Masque(Header$)
76         Affichage(T%)
77     ELSE
78         IF A$="F" AND T%<Taille%L THEN
79             Sauvegarder(T%)
80             T%=T%+1
81             Masque(Header$)
82             Affichage(T%)
83         ENDIF
84     ENDIF
85 UNTIL A$="R"
86 Sauvegarder(T%)'pour ne pas risquer de perdre des données
87 RETURN
88 '
89-Corriger
90 Header$="***** Correction d'une fiche *****"
91 PRINT FN Screen$("f")
92 T%=1
93 Masque(Header$)
94 REPEAT
95     Affichage(T%)
96     PRINT @(19,4):FN Screen$("p");"F";FN Screen$("q");"iche suivante ";
97     PRINT FN Screen$("p");"D";FN Screen$("q");"erniere fiche ";
98     PRINT FN Screen$("p");"C";FN Screen$("q");"orrection ";
99     PRINT FN Screen$("p");"E";FN Screen$("q");"ffacer ";

```



```

100 PRINT FN Screen$("p");"R";FN Screen$("q");"etour au menu principal"
101 '
102 A$=""
103 REPEAT
104   A$= INKEY$
105   IF A$<>"" THEN
106     A$= UPPER$( RIGHT$(A$,1))
107   ENDIF
108 UNTIL A$="F" OR A$="D" OR A$="C" OR A$="E" OR A$="R"
109 '
110 IF (A$="F") AND (TX<Taille%L) AND (Nom$(TX+1)<> "") THEN
111   TX=TX+1
112 ELSE
113   IF A$="F" THEN
114     FORM_ALERT (1,Erreurs$)
115   ENDIF
116 ENDIF
117 IF A$="D" AND TX>1 THEN
118   TX=TX-1
119 ELSE
120   IF A$="D" THEN
121     FORM_ALERT (1,Erreurs$)
122   ENDIF
123 ENDIF
124 IF A$="C" THEN
125   PRINT FN Screen$("e")
126   PRINT @(19,1):" *78
127   Saisie(TX)
128   Sauvegarder(TX)
129   PRINT FN Screen$("f")
130 ENDIF
131 IF A$="E" THEN
132   MOUSEON
133   FORM_ALERT (2,"[2][faut-il vraiment effacer?][oui | non]".ButX)
134   MOUSEOFF
135   IF ButX=1 THEN
136     Delete(TX)
137   ENDIF
138 ENDIF
139 UNTIL A$="R"
140 RETURN
141 '
142-Rechercher
143 Header$="***** Recherche d'une fiche *****"
144 Nom$(0)="" : Prenoms$(0)="" : Rue$(0)="" : Postal$(0)=""
145 Ville$(0)="" : Tel$(0)="" : Neles$(0)=""
146 TX=0
147 Masque(Header$)
148 PRINT FN Screen$("e")
149 INPUT @(7,20);Nom$(0) USING "a+ +-u".Ret%L,15
150 PRINT FN Screen$("f")
151 TX=1
152 REPEAT
153   WHILE Nom$(TX)<>Nom$(0)+ SPACES(15- LEN(Nom$(0))) AND TX<Taille%L
154     TX=TX+1
155   WEND
156   IF Nom$(TX)=Nom$(0)+ SPACES(15- LEN(Nom$(0))) THEN
157     Affichage(TX)
158   ELSE

```

```

159 FORM_ALERT (1."[1][Fiche non trouvée][Que faire?]")
160 ENDIF
161 PRINT @ (19,15):FN Screen$("p");"C";FN Screen$("q");
162 PRINT "continuer ";FN Screen$("p");"N";FN Screen$("q");
163 PRINT "nouvelle recherche ";FN Screen$("p");"R";FN Screen$("q");
164 PRINT "etour au menu principal"
165 AS=""
166 REPEAT
167   AS= INKEY$
168   IF AS<>"" THEN
169     AS= UPPER$( RIGHT$(AS,1))
170   ENDIF
171   UNTIL AS="C" OR AS="N" OR AS="R"
172   IF AS="C" AND T%<Taille%L THEN
173     T%=T%+1
174   ENDIF
175   IF AS="N" THEN
176     EXIT TO Rechercher
177   ENDIF
178 UNTIL AS="R"
179 RETURN
180 '
181-Sauvegarder
182 CLS
183 Chemin$="A:\":Nom$="FICHER.REL":Ext$="*.REL"
184 Do_File(Nom$,Chemin$,Ext$,Nom$,Chemin$,Ext$,Touche%)
185 IF Touche% AND Nom$<>"" THEN
186   CLS
187   'prendre le chemin actuel pour directory
188   CHDIR (Chemin$)
189   'et vérifier si existe déjà un fichier du même nom
190   Rename(Nom$)
191   'puis sauver fichier sous le nom indiqué
192   OPEN "R",1,Nom$.117
193   FIELD 1,117 AS Buffers
194   LSET Buffers= SPACE$(117)
195   FOR T%=1 TO Taille%L
196     PUT 1,T%
197   NEXT T%
198   CLOSE 1
199   FORM_ALERT (1.Avis%)
200 ENDIF
201 RETURN
202 '
203-Charger
204 CLS
205 IF Chemin$="" THEN
206   'pas encore de chemin indiqué -> attribuer un chemin
207   Chemin$="A:\":Nom$="FICHER.REL":Ext$="*.REL"
208 ENDIF
209 Do_File(Nom$,Chemin$,Ext$,Nom$,Chemin$,Ext$,Touche%)
210 IF Touche% AND Nom$<>"" THEN
211   CLS
212   CHDIR (Chemin$)'nouveau répertoire principal
213   IF NOT FN Exist%L(Nom$) THEN
214     FORM_ALERT (1Erreur_2%)
215   ELSE
216     Reset_All'effacer les données existantes
217     OPEN "r",1,Nom$.117

```



```

218 FIELD 1,15 AS No$,15 AS Pr$,31 AS Ru$,28 AS Vi$,27 AS Rest$
219 'la largeur de l'imprimante étant insuffisante, nous sommes obligés
220 'de réunir les derniers champs sous une seule variable-buffer
221 'Vi$ -> Postal$ (5 caractères) et Ville$ (23 caractères)
222 'Rest$ -> tel (17 caractères) et né le (10 caractères)
223 TX=0
224 REPEAT
225     TX=TX+1
226     GET 1,TX
227     Nom$(TX)=No$
228     Prenom$(TX)=Pr$
229     Rue$(TX)=Ru$
230     Postal$(TX)= LEFT$(Vi$,5)
231     Ville$(TX)= MID$(Vi$,6)
232     Tel$(TX)= LEFT$(Rest$,17)
233     Nele$(TX)= MID$(Rest$,12)
234 UNTIL Nom$(TX)= SPACES(15) OR TX=TailleXL
235 Efface(TX)
236 CLOSE 1
237 ENDIF
238 ENDIF
239 RETURN
240
241 DEF PROC Masque(Texte$)
242     LOCAL TX
243     CLS
244     PRINT @(2,(78- LEN(Texte$))/2):Texte$
245     PRINT @(5,10):"***60
246     FOR TX=1 TO 9: PRINT @(5+TX,10):"*":@(5+TX,69):"*": NEXT TX
247     PRINT @(15,10):"***60
248     PRINT @(7,15):"Nom: _____ Prenom: _____"
249     PRINT @(9,15):"No et rue: _____"
250     PRINT @(11,15):"Code postal: _____ Ville: _____"
251     PRINT @(13,15):"Tel: _____ Date de naiss.: _____"
252 RETURN
253
254 DEF PROC Affichage(Numero%)
255     PRINT @(7,20):Nom$(Numero%);
256     PRINT STRING$(15- LEN(Nom$(Numero%)),"_")
257     PRINT @(7,52):Prenom$(Numero%);
258     PRINT STRING$(15- LEN(Prenom$(Numero%)),"_")
259     PRINT @(9,26):Rue$(Numero%);
260     PRINT STRING$(31- LEN(Rue$(Numero%)),"_")
261     PRINT @(11,28):Postal$(Numero%);
262     PRINT STRING$(5- LEN(Postal$(Numero%)),"_")
263     PRINT @(11,41):Ville$(Numero%);
264     PRINT STRING$(23- LEN(Ville$(Numero%)),"_")
265     PRINT @(13,20):Tel$(Numero%);
266     PRINT STRING$(17- LEN(Tel$(Numero%)),"_")
267     PRINT @(13,54):Nele$(Numero%);
268     PRINT STRING$(10- LEN(Nele$(Numero%)),"_")
269 RETURN
270
271 DEF PROC Saisie(Numerp%)
272     LOCAL Back$="s"+ CHR$(72)+"s"+ CHR$(80)
273     -Nom
274     INPUT @(7,20):Nom$(Numero%) USING "a+ +-u"+Back$,Ret%L,15
275     IF (Ret%L AND $FF0000)=$480000 THEN
276         GOTO Nele

```

```

277 ENDIF
278 -Prenom
279 INPUT @(7.52):Prenom$(Numero%) USING "a+ +-u"+Back$.Ret%L.15
280 IF (Ret%L AND $FF0000)=$480000 THEN
281     GOTO Nom
282 ENDIF
283 -Street
284 INPUT @(9.26):Rue$(Numero%) USING "0a+ +-+."+Back$.Ret%L.31
285 IF (Ret%L AND $FF0000)=$480000 THEN
286     GOTO Prenom
287 ENDIF
288 -Postal
289 INPUT @(11.28):Postal$(Numero%) USING "0>"+Back$.Ret%L.5
290 IF (Ret%L AND $FF0000)=$480000 THEN
291     GOTO Street
292 ENDIF
293 -Ville
294 INPUT @(11.41):Ville$(Numero%) USING "a0+/-"+Back$.Ret%L.23
295 IF (Ret%L AND $FF0000)=$480000 THEN
296     GOTO Postal
297 ENOIF
298 -Telephone
299 INPUT @(13.20):Tel$(Numero%) USING "0+--+/c-/"+Back$.Ret%L.17
300 IF (Ret%L AND $FF0000)=$480000 THEN
301     GOTO Ville
302 ENDIF
303 -Nele
304 INPUT @(13.54):Nele$(Numero%) USING "0+."+Back$.Ret%L.10
305 IF (Ret%L AND $FF0000)=$480000 THEN
306     GOTO Telephone
307 ELSE
308     IF (Ret%L AND $FF0000)=$500000 THEN
309         GOTO Nom
310     ELSE
311         ENDIF
312     ENDIF
313 RETURN
314 *
315 DEF PROC Delete(Numero%)
316 LOCAL Quantite%-1,T%.Dummy%-Numero%*reperer les fiches à effacer
317 'rechercher le nombre d'enregistrements
318 WHILE (Nom$(Quantite%+1))<>" " AND Quantite%<Taille%L
319     Quantite%-Quantite%+1
320 WEND
321 IF Numero%=Quantite% THEN
322     Efface(Numero%)
323     Sauvegarder(Numero%)
324 ELSE
325     WHILE (Numero%<>Quantite%)
326         Nom$(Numero%)=Nom$(Numero%+1)
327         Prenom$(Numero%)=Prenom$(Numero%+1)
328         Rue$(Numero%)=Rue$(Numero%+1)
329         Ville$(Numero%)=Ville$(Numero%+1)
330         Postal$(Numero%)=Postal$(Numero%+1)
331         Tel$(Numero%)=Tel$(Numero%+1)
332         Nele$(Numero%)=Nele$(Numero%+1)
333         Numero%-Numero%+1
334     WEND
335     Efface(Numero%)

```



```

336 FOR TX=Dummy% TO Numero%
337   Sauvegarder(TX)
338 NEXT TX
339 ENDIF
340 RETURN
341 *
342 DEF PROC Efface(Numero%)
343   Nom$(Numero%)="":Prenom$(Numero%)="":Rue$(Numero%)="":
344   Postal$(Numero%)="":Ville$(Numero%)="":Tel$(Numero%)="":
345   Nele$(Numero%)="":
346 RETURN
347 *
348 DEF PROC Reset_All' pour effacer tous les enregist. dans la RAM
349   LOCAL TX=1
350   WHILE Nom$(TX)<>" " AND TX<Taille%L
351     Efface(TX)
352     TX=TX+1
353   WEND
354 RETURN
355 *
356 DEF PROC Do_File(Nom$,Path$,Post$,R_Nom$,R_Path$,R_Post$,R_Tou%)
357   'utiliser des variables locales afin de pouvoir intégrer cette
358   'procédure dans d'autres programmes.
359   LOCAL R_Path$=Path$,R_Nom$=Nom$,TX=Drive%,Pointer%L,Ret%L
360   IF Path$="" THEN 'si aucun chemin indiqué, en élaborer un
361     Path$=" "64'en respectant les conventions GEMDOS
362     Pointer%L= LPEEK( VARPTR(Path$))+ LPEEK( SEGPTR +28)
363     ' après quoi chercher le chemin actuel
364     GEMDOS (,71, HIGH(Pointer%L), LOW(Pointer%L),0)
365     ' puis retrancher correctement
366     Path$= LEFT$(Path$, INSTR(Path$+ CHR$(0), CHR$(0))-1)
367     GEMDOS (Drive%,25)' identificateur de l'unité de disque activée
368     Path$= CHR$(65+Drive%)+": "+Path$+"\"
369   ENDIF
370   IF Post$="" THEN 'on se passe difficilement d'une extension
371     Path$=Path$+"*.*" ajouter une extension
372   ELSE
373     Path$=Path$+Post$
374   ENDIF
375   PRINT CHR$(27);"f" désactiver le curseur et activer la souris
376   MOUSEON
377   FILESELECT (Path$.Nom$,Ret%L)
378   MOUSEOFF
379   PRINT CHR$(27);"e" réactiver le curseur
380   IF Ret%L=0 THEN ' sélection de ANNULER
381     Path$=R_Path$' reprendre l'ancien chemin
382     Nom$=R_Nom$
383     Tou%=0
384   ELSE
385     ' rechercher le premier backslash '\'
386     TX= LEN(Path$)
387     WHILE TX>0 AND MID$(Path$,TX,1)<>"\"
388       TX=TX-1
389     WEND
390     Path$= LEFT$(Path$,TX)
391     Tou%=1
392   ENDIF
393 RETURN
394 *

```

```

395 DEF PROC Sauvegarder(Numero%)
396   'sauvegarder l'enregistrement 'numero%' sur disquette
397   OPEN "R".1,Nom$.117
398   FIELD 1.15 AS No$.15 AS Pr$.31 AS Ru$.28 AS Po$.27 AS Rest$
399   'ici aussi, la largeur de l'imprimante nous contraint à réunir
400   'plusieurs champs sous une même variable
401   LSET No$=Nom$(Numero%)
402   LSET Pr$=Prenom$(Numero%)
403   LSET Ru$=Rue$(Numero%)
404   LSET Po$=Postal$(Numero%)
405   'impossible de faire autrement
406   MID$(Po$.5)=Ville$(Numero%)
407   LSET Rest$=Tel$(Numero%)
408   MID$(Rest$.12)=Noles$(Numero%)
409   'puis sauvegarder sur la disquette
410   PUT 1,Numero%
411   CLOSE 1
412   RETURN
413
414 DEF PROC Baptiser(R Touche%)
415   IF Nom$="" THEN
416     Nom$="FICHIER.REL"
417     Chemin$="A:\":Ext$="*.REL"
418     Do_File(Nom$,Chemin$,Ext$,Nom$,Chemin$,Ext$,Touche%)
419     IF Touche% AND Nom$<>"" THEN
420       CHDIR Chemin$
421     ENDIF
422   ELSE
423     Touche%-1
424   ENDIF
425   RETURN
426
427 DEF FN Exist%L(Nom$)
428   LOCAL T%
429   '
430   OPEN "F".1,Nom$.4
431   IF EOF(1) THEN
432     'fichier non trouvé sur la disquette
433     '-> envoyer le message d'erreur (0)
434     CLOSE 1
435     RETURN (0)
436   ELSE
437     'préparer un buffer DTA
438     FIELD 1.30 AS Buffer$.14 AS File$
439     GET 1,0
440     'fichier bien retrouvé -> retourner valeur '-1'
441     IF Nom$= LEFT$(File$, INSTR(File$, CHR$(0))-1) THEN
442       CLOSE 1
443       RETURN (-1)
444     ENDIF
445   ENDIF
446   CLOSE 1
447   'le fichier n'existe pas
448   RETURN (0)
449 DEF PROC Rename(Filename$)
450   IF FN Exist%L(Filename$) THEN
451     'le fichier existe bien -> est-il déjà flanqué d'un back-up?
452     IF FN Exist%L( LEFT$(Filename$, INSTR(Filename$, ".")+ "BAK") )
453       'effacer le back-up existant ...

```



```
454         THEN KILL LEFT$(Filename$, INSTR(Filename$, ".")+"BAK"  
455     ENDIF  
456     ' confectionner un back-up  
457     BACKUP (Filename$)  
458 ENDIF  
459 RETURN
```

3.12. Pourquoi réinventer la roue ?

Certes, la programmation est une belle chose, mais pourquoi se casser la tête sur des problèmes que d'autres ont déjà solutionnés ? Ou encore : pourquoi réécrire dans chaque programme les mêmes fonctions lorsqu'elles se sont avérées irréprochables ? Ne serait-il pas plus judicieux de sauvegarder dans des fichiers particuliers certaines fonctions très utiles, utilisées et testées dans des programmes professionnels, de façon à pouvoir les réinsérer dans n'importe quel autre programme ultérieurement ? Voilà qui nous épargnerait une masse considérable de travail !

On appelle 'bibliothèques' ('libraries') ces fichiers contenant toutes sortes de fonctions qu'on appelle au fur et à mesure des besoins.

Ces bibliothèques sont très répandues dans le monde des langages compilés : que serait le langage C sans ses bibliothèques ? quasiment rien, puisqu'il n'y aurait même pas une instruction pour entrer et sortir des données ! Et bien que l'Omikron Basic brille par le nombre de ses instructions, le recours à des bibliothèques facilitera considérablement l'écriture de bons programmes. C'est pourquoi vous trouvez, livrée avec l'interpréteur, une bibliothèque GEM, qui vous permet d'appeler un certain nombre de fonctions du GEM : nous y reviendrons plus loin. Sachez seulement qu'en tant que programmeur, vous n'avez plus à vous soucier de savoir comment programmer telle ou telle fonction GEM : il vous suffira de les reprendre dans la bibliothèque.

Pour conclure ce chapitre traitant de la gestion des fichiers, je voudrais vous faire cadeau d'une bibliothèque très utile : elle vous simplifie la manipulation de la boîte de sélection d'objet, vous permet de confectionner des fichiers back-up et contrôle si un fichier donné se trouve déjà ou non sur la disquette.

Comment travailler avec une telle bibliothèque ? c'est très simple :

- commencez par écrire votre programme, dans lequel vous pouvez inclure des appels de fonctions se trouvant dans la bibliothèque ;
- comme ces fonctions sont indispensables lorsque l'interpréteur exécute le programme, vous les ajoutez purement et simplement à la fin de votre texte : les fonctions sont définies et donc exécutables ;
- sauvegardez le programme ainsi obtenu sur une disquette, avec la bibliothèque : votre programme devient un programme normal, exécutable dès son chargement.

Lorsque vous réunissez les deux parties de programme, vous devez respecter les règles suivantes :

- pour réunir les deux programmes, utilisez l'instruction `MERGE <nom_du_programme>` ; la mémoire vive doit à ce moment déjà contenir la première partie du programme, qui sera en général celle que vous avez écrite vous-même et qui fait appel aux fonctions contenues dans la bibliothèque ;
- l'instruction `MERGE <nom_du_programme>` se charge d'ajouter le programme (en fait la bibliothèque) désigné à la fin du programme déjà présent en mémoire vive ;
- le terme 'ajouter' est un peu abusif, car les deux parties sont véritablement 'soudées' l'une à l'autre : en effet, l'instruction `MERGE` insère le programme rechargé (en code ASCII, donc auparavant sauvegardé par `SAVE,A`) dans le programme présent en mémoire vive en respectant la numérotation des lignes. Voici un petit exemple concret pour rendre cela plus clair :

Admettons que le programme suivant se trouve en mémoire vive :


```

10 <ligne 1>
20 <ligne 2>
44 <ligne 3>
50 <ligne 4>
60 END

```

et que le programme <Fichier.BAS> se trouve lui sur la disquette :

```

1 <Fichier.BAS, ligne 1>
2 <Fichier.BAS, ligne 2>
11 <Fichier.BAS, ligne 3>
22 <Fichier.BAS, ligne 4>
51 <Fichier.BAS, ligne 5>

```

Voilà l'aspect des deux fichiers 'fondus' à l'aide de l'instruction MERGE :

```

1 <Fichier.BAS, ligne 1>
2 <Fichier.BAS, ligne 2>
10 <ligne 1>
11 <Fichier.BAS, ligne 3>
20 <ligne 2>
22 <Fichier.BAS, ligne 4>
44 <ligne 3>
50 <ligne 4>
51 <Fichier.BAS, ligne 5>
60 END

```

C'est pourquoi il faut soigneusement veiller à ce que les deux programmes ne contiennent pas les mêmes numéros de ligne ! La bibliothèque 'FIC_LIB.BAS' commence par la ligne 1000. Si vous souhaitez l'ajouter à la fin d'un programme comportant plus de 1000 lignes, prenez soin de la renuméroter à l'aide de l'instruction RENUM avant de la charger. N'oubliez pas que le texte du programme sur disquette doit se trouver en format ASCII pour que vous puissiez recourir à MERGE. Voici maintenant la bibliothèque en question :

```

1000 *****
1010 **                FIC_LIB.BAS                *
1020 **-----**
1030 ** Auteur: Michael Maier      Version 1.00      Date: 15.07.1990  *
1040 **   Programme joint au livre de l'Omikron Basic                *
1050 **                (c) MICROAPPLICATION                *
1060 *****
1070 *
1080 *
1090 *voici des procédures et fonctions que vous pouvez réinsérer dans
1100 *vos programmes, ce qui vous évite de devoir toujours tout
1110 *reprendre depuis la Genèse
1120 *
1130 * Procédure DO_FILE(): gestion de la boîte de sélection d'objet
1140 *-----*
1150 *
1160 * Paramètres nécessaires:  Nom$: nom du fichier, à entrer dans le
1170 *                           champ de saisie 'SELECTION' après
1180 *                           avoir appelé la fonction
1190 *                           Path$: chemin d'accès souhaité ; si vous
1200 *                           n'entrez aucun nom, la fonction
1210 *                           reprend le chemin actuel
1220 *                           Post$: extension du chemin: ("*.*)"
1230 *                           Les trois autres paramètres renvoient les
1240 *                           chaînes de caractères sélectionnées
1250 *                           Tou$: numéro du 'button' sur lequel
1260 *                           l'utilisateur a cliqué
1270 *                           '0' -> ANNULER
1280 *                           '1' -> CONFIRMER
1290 *
1300 *
1310 DEF PROC Do_File(Nom$,Path$,Post$,R_Nom$,R_Path$,R_Post$,R_Tou%)
1320   'utiliser des variables locales afin de pouvoir intégrer cette
1330   'procédure dans d'autres programmes.
1340   LOCAL R_Path$=Path$,R_Nom$=Nom$.TX,Drive%,Pointer%L,Ret%L
1350   IF Path$="" THEN 'si aucun chemin indiqué, en élaborer un
1360     Path$=" "*64 'en respectant les conventions GEMDOS
1370     Pointer%L= LPEEK( VARPTR(Path$))+ LPEEK( SEGPTR +28)
1380     ' après quoi chercher le chemin actuel
1390     GEMDOS (,71, HIGH(Pointer%L), LOW(Pointer%L),0)
1400     ' puis retrancher correctement
1410     Path$= LEFT$(Path$, INSTR(Path$+ CHR$(0), CHR$(0))-1)
1420     GEMDOS (Drive%,25) ' identificateur de l'unité de disque activée
1430     Path$= CHR$(65+Drive%)+": "+Path$+"\ "
1440   ENDIF
1450   IF Post$="" THEN 'on se passe difficilement d'une extension
1460     Path$=Path$+"*.*)" ' ajouter une extension
1470   ELSE
1480     Path$=Path$+Post$
1490   ENDIF
1500   PRINT CHR$(27);"f" 'désactiver le curseur et activer la souris
1510   MOUSEON
1520   FILESELECT (Path$,Nom$,Ret%L)
1530   MOUSEOFF
1540   PRINT CHR$(27);"e" 'réactiver le curseur
1550   IF Ret%L=0 THEN ' sélection de ANNULER
1560     Path$=R_Path$ ' reprendre l'ancien chemin
1570     Nom$=R_Nom$
1580     Tou%=0

```



```

1590 ELSE
1600   ' rechercher le premier backslash '\'
1610   TX= LEN(Path$)
1620   WHILE TX>0 AND MID$(Path$,TX,1)<>"\"
1630     TX=TX-1
1640   WEND
1650   Path$= LEFT$(Path$,TX)
1660   Tou%=1
1670 ENDIF
1680 RETURN
1690 '
1700 '*****
1710 '
1720 ' Fonction FN EXIST(<nom de fichier>)
1730 ' -----
1740 '
1750 ' Cette fonction sert à vérifier s'il existe déjà sur la
1760 ' disquette un fichier 'Nom$'
1770 ' fichier existe      => valeur retournée = '-1' (vrai)
1780 ' fichier n'existe pas => valeur retournée = '0' (faux)
1790 '
1800 ' appel de la fonction: [<variable>] = FN EXIST(<nom de fichier>)
1810 ' -----
1820 '
1830 DEF FN Exist%L(Nom$)
1840   LOCAL TX
1850   '
1860   OPEN "F",1,Nom$,4
1870   IF EOF(1) THEN
1880     'fichier non trouvé sur la disquette
1890     '=> envoyer le message d'erreur (0)
1900     CLOSE 1
1910     RETURN (0)
1920   ELSE
1930     ' préparer un buffer DTA
1940     FIELD 1,30 AS Buffer$,14 AS File$
1950     GET 1,0
1960     'fichier bien retrouvé => retourner valeur '-1'
1970     IF Nom$= LEFT$(File$, INSTR(File$, CHR$(0))-1) THEN
1980       CLOSE 1
1990       RETURN (-1)
2000     ENDIF
2010   ENDIF
2020 CLOSE 1
2030 'le fichier n'existe pas
2040 RETURN (0)
2050 '
2060 '*****
2070 '
2080 ' Procédure: RENAME(<nom de fichier>)
2090 ' -----
2100 '
2110 ' La procédure RENAME contrôle si un fichier <nom de fichier>
2120 ' existe effectivement sur la disquette
2130 ' Si tel est le cas, il en confectionne un back-up, et supprime
2140 ' un éventuel back-up de même nom déjà présent
2150 '
2160 ' ATTENTION ! pour que cette procédure fonctionne, il faut qu'elle
2170 ' soit accompagnée de la fonction EXIST() de cette bibliothèque

```

```

2180 *
2190 * Pour appeler cette procédure: Rename(<nom de fichier>)
2200 * -----
2210 *
2220 DEF PROC Rename(Filename$)
2230   IF FN Exist%L(Filename$) THEN
2240     ' le fichier existe bien => est-il déjà flanqué d'un back-up ?
2250     IF FN Exist%L( LEFT$(Filename$, INSTR(Filename$,".")+ "BAK") )
2260       ' effacer le back-up existant ...
2270       THEN KILL LEFT$(Filename$, INSTR(Filename$,".")+ "BAK"
2280     ENDIF
2290     ' confectionner un back-up
2300     BACKUP (Filename$)
2310   ENDIF
2320 RETURN

```

3.13. Noir sur blanc : la sortie impression

Jusqu'ici, nous avons dirigé toutes nos sorties de données soit vers l'écran du moniteur soit vers une disquette après avoir ouvert un canal à l'aide de l'instruction OPEN. Il arrive cependant fréquemment que l'on ait besoin de disposer d'une impression des données sur papier. Comment faire ?

Il vous suffit de retourner quelques pages en arrière, au début de ce chapitre, pour trouver l'instruction

```
OPEN "P", <numéro de canal>
```

qui permet d'ouvrir un canal faisant transiter les données vers l'imprimante :

```

OPEN "P", 4
PRINT #4, "voici une ligne à sortir sur l'imprimante !"
CLOSE 4

```

Il semblerait qu'il n'y ait pas grand chose à ajouter sur ce sujet, mais en fait il existe encore d'autres instructions permettant de sortir des données sur l'imprimante. Signalons d'abord

LPRINT

qui correspond à PRINT, mais permet d'envoyer les données correspondantes immédiatement vers l'imprimante, exemple :

LPRINT "voici une ligne à sortir immédiatement sur l'imprimante"

Il est inutile d'ouvrir auparavant un canal par OPEN, car la sortie des données se fait toujours par le port d'impression (connexion de l'imprimante) lorsqu'intervient cette instruction, qui a d'ailleurs encore d'autres avantages.

Les imprimantes EPSON et compatibles EPSON sont paramétrées en fonction du standard américain, si bien qu'elles rencontrent toujours quelques problèmes lors de la restitution des caractères propres à des alphabets particuliers, comme par exemple en français les caractères accentués (éèàù) ou le caractère 'ç'. Ceux-ci sont remplacés par des alinéas (ouvrant ou fermant) ou tout simplement ignorés ! Les logiciels de traitement de texte remédient à ce problème en transformant les codes ASCII des caractères problématiques avant de les envoyer vers l'imprimante. Mais vous, en tant que programmeur, vous travaillez en BASIC et non sous un traitement de texte ! Bien sûr, vous pourriez d'abord sortir vos textes vers un fichier de sauvegarde sur disquette, pour ensuite les reprendre et les imprimer à partir d'un traitement de texte, mais cela semble bien fastidieux ! Alors que faire ?

La solution se trouve dans votre manuel de l'Omikron Basic : il suffit de recourir à l'instruction MODE LPRINT et d'y sélectionner le mode correspondant à l'alphabet national souhaité (pour nous, le français, mais on peut demander d'autres alphabets) en écrivant par exemple :

MODE LPRINT "F"

les caractères spéciaux français sont ainsi transcodés avant d'être envoyés vers l'imprimante. Remarquez bien que cette instruction n'influence que LPRINT et pas les autres fonctions : ainsi, une sortie de données engendrées par PRINT# n'en sera pas affectée. Si votre imprimante, après une instruction LPRINT, ne restitue qu'un texte tronqué, je vous recommande de repasser dans un autre alphabet, comme par exemple GB ou USA.

Vous pouvez aussi décider de sortir toutes vos données (que ce soit par exemple à l'aide de PRINT ou LPRINT) vers un fichier <numéro de canal> en utilisant

```
CMD<numéro de canal>
```

En effet, si vous souhaitez laisser le choix à l'utilisateur entre une sortie sur écran (ce qui économise le papier et le ruban) et une sortie sur imprimante, vous devriez en principe écrire deux routines différentes : l'une destinée à l'affichage sur moniteur et dans laquelle ne figurent que des instructions PRINT, et l'autre assurant la sortie-impression, ne comprenant que des instructions LPRINT. En écrivant

```
PRINT "Bonjour"
```

vous voyez immédiatement apparaître le mot 'Bonjour' sur votre écran ; il vous suffit de détourner la sortie des données pour que la même instruction fasse sortir le mot 'Bonjour' sur votre imprimante :

```
OPEN "P",1      ' d'abord ouvrir le canal de
                  l'imprimante
PRINT "Bonjour" ' et vive l'écran du moniteur
CMD 1           ' à partir de là, les données seront
                  détournées
                  vers l'imprimante et
PRINT "Bonjour" ' sera imprimé sur papier
CLOSE 1
```

L'instruction CMD vous permet aussi de détourner les données vers un fichier disque pour pouvoir les reprendre ensuite sous un traitement de texte : on utilise pour cela LLIST (fonctionne comme LIST mais en transmettant les données vers l'imprimante) en détournant les données vers un fichier ouvert auparavant :

```
OPEN "O".4."FICHIER.BAS"
CMD 4
LLIST           ' vous pouvez préciser par exp 100-200 si
                  vous ne
                  souhaitez imprimer qu'un morceau du texte
CLOSE 4
```


Il est préférable d'utiliser l'instruction **LLIST** (et non **LIST**) car cette instruction permet de ne pas sauvegarder dans le fichier les codes de commande propres à l'affichage-écran, qui pourraient contrarier le logiciel de traitement de texte sous lequel vous allez reprendre votre texte de programme. Il en va d'ailleurs de même pour toutes les autres instructions que vous pouvez faire précéder d'un 'L' lorsqu'il s'agit de sortir vers l'imprimante (exp : **DUMP** et **LDUMP**).

L'instruction **PRINT @(ligne,colonne)** vous permet de positionner exactement votre texte à l'écran. Le caractère arrobase ('@') représente un code de commande pour l'affichage à l'écran, qui se charge du positionnement du curseur. Par contre, vous ne pouvez écrire **LPRINT @()** puisque l'imprimante ne reconnaît pas les mêmes codes de commande que ceux du moniteur. Elle identifie par contre d'autres codes permettant la mise-en-page d'un texte.

Tous les codes de commande propres aux imprimantes **EPSON** ou compatibles **EPSON** commencent par une séquence **ESCAPE**, qui avertit l'imprimante qu'elle va bientôt recevoir des données importantes, comme par exemple un changement de type d'impression.

Vous pouvez par exemple activer le type de caractères 'elite' et la largeur 12 CPI (= caractères par inch) en utilisant le code de commande

`CHR$(27);CHR$(77);` ou encore `CHR$(27);"M";`

que vous transmettez ensuite à l'imprimante, par **LPRINT** ou **PRINT#** etc... Notez que sur une imprimante **NEC**, ce même code de commande ne bascule pas l'appareil en mode 'elite' mais modifie bien la largeur des caractères, qui passe sur 12 CPI (caractères par inch).

3.14. Equation de recherche complexe avec OR et AND

Cette version de notre programme de gestion du fichier d'adresse ne vous permet ensuite de ne rechercher les adresses que par le nom de famille ; si votre fichier contient plusieurs homonymes (plusieurs DUPOND par exemple) vous devrez le 'feuilleter' fiche par fiche jusqu'à retrouver la bonne adresse. Dans une application professionnelle, il faudrait pouvoir au moins préciser le prénom. Le programme rechercherait alors une fiche qui porte le même nom et le même prénom dans les deux champs respectifs :

```
REPEAT
....
....
UNTIL Nom$(0) = Nom$(T%) AND Prenom$(0) = Prenom$(T%)
```

Il faudrait aussi disposer de la possibilité de retrouver toutes les fiches portant le nom 'DUPONT' ou le prénom 'Alfred' : on utilise alors l'opérateur OR

```
REPEAT
...
...
UNTIL Nom$(0) = Nom$(T%) OR Prenom$(0) = Prenom$(T%)
```


Chapitre 4

Le système d'exploitation de l'Atari ST

Bien que l'Omikron Basic offre une foule d'instructions, les programmeurs ne peuvent se passer de recourir parfois au système d'exploitation de l'Atari-ST. Il est par exemple impossible d'intégrer dans un programme le formatage d'une disquette ou la lecture du répertoire d'un disque sans passer par le système d'exploitation (operating system).

Qui plus est, l'Omikron Basic contient certaines variables dont on ne peut qu'interroger le contenu (il est impossible de leur passer une valeur) et qui vous renseignent sur la position du pointeur de la souris, la date et heure-système, la ligne sur laquelle se trouve le curseur etc. Nous allons maintenant étudier tout cela plus en détail.

4.1. Les variables-système

En Omikron Basic, ces variables-système portent des noms 'réservés' et vous ne pouvez leur passer des valeurs, contrairement aux variables ordinaires. Certaines personnes les désignent aussi sous le nom de 'fonctions' puisque ces variables-système retournent une valeur. C'est un abus de langage, qui peut être source de confusion :

en effet, une fonction se voit affecter un argument (placé entre parenthèses juste après le nom de la fonction) servant à effectuer certains calculs, ce qui n'est pas le cas pour les variables-système.

□ MOUSEX

Cette variable contient la position horizontale momentanée du curseur de la souris (ordonnée X) ; le moindre déplacement horizontal de la souris fait automatiquement varier son contenu.

□ MOUSEY

Cette variable contient la position verticale momentanée du curseur de la souris (abscisse Y).

□ MOUSEBUT

La souris ne fait pas seulement varier la position de son curseur : elle est également munie de deux touches, en anglais : 'mousebuttons'. La variable-système **MOUSEBUT** vous renseigne sur l'état de ces deux touches de la souris :

Contenu de MOUSEBUT	Signification
0	aucune touche appuyée
1	appui sur la touche gauche
2	appui sur la touche droite
3	appui simultané sur les deux touches

A quoi servent ces informations ? Imaginez que vous ayez confectionné une 'image' à l'aide d'un programme graphique, image que vous entendez réutiliser comme masque dans un autre programme. Ce masque contient - comme dans GEM - des petits rectangles entourant des symboles : l'utilisateur doit cliquer sur ces symboles pour lancer telle ou telle fonction. A l'aide des

variables-système, votre programme peut repérer sur quel symbole s'est produit le clic ; une boucle permet d'attendre que l'utilisateur appuie sur l'une des touches de la souris :

```
REPEAT
```

```
...
```

```
UNTIL MOUSEBUT=1
```

La boucle vide suffirait par elle-même, car vous pourriez récupérer ultérieurement les coordonnées du curseur de la souris à l'aide de MOUSEX et MOUSEY. Mais que se passerait-il lorsqu'un utilisateur cliquerait rapidement sur une des cases pour déplacer immédiatement la souris ? Les variables-système retourneraient alors des valeurs fausses. C'est pourquoi il est plus prudent de confier à une autre variable, tout de suite dans la boucle, les coordonnées du curseur de la souris. Qui plus est, il est plus rapide d'écrire X et Y dans la routine guettant l'état des touches de la souris que de les reprendre ensuite dans les variables-système :

```
REPEAT
```

```
Y%=MOUSEY
```

```
X%=MOUSEX
```

```
UNTIL MOUSEBUT=1
```

puis de guetter ce qui se passe dans un rectangle bien déterminé (contenant le symbole), possédant par exemple les coordonnées X 100,150 et Y 30,90, dont les sommets occupent donc les positions suivantes :

coin supérieur gauche (100,30)

coin supérieur droit (150,30)

coin inférieur gauche (100,90)

coin inférieur droit (150,90)

```
IF X%>= 100 AND X% <= 150 THEN
```

```
IF Y% >= 30 AND Y% <= 90 THEN
```

```
GOSUB Sous_routine
```

```
ENDIF
```

```
ENDIF
```

Les conditions IF servent à tester si le curseur de la souris se trouve à l'intérieur du champ délimité par certaines coordonnées, c'est-à-dire à l'intérieur du rectangle contenant le symbole. A l'intérieur : cela signifie que la position X du curseur de la souris est supérieure à la limite gauche et inférieure à la limite droite du rectangle (d'où l'utilisation du AND logique) ; il faut ensuite tester de la même manière les coordonnées Y du curseur de la souris pour établir avec certitude que le curseur se trouve à l'intérieur du rectangle.

□ TIMER

La variable TIMER contient l'heure système (heure-minute seconde) qui s'est écoulée depuis que vous avez allumé votre configuration (en 1/200ème de seconde). Vous utilisez cette variable lorsque vous souhaitez effectuer un calcul se répétant durant une certaine durée :

```
Depart=TIMER      'consigner l'heure de début
FOR T%=1 TO 30000
  PRINT T%
NEXT T%
PRINT (TIMER-Depart)/200 'durée écoulée en secondes
```

□ TIME\$

Il est impossible, avons-nous dit, d'affecter une valeur à une variable-système ! mais voilà, il n'y a pas de règle sans exception, comme nous allons le voir tout de suite. TIME\$ retourne l'heure-système actuelle sous la forme "HH:MM:SS", soit deux chiffres pour les heures (numérotées jusqu'à 24) puis deux pour les minutes et deux pour les secondes, chaque indication étant séparée de la suivante par un double-point.

Pour voir l'heure-système actuelle sur votre écran, vous écrivez et envoyez

```
PRINT TIME$
```

mais si vous ne possédez pas un MEGA-ST (qui sont équipés d'une horloge autonome avec pile, mesurant en continu le temps réel) vous devez penser d'abord à mettre votre configuration à l'heure à chaque

réinitialisation du système. En effet, l'heure et la date système sont sauvegardés à chaque sauvegarde de fichier sur le disque ou la disquette, avec le nom de fichier : si vous ne mettez pas cela à jour lors de la mise en route de la configuration, le système reprend les valeurs par défaut, qui sont forcément fausses.

C'est pourquoi il est possible de passer une valeur à la variable-système `TIME$` (de même qu'à la variable `DATE$`) pour qu'elle contienne bien l'heure actuelle. Par exemple, pour mettre la configuration sur 10H30mn, vous écrivez :

```
TIME$="10:30:00"
```

cela suffit, et ne modifie pas le contenu de `TIMER`.

□ `DATE$`

Alors que `TIME$` fournit l'heure-système, `DATE$` fournit la date, selon le format spécifié. En mode "D" (allemand) ou "F" (français) la date s'écrit sous la forme "JJMAA" (deux chiffres pour le Jour, deux pour le Mois et deux pour l'Année) ; en mode "USA", vous obtenez le format MMJJAA (mois-jour-année). Vous pouvez affecter une date à la variable-système `DATE$`.

□ `PI`

Cette constante représente le nombre 3,14159265 etc., très usuel en mathématiques. La constante `PI` est traitée comme un nombre décimal à double précision. Vous pouvez ramenez les calculs à une précision simple en attribuant cette constante à une variable single-float :

```
PII=PI
```

et continuer vos calculs avec `PI!` ou définir un `PI` comme une variable simple :

```
CSNG(PI)
```

□ CSRLIN

Cette variable représente la ligne sur laquelle se trouve momentanément le curseur d'écran : la ligne supérieure étant désignée par 1 et la ligne inférieure par le numéro 25.

□ ERR, ERL, ERR\$

vous connaissez déjà toutes ces variables : elles nous ont servi au traitement des erreurs (error handling) en Omikron Basic, elles fournissent le numéro de l'erreur commise (ERR), la ligne à laquelle elle survint (ERL) et enfin le message en toutes lettres (ERR\$).

4.2. TOS, GEMDOS, BIOS et XBIOS

Rassurez-vous, il ne s'agit pas d'un extrait de l'annuaire téléphonique chinois, mais d'abréviations désignant certaines parties du système d'exploitation de l'Atari-ST. Mais qu'est-ce qu'un système d'exploitation ? L'amateur novice considère comme une évidence de pouvoir provoquer telle ou telle réaction rien qu'en appuyant sur certaines touches, de voir son curseur tourner sur l'écran en déplaçant frénétiquement sa souris, ou de pouvoir 'ouvrir' une fenêtre (comme c'est le cas avec l'Atari-ST) en effectuant un double-clic sur telle icône, fenêtre qu'il peut ensuite agrandir, rétrécir ou déplacer.

Le coeur de l'ordinateur est le processeur : ce dernier peut certes jongler avec les bits et les octets en les envoyant dans la mémoire vive et il maîtrise même la manipulation de bits isolés ou d'opérations arithmétiques. Mais dès qu'il s'agit d'entrer ou de sortir des données, il est obligé de recourir aux services de certains composants périphériques, que ce soit le floppycontroller (qui contrôle les allers-venues des données avec le lecteur de disquette) ou d'autres composants se chargeant de lui apporter des données puis de les réexpédier vers un périphérique de sortie.

Les constructeurs d'ordinateurs doivent donc en quelque sorte les 'éduquer' et leur apprendre toutes ces manoeuvres de base : l'ordinateur doit savoir comment s'adresser à telle ou telle interface et comment traiter une réponse fournie par un des périphériques. C'est justement le rôle du système d'exploitation, appelé TOS (Tramiel Operating System) en l'honneur de son concepteur, Jack Tramiel, directeur de Atari.

A partir de la gamme des ST, ce système d'exploitation TOS est enregistré dans des ROM (read-only-memory) : il s'agit de mémoires non-volatiles, montées dans l'ordinateur, dont le contenu ne peut être effacé même lorsque vous coupez le courant brutalement. Le TOS en lui-même se compose d'un dérivé de CP/M-68K (quel nom !) et du GEM (graphics environment manager = gestionnaire d'environnement graphique). Le dérivé CP/M se charge de la gestion des communications avec l'extérieur, le clavier étant aussi considéré comme un périphérique. Par rapport à son prédécesseur le CP/M-68K, ce dérivé a été enrichi de quelques fonctionnalités supplémentaires pour que l'ordinateur puisse gérer l'interface MIDI et la souris.

Comme tous les autres systèmes d'exploitation (y compris le MS-DOS des compatibles IBM), celui de l'Atari se divise en une partie 'généraliste' et une partie plus spécifique. La partie 'généraliste' du ST s'appelle GEMDOS (GEM-Disc operating system, appellation prêtant à confusion, car ce GEM n'a absolument rien de commun avec l'autre GEM de gestion graphique d'environnement) tandis que le BIOS (Basic input output system) s'occupe de la partie plus spécialement consacrée à ce type de configuration. D'autres fonctions Atari sont encore contenues dans le troisième module, appelé XBIOS (eXtended Basic Input Output system).

La partie GEM du système d'exploitation se compose du 'bureau GEM' (ou bureau utilisateur), des programmes accessoires (accessories), du GEM-VDI (virtual device interface) et du GEM-AES (application environment system). Le VDI gère toutes les fonctions graphiques élémentaires, comme le dessin d'une ligne, d'un cercle, l'épaisseur du trait ou le motif de remplissage de l'arrière-fonds etc.

L'AES gère par contre la communication graphique entre l'homme et la machine. Ce sont les routines AES qui 'construisent' le bureau GEM tel qu'il apparaît à l'écran juste après la mise sous tension de la configuration Atari. Le GEM, adaptation graphique du TOS, transforme les instructions de l'utilisateur en codes de commande compréhensibles pour le TOS. A l'intérieur du TOS, le GEMDOS occupe une place prééminente par rapport aux deux autres modules : il est chargé de l'exécution des instructions en question, alors que BIOS et XBIOS sont en quelque sorte ses deux esclaves. C'est là tout ce que je voulais vous dire au sujet du système d'exploitation (reportez-vous aussi au schéma explicatif 4.1). Dans ce chapitre, nous allons surtout nous occuper du TOS, GEM viendra plus tard.

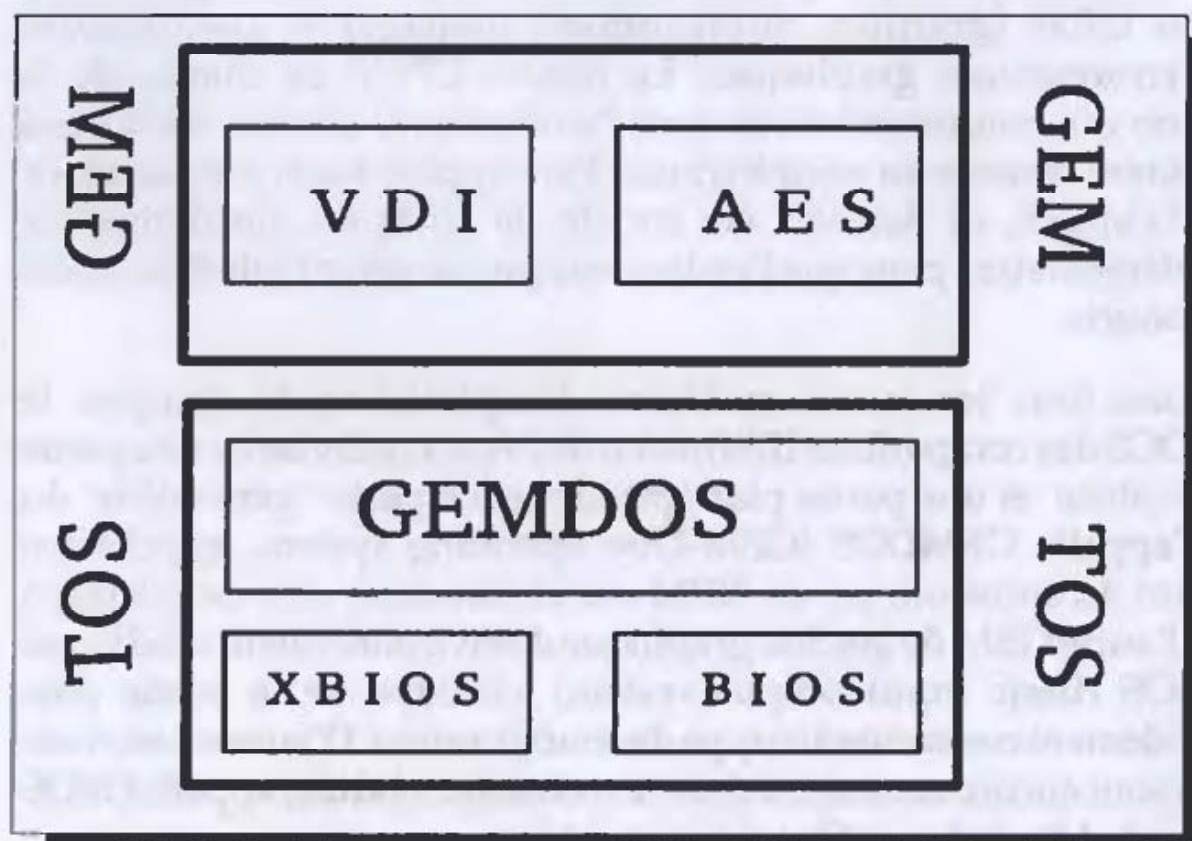


Figure 4.1 : structure du système d'exploitation de l'Atari ST.

4.3. GEMDOS

Pour appeler une fonction GEM, vous écrivez

GEMDOS([Valeur_retour],<Numero de fonction>,[Paramètre])

Le GEMDOS contient un grand nombre de routines identifiables grâce à des numéros compris entre 0 et 87 ; si nécessaire, vous pouvez aussi entrer un ou plusieurs paramètres (séparés par des virgules). Voici les fonctions utilisables sous Omikron Basic (R désigne la valeur en retour) :

GEMDOS(R,2,caractère)

Cconout

Cette routine sert à afficher un caractère à l'écran ; il est préférable de se servir de l'instruction PRINT.

GEMDOS(R,3)

Cauxin

Entrée d'un caractère provenant de l'interface série RS232.

GEMDOS(R,4,caractère)

Cauxout

Sortie d'un caractère vers l'interface série.

GEMDOS(R,5,caractère)

Cprnout

Sortie d'un caractère vers l'imprimante.

GEMDOS(R,10,HIGH(buffer),LOW(buffer))

Cconrs

Permet de lire une ligne dans le buffer, qui n'est en fait qu'un pointeur vers le véritable buffer de saisie ; les deux premiers octets contiennent la longueur maximale de la saisie ainsi que le nombre de caractères effectivement entrés ; la combinaison des touches <control>1<C> provoque une interruption du programme durant l'exécution de cette fonction ; voici un exemple :

```

Saisie(10, Contenu$)
...
END
DEF PROC Saisie (Longueur%, R Texte$)
  Local Adr
  ' longueur maxi de la saisie, et reste de la chaîne de
  caractères
  Texte$=CHR$(Longueur%)+CHR$(0)*(Longueur%+2)
  ' calcule l'adresse de la chaîne de caractères
  Adr = Lpeek( VARPTR(Texte$))+Lpeek(SEGPTR+28)
  GEMDOS(.10,HIGH(Adr),LOW(Adr))
  Texte$=MID$(Texte$,3,ASC(MID$(Texte$,2)))
RETURN

```

Cette procédure permet de lire une chaîne de 10 caractères maximum et de la repasser au programme principal ; on trouve, avant le premier caractère, la longueur maximale admise pour la saisie suivie, dans le deuxième octet, du nombre de caractères effectivement saisis. Avant l'appel de la routine, la longueur maxi de saisie vient s'inscrire dans le premier octet de la chaîne ; le restant de la chaîne de caractères (2 + longueur maxi) est occupé temporairement par un caractère de remplissage pour éviter que des secteurs importants de la mémoire ne soient écrasés par erreur par d'autres données. Comme la routine attend un pointeur vers ce string, elle se charge de le calculer et de le mémoriser dans 'Adr'. Une fois la chaîne saisie, la routine renvoie au programme principal la chaîne commençant à la troisième position et d'une longueur identique à celle qui est indiquée dans le 2ème octet.

GEMDOS(R,14,unité de disque)

Dsetdrv

Définition de l'unité de disque(tte) activée, selon le tableau suivant :

0	Disque A (unité de disquette incorporée à l'Atari)
1	Disque B (unité de disquette externe)
2	Disque C (généralement : disque dur)

R contient le numéro de l'unité de disque active avant l'appel de la fonction ; tout cela équivaut à la fonction

CHDIR CHR\$(65+Unité de disque)+":."

GEMDOS(R,16)

Cconos

Vérifie si l'écran est en état d'afficher des données ; comme c'est en principe toujours le cas, R contient la valeur \$FFFF ; lorsque ce n'est pas le cas (ce qui n'arrive en principe jamais) la fonction retourne la valeur 0.

GEMDOS(R,17)

Cprnos

Vérifie si l'imprimante est en état d'imprimer des données ; lorsque ce n'est pas le cas la fonction retourne la valeur 0.

GEMDOS(R,18)

Cauxis

Retourne la valeur 0 lorsque l'interface série ne délivre aucun caractère.

GEMDOS(R,19)

Cauxos

Retourne la valeur 0 lorsqu'aucun caractère ne peut transiter par l'interface série.

GEMDOS(R,25)

Dgetdrv

Retourne le numéro de l'unité de disque(tte) active (voir Dsetdrv).

GEMDOS(,26,HIGH(buffer),LOW(buffer))

Fsetdta

Détermine l'adresse du buffer DTA (disc transfer adress) ; ce buffer de 44 octets sert de mémoire tampon lors des opérations exigeant l'accès à une unité de disque(tte), comme par exp. la lecture du répertoire ; voici son contenu :

Octet	Contenu
0-20	réservé au GEMDOS
21	attribut du fichier
22-23	heure-système de création du fichier
24-25	date-système de création du fichier
26-29	taille du fichier
30-43	nom du fichier

On peut aussi lire ces 44 octets en entrant 'GET 1,enregistrement' après OPEN "F",1,"*.*" mais il faut alors installer un buffer correspondant à l'aide de FIELD 1,... à partir duquel il est possible de reprendre les données ; cette méthode est peu pratique en Omikron Basic.

GEMDOS(R,47)

Fgetdta

Cette fonction retourne l'adresse du buffer DTA.

GEMDOS(R,48)

Sversion

Retourne le numéro de la version du système d'exploitation.

GEMDOS(R,54,HIGH(buffer),LOW(buffer),Drv)

Dfree

Equivaut largement à l'instruction FRE(CHR\$(65+Drv)+":") et sert à calculer la place encore disponible sur le disque ou la disquette se trouvant dans le lecteur Drv ; cette fonction retourne un pointeur dirigé vers un buffer contenant quatre mots-longs :

- nombre de clusters encore libres (b_free)
- nombre total de clusters sur le disque (b_total)
- nombre d'octets par secteur (b_sectsize)
- nombre de secteurs par cluster (2) (b_clsize)

Ceci permet donc de calculer aussi bien la place encore disponible que la place déjà occupée sur le disque :

place disponible $b_free * b_seclsize * b_clsize$
 capacité du disque $b_total * b_seclsize * b_clsize$
 place occupée capacité du disque - place encore libre

Drv contient le numéro de l'unité de disque dont on calcule la place encore disponible, selon le tableau suivant :

0	unité de disque actuelle (par défaut)
1	unité A
2	unité B
..	
...	
etc	

GEMDOS(R,57,HIGH(adresse),LOW(adresse)

Dcreate

Sert à installer un dossier (subdirectory) sur le disque ; le nom du dossier comprend au maximum 8 caractères plus 3 caractères d'extension (séparés des huit premiers par un point). On peut aussi indiquer le chemin d'un dossier déjà existant : la fonction Dcreat installe alors un dossier à l'intérieur d'un autre. 'Adresse' renvoie à un string devant se terminer (comme en langage C) par un octet nul et qui contient le nom du dossier. On peut aussi rechercher l'adresse en écrivant :

Adresse = LPEEK(VARPTR(STRING\$)) + LPEEK(SEGPTR+28)

ce qui correspond à l'instruction MKDIR de l'Omikron Basic.

GEMDOS(R,58,HIGH(adresse),LOW(adresse))

Ddelete

Sert à effacer le dossier (subdirectory) dont le nom est indiqué dans le string sur lequel est dirigé le pointeur 'adresse' et qui se termine par un octet nul ; lorsque le dossier en question contient encore des fichiers, l'ordinateur envoie un message d'avertissement : il faut alors commencer par effacer tous les fichiers se trouvant dans le dossier à effacer. Equivaut à RMDIR.

GEMDOS(R,59,HIGH(adresse),LOW(adresse))

Dsetpath

Cette fonction définit un des dossiers comme étant le directory actuel ; le nouveau chemin d'accès est contenu dans un string sous 'adresse' et se termine par un octet nul. Equivaut à CHDIR.

GEMDOS(R,60,HIGH(adresse),LOW(adresse),Atr)

Fcreate

Sert à installer un nouveau fichier, dont le nom doit se trouver sous 'adresse' dans un string (se terminant par un octet nul). La valeur en retour s'appelle un 'file-handle' que nous allons retrouver dans les fonctions ci-dessous. Lorsque le système a pu effectivement créer le fichier, il retourne un file-handle (ou 'handle' tout court) ≥ 6 , sinon il retourne une valeur négative. L'apparition d'une erreur numéro -36 peut signifier que le système ne parvient plus à créer de nouveau fichier car le répertoire est déjà plein. Equivaut à OPEN "O",... en Omikron Basic. 'Atr' précise les attributs du fichier, selon le tableau suivant :

Atr	Attribut
0	vous pouvez lire le fichier et écrire dedans
1	après la fermeture du fichier
2	permet de créer un fichier caché (hidden-file) qui n'apparaîtra pas dans le répertoire
4	permet de créer un fichier système
8	donner un nom au disque ou à la disquette.

S'il existe déjà un fichier de même nom sur le disque, il se voit affecter une longueur 0, ce qui revient à le faire disparaître ; pour créer un dossier (attribut 16), reportez-vous à la fonction Dcreate.

<i>GEMDOS(R,61,HIGH(adresse),LOW(adresse),Mod)</i>	<i>Fopen</i>
--	--------------

Sert à ouvrir un fichier ; Mod = 0 équivaut à OPEN "T", Mod = 1 équivaut à OPEN "A" (écriture dans le fichier) et Mod = 2 provoque l'ouverture d'un fichier en autorisant la lecture et l'écriture ; pour les paramètres, reportez-vous à Fcreate.

<i>GEMDOS(R,62,Handle)</i>	<i>Fclose</i>
----------------------------	---------------

Sert à refermer le fichier portant le numéro 'handle', ainsi que le paramètre de création (Fcreate) ou d'ouverture (Fopen).

<i>GEMDOS(R,63,Handle,HIGH(quantité), LOW(quantité),HIGH(adresse),LOW(adresse))</i>	<i>Fread</i>
---	--------------

Sert à lire une certaine 'quantité' d'octets contenue dans le fichier désigné par son numéro 'handle' puis à les enregistrer dans un string à partir de 'adresse' ; après quoi R contient soit les octets lus soit un message d'erreur.

<i>GEMDOS(R,64,Handle,HIGH(quantité), LOW(quantité),HIGH(adresse),LOW(adresse))</i>	<i>Fwrite</i>
---	---------------

Sert à inscrire la 'quantité' d'octets se trouvant dans le buffer (string) désigné par le pointeur 'adresse', dans le fichier désigné par son 'handle'.

<i>GEMDOS(R,65,HIGH(adresse),LOW(adresse))</i>	<i>Fdelete</i>
--	----------------

Sert à effacer un fichier disque ; R contient soit un message d'erreur soit 0 si tout s'est bien passé ; équivaut à l'instruction KILL "<fichier>".

*GEMDOS(R,66,HIGH(quantité),LOW(quantité),
Handle,Modus)*

Fseek

Normalement, un fichier est écrit en mode séquentiel ; cette fonction permet de placer un pointeur sur une certaine position à l'intérieur d'un fichier, en recourant à trois paramètres :

- décalage du pointeur de 'quantité' d'octets
- numéro handle du fichier de Fopen ou Fcreate
- mode à choisir parmi les trois possibilités suivantes :

0	décalage depuis le début du fichier
1	décalage à partir de la position actuelle
2	décalage en 'remontant' depuis la fin du fichier.

En mode 0, 'quantité' doit être un nombre positif ; il peut être positif ou négatif en mode 1 (selon le sens du déplacement du pointeur) et toujours négatif en mode 2 puisqu'on 'remonte' depuis la fin du fichier. R contient un message d'erreur lorsqu'on dépasse la fin du fichier.

*GEMDOS(R,67,HIGH(adresse),LOW(adresse),
Mode,Attribut)*

Fattrib

Cette fonction permet de lire l'attribut conféré à un fichier et éventuellement de le modifier ; on lui transmet l'adresse du nom du fichier, le mode (0 => recherche de l'attribut, 1 => affecter un attribut) et le nouvel attribut que l'on veut affecter au fichier :

0	lecture et écriture autorisées
1	lecture seule autorisée
2	fichier caché (hidden-file)
3	fichier système
8	nom du disque (volume label)
16	dossier

Cette fonction ne permet pas de modifier les attributs d'un dossier ou le nom d'un disque. Voici un exemple d'utilisation :

```
DEF PROC Attribut(Fichier$.Attribut%)
  LOCAL Dummy%, Adr
  Fichier$ = Fichier$+CHR$(0)      'pour ne pas se planter
  Adr= LPEEK(SEGPTR+28)+ LPEEK( VARPTR(Fichier$))
  GEMDOS(Dummy%,HIGH(Adr),LOW(Adr),1,Attribut%)
  IF Dummy% <0 THEN                ' il y a comme une erreur !
    FORM ALERT(1,"[3][ ERREUR ! ][ Annuler ]")
  ENDIF
RETURN
```

GEMDOS(R,69,Device)

Fdup

Cette fonction permet d'attribuer un numéro de handle de fichier à un périphérique.

GEMDOS(R,70,Device,Handle)

Ffcorce

Ces deux routines permettent de détourner vers le canal standard de sortie une des routines GEMDOS d'entrée ou sortie des données ; 'Device' désigne :

1	l'écran et le clavier
2	l'interface série
3	l'imprimante

<i>GEMDOS(R,71,HIGH(adresse),LOW(adresse),Drv)</i>	<i>Dgetpath</i>
--	-----------------

Inscrit le chemin d'accès actuel vers l'unité de disque Drv dans un buffer de 64 octets se trouvant sous 'adresse'. L'identificateur de l'unité de disque n'est pas écrit au début du chemin d'accès, il faut l'ajouter par la suite.

<i>GEMDOS(R,72,HIGH(quantité),LOW(quantité))</i>	<i>Malloc</i>
--	---------------

Cette fonction sert à réserver un espace mémoire de 'quantité' d'octets ; R contient l'adresse de l'espace réservé dans la mémoire vive. Cette fonction équivaut largement à l'instruction MEMORY, à la différence qu'un espace réservé à l'aide de Malloc n'est pas effacé par un CLEAR.

<i>GEMDOS(R,73,HIGH(adresse),LOW(adresse))</i>	<i>Mfree</i>
--	--------------

Sert à libérer l'espace réservé auparavant par Malloc.

<i>GEMDOS(R,78,HIGH(adresse),LOW(adresse),Atr)</i>	<i>Fsfirst</i>
--	----------------

Sert à rechercher dans le répertoire le premier intitulé flanqué de l'attribut 'Atr' ; ce dernier est alors inscrit dans le buffer DTA où on peut le reprendre ; il est conseillé d'utiliser plutôt OPEN "F",... de l'Omikron Basic.

<i>GEMDOS(R,79)</i>	<i>Fsnext</i>
---------------------	---------------

Sert à rechercher le premier intitulé du répertoire rencontré après l'appel de la fonction ; là aussi, il est conseillé d'utiliser plutôt OPEN "F",... de l'Omikron Basic.

<i>GEMDOS(R,86,0,HIGH(ancien), LOW(ancien),HIGH(nouveau),LOW(nouveau))</i>	<i>Frename</i>
--	----------------

Sert à renommer un fichier ; mieux vaut utiliser NAME...AS.

<i>GEMDOS(R,87,HIGH(adresse),LOW(adresse), Handle,Mode)</i>	<i>Fdattime</i>
---	-----------------

Sert à modifier la date et l'heure système d'un fichier ; à l'appel de la fonction, on lui passe le numéro handle du fichier ouvert (Fopen) ainsi que l'adresse d'un buffer de 4 octets dans lequel on écrit ces données (Mode = 0) ou dont le contenu doit remplacer les anciennes données (mode = 1).

4.4. BIOS

Le BIOS (Basic input output system) gère les routines d'entrée et sortie des données ; on l'appelle par la fonction

BIOS([valeur en retour],<numéro de la fonction>,[paramètre])

Voici les routines BIOS implémentées dans le système d'exploitation de l'Atari ST :

<i>BIOS(,0,HIGH(adresse),LOW(adresse))</i>	<i>Getmpb</i>
--	---------------

Cette routine se charge de la gestion GEMDOS de la mémoire vive ; elle retourne l'adresse d'un bloc mémoire avec le 'memory parameter bloc' qui contient trois autres pointeurs renvoyant chacun à une structure ; vous ne pouvez guère vous en servir sans disposer de connaissances approfondies sur le système d'exploitation.

<i>BIOS(R,1,Device)</i>	<i>Bconstat</i>
-------------------------	-----------------

Cette fonction vérifie s'il existe un caractère à traiter sur le périphérique désigné par son device ; le device peut prendre les valeurs suivantes :

0	interface parallèle (PRT)
1	interface série (AUX
2	clavier écran (ne pas utiliser en Omikron Basic)
3	interface MIDI.
4	Port clavier (IKBD)

BIOS(R,2,Device)

Bconin

Lit un caractère transmis par le périphérique désigné par son device ; en Omikron Basic, on évite de se servir de cette fonction pour le clavier.

0	interface parallèle (port centronics
1	interface série RS232
2	clavier et écran
3	interface MIDI.

BIOS(,3,Device,Caractère)

Bconout

Sortie d'un caractère vers le périphérique désigné ; sous Omikron Basic, on évite de se servir de cette fonction pour la sortie vers l'écran.

0	interface parallèle (port centronics
1	interface série RS232
2	clavier et écran
3	interface MIDI
4	processeur du clavier (attention !)

<i>BIOS(R,4,Flag,HIGH(adresse),LOW(adresse), Nbre de secteurs,Numéro de secteur,Drive)</i>	<i>Rwabs</i>
--	--------------

Lorsque Flag = 0, cette fonction sert à recopier un nombre de secteurs (<Nbre de secteurs>) décomptés à partir du secteur logique portant le numéro <numéro de secteur> dans le buffer commençant à <adresse>.

Lorsque Flag = 1, le contenu du buffer va s'inscrire sur la disquette ou le disque ; Flag = 2 et Flag = 3 : similaire respectivement à 1 et 2, à la différence qu'un changement de disquette est alors ignoré.

<i>BIOS(R,5,Numéro,HIGH(adresse),LOW(adresse))</i>	<i>Setexec</i>
--	----------------

Cette fonction sert à modifier un vecteur d'anomalie (exception vector), <numéro> contenant le numéro du vecteur à modifier. L'adresse de la routine appelée à remplacer l'ancien vecteur doit se trouver sous <adresse>. Entrez la valeur '-1' à la place de <numéro> lorsque vous souhaitez lire un des 256 vecteurs d'anomalie du processeur 68000 ou l'un des 8 vecteurs GEM.

<i>BIOS(R,6)</i>	<i>Tickcal</i>
------------------	----------------

Retourne la durée (mesurée en millisecondes) qui s'est écoulée entre deux appels du timer.

<i>BIOS(R,7,Drive)</i>	<i>Getbpb</i>
------------------------	---------------

Retourne l'adresse du BIOS-parameter-bloc pour l'unité de disque indiquée ; ce bloc comprend 9 integers :

- Recsiz** taille d'un secteur exprimée en octets
- Clsiz** taille d'un cluster, exprimée en secteurs
- Clsizb** taille d'un cluster, exprimée en octets
- Rdlen** longueur du répertoire, exprimée en secteurs
- Fsiz** taille de la FAT (file allocation table) exprimée en secteurs ; la FAT répertorie les clusters dans lesquels un fichier est sauvegardé et sert aussi à trouver le cluster suivant
- Fatrec** numéro de secteur de la copie de la FAT
- Datrec** numéro de secteur du premier cluster de données
- Numcl** nombre total de cluster de données sur le disque
- Bflags** flags divers.
- 'Drive' peut prendre les quatre valeurs suivantes :

0	unité de disquette A
1	unité de disquette B
2	unité de disque C (disque dur)

<i>BIOS(R,8,Device)</i>

<i>Bcostat</i>

Vérifie si le périphérique de sortie est prêt à recevoir le caractère suivant ; valeur de retour possible :

-1 le périphérique est prêt

0 le périphérique n'est pas (encore) prêt

Le périphérique est désigné par un des devices suivants :

0	interface parallèle (port centronics)
1	interface série RS232
2	clavier et écran
3	interface MIDI
4	processeur du clavier.

BIOS(R,9,Drive)

Mediach

Vérifie s'il y a eu ou non une permutation de disquette dans l'unité de disquette indiquée ; valeur de retour :

0	la disquette n'a pas été échangée
1	il se pourrait que la disquette ait été échangée
2	la disquette a bien été échangée contre une autre.

BIOS(R,10)

Drvmap

Retourne la configuration de l'unité de disque, d'où il résulte un vecteur dans R, dans lequel chaque bit positif désigne une unité de disque connectée :

Bit 0	=> unité A connectée
Bit 1	=> unité B connectée
Bit 2	=> unité C connectée

BIOS(R,11,Status)

Kbshift

Retourne ou modifie le statut des touches spéciales du clavier ; lorsque la variable 'status' n'est pas négative, les touches spéciales sont reparamétrées selon le bit contenu dans 'status' ; lorsque 'status' prend la valeur -1, la fonction retourne un vecteur bit selon le tableau suivant :

Bit 0	=> touche <Shift droite>
Bit 1	=> touche <Shift gauche>
Bit 2	=> touche <control>
Bit 3	=> touche <alternate>
Bit 4	=> <capslock> activé
Bit 5	=> <Clr/Home> (touche droite de la souris)
Bit 6	=> <insert> (touche gauche de la souris)
Bit 7	=> inutilisé, toujours 0.

4.5. XBIOS

Le XBIOS (eXtended Basic Input Output System) gère les fonctions supplémentaires implémentées dans l'Atari-ST : la mémoire de l'écran, l'interface MIDI, l'interface de la souris, le paramétrage des couleurs du moniteur. On l'appelle en écrivant :

XBIOS ([valeur de retour], <numéro de fonction>, [paramètre])

XBIOS(0,...)

Initmous

Cette routine sert à initialiser les routines gérant la souris, cette fonction n'étant pas toujours compatible avec le GEM.

XBIOS(R, 1, Quantité)

Ssbrk

Sert à réserver de la place en mémoire vive pour un module ROM.

XBIOS(R, 2)

Physbase

Retourne l'adresse de l'écran actuellement activé.

XBIOS(R,3)**Logbase**

Retourne l'adresse de l'écran servant actuellement à l'affichage.

XBIOS(R,4)**Getrez**

Retourne la résolution de l'écran :

0	résolution basse (Lores) 320 * 200 pixel, 16 couleurs
1	résolution moyenne (Midres) 640 * 200 pixel, 4 couleurs
2	résolution haute (Hires) 640 * 400 pixel, 2 couleurs

**XBIOS(,5,HIGH(log_adr),LOW(log_adr),HIGH(phys_adr),
LOW(phys_adr),Résolution)**

Setscreen

Cette fonction permet de modifier le paramétrage de l'écran du moniteur ; 'log_adr' et 'phys_adr' contiennent respectivement l'adresse de l'écran logique et l'adresse de l'écran physique, retournées l'une par XBIOS(2) et l'autre par XBIOS(3). 'Résolution' permet de choisir entre les taux de résolution ci-dessous :

0	résolution basse (Lores) 320 * 200 pixel, 16 couleurs
1	résolution moyenne (Midres) 640 * 200 pixel, 4 couleurs
2	résolution haute (Hires) 640 * 400 pixel, 2 couleurs

XBIOS(,6,HIGH(adresse),LOW(adresse))

Setpalette

'adresse' renvoie à l'adresse (paire) de début d'un tableau comprenant 16 couleurs ; ce tableau contient pour chaque couleur un integer (word) ; voir l'instruction PALETTE en Omikron Basic.

XBIOS(R,7,Couleur,Valeur de la couleur)

SetColor

Cette fonction permet de modifier précisément une couleur ; on passe au paramètre 'couleur' un des numéros de couleur compris entre 1 et 15, la valeur de la nouvelle couleur pouvant être comprise entre 0 et \$777. Pour lire la valeur d'une couleur particulière, attribuez la valeur -1 au paramètre 'couleur'.

*BIOS(R,8,HIGH(adresse),LOW(adresse),0,0,Drv,Secteur,
Piste,Face,Quantité)*

Floprd

Cette fonction permet de charger certains secteurs du disque Drv dans le buffer sur lequel est dirigé le pointeur 'adresse'. Sous 'secteur', vous indiquez le premier secteur à lire (0 - 9), 'piste' étant comprise entre 0 et 79 pour des disques de 80 pistes (valeurs que l'on peut dépasser en formatant la disquette différemment, maxi 82). 'Face' contient la valeur 0 pour désigner la face 1, et 1 pour désigner la face 2 (disquettes de 720 Ko, lecteur SF314) ; 'quantité' précise le nombre de secteurs consécutifs à lire.

XBIOS(R,9,...)

Flopwr

Les paramètres de cette fonction correspondent à ceux de XBIOS(8) ; les données contenues dans le buffer sont ici écrites sur le disque.

*XBIOS(R,10,HIGH(adresse),LOW(adresse),0,0,Drv,Nbre de
secteurs,Piste,Face,Interleave,\$8765,\$4321,Formatage) Ffopfmt*

Cette fonction sert à formater une piste ; le formatage des disquettes est une opération très importante, puisque c'est là que vous allez sauvegarder vos données pour pouvoir ensuite les retrouver lorsque ce sera nécessaire. Voici les paramètres à indiquer :

Paramètre	Signification
Adresse	pointeur vers un buffer (string) contenant les données de formatage d'une piste (track) ; le buffer doit contenir au moins 8 Ko pour 9 secteurs.
Drv	numéro de l'unité de disque (0 = A, 1 = B etc)
Nbre de secteurs	nombre de secteurs par piste ; normalement : 9 mais on devrait pouvoir en mettre 10
Piste	numéro de la piste à formater (0 à 79)
Face	face de la disquette (0 ou 1)
Interleave	précise l'ordre dans lequel seront consignés les secteurs sur la disquette ; interleave = 1 => ordre : 1-2-3-4-5-6-7-8-9 interleave = 2 => ordre : 1-3-5-7-9-2-4-6-8 on utilise en général 'interleave = 1'
Formatage	cette valeur s'inscrit sur la disquette en tant qu'octet de données ; Atari recommande d'utiliser \$E5E5 et d'éviter en tout cas F0 à F5 car ce sont des instructions pour le floppy-controller.

Si le formatage se déroule correctement, on obtient une valeur en retour = 0. Cette valeur est négative (exp: -16: bad sectors) lorsqu'une erreur apparaît durant le formatage : le buffer contient alors la liste de tous les secteurs défectueux dans la piste.

XBIOS(R,11)

Getdsb

Non utilisé, retourne constamment la valeur 0.

XBIOS(,12,Longueur,HIGH(adresse),LOW(adresse))

Midiws

Ecrit les données se trouvant sous 'adresse' (string) dans le port MIDI.

XBIOS(R,14,Device)

lore

Cette fonction retourne un pointeur vers l'enregistrement se trouvant dans le buffer, pour l'entrée et la sortie vers un périphérique désigné par son 'device'

0	buffer d'entrée-sortie pour l'interface RS232
1	clavier
2	MIDI.

Le buffer lui-même a la structure suivante (à part Ibuf, il se compose uniquement de nombres entiers) :

Variable	Signification
Ibuf (Long)	pointeur vers le buffer d'entrée des données
Ibufsize	taille
Ibufhd	position suivante d'écriture dans le buffer
Ibuftl	adresse à laquelle on peut lire le buffer
Ibufflow	le buffer accepte de nouvelles données tant que son contenu est inférieur à cette variable
Ibufhi	le buffer est plein

XBIOS(,15,Baud,Com_Par,Usart_Reg,Recept_Stat,Trans_Stat,Sysnchr_Stat)
--

Rsconf

Cette fonction permet de configurer le port série RS232, et voici la signification de ces paramètres :

Paramètres	Signification
Baudrate (0-15)	vitesse de transmission des données : 19200, 9600, 4800, 3600, 2400, 2000, 1800, 1200, 600, 300, 200, 150, 134, 110, 75, 50
Com_Par	communication des données, selon les modes : 0 : pas de handshake 1 : XON/XOFF 2 : RTS/CTS 3 : XON/XOFF et RTS/CTS (modes 1 et 2)
Usart_Reg	valeur du registre USART dans MFP : Bit 1: 0 = odd 1 = even parity Bit 2: 0 = no parity 1 = parity Bit 3,4: 0-3: synchron, 1 stopbit 1.5 stopbits, 2 stopbits Bit 5,6: 0-3: 8,7,6,5 bits Bit 7: 0: ne pas diviser la fréquence 1: diviser la fréquence
Recept_Stat	Bit 0: 1 = réception RS232 activée Bit 1: transmettre les caractères SCR
Trans_Stat	Bit 0: 1 = émission RS232 activée Bit 1,2: 0 = sortie 1 = H 2 = L 3 = sortie sur l'entrée Bit 3: 1 = envoyer un break Bit 5: 1 = activer la réception après réception d'un caractère.

*XBIOS(R,16,HIGH(norm),LOW(norm),HIGH(shift),
LOW(shift),HIGH(lock),LOW(lock))*

Keytbl

Cette fonction permet de modifier l'affectation des caractères aux touches du clavier de l'Atari. Le TOS gère trois tableaux, dans lesquels sont consignés les codes SCAN affectés à chacune des touches du clavier ainsi que les codes ASCII correspondants : un tableau correspond à l'état du clavier lorsque la touche <shift> n'est pas appuyée, un tableau correspondant au clavier avec la touche <shift> appuyée et un tableau correspondant à l'état du clavier une fois la touche <capslock> activée. Chacun de ces tableaux contient 128 octets. Les paramètres indiqués ci-dessus sont des pointeurs vers le tableau correspondant. Le tableau n'est pas modifié lorsqu'on transmet la valeur -1 à la place de l'adresse. Pour modifier l'affectation des touches du clavier, vous écrivez les trois strings contenant les nouveaux tableaux puis vous dirigez sur eux les trois pointeurs du système d'exploitation à l'aide de XBIOS(R,16...). Vous pouvez ensuite annuler cette réaffectation des pointeurs à l'aide de XBIOS(,14).

XBIOS(R,17)

Random

Cette fonction retourne dans R un nombre aléatoire.

*XBIOS(,18,HIGH(adresse),LOW(adresse),HIGH(ser),
LOW(ser),Type,Exec)*

Protobt

Cette fonction sert à créer ou modifier le secteur de rebootage (réinitialisation) : 'adresse' indique l'endroit où se trouve le secteur boot dans la mémoire vive (taille : 512 octets) ; <ser> contient le nouveau numéro de série de la disquette, sur 24 bits, ou -1 lorsque ce numéro n'est pas modifié.

Voici les différents paramètres désignant des types de disquette :

0	40 pistes, une seule face (180 Ko) = 40 SS
1	40 pistes, double-face (360 Ko) (format IBM) = 40 DS
2	80 pistes, simple face (360 Ko) (Atari SF 354) = 80 SS
3	80 pistes, double-face (720 Ko) (Atari SF 314) = 80 DS
-1	pas de modification du type de disquette

'exec' indique si le secteur boot doit être exécutable ou non :

0	secteur boot non exécutable
1	secteur boot exécutable
-1	secteur boot non modifié.

Le secteur boot d'une disquette contient les informations suivantes :

Octets	Contenu	40 SS	40 DS	80 SS	80 DS
0-1	branchement sur le programme d'initialisation				
2-7	"LOADER"				
8-10	numéro de série sur 24 bits				
11-12	BPS	512	512	512	512
13	SPC	1	2	2	2
14-15	RES	1	1	1	1
16	NFATS	2	2	2	2
17-18	NDIRS	64	112	112	112
19-20	NSECTS	360	720	720	1440
21	MEDIA	252	253	248	249
22-23	SPF	2	2	5	5
24-25	SPT	9	9	9	9
26-27	NSIDES	1	2	1	2

et voici la signification de ces abréviations :

BPS	octets contenus dans un secteur
SPC	nombre de secteurs par cluster
RES	quantité de secteurs réservés au début de la disquette (Boot compris)
NFATS	nombre de FATs sur la disquette
NDIRS	nombre d'intitulés autorisés dans le répertoire
NSECTS	nombre de secteurs sur la disquette
MEDIA	Media descriptor byte (non utilisé)
SPF	nombre de secteurs dans chaque FAT
SPT	nombre de secteurs par piste (9)
NSIDES	nombre de face(s) de la disquette.

Cette fonction sert à créer le secteur d'initialisation indispensable sur une disquette après son formatage, puis à l'écrire sur la disquette au moyen de la fonction Flopwr. Reportez-vous aussi à la routine 'FORMAT' dont nous reparlerons dans le prochain paragraphe (4.6).

<i>XBIOS(R,19,HIGH(adresse),LOW(adresse),0,0, Unité_de_disque,Secteur,Piste,Face,Quantité)</i>	<i>Flopver</i>
--	----------------

Sert à vérifier la bonne lisibilité des secteurs désignés ; la syntaxe est la même que celle de Floprd.

<i>XBIOS(R,20)</i>	<i>Scrdmp</i>
--------------------	---------------

Sert à confectionner une recopie de l'écran (hardcopy).

<i>XBIOS(R,21,Fonction,Vitesse)</i>	<i>Cursconf</i>
-------------------------------------	-----------------

Sert à activer/désactiver le curseur, ainsi qu'à régler la vitesse du clignotement à l'écran

Fonction	Effet provoqué
0	désactiver le curseur
1	activer le curseur
2	faire clignoter le curseur
3	arrêter le clignotement du curseur
4	déterminer la vitesse du clignotement
5	demander la vitesse du clignotement.

XBIOS(,22,HIGH(Calendrier),LOW(Calendrier))

Settime

Sert à mettre à jour le 'calendrier', dont voici la structure :

Bits	Contenu
0 - 4	indication des secondes
5 - 10	indication des minutes (0 - 59)
11 - 15	indication des heures (0 - 23)
16 - 20	indication des jours (1 - 31)
21 - 24	indication des mois (1 - 12)
25 - 31	indication de l'année (+1980)(0-119)

XBIOS(R,23)

Gettime

Rechercher l'heure et la date, selon le format décrit ci-dessus.

XBIOS(,24)

Bioskeys

Annuler les modifications faites à l'aide de la fonction XBIOS(16).

<i>XBIOS(,25,longueur,HIGH(adresse),LOW(adresse))</i>	<i>Ikbdws</i>
---	---------------

Sert à transmettre au processeur du clavier les instructions se trouvant dans un buffer sous <adresse> ; <longueur> représente la quantité -1 des données présentes dans le buffer.

<i>XBIOS(,26,Numero)</i>	<i>Jdisint</i>
--------------------------	----------------

Sert à stopper une interruption du MFP.

<i>XBIOS(,27,Numero)</i>	<i>Jenabint</i>
--------------------------	-----------------

Libère une interruption du MFP.

<i>XBIOS(R,28,Données,Numero de registre)</i>	<i>Giaccess</i>
---	-----------------

Sert à écrire dans le registre de la puce sonore (sound chip) indiqué ou à en lire le registre. Les numéros de registres sont compris entre 0 et 15 ; lorsque le bit 7 est positif dans le numéro de registre, cela signifie qu'il s'agit d'un accès par écrit.

<i>XBIOS(,29,Numero de bit)</i>	<i>Offgibit</i>
---------------------------------	-----------------

Effacer un bit dans le port A de la puce sonore.

<i>XBIOS(,30,Numero de bit)</i>	<i>Ongibit</i>
---------------------------------	----------------

Rendre un bit positif dans la puce sonore.

<i>XBIOS(,32,HIGH(adresse),LOW(adresse))</i>	<i>Dosound</i>
--	----------------

Emission de la séquence sonore se trouvant dans la mémoire vive, dans un tableau sous <adresse>.

XBIOS(,33,Paramétrage)**Setprt**

Paramétrage selon le type de l'imprimante :

Bit numéro	Bit à 0	Bit à 1
0	imprimante matricielle	impr. à marguerite
1	imprimante couleur	impr. noir-et-blanc
2	imprimante Atari	impr. EPSON
3	impression Brouillon	impression NLQ
4	impr connectée Centronics	impr connectée RS232
5	papier continu	feuille-à-feuille

XBIOS(R,34)**Kbdvbase**

Retourne un pointeur vers un tableau de pointeurs :

- saisie MIDI
- erreur clavier
- erreur MIDI
- statut IKBD
- routines de la souris
- routine de l'heure
- routine pour le joystick

XBIOS(R,35,Délai,Vitesse)**Kbrate**

Dans la mesure où on ne lui passe pas la valeur -1, cette routine permet de régler le délai et la vitesse de répétition d'un même caractère lorsqu'on laisse la touche enfoncée un certain temps (repeat

function) ; la fonction retourne dans R les valeurs actives jusque là : les bits 0 à 7 indiquent la vitesse de répétition et les bits 8 à 15 le délai actuel.

<i>XBIOS(,26,HIGH(adresse),LOW(adresse))</i>
--

<i>Prtblk</i>

Autre routine servant à confectionner une copie d'écran, mais plus difficile à manipuler que la précédente XBIOS(R,20).

<i>XBIOS(,37)</i>

<i>Vsync</i>

Attend que se produise un effacement de trame (vertical blank).

<i>XBIOS(,38,HIGH(adresse),LOW(adresse))</i>
--

<i>Supexec</i>

Exécution en mode superviseur d'un programme-machine se trouvant dans la mémoire vive à partir de <adresse>.

<i>XBIOS(,39)</i>

<i>Puntaes</i>

Efface l'AES dans le cas des systèmes d'exploitation réinitialisés depuis la disquette ; cette instruction doit être appelée depuis le dossier AUTO.

4.6. Programmation sous le système d'exploitation

A l'aide de deux exemples concrets, je voudrais maintenant vous montrer comment utiliser certaines routines du système d'exploitation, et j'en profiterais pour vous donner d'autres informations.

□ Comment formater une disquette

Avant de pouvoir enregistrer des données sur une disquette, il convient de la formater. Cela crée en quelque sorte des 'rayonnages' sur lesquels vous pourrez ensuite 'ranger' vos données. Une fois formatée, la disquette se divise en effet en pistes (tracks), lesquelles se subdivisent à leur tour en secteurs logiques. Il est possible de formater différemment la même disquette, en faisant varier le nombre de pistes et de secteurs par piste, de la formater en simple face ou en double face : toutes ces informations se retrouvent dans le secteur d'initialisation ou 'secteur de rebootage' en bon français informatique (bootsector). La disquette contient en plus une FAT (file allocation table) en double-exemplaire, sorte de table des matières répertoriant les secteurs de la disquette déjà occupés, ceci afin d'éviter d'utiliser le même secteur deux fois pour des données différentes (ce qui serait catastrophique !).

L'Atari-ST vous offre, au travers du GEM, une possibilité très commode de procéder au formatage des disquettes dès que vous vous trouvez sous le 'bureau GEM'. Par contre, si vous souhaitez formater une disquette lorsque vous êtes en train de travailler sous un logiciel, il vous faut appeler vous-même les routines nécessaires dans le système d'exploitation. C'est précisément ce que nous allons faire maintenant.

La première étape consiste à créer un buffer, dont le TOS aura besoin pour formater la disquette. Nous ne nous attarderons pas sur son contenu, qui n'est guère intéressant. Seule chose importante : ce buffer ne doit en aucun cas s'avérer trop petit ; une taille de 11 Ko

devrait amplement nous suffire. Etape suivante : nous devons veiller à ce que le buffer en question ne soit en aucun cas déplacé. Comment une telle chose pourrait-elle se produire ?

Essayez de vous représenter la mémoire vive de votre Atari sous la forme d'une liste continue, à une seule dimension : je veux dire par là que tous les octets sont en quelque sorte alignés les uns à la suite des autres. L'ordinateur 'range' les chaînes de caractères dans cette mémoire vive : pour éviter de gaspiller de la place, il aligne les différentes chaînes l'une derrière l'autre. Lorsque vous attribuez un nouveau contenu à une chaîne de caractères, et que ce nouveau contenu dépasse la longueur de l'ancienne chaîne, l'ordinateur devrait procéder en trois étapes :

- ❶ il décalerait les chaînes de caractères restantes d'une position, afin de dégager de la place
- ❷ il remplacerait l'ancienne chaîne par la nouvelle
- ❸ comme ce processus reviendrait à modifier toutes les adresses des chaînes de caractères concernées, il les rectifierait.

Naturellement, aucun ordinateur ne procède ainsi, car cela engendrerait un travail énorme. La nouvelle chaîne de caractères est tout simplement 'accrochée' derrière les autres chaînes, tandis que l'ancienne chaîne reste à sa place, mais sous forme de 'rature' qui n'est plus accessible : la nouvelle chaîne se trouve donc à un autre endroit de la mémoire.

A force de juxtaposer ainsi les chaînes de caractères, il arrive toujours un moment où l'espace mémoire, aussi important soit-il, est saturé : un grand nettoyage devient inévitable pour éliminer les multiples ratures. Ce grand ménage de printemps s'appelle en informatique 'garbage collection' (ramassage des ordures). Dans notre cas, nous ne pouvons absolument pas lancer ce nettoyage depuis l'Omikron Basic, puisque nous devons transmettre l'adresse du buffer, que nous venons d'installer, au système d'exploitation, et que cette adresse serait alors faussée.

Pour nous en sortir, nous allons recourir à une petite astuce et lancer d'abord une 'garbage collection', qui forcera l'interpréteur à faire le ménage après quoi il ne touchera plus à l'adresse de notre buffer. Pour lancer une 'garbage collection', nous écrivons :

```
FRE(" ")
```

Cette fonction retourne une valeur indiquant la place mémoire disponible après élimination de toutes les 'ratures', ce qui ne nous concerne guère, d'où son affectation à la variable 'dummy'. Le reste n'est plus que routine, et ce, au vrai sens du terme ! en effet, la routine XBIOS 10 nous permet de formater la disquette, piste par piste ; dans le cas d'une disquette double-face, il me semble judicieux de ne pas formater les deux faces l'une à la suite de l'autre, mais alternativement : d'où les deux boucles FOR...NEXT imbriquées l'une dans l'autre.

La fonction retourne un code d'erreur (valeur négative) lorsqu'une piste ne peut être formatée correctement (ce qui arrive). Comme il nous semble important que l'utilisateur le sache, nous demandons l'affichage de ce message à l'écran.

L'étape suivante consiste à créer le secteur d'initialisation. Le système d'exploitation identifie les différentes disquettes d'après leur numéro de série : nous allons donc générer un numéro aléatoire sur 24 bits qui remplacera ce numéro de série. Pour cela, nous faisons de nouveau appel à une routine XBIOS. Une fois ce numéro de série obtenu, nous utilisons la fonction XBIOS 19 (Protobt) pour créer le secteur de rebootage. Nous créons d'abord un secteur boot standard (que nous connaissons déjà, voir ci-dessus) que nous adaptons ensuite à nos besoins.

Vous connaissez l'instruction MKI\$, puisque nous l'avons étudiée dans le chapitre consacré à la gestion des fichiers. Cette fonction sert à transformer un entier en une chaîne de caractères longue de deux octets, ce qui nous permettait d'utiliser des nombres dans un fichier relatif. Mais comment l'ordinateur s'y prend-il pour transformer ainsi un nombre en une chaîne de caractères ?

L'ordinateur fonctionne selon le système binaire ; ce système se compose d'unités d'information minimale nommées 'bits'. Les bits sont réunis par paquet de huit pour devenir un octet (byte), ce qui permet de représenter les nombres de 0 à 255. Que faire pour les nombres supérieurs à 255 ? prendre un deuxième octet ayant la valeur 256. Comme cet octet augmente d'une unité lorsque la capacité de l'octet précédent est saturée, on l'appelle 'high-byte' (octet haut) et son collègue devient du même coup le 'low-byte' (octet bas). L'instruction MKI\$ transforme un nombre pour lui faire prendre ce format.

En clair : MKI\$ transforme un entier en une chaîne de caractères de deux octets respectant le format suivant :

HIGHBYTE LOWBYTE

Vous pouvez vérifier cela par vous-même, en convertissant un nombre pour ensuite décomposer en deux parties la chaîne de caractères (Left\$(...,1) ou Right\$(...,1) dont vous reprenez les codes ASCII. La formule

$$\text{ASC}(\text{HIGHBYTE}) * 256 + \text{ASC}(\text{LOWBYTE})$$

vous permet en principe de retrouver le nombre d'origine. L'ordinateur s'y prend exactement de la même façon, et nous pouvons nous servir de cette fonction (et de celles qui lui sont apparentées) pour atteindre d'autres objectifs. Un bon exemple est justement fourni par la routine de formatage : cette fonction nous permet d'écrire, dans le buffer du secteur boot, le nombre de faces ainsi que le nombre de secteurs que comporte la disquette.

Attention à un dernier petit détail : MKI\$ transforme son argument selon le format HIGHBYTE LOWBYTE, et il se trouve que c'est le même format que celui qui est utilisé par le processeur 68000. Notez cependant que tous les processeurs ne fonctionnent pas sur ce format.

Alors qu'Atari a choisi, avec le TOS, un système d'exploitation qui lui est particulier, il a utilisé par contre un format d'enregistrement emprunté au monde des compatibles IBM et de leur système d'exploitation MS-DOS. Cela signifie que l'Atari-ST peut au moins

lire les disquettes enregistrées par un ordinateur MS-DOS (l'inverse est par contre impossible avec les anciennes ROM), sous réserve que les disquettes aient la bonne taille : comment voulez-vous insérer une disquette 5"1/4 d'un IBM-PC dans le lecteur 3"1/2 de votre Atari ?

Les processeurs de la gamme des compatibles IBM (de la firme Intel) recourent à un format différent de celui du processeur des ordinateurs Atari (Motorola 68000) : LOWBYTE HIGHBYTE. Conséquence : le processeur de l'Atari doit d'abord 'retourner' toutes les données avant de les inscrire sur une disquette ; lors de la lecture, il recommence ce petit jeu puisque le processeur n'admet pas le format sous lequel les données ont été enregistrées. D'où l'utilité de l'instruction MIRROR\$ qui sert à intervertir l'ordre des deux octets avant leur entrée dans le buffer.

Une fois le secteur boot créé, il ne nous reste plus qu'à aller l'inscrire sur la disquette. Le travail est ainsi terminé, ainsi que la procédure elle-même. Veuillez vous reporter au texte du programme pour ce qui concerne les paramètres utilisés. Cette routine vous permet d'ailleurs de 'gonfler' les capacités de stockage sur votre disquette, c'est-à-dire d'y créer 82 ou 83 pistes de chacune 10 secteurs (à supposer que votre lecteur suive !).

```

100 *****
110 *                                     FORMAT.BAS                                     *
120 *-----*
130 * Auteur: Michael Maier      Version 1.00      Date: 16.07.1990  *
140 * Programme joint au 'livre de l'Omikron Basic'                  *
150 * (c) 1988 MICROAPPLICATION                                     *
160 *****
170 *
180 *
190 * ----- - voici une procédure servant à formater une disquette - -----
200 *
210 * pour la lancer:
220 * Format(drv%,sides%,tracks%,spt%)
230 *
240 * Paramètres:
250 *     drv%      :  numéro de l'unité de disquette
260 *                  0 -> unité A
270 *                  1 -> unité B
280 *     sides%    :  1 -> disquette une seule face (SF354)
290 *                  2 -> disquette double-face (SF 314)
300 *     tracks%   :  nombre de pistes (80 - 83)
310 *     spt%      :  nombre de secteurs par piste (9 ou 10)
320 *
330 *
```

```

340 DEF PROC Format(De%,Sid%,Trk%,St%)
350   LOCAL TX,S%,Du%L,A%L,Nsects%,Buffer$
360   '
370   ' réserver assez de place pour 10 secteurs
380   '
390   Buffer$=SPACE$(11000)
400   '
410   'attention: il ne faut pas que le buffer soit déplacé !
420   '
430   Du%L= FRE("")
440   '
450   ' calcul de l'adresse
460   '
470   A%L= LPEEK( VARPTR(Buffer$))+ LPEEK( SEGPTR +28)
480   FOR TX=0 TO Trk%-1
490     FOR S%=0 TO Sid%-1
500       'formatage par piste, alternativement facel/face2
510       XBIOS(Du%L,10,HIGH(A),LOW(A),0,0,De%,St%,TX,S%,1,$-7898,$4321,0)
520       ' si valeur de retour est un nbre négatif -> erreur
530       IF Du%L<0 THEN
540         PRINT
550         PRINT "erreur de formatage sur la face:":S%:" Piste:":TX
560       ENDIF
570     NEXT S%
580   NEXT TX
590   '
600   ' nombre de secteurs sur cette disquette, selon la formule:
610   ' Quantité = quantité tracks * quantité de faces * steps par track
620   '
630   Nsects%=Trk%*Sid%*St%
640   '
650   'générer un nombre aléatoire de 24 bits pour le no de série
660   '
670   XBIOS (Du%L,17)
680   '
690   'créer un secteur boot dans le buffer
700   '
710   Buffer$= SPACE$(513)
720   A%L= LPEEK( SEGPTR +28)+ LPEEK( VARPTR(Buffer$))
730   XBIOS (.18, HIGH(A%L), LOW(A%L), HIGH(Du%L), LOW(Du%L),3,0)
740   '
750   'reste à adapter cela à nos besoins, mais en format LOW-HIGH
760   '
770   MID$( Buffer$,20,2)= MIRROR$( MKI$(Nsects%))
780   MID$( Buffer$,25,2)= MIRROR$( MKI$(Sid%))
790   '
800   'enfin, inscrire sur la disquette
810   '
820   XBIOS (.9, HIGH(A%L), LOW(A%L),0,0,0,1,0,0,1)
830   '
840   RETURN

```


❑ Comment lire le répertoire d'une disquette

L'Omikron Basic vous offre deux instructions permettant de lire le répertoire d'une disquette à l'intérieur d'un programme. Vous pouvez utiliser l'une des deux routines du système d'exploitation

FSFIRST ou FSNEXT

ou l'instruction OPEN "F", qui n'a certes rien de commun avec le système d'exploitation mais fonctionne de la même manière. Nous verrons l'utilisation des deux routines dans le prochain chapitre consacré à la programmation sous GEM et à la boîte de sélection d'objet. Pour l'instant, nous nous intéressons à la possibilité de lire le répertoire d'une disquette à l'aide de OPEN "F", d'autant plus que le manuel de l'Omikron Basic est totalement muet à ce sujet.

Dans le texte de programme ci-dessous, bien que l'on utilise, après OPEN 'F', l'instruction GET pour lire chacun des intitulés du répertoire, le nombre 63 concluant l'instruction OPEN (ligne 13) n'a rien à voir avec la longueur de l'enregistrement, contrairement à ce que l'on pourrait supposer. Il s'agit en fait d'une combinaison des attributs de fichier possibles dont il faut tenir compte lors de la recherche :

Valeur	Type de fichier
1	fichier protégé contre l'écriture
2	fichier caché (hidden file)
4	fichier système
8	nom de la disquette (volume label)
16	dossier
32	fichier correctement refermé

Lorsque vous recherchez par exemple le nom de la disquette, il vous suffit d'entrer le nombre 8. Il est possible de combiner les divers attributs en additionnant purement et simplement les valeurs correspondantes. On remplace habituellement le nom du fichier par deux troncatures '*.*', ce qui permet de lire tous les noms de fichiers

arborant les attributs sélectionnés. Si par exemple vous ne souhaitez retrouver que les noms des fichiers contenant des programmes en Basic, vous entrez '*.BAS'.

La première tâche du programme consiste à vérifier s'il existe bien, sur la disquette, des fichiers possédant les attributs et/ou le nom de fichier spécifiés. Pour cela, nous contrôlons si la routine n'a pas encore dépassé la fin du fichier (sachez en effet que tous les noms de fichiers sur la disquette sont réunis en un seul fichier) :

```
NOT EOF(<canal>)
```

Une boucle sans fin nous sert ensuite à lire tous les intitulés un par un grâce à l'instruction GET : tant que la fin de fichier EOF n'est pas dépassée, il s'agit d'un nom de fichier valide, qui peut s'afficher ; si tel n'est pas le cas, on ressort de la boucle par EXIT.

L'instruction GET dépose les données recueillies dans un buffer défini par FIELD, qui doit avoir une taille de 44 octets. Sa structure ressemble à celle du buffer DTA, dans lequel FSFIRST et FSNEXT 'empilent' les données. On trouve donc, à partir de l'octet 31, le nom de fichier se terminant par un octet nul, dont on provoque l'affichage par un PRINT, après quoi il est effacé.

```

0 *****
1 *                                     DIR_1.BAS                                     *
2 *-----*
3 * Auteur: Michael Maier      Version 1.00      Date: 16.07.1990 *
4 * Programme joint au 'Livre de l'Omkron Basic' *
5 * (c) 1988 MICROAPPLICATION *
6 *****
7
8      procédure servant à lire le répertoire d'une disquette
9
10
11 * commençons d'abord par ouvrir le 'fichier'
12
13 OPEN "F".1."*.*".63
14 * vérifier s'il existe des intitulés
15 IF NOT EOF(1) THEN
16   * dimensionner le tableau pour permettre la lecture avec GET
17   * le nom de fichier commence à l'octet 31 et se termine par
18   * un octet nul 'CHR$(0)' (comme en langage C)
19   FIELD 1,30 AS Buffer$.14 AS Nom$
20   TX = 0
21   REPEAT
22     GET 1,TX
23     * si l'intitulé correspond aux critères recherchés, l'afficher

```



```
24 IF NOT EOF(1) THEN
25   ' ne tenir compte du string que jusque l'octet nul
26   PRINT LEFT$(Nom$, INSTR(Nom$, CHR$(0))-1)
27 ELSE
28   ' lorsque fin du directory atteinte -> sortir de la boucle
29   EXIT
30 ENDIF
31 ' lire l'intitulé suivant
32 TX=TX+1
33 UNTIL 0
34 ENDIF
35 CLOSE 1
```


Chapitre 5

La programmation graphique

Le graphisme est un domaine que tous les fans d'informatique finissent par aborder un jour ou l'autre, surtout s'ils possèdent un ordinateur se distinguant par son taux de résolution particulièrement élevé. Je suppose que tout lecteur de ce manuel aura déjà eu entre ses mains le manuel d'un de ces logiciels graphiques transformant l'Atari en outil de création picturale. Au contraire des logiciels manipulant des caractères, ces logiciels de création graphique sont généralement 'orientés pixels', c'est-à-dire qu'ils manipulent d'une façon ou d'une autre le contenu de l'écran. On ne peut plus revenir sur cette manipulation une fois qu'elle a été exécutée.

Un autre groupe de logiciels graphiques sont dits 'orientés objets', ce qui signifie qu'ils gèrent des objets graphiques sous deux aspects différents : premièrement comme leurs collègues orientés pixels et deuxièmement comme des descriptions d'objets, de graphismes. Ce genre de production graphique peut être éditée (retravaillée) sans influencer les autres objets. Le problème étant que ces programmes sont beaucoup plus difficiles à écrire : impossible de s'en tirer sans les vecteurs et les matrices. C'est pourquoi nous allons devoir aborder un peu le calcul matriciel dans les pages qui vont suivre, mais commençons par le plus simple :

5.1. Les commandes graphiques simples

Je n'ai pas l'intention ici de répéter ce que vous trouvez déjà dans le manuel de l'Omikron Basic ; je préfère vous expliquer en détail le fonctionnement de quelques instructions graphiques.

□ La résolution graphique du moniteur Atari

La résolution de l'écran varie en fonction du moniteur (noir/blanc ou couleur) et en fonction du taux de résolution que vous avez vous-même sélectionné. Le terme 'résolution graphique' désigne le nombre de points composant le contenu de l'écran. L'Atari ST peut gérer trois degrés de résolution :

640*400 point en mode HIRES (haute résolution monochrome)

640*200 points en quatre couleurs (moyenne résolution)

320*200 points en seize couleurs (basse résolution).

Vous constatez que, si vous pouvez choisir entre deux résolutions lorsque vous possédez un moniteur couleur, cela n'est pas possible avec un moniteur noir et blanc : ce dernier vous fait obligatoirement bénéficier du taux de résolution le plus élevé : HIRES (High RESolution). Dans le cas le plus simple, la haute résolution, chaque point nécessite exactement un bit, ce qui nous donne 32000 octets pour rendre compte du contenu de l'écran. Dans le cas d'un écran couleur, il convient d'ajouter des informations supplémentaires (concernant chaque couleur) ce qui se fait aux dépens de la résolution qui est alors moins élevée.

Le coin supérieur gauche de l'écran répond aux coordonnées (0,0) et le coin inférieur droit, en haute résolution, aux coordonnées (639,399). Seule exception : lorsque vous ouvrez une fenêtre, c'est le coin supérieur gauche de cette fenêtre qui devient le point (logique) zéro. Ces coordonnées interviennent ensuite dans toutes les instructions ; pour ma part, dans les lignes qui suivent, je me réfère toujours au degré de résolution le plus élevé, du moniteur noir et blanc.

Tout graphique se compose d'un quadrillage de points, chaque point pouvant être 'actif' (noirci) ou 'inactif' (clair). La densité plus ou moins grande des points permet de rendre différents effets ; c'est ainsi que les gris sont de plus en plus soutenus, jusqu'au noir complet. On comprend alors qu'une ligne ou un trait ne sont rien d'autre que des points juxtaposés avec une grande densité, exactement comme des carrés ou des triangles.

□ Les points, les lignes et les rectangles

Vous activez (= noircissez) un point répondant aux coordonnées (X,Y) à l'aide de l'instruction

```
DRAW X,Y
```

La même instruction sert aussi à confectionner une ligne (se composant de nombreux points consécutifs), en écrivant :

```
DRAW X0,Y0 TO X1,Y1
```

Attention : L'instruction DRAW fait alors appel aux valeurs entrées à l'aide des instructions

```
MODE -
```

```
LINE COLOR -
```

```
LINE STYLE -
```

Cette même instruction DRAW vous permettrait de dessiner un rectangle (ou un carré) en reliant tout simplement les quatre lignes nécessaires. Il existe cependant une instruction beaucoup plus simple :

```
BOX X0,Y0 TO X1,Y1
```

flanquée de sa proche parente :

```
BOX X,Y,Largeur,Hauteur
```

Ces deux instructions permettent l'une comme l'autre de dessiner un rectangle sur votre écran (un carré n'étant qu'une variante de rectangle aux quatre côtés de longueur identique). Dans le premier cas, vous indiquez les coordonnées des deux points diagonalement opposés (coin supérieur gauche, coin inférieur droit), dans le

deuxième cas, vous indiquez les coordonnées du coin supérieur gauche puis la largeur et la hauteur du rectangle, ce qui peut se ramener à la première syntaxe :

```
BOX X,Y TO X+Hauteur,Y+Largeur
```

Vous pouvez ensuite remplir la surface délimitée par un motif ou une couleur, à l'aide de l'instruction

```
FILL X,Y,Couleur
```

Pour vous représenter l'effet de cette instruction, imaginez un pot de peinture que l'on renverserait à l'intérieur de la surface délimitée par les coordonnées (X,Y) et qui peu à peu la remplit. La moindre faille dans le périmètre de la surface délimitée fait que la 'peinture' s'échappe et remplit tout l'écran. Il est recommandé d'utiliser -1 comme valeur limite de 'Couleur' faute de quoi GEM fournit des résultats peu satisfaisants. Comme l'algorithme de remplissage est assez lent, il faut un certain temps avant que le rectangle ne soit rempli. On en est venu à implémenter une instruction qui remplit le rectangle d'un motif (ou d'une couleur) dès sa création :

```
PBOX X0,Y0 TO X1,Y1  
PBOX X,Y,Largeur,Hauteur
```

Il est possible de dessiner un rectangle à coins arrondis (des coins ronds ?)

```
RBOX X0,Y0 TO X1,Y1  
RBOX X,Y,Largeur,Hauteur
```

avec son cousin garni d'emblée par un motif de remplissage :

```
PRBOX X0,Y0 TO X1,Y1  
PRBOX X,Y,Largeur,Hauteur
```

Pour indiquer le motif ou la couleur de remplissage, vous utilisez les instructions :

```
FILL STYLE -  
FILL COLOR -
```


5.2. BITBLT

Cette instruction, que l'on prononce 'bit-blit, sert à recopier le contenu de certains secteurs de la mémoire vive, et plus précisément ceux qui contiennent des 'morceaux' de l'écran ; utilisations possibles :

- ① recopie d'une partie de l'écran dans une autre partie de l'écran
- ② recopie d'une partie de l'écran dans la mémoire vive
- ③ recopie d'une partie de la mémoire vive sur l'écran
- ④ recopie d'une partie de la mémoire vive dans une autre partie de la mémoire vive.

Commençons par le commencement : pour recopier un extrait de l'écran d'une position à une autre, il faut commencer par délimiter la surface du 'morceau' (du bloc) que l'on souhaite recopier. En écrivant :

```
BITBLT X0,Y0,Largeur0,Hauteur0 TO X1,Y1,Largeur1,Hauteur1
```

on recopie le rectangle délimité par la position de son coin supérieur gauche (X0,Y0) ainsi que sa largeur et sa hauteur à la position indiquée derrière 'TO'. Il n'est pas indispensable que les deux rectangles aient la même taille : c'est le plus petit rectangle qui est recopié, et il peut fort bien prendre la hauteur 0 tout en adoptant la largeur 1. On peut de plus préciser un 'mode' de liaison entre les deux rectangles, mode qui relie logiquement un point S du rectangle source à un point D du rectangle cible, en respectant la syntaxe suivante :

```
BIBBLT X0,Y0,Largeur0,Hauteur0 TO X1,Y1,Largeur1,Hauteur1,Mode
```

Voici les modes possibles :

Mode	Liaison logique résultante
0	0
1	S AND D
2	S AND (NOT)D
3	S (Mode remplacement)
4	(NOT S) AND D
5	D
6	S XOR D
7	S OR D
8	NOT (S OR D)
9	NOT (S XOR D)
10	NOT D
11	S OR (NOT D)
12	NOT S
13	(NOT S) OR D
14	NOT (S OR D)
15	1

L'instruction BITBLT ne sert pas seulement à recopier un morceau d'écran d'un endroit à l'autre ; elle peut aussi protéger son contenu dans la mémoire vive pour qu'il ne soit pas écrasé par de nouvelles données (cette instruction est donc très importante, et nous y reviendrons dans le chapitre suivant consacré à la programmation sous GEM). Ceci permet de sauver temporairement le contenu des surfaces recouvertes par des formulaires de mise-en-page ou autres objets graphiques, en les conservant dans la mémoire vive, dans laquelle on réserve un espace de stockage à l'aide de la fonction :

Adresse - MEMORY(<nombre d'octets>)

ce qui a pour effet de dégager un espace-mémoire du 'nombre d'octets' indiqué. La fonction retourne alors l'adresse à laquelle débute l'espace mémoire. Reste à expliquer comment calculer la taille (en octets) de l'espace à réserver pour un bloc BITBLT dans la mémoire de travail. C'est très simple, il se calcule selon la formule :

$$\text{Octets nécessaires} = (\text{Largeur} + 15) \text{ SHR } 4 * \text{Hauteur} * 2 + 6$$

après quoi on peut transférer le bloc dans l'espace réservé :

BITBLT X,Y,Largeur,Hauteur TO Adresse

Ceci vous permet par exemple d'ouvrir une fenêtre (de menu, de mise en page etc), ce qui détruit le contenu précédent de l'écran, qui est restauré sans aucun problème (après fermeture de la fenêtre) en recopiant l'ancien contenu de l'écran depuis l'espace où il se trouve en mémoire vive :

BITBLT Adresse TO X,Y,Largeur,Hauteur [.Mode]

Cette syntaxe vous permet également de confectionner des graphiques dans la mémoire vive et de demander ensuite leur affichage à l'écran. Il faut cependant alors tenir compte du format utilisé par BITBLT pour stocker les données en mémoire : en effet, l'espace mémoire commencent par trois mots

- nombre de niveaux graphiques * 2
- largeur de l'extrait mesurée en pixels
- hauteur de l'extrait mesurée en pixel

Vous n'entrez ces données 'à la main' que si vous souhaitez élaborer de toutes pièces un graphique dans la mémoire vive. Dans tous les autres cas, c'est l'instruction BITBLT qui se charge de ce travail lors de la manipulation des données dans la mémoire vive.

La dernière variante citée au début de ce paragraphe mentionnait la possibilité de recopier le contenu d'un espace mémoire dans un autre espace mémoire. Utilité de cette manipulation : recopier des fragments de l'écran, confectionnés à l'origine en haute résolution (HIRES), dans un écran couleur :

BITBLT Secteur1 TO Secteur2,COLOR Couleur

La couleur entrée comme paramètre est celle que doit prendre un point lorsqu'il est activé.

5.3. L'affichage à l'écran

Comme vous le savez maintenant, il est possible de transférer et d'archiver dans la mémoire vive le contenu d'un fragment de l'écran ou de l'écran tout entier, ce qui le conserve au moins jusqu'à une éventuelle réinitialisation de la configuration, après quoi il est perdu. Il arrive cependant que l'on souhaite conserver le contenu d'un écran sur une disquette ou un disque dur, afin de pouvoir le recharger plus tard, surtout s'il s'agit d'une création personnelle que vous tenez à préserver de la destruction.

Il arrive encore plus souvent que l'on souhaite conserver des graphismes servant par exemple de masque ou de fenêtre d'accueil, pour les réintégrer dans un texte de programme. Il convient alors de sauvegarder le contenu de l'écran en 32000 octets (c'est-à-dire non comprimé) pour pouvoir le recharger en début de programme à l'aide de l'instruction adéquate. Il existe une instruction servant à sauvegarder le contenu de l'écran (32000 octets) sur disque :

BSAVE "Nom de fichier"

Pour le recharger, il suffit ensuite d'entrer l'instruction

BLOAD "Nom de fichier"

pour voir réapparaître à l'écran le graphisme sauvegardé dans le fichier. Sachez que le logiciel graphique DOODLE vous permet de sauvegarder vos graphismes sous forme non comprimée, ce qui permet de les recharger tout simplement par BLOAD. Sachez également que BSAVE et BLOAD peuvent être accompagnées de paramètres affinant quelque peu leur utilisation :

BLOAD "Nom de fichier",<adresse>

sert ainsi à recharger le fichier sous l'adresse spécifiée en mémoire vive ; j'espère que vous avez pensé auparavant à sauver l'ancien contenu de cet espace mémoire à l'aide de l'instruction MEMORY !

L'instruction BSAVE peut aussi servir à sauvegarder dans un fichier les données à partir d'une certaine adresse, mais vous devez alors préciser le nombre d'octets à enregistrer :

BSAVE "Nom de fichier", <adresse>, <quantité d'octets>

Les instructions BSAVE et BLOAD permettent de recharger des programmes machine dans la mémoire vive, sous des adresses précises ; vous pouvez aussi sauvegarder des contenus de variables en recopiant carrément tout le contenu de la mémoire vive sur une disquette, etc.

5.4. Déplacer et retourner des graphiques

A titre d'exemple, commençons par l'objet graphique le plus simple qui soit, à savoir un point ; vous pouvez, si vous le souhaitez, le remplacer par les quatre coins d'un rectangle ou le centre d'un cercle.

☐ Comment déplacer un objet

Comme la position de tout point est décrite précisément par ses coordonnées (X,Y), il suffit de procéder à des additions ou soustractions pour le déplacer (voir figure 5.1).

On peut résumer cela en une formule :

$$\begin{aligned} X2 &= X1 + \text{Déplacement} \\ Y2 &= Y1 + \text{Déplacement} \end{aligned}$$

Après son déplacement (sa translation), le point qui avait les coordonnées X1,Y1 possède désormais les coordonnées X2,Y2. On peut pousser cette translation à l'extrême et ramener tout d'abord l'objet à l'origine du système de coordonnées pour l'amener à sa

nouvelle position dans une deuxième étape. Il est ainsi possible de déplacer un rectangle au 'bout' de la flèche de la souris pour l'amener à sa nouvelle position.

☐ Comment tourner un objet

Pour faire tourner un objet sur lui-même (rotation), on recourt aux fonctions trigonométriques ; si nous avons utilisé l'addition et la soustraction pour déplacer l'objet, il nous faut maintenant le faire tourner selon un certain angle (voir figure 5.2).

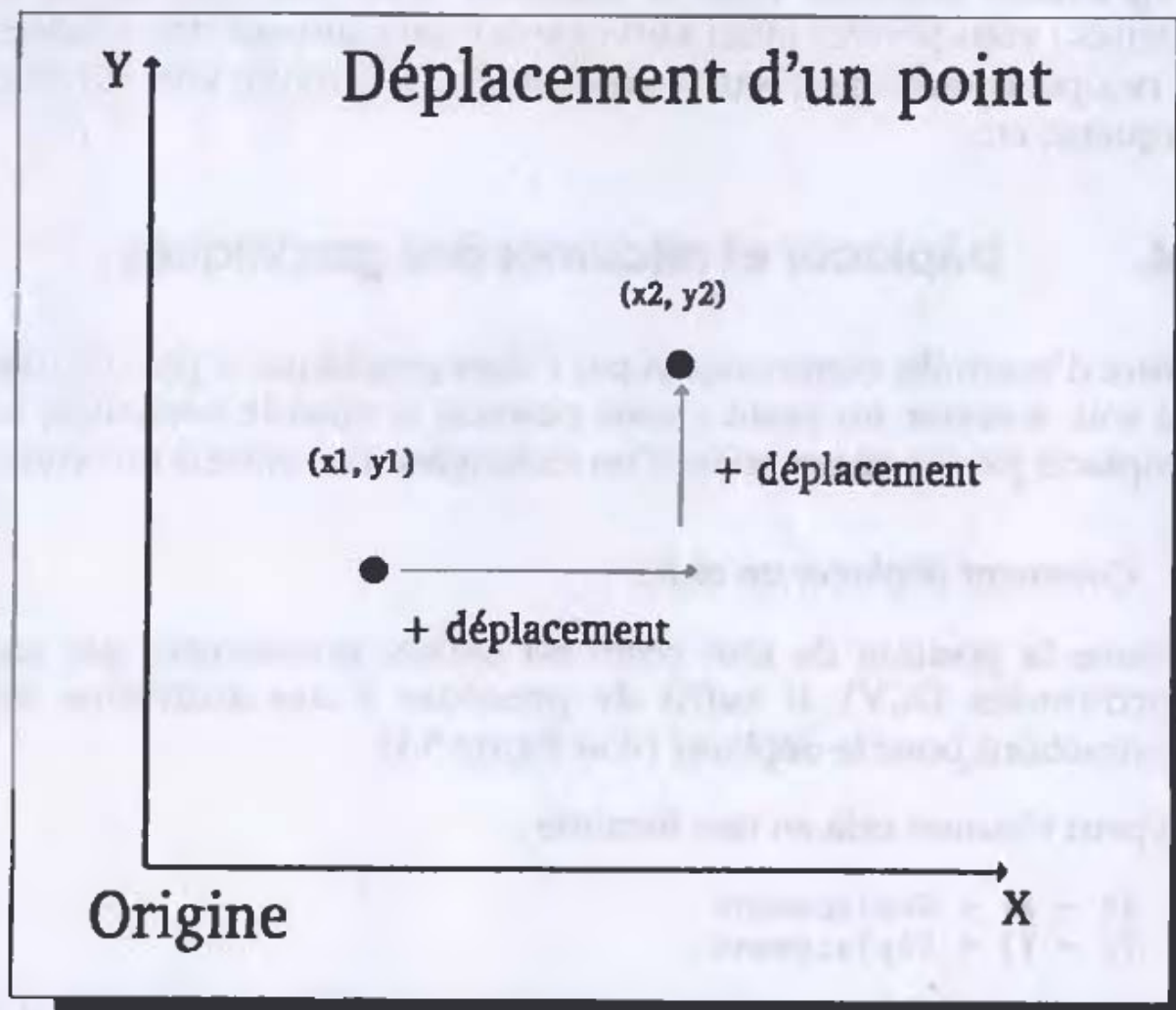


Figure 5.1 : Translation d'un point

Je souhaite vous épargner et m'éviter toutes les explications mathématiques relatives à cette opération de rotation, et je vous donne donc tout simplement la formule à respecter :

$$X2 = X1 * \cos(B) - Y1 * \sin(B)$$

$$Y2 = X1 * \sin(B) + Y1 * \cos(B)$$

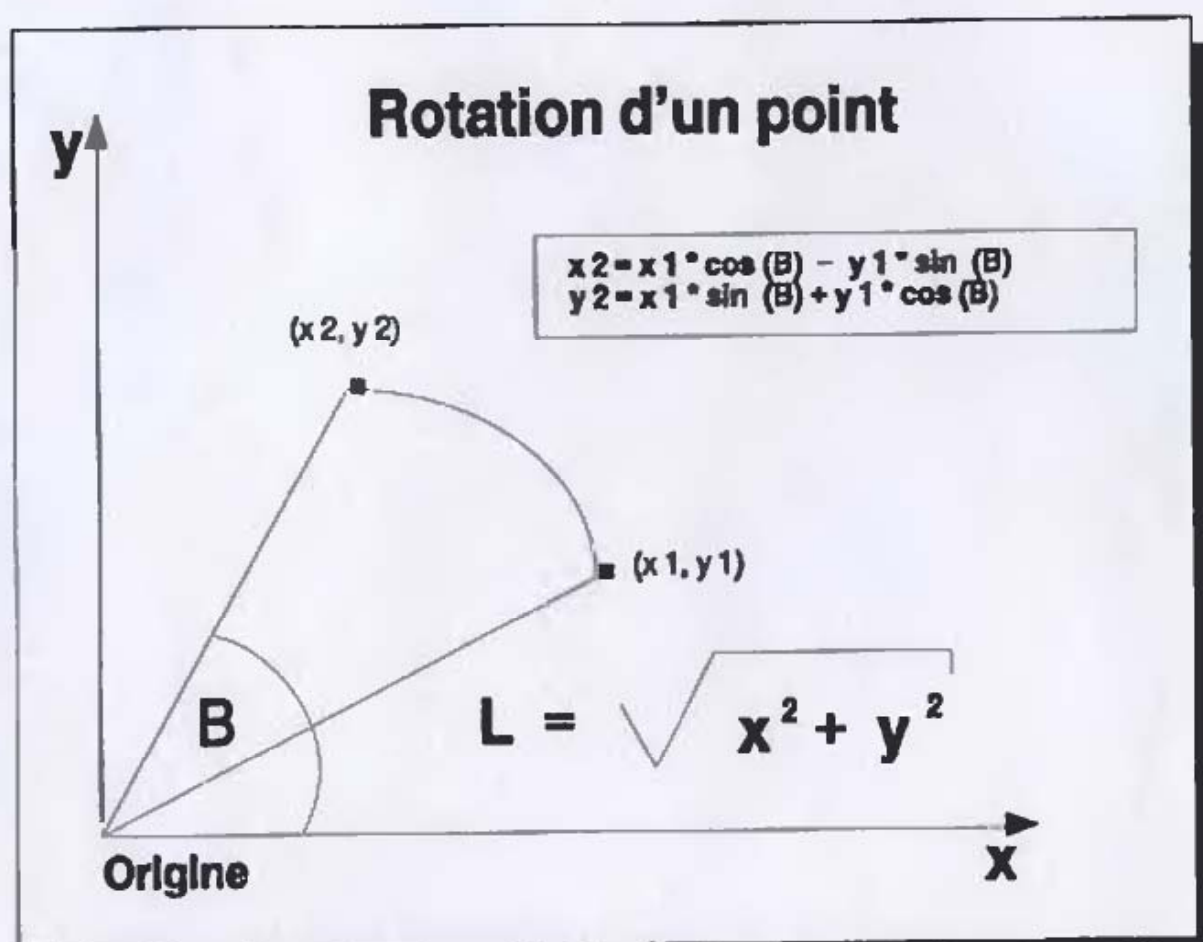


Figure 5.2 : Rotation d'un point

Il est important de noter que les coordonnées des points sont données en degrés et minutes. Les angles sont donnés en degrés et minutes. Les longueurs sont données en mètres.

$$\begin{aligned} \text{Longueur} &= \sqrt{(\Delta X)^2 + (\Delta Y)^2} \\ \text{Angle} &= \arctan\left(\frac{\Delta Y}{\Delta X}\right) \end{aligned}$$

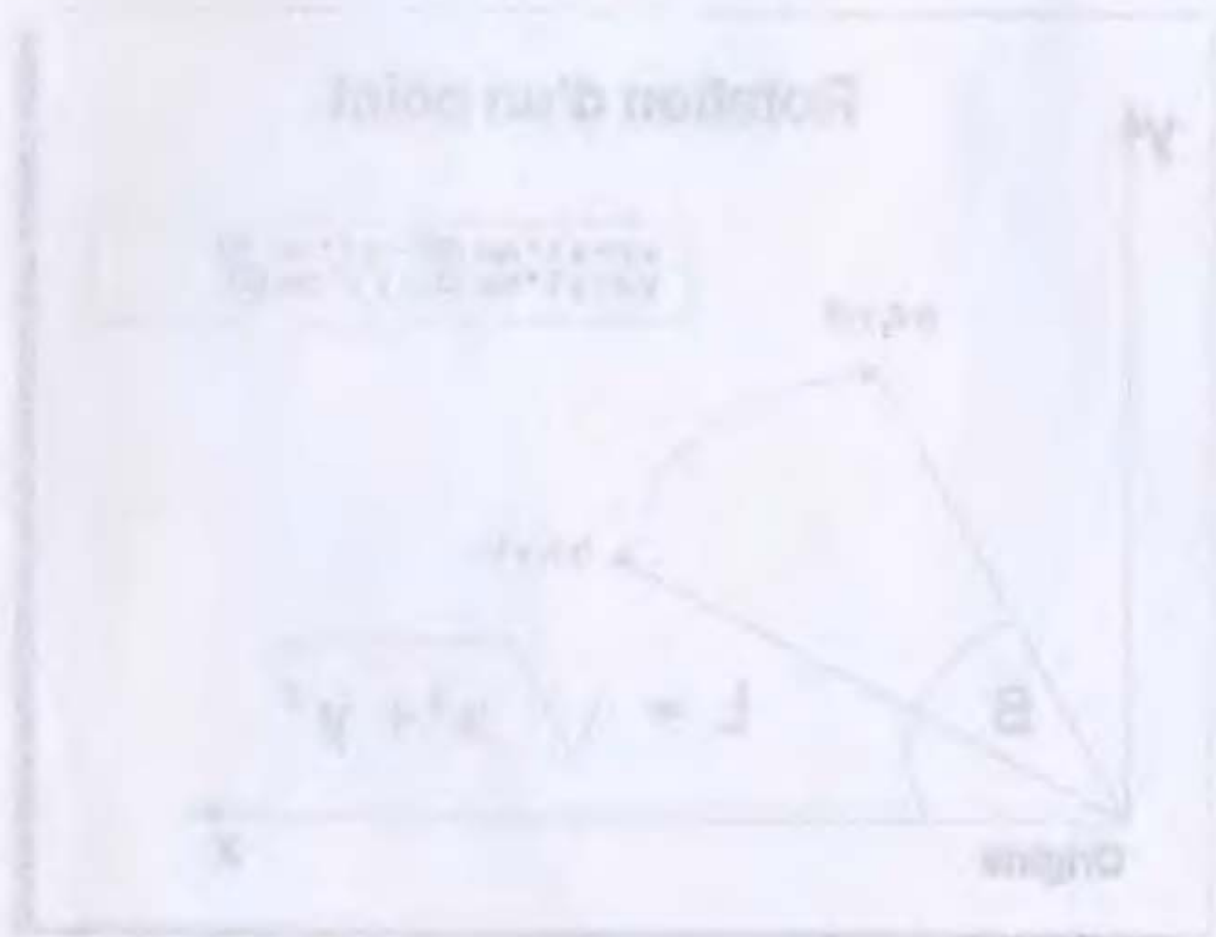


Figure 1.1 : Calcul de la distance

Chapitre 6

GEM

Dans l'Atari-ST, le GEM (Graphics Environment Manager = gestionnaire d'environnement graphique) met à votre disposition de nombreuses routines, qui vont des plus simples (VDI) aux plus complexes facilitant considérablement le travail (RCS = ressources). Ces 'ressources' sont des fichiers contenant divers objets que l'on peut faire apparaître à l'écran. Le GEM se charge entièrement de la gestion de ces formulaires et masques (se composant eux-mêmes de divers objets graphiques) et permet de plus d'en sélectionner ou d'en activer certains (par exemple les 'boxes' ou 'boîtes') à l'aide de la souris ou en entrant un texte dans un masque. Le programme principal reprend le contrôle du déroulement des opérations après que la condition d'interruption de la routine ait été remplie (par exemple, le fait de cliquer sur un bouton ANNULER). On peut alors vérifier les options sélectionnées par l'utilisateur dans le formulaire géré par le GEM.

Il convient de respecter certaines conventions pour que le GEM puisse gérer correctement de tels masques ou formulaires. L'ordinateur doit savoir quel est le rapport existant entre les objets et posséder une représentation graphique de l'ensemble de l'objet et des éléments dont il se compose. C'est pourquoi chaque élément est identifié par un intitulé de 24 octets, et est répertorié dans une liste

d'objets graphiques qui contient les coordonnées (X,Y, Largeur, Hauteur) des objets précédents ou suivants, ainsi que d'autres informations. Cette liste d'objets peut elle-aussi être sauvegardée dans un fichier-disque pour être rechargée (en début de programme) si nécessaire. Comme il est assez pénible et difficile de créer une telle liste d'objets, il est possible de la faire confectionner par un 'ressource construction set' (RCS). C'est ce que nous allons étudier plus en détail dans ce chapitre.

6.1. Comment travailler avec RCS

Comme nous venons de le voir, il existe une liste répertoriant tous les éléments graphiques composant un 'formulaire' (masque de mise-en-page de l'écran). Chaque élément est identifié par un intitulé de 24 octets qui contient toutes les informations relatives à l'objet concerné. Cette liste d'objets est gérée dans la mémoire vive comme une arborescence.

Les informaticiens appellent cela une arborescence, car ce genre de liste se décompose (comme un arbre) en différents niveaux, depuis la racine (root) jusqu'aux branches et ramifications. Illustrons cela par un exemple concret.

Imaginez une boîte (box) contenant deux boîtes plus petites ; la boîte de droite contient à son tour une boîte encore plus petite. Nous avons alors trois niveaux :

- premier niveau : la grande boîte
- deuxième niveau : les deux boîtes
- troisième niveau : la boîte la plus petite, dans celle de droite.

Le premier niveau est considéré comme la 'racine', puisque c'est à partir de lui que l'on construit les autres niveaux ; c'est en même temps l'arrière-fond de l'ensemble du formulaire. On peut considérer que les branches sont les deux boîtes plus petites, se

trouvant dans la plus grande, et qui sont toutes deux au même niveau. On les désigne aussi par le mot 'enfant' ou 'descendant' (children), la boîte de départ étant alors le 'parent' (anglais : parent).

Comment représenter ces rapports généalogiques ? C'est très simple : un objet dont les limites ne dépassent pas celles d'un autre objet plus grand dans lequel il se trouve est considéré comme d'un niveau hiérarchiquement inférieur. Tout déplacement de l'objet 'parent' mène logiquement à un déplacement semblable des objets qui lui sont soumis ('enfants'). Illustrons cela par un schéma :

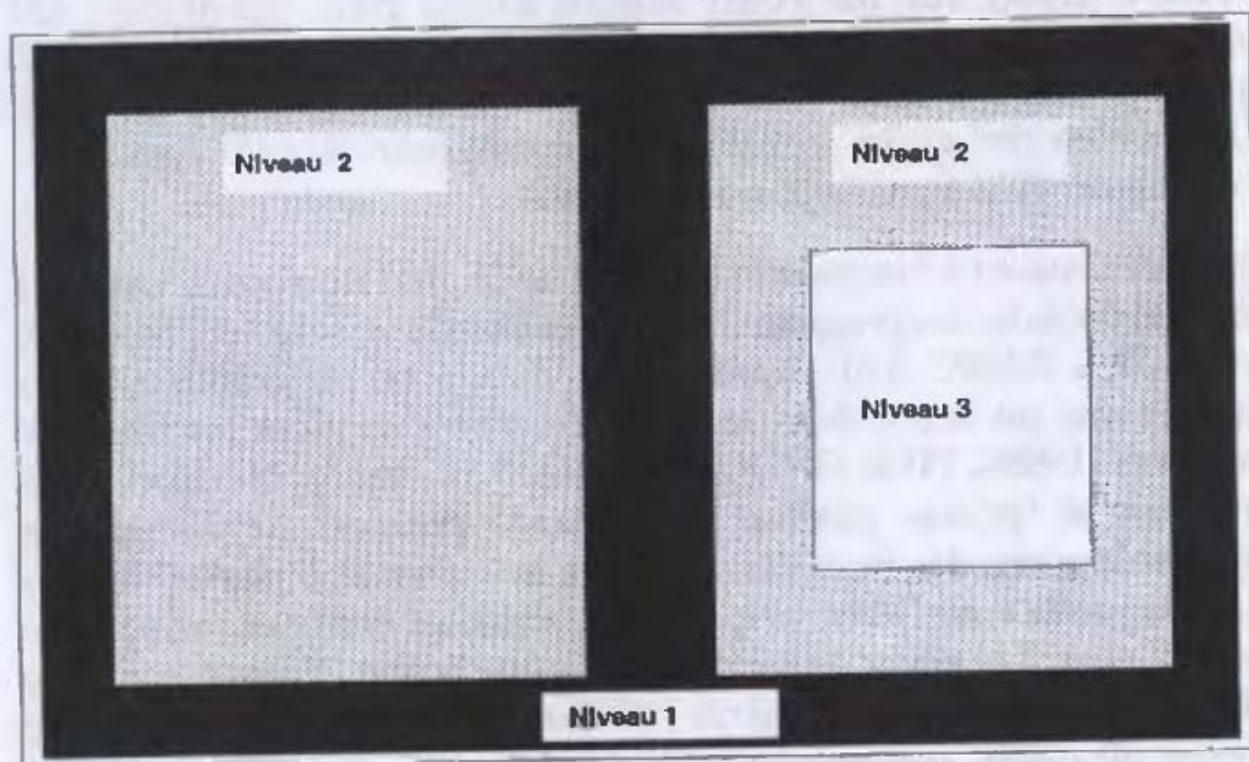


Figure 6.1 : Arborescence à trois niveaux.

Il est tout simplement horrible d'avoir à écrire soi-même 'à la main' une telle arborescence graphique. Cela devient par contre un jeu d'enfant lorsqu'on dispose d'un RCS (ressource construction set), véritable boîte-à-outils du programmeur harassé. Vous devez absolument vous procurer un programme de ce type si vous tenez à embellir vos propres programmes de créations graphiques recourant au GEM. Le RCS de Digital Research est quasi donné : voyez cela avec votre revendeur Atari.

D'autre part, les firmes commercialisant des logiciels savent bien comme il est difficile de créer des 'ressources', et beaucoup d'entre elles diffusent des langages de programmation ressemblant au RCS. Vous pouvez ainsi détourner le RCS du compilateur C de Mégamax. Personnellement, je ne ferai pas de publicité pour l'un ou l'autre produit, car j'utilise plusieurs RCS différents dans mon travail quotidien. Il est ainsi possible d'exploiter au mieux leurs possibilités et de créer des objets graphiques théoriquement inconcevables.

Le RCS de Digital Ressource n'est pas utilisable dans son ancienne version (1.xx) sur les Atari MEGA-ST ou STE, car il ne peut fonctionner avec le Blitter-TOS (plus précisément : c'est le Blitter-TOS qui ne peut fonctionner avec ce programme, ce qui revient au même). On ne s'en tire qu'en initialisant la configuration avec l'ancienne version du TOS à partir d'une disquette.

Mais venons-en à l'utilisation du RCS de Digital Ressource, car c'est le plus répandu des programmes de création de ressources (il est livré avec GFA BASIC 3.0). Après avoir chargé ce programme, vous arrivez sur un écran dont la barre de menu contient les intitulés suivants : DESK, FILE, OPTIONS et GLOBAL. Sur le côté droit, vous trouvez le 'presse papier' (clipboard) permettant de stocker provisoirement des données ainsi que la 'corbeille à papier' (trash) dans laquelle vous 'jetez' ce que vous souhaitez éliminer : on appelle généralement ce genre de pictogramme une 'icône'. Votre écran vous affiche également deux fenêtres (windows) : la fenêtre inférieure, de forme allongée, présente les différentes arborescences que vous pouvez construire à l'aide du RCS et nous l'appellerons 'fenêtre des objets'. La fenêtre supérieure sert à élaborer un formulaire, et nous l'appellerons 'fenêtre de travail'. Pour pouvoir retravailler et modifier un objet, il vous faut le prendre dans la fenêtre des objets et le transporter à l'aide de la souris dans la fenêtre de travail. Pour ce faire, cliquez sur le symbole concerné (par exemple FREE), maintenez la touche gauche de la souris enfoncée et tirez ce symbole jusque dans la fenêtre de travail. Ce genre de déplacement d'icône s'appelle, en bon français informatique, le 'dragging'.

Vous voyez maintenant s'ouvrir une boîte qui vous permet de redonner un nom à l'objet en question, car un intitulé comme TREE1 reste trop vague. Pourquoi renommer ces objets ?

Nous avons vu que chaque objet est répertorié dans une liste, dans laquelle il est identifié clairement par un numéro qui lui est propre. En effet, l'ordinateur ne peut guère se donner la peine de vous signaler en toutes lettres que "l'objet sur lequel l'utilisateur vient de cliquer est bien le petit rectangle strié et grisé se trouvant dans la moitié gauche supérieure de l'arborescence". Pour pouvoir manipuler un objet dans un programme (ou identifier l'objet sélectionné par un utilisateur), l'ordinateur se sert du genre de code qu'il apprécie le plus, à savoir un nombre (qui est ici un numéro) désignant et identifiant l'objet concerné.

Au cours des manipulations que vous faites subir à l'arborescence, il se peut que vous en veniez à inverser les rapports de parentés, du niveau hiérarchique 1 au niveau hiérarchique 2 (parent/enfant), ce qui revient à donner un autre numéro aux objets mis en cause. Cela peut engendrer des problèmes non négligeables : admettons que vous souhaitiez, dans votre programme, vérifier si l'utilisateur a cliqué sur l'objet numéro 2 et que vous avez pour ce faire écrit une boucle IF... THEN (Ret% représentant le numéro de l'objet sélectionné) comme celle-ci :

```
IF Ret% = 2 THEN
```

```
...
```

```
<insérer ici la réaction prévue dans le programme>
```

```
...
```

```
ENDIF
```

Cela fonctionne correctement tant que l'objet concerné ne change pas de numéro. Mais si par malheur, en modifiant l'arborescence, vous modifiez aussi le numéro de l'objet concerné, votre beau programme n'est plus opérationnel, puisque vous faites référence dans la boucle IF...THEN à un numéro d'objet qui n'est plus valable. Il ne vous reste plus qu'à modifier la boucle dans le texte même du programme : je vous souhaite bien du plaisir !

Dans le langage de programmation C (n'oublions pas que le GEM est justement issu de ce langage) il existe un processus particulier appelé préprocesseur. Ce dernier 'ausculte' l'ensemble du texte source, avant son traitement par le compilateur, et remplace certaines formulations par des instructions définies par avance. L'instruction du préprocesseur

```
#define Tree1 0
```

a pour effet de remplacer par un 0 toutes les mentions de Tree1 dans le texte du programme : partout où le préprocesseur rencontre la chaîne de caractères 'tree1', il la remplace par le nouveau texte, à savoir un zéro dans notre exemple. Je vous raconte tout cela car le RCS se sert de ce préprocesseur et génère un fichier HEADER flanqué de son extension '.h'. Comme vous vous en doutez, ce fichier Header contient des instructions empruntées au préprocesseur sous forme de #define. Chaque #define est suivi du nom de l'objet et de son numéro, ce qui simplifie considérablement la tâche du programmeur en langage C, qui n'a plus qu'à reprendre dans son texte de programme le nom attribué à l'objet pour l'appeler. Et c'est justement ce nom (nous voilà revenu à notre point de départ) que vous pouvez attribuer dans la boîte de dialogue ci-dessus mentionnée.

Comme le Basic ne dispose pas d'un auxiliaire aussi précieux et confortable que le préprocesseur, il faut nous débrouiller autrement. Nous allons passer le numéro de l'objet à une variable, et entrer le nom de la variable dans la boîte de dialogue du RCS.

Sachez que si vous travaillez avec un RCS générant un fichier pour un préprocesseur-C, il faut d'abord transformer ce fichier avant de l'intégrer dans votre texte de programme. Il vaut mieux l'intégrer en début de programme, pour que les variables soient pourvues de valeurs correctes dès le début du programme et éviter que le formulaire ne se compose que d'objets du niveau 1, la racine. Ainsi

```
#define Tree1 55
```

devient en Basic

```
Tree1 = 55
```


A partir de ce moment, vous appelez l'objet par le biais de la variable Tree1, puisque cette dernière contient le numéro de l'objet. Je recommanderais à tous ceux qui possèdent un RCS ne générant pas seulement un fichier préprocesseur pour le langage C mais aussi pour le GfA Basic d'intégrer le fichier dans leur programme par le biais du menu LOAD Block *.* (je signale que le GfA Basic 3.00 est doté d'un tel RCS, ainsi que le compilateur C du Mégamax de Application System Heidelberg). Au niveau du chemin d'accès, il faut remplacer l'extension d'origine par '.LST' si l'on tient à ce que le nom du fichier s'affiche dans le File-Selector-Box (boîte de sélection d'objet) de l'Omikron Basic.

Attribuez un nom facile à retenir à votre objet puis appuyez sur <return> ou cliquez sur OK. Vous pouvez alors retravailler l'objet choisi dans la fenêtre de travail.

La fenêtre d'objets contient, de la gauche vers la droite, les symboles suivants (que vous pouvez modifier) :

☐ Unknown (point d'interrogation)

Il ne s'agit pas vraiment d'une arborescence puisqu'il s'agit d'une structure encore inconnue. Elle est utile lorsque le RCS refuse de charger le fichier DEF complémentaire d'un fichier RCS contenant les indications relatives au type des différents objets. Dans ce cas, l'objet est désigné comme étant inconnu et doit être renommé avant de faire l'objet d'un nouveau traitement dans la fenêtre de travail.

☐ Free

Cet objet est le fondement de toutes sortes de formulaires. Vous déterminez vous-même la position de chacun des descendants de cet objet. Inconvénient : les arborescences créées sur un écran de haute résolution peuvent sembler décalées lorsque vous passez dans une autre résolution.

☐ Dialogue

S'utilise pour positionner les objets dans une trame, qui dépend de la résolution graphique momentanément activée. Sur un écran haute-résolution, les objets sont placés de telle sorte que les coordonnées X sont divisibles par huit et les coordonnées Y par seize.

☐ Menu

Cette arborescence sert à élaborer des menus déroulants (Pull-down-menus) qui 's'ouvrent' à partir du bord supérieur de l'écran.

☐ Alertbox

Cet objet permet de créer des panneaux d'avertissement (alert-box ou alarm-box) envoyant des messages d'erreur ou des demandes de confirmation (voir FORM_ALERT en Omikron Basic).

Pour confectionner votre propre objet, tirez le symbole correspondant à l'arborescence voulue depuis la fenêtre des objets jusqu'à la fenêtre de travail. Ceci étant fait, vous avez à votre disposition les objets qui composent l'ensemble de l'arborescence présente dans la fenêtre de travail et que vous devez ramener vers l'objet-parent. Vous pouvez alors les agrandir (cliquer sur le coin inférieur droit, maintenez la touche de la souris enfoncée et déplacez la souris) ou les rendre plus petits. Vous pouvez aussi modifier leur position : amenez la flèche de la souris en plein milieu de l'objet, appuyez sur la touche de la souris et déplacez l'objet. Si vous amenez l'objet dans le périmètre définissant un autre objet, vous modifiez toute la structure de l'arborescence dès que vous l'avez sorti de son périmètre d'origine. Vous pouvez par contre le déplacer sans aucun problème à l'intérieur du même objet-parent. Voici les différents types d'objets utilisables :

☐ Button (touche, bouton)

Cet objet fonctionne comme une touche ; on peut cliquer dessus à l'aide de la souris ou, sous certaines conditions, le sélectionner rien qu'en appuyant sur la touche <enter> ou <return>.

☐ String (ligne d'édition)

Cet objet peut mémoriser un texte ; on l'utilise lorsqu'on attend la saisie d'une réponse dans une boîte de dialogue (par exemple pour faire un en-tête) :

Ftext (EDIT : _____)

Par la suite, ce texte pourra être modifié par l'utilisateur.

☐ Fboxtext

Semblable à Ftext, la différence étant que le texte est présenté dans une boîte pouvant comporter si nécessaire un motif de remplissage.

☐ Ibox

Cette boîte sert de 'parent' pour d'autres objets se trouvant à leur propre niveau, toujours à l'intérieur de Ibox.

☐ Box

Box confère un encadrement à d'autres objets, la boîte pouvant être remplie par un motif.

☐ Text

Il s'agit ici d'un texte, qui, à la différence de Ftext, n'est pas modifiable par l'utilisateur.

☐ Boxchar

Cet objet contient un caractère : il peut être sélectionné à l'aide de la souris ou en appuyant sur <enter> ou <return> ; cette forme d'objet est bien adaptée pour les symboles de défilement <flèche vers le haut> <flèche vers le bas>.

☐ Boxtext

Devinez un peu ce que contient cet objet ? Très juste : un texte encadré par une boîte.

☐ Icon

Nous avons déjà rencontré ce mot 'icône' ; il s'agit par exemple des pictogrammes représentant les différentes unités de disque ou la corbeille à papier au niveau du bureau GEM. On en trouve même dans le RCS : le presse-papier (clipboard) et la corbeille.

☐ Image

Sert à visualiser l'élaboration des arborescences ; ne peut être sélectionné, au contraire des icônes !

☐ Title (titre d'un menu)

Ce type d'objet n'est admis qu'à l'intérieur d'un menu, et n'apparaît donc dans la fenêtre des objets que lorsque vous procédez à l'élaboration d'une arborescence de menu. Il convient de refermer la fenêtre de travail avant de passer à l'arborescence suivante : pour ce faire, cliquez dans la petite case de fermeture (closer) se trouvant en haut à gauche de la fenêtre ; les différents types d'arborescences réapparaissent ensuite dans la fenêtre des objets.

Il est possible de dupliquer des objets : appuyez sur la touche <shift> avant de cliquer sur l'objet voulu dans la fenêtre de travail et tirez-en une copie en maintenant la touche de la souris enfoncée.

Vous pouvez manipuler plus finement un objet en "l'ouvrant" (cliquer dessus) et en modifiant ses attributs. Nous ne parlerons pas ici de ces différents attributs, puisque nous y reviendrons plus longuement dans le prochain chapitre, lorsque nous étudierons la structure interne des ressources.

Armé de ce savoir, vous ne devriez quasiment plus rencontrer de problème pour confectionner vos propres 'ressources' avec RCS. Amusez-vous un peu à découvrir les possibilités de RCS ; d'autres programmes permettent à peu près les mêmes choses (ex : K-resource) mais il m'est impossible ici de faire le tour de tous les produits présents en ce domaine sur le marché.

L'Omikron Basic est fourni avec un RCS plus ou moins complet (plutôt moins que plus) et qui n'est pas bien fameux.

6.2. La programmation GEM sous l'Omikron Basic

Je peux vous garantir que vous allez quelque peu peiner dans ce chapitre, mais n'ayez pas peur : une fois qu'on s'est habitué à ce genre de programmation, je vous assure que rien au monde ne pourra plus vous y faire renoncer (et je n'exagère pas).

☐ Comment déclarer une application

La méthode est toujours la même, il faut commencer par activer le GEM, le gestionnaire d'environnement graphique du TOS. Cela serait assez fastidieux, mais la disquette de l'Omikron Basic vous livre une bibliothèque simplifiant considérablement la programmation sous GEM. Pour programmer sous GEM, vous devez donc commencer par charger le fichier

GEMLIB.BAS

qui met à votre disposition une foule d'instructions vous permettant d'exploiter les possibilités du GEM. La première instruction à utiliser est celle qui sert à déclarer une application :

App1_Init

et à activer le GEM. Avant de sortir du programme en cours, pensez à sortir du GEM, faute de quoi vous risquez de planter le système. La bibliothèque GEMLIB.BAS vous offre pour cela l'instruction

App1_Exit

Après avoir activé le GEM, vous voici donc à pied d'oeuvre.

Activer le GEM :

- ❶ charger le fichier RCS se trouvant sur la disquette
- ❷ rechercher l'adresse de l'arborescence
- ❸ lancer Form_Dial (sauver le contenu de la mémoire vive)
- ❹ élaborer votre formulaire à l'écran
- ❺ passer le contrôle du formulaire au GEM
- ❻ lancer Form_Dial (libérer la mémoire vive)

☐ Désactiver le GEM

Essayez de bien vous mettre ce schéma dans la tête, car toute programmation d'une boîte de dialogue se fait en respectant cette démarche.

A titre d'exemple, je me suis fixé pour objectif dans ce chapitre de confectionner ma propre boîte de sélection d'objet (File selector box), que je pourrai ensuite insérer dans un programme quelconque. Il faut d'abord créer un fichier ressource correspondant, à l'aide d'un RCS. N'hésitez pas à mettre la main à la pâte : prenez un Resource Construction Set et bricolez vous-même une boîte (voir figure 6.2).

Vous devez en tout premier lieu bien réfléchir pour savoir quelle arborescence vous allez utiliser dans votre boîte de sélection d'objet. On pense évidemment à 'Dialog' (mais 'Free' ne serait pas une erreur). Tirez l'icône 'Dialog' jusque dans la fenêtre de travail et

donnez-lui le nom FILSEL (abréviation de File Selector) qui sera facile à retenir. Pour entrer un nom, appuyez simultanément sur la touche <shift> et la touche des caractères adéquats : le programme ne tolère que des noms en majuscules pour les fichiers ressources.

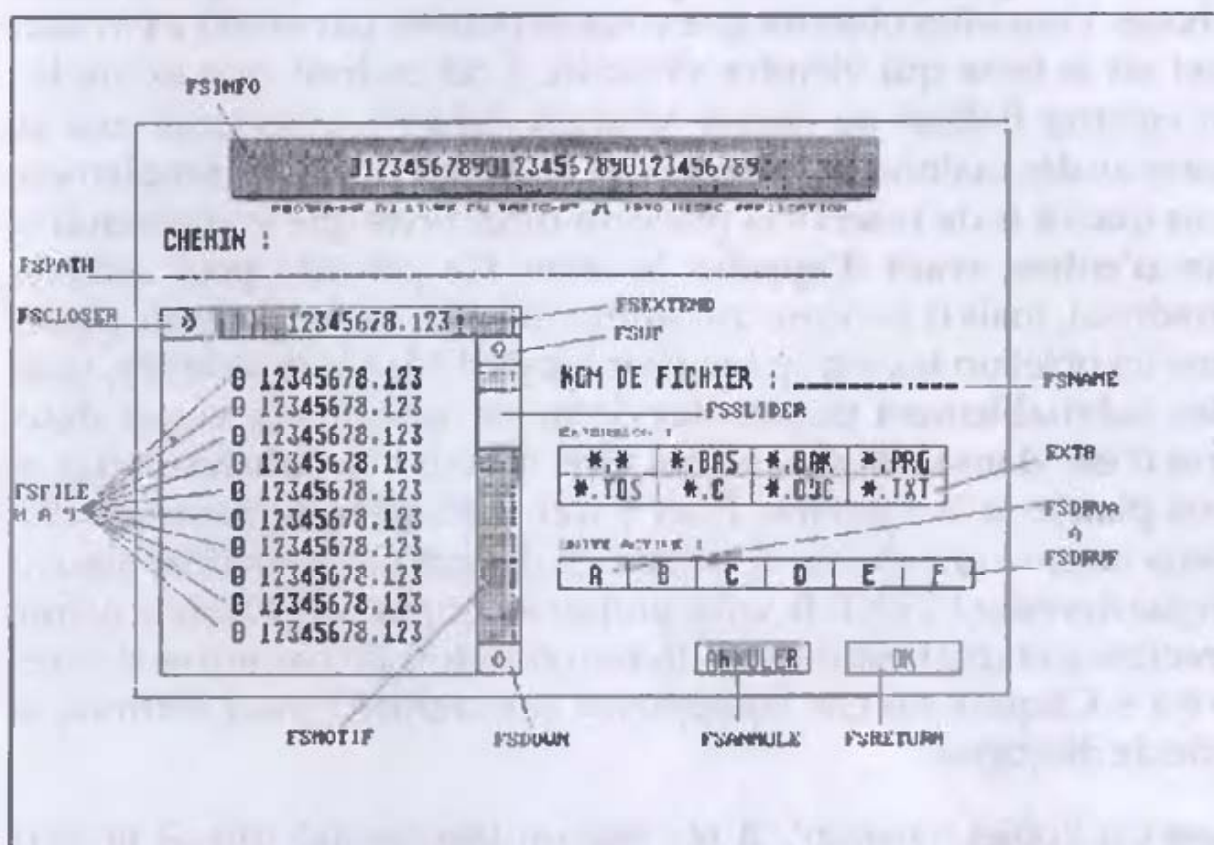


Figure 6.2 : détail de la composition d'une boîte de sélection

Voilà notre boîte de dialogue ouverte, il suffit d'y entrer les différents objets. Commençons par le haut. Le haut de la boîte contient tout d'abord la ligne d'information (FSinfo) pour laquelle nous utilisons un objet BOXTEXT, que nous baptisons, du fait de sa fonction, FSinfo. Après avoir amené cet objet dans la boîte de sélection, essayez de lui donner le même aspect que ci-dessus dans la figure 6.2. Exécutez un double-clic sur cet objet pour voir apparaître une boîte de dialogue du RCS dans laquelle vous entrez les flags et paramètres nécessaires. La ligne d'information ne réclame d'ailleurs aucun flag. Il faut tout de même cliquer sur le champ 'C' sous 'justify' (justification du texte dans la boîte) pour que le texte vienne se placer en plein milieu, à la même distance des deux bords : c'est ce qu'on appelle 'centrer' un texte (anglais : center) d'où l'abréviation 'C'. 'Border' vous permet

d'encadrer un bouton par un trait plus épais tandis que 'Background' se charge de créer un arrière-fond. Bien entendu, vous pouvez choisir un motif de remplissage à votre goût ou laisser l'arrière-fond blanc.

La dernière ligne sert à entrer, après PTEXT, le texte qui figurera dans la boîte. Vous allez objecter que vous ne pouvez pas savoir à l'avance quel est le texte qui viendra s'inscrire à cet endroit (par exemple : 'enregistrer fichier' ou encore 'charger fichier'), alors pourquoi en entrer un dès maintenant, dans le fichier ressource ? Tout simplement pour que ce texte réserve la place du futur texte que vous prendrez soin d'entrer avant d'appeler la boîte. Ce procédé peut sembler paradoxal, mais il a encore une autre utilité : si vous tentez de placer dans un objet un texte trop long par rapport à la place réservée, vous allez inévitablement perdre des données (essayez de verser deux litres d'eau dans une cruche d'un litre, et vous verrez le résultat !), et vous planterez le système. Pour parer à ces désagréments, il vaut mieux inscrire une chaîne de caractères de remplissage suffisamment longue derrière PTEXT. Je vous propose d'entrer soit 30 fois le même caractère soit (pour faciliter le décompte) 3 fois la chaîne des chiffres de 0 à 9. Cliquez sur OK ou appuyez sur <return> pour sortir de la boîte de dialogue.

Passez à l'objet 'chemin'. Il n'a pas un bien grand rôle et ne sert quasiment qu'à 'décorer' la boîte de dialogue. Tirez l'objet 'Text' dans la fenêtre de travail et remplacez le texte (double-clic sur l'objet) par le mot PATH (ou CHEMIN ou REPERTOIRE).

Plus important maintenant : la ligne d'édition du chemin d'accès, que nous nommons FSpaht. Elle se compose d'un objet du type Ftext doté du flag 'Editable'. Veuillez faire suivre 'Text' de 50 fois le même caractère (chaîne qui servira à réserver la place nécessaire pour le chemin d'accès) et faire suivre 'Template' de 50 fois le caractère P.

Il s'agit ensuite de construire la fenêtre dans laquelle viendront s'afficher les noms des fichiers. Prenez un objet 'Box' et tirez-le jusque dans la fenêtre de travail ; amenez la flèche de la souris sur le coin inférieur droit de la fenêtre, maintenez la touche gauche de la souris enfoncée et modifiez la taille de la fenêtre pour qu'elle ressemble plus

ou moins à celle de la figure 6.2 ci-dessus (qui est un peu grande). Après quoi vous enregistrez les objets suivants comme étant les 'enfants' de cette boîte :

Type	Nom	Flags	Autres précisions
Boxchar	FScloser	Touchexit	'closer' sous forme de lettre derrière CHAR (Control E)
Boxtext	FSextend	Touchexit	Justification : Centrer Arrière-fond : pointillés PTEXT: 12345678.123
Boxchar	FSup	Touchexit	CHAR: flèche vers le haut (Control A)
Boxchar	FSdown	Touchexit	CHAR: flèche vers le bas (Control B)
Box	FSmotif	Touchexit	Arrière-fond : pointillé grisé allant de FSup à FSdown
Box	FSslider	Touchexit	enfant' de FSmotif ; taille calculée par le programme
Text	FSFile0 à FSfile9	Selectable Touchexi	PTEXT: '0 12345678.123' Radio Button

Après avoir placé tous ces objets, ramenez l'objet 'parent' à la bonne taille, de façon à ce qu'il encadre tous les objets qu'il doit contenir. Vous avez ainsi terminé la partie la plus ardue de votre travail, le reste n'est plus que formalité.

La figure ci-dessous montre l'élaboration par étapes de la boîte de sélection d'objet :

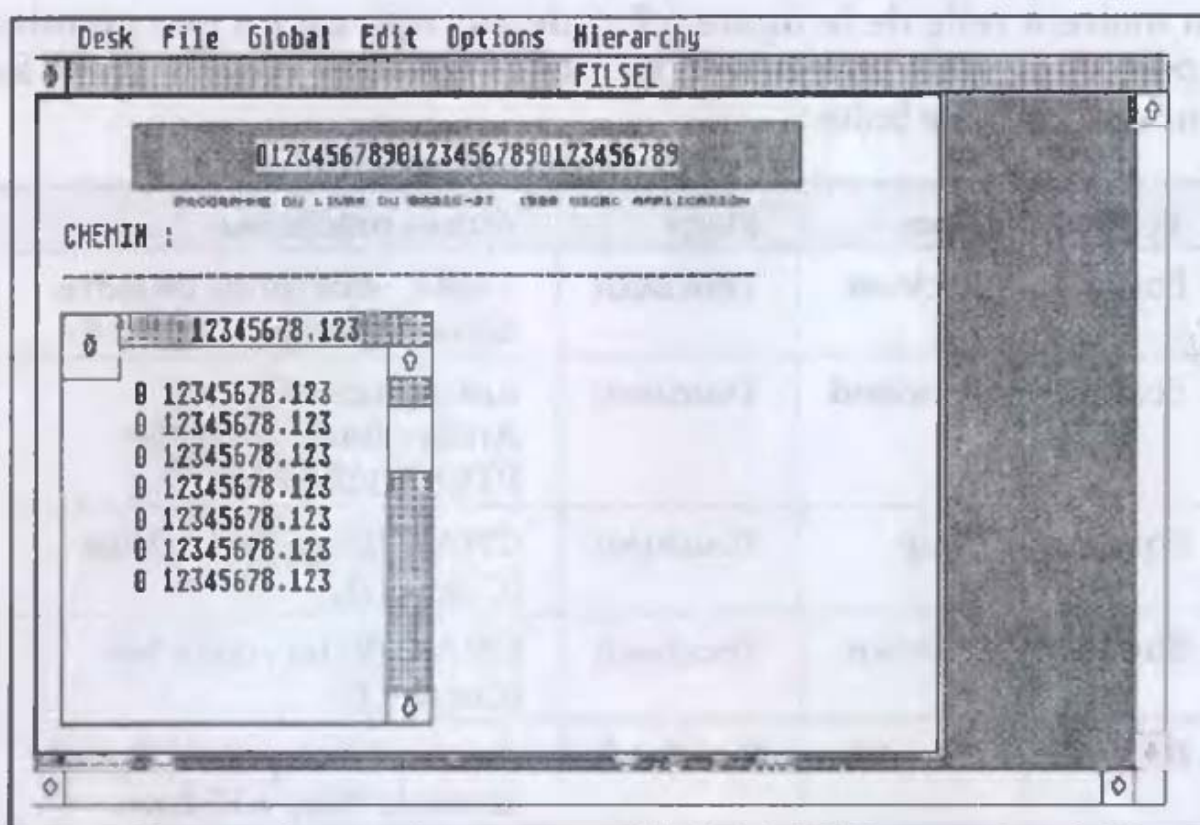


Figure 6.3 : notre boîte de sélection d'objet prend forme

Type	Nom	Flags	Autres précisions
Text	---	---	PTEXT : Fichier :
Ftext	FSname	Editable	TEXT : 12345678.123 Template : 'ppppppppp.ppp'
Box	---	---	encadre les boutons de choix de l'unité de disque. Normalement invisible, Border color : 0
Boxcha	FSdrva à FSdrvf	Selectable	CHAR: A jusque F Radio Button 'enfant' de la boîte précédente
Box	---	---	entoure chaque bouton de choix d'une extension

Type	Nom	Flags	Autres précisions
Boxtext	Ext1 à Ext8	Selectable Touche exit	contient les extensions Radio Button Le dernier objet doit compter 5 caractères puisqu'on le modifie depuis le programme
Button	FSannul	Selectable Exit	Ptext: ANNULER
Button	FSreturn	Selectable Exit	Ptext: CONFIRMER

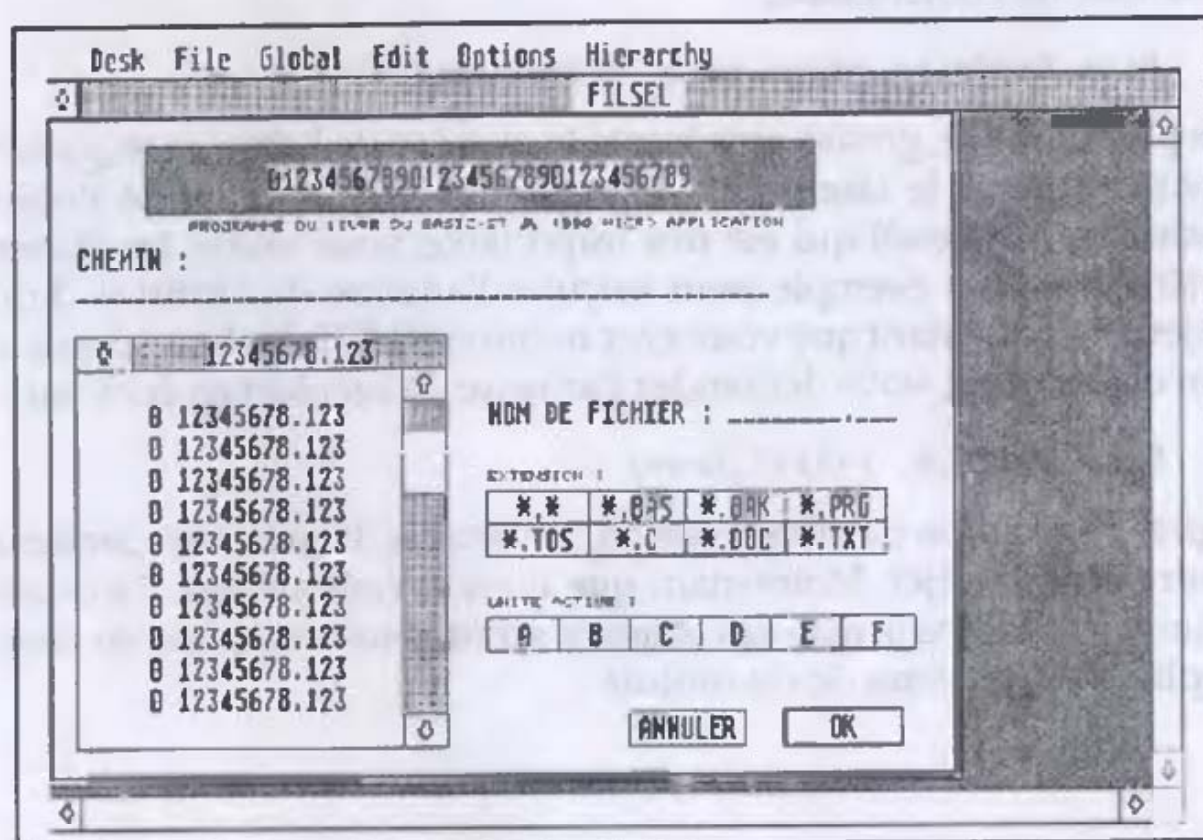


Figure 6.4 : La boîte de sélection d'objet terminée...

Une fois la boîte de sélection terminée, nous la sauvegardons sur disque(tte) et insérons le fichier HEADER élaboré sous RCS dans le programme. Si vous avez bien repris les mêmes noms et intitulés que ceux mentionnés ci-dessus, il vous suffira de rectifier (dans le listing de la boîte de sélection) un peu certains objets qui s'avèreraient mal dimensionnés.

□ Comment construire une arborescence

Dans ce qui va suivre, je suppose que vous avez bien chargé le fichier ressource et qu'il se trouve donc dans la mémoire vive de l'ordinateur. Comme chaque objet est identifié par un intitulé de 24 octets, et que ces objets sont numérotés en continu, l'objet-racine (root) portant le numéro zéro, il est facile de retrouver les données relatives à un objet isolé, à l'aide de la formule :

$$\text{Objet_racine} + 24 * \text{numéro de l'objet}$$

La bibliothèque GEM contient une fonction permettant de retrouver l'adresse de l'objet-racine :

```
Rsrc_Gaddr(re_gtype, re_gindex, re_gaddr)
```

Le paramètre `re_gindex` représente le numéro de l'objet et `re_gaddr` contient, après le lancement de la procédure, l'adresse de l'objet racine (`re_gtype=0`) qui est très importante pour toutes les étapes ultérieures (par exemple pour calculer l'adresse de l'intitulé d'un objet). En admettant que vous ayez nommé `FILSEL` l'arborescence et son objet-racine, vous demandez l'adresse de cet objet en écrivant :

```
Rsrc_Gaddr(0, Filsel, Tree)
```

Après l'exécution de cette fonction, la variable `Tree` (arbre) contient l'adresse de l'objet. Maintenant que nous savons obtenir l'adresse d'un intitulé dans la liste des objets, il serait peut-être temps de vous expliquer le contenu de cet intitulé :

Offset	Contenu
+0	objet suivant
+2	premier objet
+4	dernier objet
+6	type de l'objet
+8	flags
+10	statut
+12	spécifications
+16	coordonnées de l'objet : X
+18	coordonnées de l'objet : Y
+20	largeur de l'objet
+22	hauteur de l'objet

Pour savoir de quel type d'objet il s'agit, (et éventuellement le modifier) il suffit d'additionner l'offset indiqué, qui est ici de 6, avec l'adresse de l'objet (calculée à partir de l'adresse de l'objet-parent + 24 * numéro de l'objet) :

$$(\text{Tree} + 24 * \text{numéro_de_l'objet}) + 6$$

Ceci vous permet de manipuler n'importe quel objet, mais il vous faut encore prendre connaissance de certaines instructions que je vais vous présenter ci-dessous.

☐ Peek, Poke et leurs parents (opérations dans la mémoire)

Pour lire un des octets se trouvant dans un des compartiments de la mémoire vive, vous écrivez :

$$\text{Contenu} = \text{PEEK}(\text{Adresse})$$

<adresse> représentant l'adresse de l'octet passé à la variable <contenu>. Inversement, il est également possible d'inscrire une valeur dans un compartiment de la mémoire, en écrivant :

$$\text{POKE Adresse, Valeur}$$

la <valeur> étant alors enregistrée dans le compartiment désigné par <adresse>. PEEK et POKE servent à lire ou à écrire un octet. Elles ne sont pas utilisables pour lire ou écrire un mot entier (deux octets), vous utilisez alors :

Contenu = WPEEK(Adresse)

pour lire un mot et

WPOKE Adresse,Valeur

pour écrire un mot (= 16 bits, dans deux octets consécutifs). Il faut absolument veiller à ce que l'adresse soit paire avec WPOKE, faute de quoi le CPU (Central Process Unit = le processeur de l'Atari ST) renverra un message d'erreur dans l'adresse, ce qui vous vaut en général trois petites bombes à l'écran.

Mais n'ayez crainte : l'ordinateur se charge de ranger les mots dans sa mémoire de façon à ce qu'ils commencent toujours à une adresse paire. Vous éviterez les bombes aussi longtemps que vous ne ferez pas d'erreur.

Troisième alternative : les mots longs (quatre octets), que l'on peut aussi ranger et lire dans la mémoire à l'aide de l'instruction :

Contenu = LPEEK(Adresse)

pour la lecture et

LPOKE Adresse,Valeur

pour l'écriture. Ici aussi, il nous faut une adresse paire pour éviter de recevoir un message d'erreur. Maintenant que nous savons comment accéder à l'adresse d'un intitulé dans l'arborescence des ressources, nous avons les mains libres pour nous livrer à toutes sortes de manipulations. L'intitulé d'un objet se compose de mots (à l'exception de la spécification qui fait 4 octets) que l'on peut modifier à l'aide de WPEEK et WPOKE.

Pour connaître le type d'objet rencontré, il vous suffit de lire le compartiment correspondant (plus exactement les compartiments puisque le mot se compose de deux octets consécutifs) : ainsi

Contenu = WPEEK((Tree + 24 * Numéro_de_l'objet) + 6)

retourne le type de l'objet dans la variable <contenu>, et nous avons déjà vu les différents types d'objets possibles dans le paragraphe décrivant le RCS. Je me limite donc à vous donner un tableau contenant les numéros relatifs à chacun des types d'objet :

Numéro	Objet
20	Box
21	Text
22	Boxtext
23	Image
24	Progdef (objet défini par le programmeur)
25	Ibox
26	Button
27	Boxchar
28	String
29	Ftext
30	Fboxtext
31	Icon
32	Title

Les autres données composant l'intitulé d'un objet dans l'arborescence sont également encodées, et voici leur signification :

☐ Flags

Les flags indiquent quelles sont les caractéristiques d'un objet ; chaque bit représente l'état d'un de ces flags :

Bit	Etat du flag
0	SELECTABLE
1	DEFAULT
2	EXIT
3	EDITABLE
4	RADIO-BUTTON
5	LASTOB
6	TOUCHEXIT
7	HIDETREE
8	INDIRECT

Comme vous ne savez sans doute pas très bien ce que tout cela signifie, voici quelques explications complémentaires.

☐ SELECTABLE

L'utilisateur pourra cliquer avec la souris sur l'objet en question, ce qui le fera passer en inversion-vidéo (l'objet, pas l'utilisateur !).

☐ DEFAULT

L'objet en question pourra être sélectionné rien qu'en appuyant sur la touche <enter> ou <return> ; évidemment, il faut veiller à ce qu'une arborescence ne comprenne au maximum qu'un seul objet sélectionnable par défaut ; en général, on donne cet attribut à un bouton permettant de ressortir du formulaire.

☐ EXIT

Lorsque l'utilisateur clique sur un bouton pourvu de cet attribut, il repasse au programme principal le contrôle sur le formulaire, contrôle exercé auparavant par l'AES (vous trouverez plus loin des explications plus détaillées).

☐ EDITABLE

L'utilisateur pourra 'éditer' l'objet en question (ce qui était le cas avec 'Text') ; attention, 'éditer' est ici un anglicisme qui signifie que l'utilisateur pourra entrer des données ou les corriger.

☐ RADIO-BUTTON

Il s'agit d'un groupe de boutons (au moins deux) de même niveau hiérarchique et dotés d'une caractéristique particulière : il suffit de cliquer sur l'un des boutons pour que le bouton activé auparavant (s'affichant en inversion-vidéo) reprenne son aspect normal ; ce qui revient à dire que l'on ne peut sélectionner qu'un bouton à la fois dans ce groupe de plusieurs boutons.

☐ LASTOB

Ce flag indique si l'objet se trouve être le dernier élément de l'arborescence (liste séquentielle) ; évidemment, il n'y a qu'un seul objet de ce type par arborescence.

☐ TOUCHEXIT

Ce flag repasse le contrôle au programme principal (application) dès que l'utilisateur clique sur l'objet concerné après y avoir amené la flèche de la souris. A la différence de EXIT, le programme principal reprend ici le contrôle même si l'utilisateur n'a pas relâché la touche gauche de la souris.

☐ HIDETREE

Cet attribut rend invisible l'objet en question, qui n'est plus visible à l'écran dans le formulaire bien que toujours présent. Il suffit de supprimer ce flag pour faire immédiatement réapparaître l'objet (après avoir appelé la fonction OBJC_DRAW, évidemment).

☐ INDIRECT

Ce flag indique que la spécification relative à un objet ne représente pas la valeur effective mais un pointeur dirigé vers une valeur.

☐ Object status

Il détermine l'aspect et l'affichage d'un objet, et voici la signification des différents bits qui le composent :

Bit	Signification
0	SELECTED
1	CROSSED
2	CHECKED
3	DISABLED
4	OUTLINED
5	SHADOWED

☐ SELECTED

Lorsque le bit 0 est mis, cela signifie que l'objet est activé par avance ; cela permet d'afficher des objets en inversion-vidéo avant même que l'utilisateur n'ait cliqué dessus.

☐ CROSSED

Ce flag n'est utilisable qu'avec les objets BOX... et a pour effet de dessiner un X dans l'objet.

☐ CHECKED

Ce flag permet de dessiner un petit crochet dans l'objet (voir par exemple le petit crochet en forme de V très évasé placé devant certains items dans certains menus pour signaler que telle fonction est activée).

☐ **DISABLED**

L'objet s'affiche en grisé, ce qui indique à l'utilisateur qu'il ne peut pas s'en servir momentanément (voir certaines options de menus).

☐ **OUTLINED**

L'objet est entouré d'un cadre constitué de deux rectangles, l'un en gras et l'autre moins épais.

☐ **SHADOWED**

L'objet est garni de son 'ombre' dans son coin inférieur droit.

☐ **Spécification de l'objet**

La spécification est intéressante (pour nous) à chaque fois qu'il s'agit d'un objet permettant la saisie d'un texte ou d'un caractère quelconque. Dans ce cas, la spécification de l'objet contient un pointeur dirigé vers une structure particulière de données, nommée structure TEDINFO.

Dans les autres types d'objet, la spécification contient un pointeur dirigé vers différentes structures de données (qui ne nous concernent guère) ou vers des informations diverses concernant la couleur et l'épaisseur du trait.

☐ **La structure TEDINFO**

Cette structure contient des informations utilisées par le GEM pour contrôler la saisie ou la sortie d'un texte ; elle est utilisée avec les types d'objets suivants :

TEXT
BOXTEXT
FTEXT
FBOXTEXT

et son adresse se trouve dans l'intitulé de l'objet à la rubrique spécification. Comme la spécification s'incarne dans un mot long, n'oubliez pas d'utiliser l'instruction LPEEK() pour le lire. Voici les éléments de la structure TEDINFO :

Offset	Contenu
+0	te_ptext
+4	te_ptmplt
+8	te_pvalid
+12	te_font
+14	non-utilisé (réservé)
+16	te_just
+18	te_color
+20	non-utilisé (réservé)
+22	te_thickness
+24	te_bxtlen
+26	te_tmplen

□ te_ptext

contient un pointeur dirigé vers le texte à sortir, à afficher (adresse du texte) ; lorsque le premier caractère est un aroba (@), les caractères suivants sont considérés comme des caractères vides.

□ te_ptmplt

contient l'adresse d'un texte fonctionnant comme modèle, comme masque de saisie. Les positions de saisie mises à la disposition de l'utilisateur sont représentées obligatoirement par le signe "_". L'utilisateur peut remplacer chaque "_" par un autre caractère ; grâce à te_pvalid, on peut réduire à certaines catégories les caractères acceptables : ceci permet par exemple de n'autoriser que la saisie de majuscules ou de chiffres.

☐ **te_pvalid**

contient l'adresse d'un string limitant la saisie de façon à n'autoriser que certains types de caractères (majuscules, chiffres etc). Les caractères autorisés sont précisés par un code, et ce pour chaque position de saisie :

Code	Caractères autorisés
9	chiffres seulement
A	majuscules et espaces vides
a	majuscules, minuscules et espaces vides
N	chiffres, majuscules et espaces vides
n	chiffres, majuscules, minuscules et espaces vides
F	noms de fichiers admis par le TOS, plus '?', '*', '/', ':'
p	noms de fichiers admis par le TOS, plus '?', '*', '/', ':', '\'
P	noms de fichiers admis par le TOS, plus '\'
X	tous caractères autorisés

☐ **te_font**

contient le numéro du jeu de caractères à utiliser :

3	jeu normal
5	jeu plus petit

☐ **te_just**

précise le positionnement du texte dans l'objet :

0	aligné sur la marge gauche
1	aligné sur la marge droite
2	centré (en plein milieu)

☐ **te_color**

détermine la couleur, selon le tableau suivant :

Valeur	Couleur
0	blanc
1	noir
2	rouge
3	vert
4	bleu
5	cyan
6	jaune
7	magenta
8	blanc
9	noir
10	rouge clair
11	vert clair
12	bleu clair
13	cyan clair
14	jaune clair
15	magenta clair

en respectant le masque de bit suivant :

(\$) eeee cccc smmm rrrr

dont voici la signification :

e	couleur de l'encadrement
c	couleur des caractères
s	mode d'écriture 0 => transparent 1 => recouvrant
m	motif de remplissage 0 => pas de motif de remplissage 7 => motif recouvrant 1 à 6 => motifs de densité croissante
r	couleur de remplissage

☐ te_thickness

détermine l'épaisseur du cadre entourant la boîte :

0	pas de cadre
1-127	épaisseur du cadre intérieur
128-	épaisseur du cadre extérieur, à interpréter comme un nombre négatif : -1 à -127.

☐ te_txtlen

contient la longueur du string sur lequel pointe te_ptext ; la longueur doit être supérieure de une unité au nombre réel de caractères, car il faut tenir compte de l'octet nul (CHR\$(0)) concluant le string.

☐ te_tmplen

contient la longueur du string désigné par l'adresse se trouvant dans te_ptmplt ; ici aussi, il convient d'ajouter l'octet nul concluant le string.

Voici un petit exemple illustrant la fonction des trois premiers mots longs :

```
te_ptext  pointe sur '1522'+CHR$(0)
te_ptmplt pointe sur 'Prix FF __. __'+CHR$(0)
te_pvalid pointe sur '9999'+CHR$(0)
```

lors de l'affichage de l'objet, le texte prend la place du modèle de saisie et devient :

```
PRIX FF 15.22
```

L'utilisateur ne peut entrer que des chiffres, puisque nous avons instauré cette limitation à l'aide de `te_pvalid` (9999).

□ Objets EDIT

Armé de ces connaissances, nous allons écrire deux procédures permettant la saisie d'une chaîne de caractères dans un objet ainsi que sa sélection. Il faut veiller à n'appliquer ces routines qu'aux objets TEXT, BOXTEXT, FTEXT et FBOXTEXT puisque ce sont les seuls dont la spécification puisse renvoyer à une structure `TEIDINFO`.

Il convient de passer les paramètres suivants : d'abord le numéro de l'objet dans lequel on souhaite inscrire le string ou dans lequel on souhaite reprendre le string ainsi que le string lui-même. Pour inscrire un texte dans l'objet, on écrira :

```
Put_Text(Numéro_d'objet, "texte")
```

tandis que pour lire un texte (saisi auparavant), on écrira :

```
Get_Text(Numéro_d'objet, Variable$)
```

et on retrouvera le texte par le biais du paramètre en retour `<variable$>`.

Mais comment accéder au texte et à son adresse ? Il faut d'abord trouver l'adresse de l'intitulé correspondant à l'objet concerné dans l'arborescence d'objets. Cela ne devrait plus être qu'une formalité pour vous (nous l'avons déjà expliqué plusieurs fois) ; comme chaque intitulé occupe 24 octets, et que l'adresse de l'objet-parent se trouve dans la variable `Tree`, on calcule cette adresse grâce à la fonction `RSRC_GADDR(0,0,Tree)` :

```
Adresse = Tree + 24 * Numéro d'objet
```


Il convient d'additionner à l'adresse ainsi obtenue l'offset de la spécification. Il vous suffit de revenir quelques pages en arrière pour apprendre que cet offset est 12. La spécification (mot long) contient l'adresse de la structure TEDINFO, que l'on peut lire à l'aide de LPEEK() :

```
Tedinfo = LPEEK(Adresse+12)
```

en partant évidemment de l'adresse retrouvée à l'aide de la formule précédente. Les quatre premiers octets (à nouveau un mot long) de la structure TEDINFO renvoient à l'adresse sous laquelle se trouve le texte lui-même :

```
Text_Adresse = LPEEK(Tedinfo + 0)
```

nous avons pris soin d'augmenter scrupuleusement l'offset de 0, ce qui n'a aucune importance et pourrait donc être laissé de côté.

La variable Text_Adresse contient maintenant la véritable adresse du texte lui-même, et nous pouvons la lire, caractère par caractère, à l'aide de PEEK(). L'octet nul servira de critère de sortie de notre boucle (contenant Peek()), octet nul qui conclut un string selon les bons usages du langage C et du GEM. On peut au choix accrocher l'octet nul au string (Omikron Basic) ou le laisser tomber. Cet octet nul ne pourrait que nous gêner dans la manipulation du string ; il est par contre indispensable dès qu'il s'avère nécessaire de passer le string au système d'exploitation. La boucle ci-dessous fait croître un compteur de répétition T% jusqu'à ce qu'on atteigne l'octet nul, sans pour autant ajouter cet octet nul au string :

```
Text$=""           'contient le texte
T%= 0             'variable du compteur
'repéter la boucle tant que l'octet nul
'n'est pas atteint
WHILE PEEK(Text_Adresse+T%) <> 0
    Text$ = Text$+ CHR$( PEEK(Text_Adresse+T%))
    T%=T%+1
WEND
```

Comme nous souhaitons rendre le string (lu caractère par caractère) à la partie du programme concernée, nous définissons le paramètre comme une valeur en retour en le faisant précéder d'un R :

```

DEF PROC Get_Text(Numero%, R Text$)
  LOCAL Adr=Tree+24*Numero%
  LOCAL Tedinfo= LPEEK(Adr+12)
  LOCAL Text_Adr= LPEEK(Tedinfo)
  LOCAL T%-0
  ...
  insérer ici la boucle mentionnée ci-dessus
  ...
RETURN

```

La deuxième procédure sera construite sur le même modèle, et elle servira à insérer un texte dans un objet. Il convient d'ajouter un octet nul à la fin du string (s'il n'y est pas déjà) ; il est indispensable en effet que cet octet nul se trouve bien à la fin de la chaîne de caractères transmise comme paramètre.

Astuce : Vous pouvez être quasi certain de provoquer un plantage général du système si vous oubliez ce fameux octet nul ; pour parer à ce risque, vous pouvez intégrer dans votre procédure une condition de vérification, qui contrôle si le string se termine bien par un octet nul :

```

IF RIGHT$(Text$,1) <> CHR$(0)
  Text$=Text$+ CHR$(0)
ENDIF
..
ici commence la boucle servant à insérer la chaîne de
caractères
dans un objet
...

```

L'écriture d'un string dans l'objet se déroule de la même façon, à la différence que chaque caractère est converti en un nombre (code ASCII) puis rangé dans les compartiments de la mémoire vive commençant à 'adresse' :


```

DEF PROC Put_Text(Numero%,Text$)
  LOCAL Adr=Tree+24*Numero%
  LOCAL Tedinfo= LPEEK(Adr+12)
  LOCAL Text_Adr= LPEEK(Tedinfo)
  LOCAL T%=0
  'inscrire caractère par caractère, grâce à POKE.
  'jusqu'à atteindre la fin du string
  WHILE T% < LEN(Text$)
    POKE Text_Adr+T%, ASC( MID$(Text$,T%+1,1))
    T%=T%+1
  WEND
RETURN

```

Pour éviter les mauvaises surprises et le plantage du système, je vous conseille de veiller à ce que la chaîne de caractères à insérer ne soit en aucun cas plus longue que la place réservée pour elle dans le RCS, faute de quoi vous pourriez même endommager d'autres données importantes contenues dans le fichier RCS.

❑ La gestion du formulaire

Nous allons maintenant étudier de plus près un thème que nous n'avons qu'effleuré : la gestion du formulaire, ce 'schéma F' selon lequel sont gérés les formulaires. Je me permets de vous rappeler les différentes étapes :

- ❶ charger le fichier RCS sauvegardé sur la disquette ; vous vous servez pour ce faire de la routine `RSRC_LOAD`, implémentée dans l'AES.
- ❷ lire l'adresse de l'arborescence que vous venez de charger grâce à `RSRC_GADDR`.
- ❸ vous pouvez, mais ce n'est pas obligatoire, appeler `FORM_CENTER`, qui se charge de calculer les coordonnées du formulaire de façon à ce que ce dernier vienne s'afficher en plein milieu de l'écran.
- ❹ `FORM_DIAL` sert ensuite à réserver suffisamment de place dans la mémoire d'écran.

- ⑤ deuxième appel de FORM_DIAL : un paramètre évoluant provoque l'apparition d'un rectangle se 'dilatant' (pur effet d'optique !)
- ⑥ dessin du formulaire lui-même à l'aide de OBJC_DRAW.

Une fois qu'il est complètement affiché à l'écran, on utilise la fonction FORM_DO pour repasser le contrôle du formulaire à l'AES ; l'AES rendra ce contrôle au programme (à l'application) en cours lorsqu'un objet doté de l'attribut EXIT, TOUCHEEXIT ou DEFAULT sera sélectionné soit par un clic de la souris soit par un appui sur la touche <enter> ou <return>. L'application décide alors de la suite à donner à cet évènement, et analyse pour cela le code (c'est à dire le numéro d'objet) de l'objet ayant entraîné l'interruption du contrôle de l'AES.

Une fois que le formulaire a rempli son rôle, il faut penser à le 'démonter' en recourant à FORM_DIAL, qui va faire apparaître à l'écran un rectangle se rétrécissant.

FORM_DIAL veille enfin à libérer l'espace de mémoire d'écran occupé depuis le premier appel de la fonction ; comme le GEM ne prévoit aucune possibilité de restauration de la surface recouverte par l'affichage du formulaire, il faut que l'application reconstitue le contenu de l'écran. Nous avons vu qu'il suffit de se servir de l'instruction BITBLT (bit-blit) pour stocker temporairement le contenu de la surface concernée, contenu qui est réaffiché à l'écran une fois que le formulaire est refermé.

Voici une présentation détaillée des routines servant à cette gestion du formulaire.

☐ Rsrc_Load(Re_Lname\$,Re_Lreturn)

Cette fonction sert à charger un fichier ressource dans la mémoire vive de l'ordinateur, la variable Re_Lname\$ devant contenir le nom du fichier ressource. L'ordinateur utilise la variable Re_Lreturn pour indiquer si le processus de chargement s'est bien déroulé : en cas d'erreur, cette variable contient la valeur 0, lorsque tout se passe correctement, elle contient la valeur 1.

Il est conseillé de contrôler le contenu de cette variable juste après l'exécution de la fonction, de façon à ressortir du programme si le fichier ne s'est pas bien chargé :

```
Rsrc_Load(Name$.Ret)
```

```
IF RET = 0 THEN
```

```
  'envoyer un message d'erreur
```

```
  FORM_ALERT(1,[3][Fatal error][désolé])
```

```
  'sortir du GEM
```

```
  Appl_Exit
```

```
END
```

```
ENDIF
```

□ Rscr_Gaddr(Re_Gtype,Re_Gindex,Re_Gaddr)

Comme vous le savez déjà, cette fonction retourne l'adresse d'une structure ou d'un objet dans la mémoire vive. Re_Gtype sert à préciser le type de structure à rechercher, selon le code suivant :

```
0 -> arborescence
```

```
1 -> objet
```

Le numéro de l'objet recherché est indiqué sous Re_Gindex. La fonction retourne l'adresse de l'objet recherché dans Re_Gaddr. En règle général, s'il s'agit de l'adresse d'une arborescence, elle est passée à une variable nommée 'Tree'. Nous avons vu que l'adresse de l'objet parent (root) permet ensuite de retrouver celle des différents objets.

□ Form_Center(Tree,X_Obj,Y_Obj,W_Obj,H_Obj)

Cette fonction sert à calculer les coordonnées d'un formulaire, de façon à ce qu'il vienne s'afficher en plein milieu de l'écran. Il convient d'indiquer l'adresse de l'arborescence dans la variable 'Tree', et la fonction retourne :

dans X_Obj	les coordonnées X du coin supérieur gauche
dans Y_Obj	les coordonnées Y du coin supérieur gauche
dans W_Obj	la largeur de la boîte
dans H_Obj	la hauteur de la boîte

□ **Form_Dial(Flag,X_Obj,Y_Obj,W_Obj,H_Obj)**

Cette fonction a de nombreux talents puisqu'elle exécute quatre tâches différentes selon le contenu de flag :

0	réserver de la place en mémoire d'écran
1	dessiner un rectangle en expansion
2	dessiner un rectangle allant en diminuant
3	libérer la place mémoire

Les quatre autres paramètres ont la même signification que dans la fonction précédente : coordonnées X et Y du coin supérieur gauche, largeur et hauteur de la boîte.

□ **Objc_Draw(Start,Niveaux,X_Obj,Y_Obj,W_Obj,H_Obj,Tree)**

Cette fonction sert à dessiner un formulaire à l'écran :

Start	contient l'indice du premier objet à dessiner
Niveaux	contient le nombre de niveaux d'objets (max 8)
X_Obj	contient les coordonnées X du coin supérieur gauche
Y_Obj	contient les coordonnées Y du coin supérieur gauche
W_Obj	contient la largeur du formulaire
H_Obj	contient la hauteur du formulaire
Tree	contient l'adresse de l'arborescence.

⚠ Attention : Il faut bien être conscient du fait que le GEM ne sauvegardera pas automatiquement le contenu de la surface de l'écran qui va être occupée par le formulaire ; vous devez donc vous-même veiller à le sauvegarder avant de provoquer l'affichage du formulaire.

☐ **Form_DO(Start,Tree,Return)**

Cette fonction a pour effet de retirer à l'application le contrôle du formulaire pour le passer à l'AES. **Start** contient l'indice de l'objet sur lequel le curseur va venir se positionner (dans un champ d'édition, ce curseur prend la forme d'un mince trait vertical) ; il convient de lui donner la valeur 0 lorsque le formulaire ne contient aucun champ éditable. **Tree** représente ici aussi l'adresse de l'arborescence. La variable **Return** retourne le numéro (indice) de l'objet ayant provoqué la sortie du formulaire (exit), après quoi le programme peut réagir en conséquence.

⚠ Attention : Avant d'appeler cette fonction, il est prudent de s'assurer que l'arborescence contient bien au moins un objet permettant de redonner le contrôle à l'application, c'est-à-dire un objet pourvu de l'un des trois attributs EXIT, TOUCHEXIT ou DEFAULT. Faute de quoi vous n'auriez plus aucune possibilité de retirer le contrôle à l'AES, et vous seriez bien obligé pour vous en sortir de procéder à une réinitialisation, ce qui revient à faire disparaître le programme de la mémoire vive (j'espère que vous l'avez sauvegardé auparavant sur une disquette !).

☐ **Comment lire et conférer des attributs d'objet**

Avant que la fonction FORM_DO n'ôte le contrôle à l'AES pour le rendre à l'application, l'utilisateur peut saisir du texte dans un champ d'édition, il peut cliquer sur différents objets (par exemple sur une des cases d'un bouton radio) et ainsi procéder à divers paramétrages.

Un fois ressorti de l'AES par l'objet 'exit', c'est au programme lui-même de reprendre ces paramétrages pour les exploiter : il doit donc reprendre les données saisies avant que le formulaire ne disparaisse.

Un tel contrôle n'est pas indispensable aussi longtemps que tous les objets sélectionnables (i.e. : sur lesquels l'utilisateur peut cliquer) sont dotés de l'attribut 'exit'. Il faut tout de même veiller à ce que l'objet reprenne son aspect non-activé avant la fermeture du formulaire, de façon à ce qu'il n'apparaisse pas comme étant activé (inversion-vidéo) lors de la prochaine ouverture du formulaire. Ainsi par exemple, si vous avez créé un objet 'SORTIE' servant à ressortir du formulaire, il sera en inversion-vidéo lorsque l'utilisateur aura cliqué dessus. Lors de l'appel suivant du formulaire, ce bouton sera toujours activé vu l'état du bit zéro (= SELECTED) qui définit le statut de l'objet, et il faudrait alors effectuer un double-clic sur cet objet pour ressortir du formulaire. D'où il ressort qu'il est indispensable de restaurer l'ancien état (en principe non-activé) de tous les objets avant de refermer un formulaire. Je vous ai déjà donné les explications théoriques nécessaires et je peux donc passer directement à la mise en pratique.

Un objet s'affiche en inversion-vidéo lorsque le bit 0 est mis dans son statut. Pour replacer cet objet dans son état d'origine (dé-sélection) il suffit de masquer ce bit, et nous savons que nous pouvons utiliser pour ce faire l'opérateur logique AND. Comme il ne faut pas toucher à l'état du reste des bits composant le statut, nous remplissons le masque de valeurs 1 ; seul le premier bit contient un 0, le résultat du traitement avec AND étant alors toujours un 0 pour le premier bit. Ainsi :

```
Nimportequoi AND %11111110
```

mettra toujours le premier bit (bit zéro) sur la valeur 0, les autres bits restant inchangés. Pour y voir plus clair, nous convertissons le nombre binaire en valeur décimale, ce qui nous donne 254.

On procède de façon exactement inverse lorsqu'il s'agit de conférer le statut SELECTED à un objet directement depuis l'application. Il convient alors de mettre ce bit zéro sur la valeur 1, quel que soit son contenu antérieur. L'opérateur logique AND ne nous est guère utile, il vaut mieux recourir à l'opérateur OR. Nous écrivons :

```
Nimportequoi OR %00000001
```

Attention : Alors qu'avec AND seul le bit 0 était sur 0 dans le masque, cela mènerait à une mini-catastrophe avec OR, puisque tous les bits seraient mis, sauf celui qui convient. Il faut donc inverser les bits avant de les soumettre à l'action de l'opérateur OR.

Ces quelques explications étant données, nous pouvons manipuler efficacement le statut d'un des objets du formulaire. Et comme nous aurons à le faire assez souvent, il est plus simple d'écrire une petite procédure exécutant cette tâche pour nous. Un flag utilisé comme paramètre permet de lui indiquer si l'objet en question doit être activé ou désactivé :

1	objet activé
2	objet désactivé

Le paramètre 'Numero%' contient bien sûr le numéro de l'objet concerné dans la liste :

```
DEF PROC Select(Numero%,Flag%)
LOCAL Adr%L=Tree%L+24*Numero%
IF Flag%=1 THEN
    LOCAL Ob_State%L= WPEEK(Adr%L+10) OR 1
ELSE
    LOCAL Ob_State%L= WPEEK(Adr%L+10) AND 254
ENDIF
Obj_Change(Numero%,Ob_State%L,Xobj%L,Yobj%L,Wobj%L,Hobj%L
.Tree%L)
RETURN
```

Après avoir calculé l'adresse du statut de l'objet concerné,

le programme lit le statut en question grâce à `WPEEK()` en se souciant spécialement de l'état du bit zéro, qui est d'abord mis (OR) puis masqué (AND). Le résultat est stocké temporairement dans la variable `Ob_State%L`. Nous ne sommes encore qu'à mi-chemin du but, puisqu'il nous reste encore à repasser le nouveau statut dans la structure de l'objet. Il existe pour cela une fonction GEM :

```
Objc_Change(Indice,Nouveau_statut,X_Obj,Y_Obj,W_Obj,H_Obj,Tree)
```

<indice> étant à remplacer par le numéro de l'objet dont le statut doit être remplacé par <nouveau_statut>. Les autres paramètres précisent les dimensions de l'objet à modifier ainsi que la référence de l'objet-parent 'Tree'. Il existe une variante de cette fonction, dans laquelle on se dispense d'entrer les coordonnées de l'objet en se limitant à indiquer l'objet-parent :

```
Obj_Change(Indice,Nouveau_statut,Tree)
```

ce qui revient à remplacer par un <nouveau_statut> l'ancien statut de l'objet désigné par son numéro <indice> dans l'objet-parent.

Je voudrais maintenant prendre un deuxième exemple pour vous montrer l'utilité de `ENABLED` et `DISABLED` dans le statut de l'objet. Ces deux éléments permettent de 'verrouiller' et 'déverrouiller' un objet, ce qui revient à laisser ou non la possibilité à l'utilisateur de sélectionner tel ou tel objet en cliquant dessus avec la souris. Ceci est symbolisé par le fait que l'objet en question, lorsqu'il n'est plus activable, s'affiche en gris sur blanc ('affichage affaibli') et non plus en noir sur blanc. Ceci montre à l'utilisateur qu'il doit passer par une autre fonction avant de lancer la fonction qui correspond à l'objet affaibli.

Dans cette opération, il nous faut prendre en compte non seulement le statut de l'objet mais aussi les flags qui l'accompagnent. En effet, le statut comprend le flag `DISABLED` (bit 3), tandis que les flags de l'objet contiennent les bits autorisant la sélection et la sortie d'un objet.


```

DEF PROC Enable(Numero%,Flag%)
LOCAL ADR=Tree+24*Numero%
IF Flag%=1 THEN
    WPOKE ADR+8, WPEEK(ADR+8) OR 1
    Obj_Change(Numero%, WPEEK(ADR+10) AND
247,Xobj,Yobj,Wobj,Hobj,Tree)
ELSE
    WPOKE ADR+8, WPEEK(ADR+8) AND 254
    Obj_Change(Numero%, WPEEK(ADR+10) OR
8,Xobj,Yobj,Wobj,Hobj,Tree)
ENDIF
RETURN

```

Examinons d'abord les flags : selon son état, le bit 0 autorise ou non la sélection d'un objet à l'aide de la souris. Il suffit de le masquer pour que l'objet ne puisse plus être sélectionné à l'aide de la souris. Inversement, il faut le remettre pour que l'objet soit sélectionnable par la souris. La formule

```
WPOKE ADR+8, WPEEK(ADR+8) OR 1 [AND 254]
```

permet de lire les flags, de leur faire subir une opération logique (opérateur OR puis AND) et de les réécrire immédiatement. Dans l'étape suivante, nous nous soucions du statut de l'objet, ce qui est un peu plus compliqué.

Le bit concerné est ici le bit 3 : lorsqu'il est mis, il représente la valeur 8 (2^3). Pour mettre tous les bits jusqu'au troisième, il convient d'inverser le nombre 8 : cette négation provoque une inversion des valeurs des bits, ce qui donne le nombre 247 ($255 - 2^3$). Nous repassons le nouveau statut ainsi obtenu à la fonction OBJ_CHANGE, qui se chargera de modifier en conséquence le statut de l'objet.

La fonction Enable() sert à définir l'état de l'objet en fonction de l'état du flag :

0	disabled
1	enabled

cette fonction se charge en même temps d'affaiblir (affichage en grisé) le texte contenu dans l'objet : l'utilisateur voit qu'il ne peut momentanément sélectionner l'objet en question. Nous utiliserons une variante de cette fonction dans la boîte de sélection d'objet : elle nous servira à verrouiller l'accès à des unités de disque(tte) non connectées. Il nous faudra procéder à quelques modifications de détail pour parer aux multiples utilisations possibles de la boîte de sélection.

6.3. Les 'ascenseurs' (sliders)

Le GEM rend possible l'utilisation des 'ascenseurs' (sliders) : il s'agit de ces coulisseaux placés généralement sur le bord droit d'une fenêtre et qui permettent de se déplacer (de 'monter' ou 'descendre') dans une liste ou un texte. Un ascenseur résulte de la combinaison d'un objet-parent (parent) contenant un objet plus petit (child) pouvant coulisser dans l'objet-parent.

Le coulisseau doit absolument être doté de l'attribut TOUCHEXIT, ainsi d'ailleurs que l'objet-parent, faute de quoi le coulisseau ne pourrait guère ... coulisser ! Une fois les deux objets créés - parent et enfant - le reste n'est plus qu'une question de détail.

L'AES dispose de la fonction `Graf_Slidebox`, qui permet de faire coulisser un objet à l'intérieur d'un autre. Le coulisseau se déplace avec la souris tant que sa touche gauche reste appuyée. La fonction est accompagnée des paramètres suivants :

`Graf_Slidebox(Parent.Child,Direction,Position)`

parent numéro indice de l'objet parent

child numéro indice de l'objet enfant (le coulisseau)

direction déplacement possible du coulisseau :
0 => horizontalement
1 => verticalement

position valeur retournée par la fonction, qui indique la position relative du coulisseau par rapport à l'objet-parent ; la valeur 0 indique que le coulisseau se trouve tout à fait à gauche ou tout en haut, tandis que la valeur 1000 indique qu'il se trouve tout à fait à droite ou tout en bas.

Comme la 'position' est indiquée de façon relative par rapport à l'objet-parent, il faut d'abord la calculer : on commence pour cela par soustraire la largeur du coulisseau (child) de la largeur totale de la coulisse (parent) ; le résultat ainsi obtenu est multiplié par le paramètre retourné (la position) puis divisé par 1000. Ce qui nous donne la formule :

$$\text{Position} = (\text{Largeur-parent} - \text{Largeur-enfant}) * \text{Position} / 1000$$

L'ascenseur est géré par le GEM ; c'est à nous par contre de nous soucier de la taille véritable que prendra le coulisseau. En effet, la distance que doit parcourir le coulisseau dépend en grande partie (dans une boîte de sélection d'objet) du nombre d'intitulés (noms de fichier et de dossier) contenus dans le répertoire de l'unité de disque : nous devons donc en tenir compte pour calculer la taille du coulisseau.

Lorsque la fenêtre peut contenir tous les intitulés du répertoire, le coulisseau prend la taille de l'ascenseur global, puisqu'il est alors inutile de s'en servir.

Plus le répertoire contient d'intitulés, plus le coulisseau sera petit, pour que son 'rayon d'action' s'agrandisse.

Il faut de plus veiller à ce que le coulisseau ait une largeur compatible avec celle de l'objet-parent, ce que le RCS ne permet qu'à certaines conditions. Toutes ces données se trouvent dans l'intitulé de l'objet concerné et peuvent être manipulées. Nous prendrons un exemple concret, en créant une boîte de sélection d'objet contenant un ascenseur (paragraphe 6.5).

6.4. Les menus déroulants

Il y a longtemps maintenant que les propriétaires d'Atari-ST se sont habitués à ces barres de menus, qui permettent de 'dérouler' un menu dès qu'on effleure l'un des intitulés à l'aide de la flèche de la souris. L'utilisateur a ainsi accès à un certain nombre de fonctions. Comme il s'agit là encore de l'une des nombreuses conquêtes du GEM, cela signifie que le RCS vous permet de créer vos propres menus déroulants.

Le premier intitulé, celui qui se trouve le plus à gauche dans la barre des menus, est généralement appelé 'Bureau' (ou 'Desk' en anglais). Il abrite généralement les accessoires chargés automatiquement à chaque initialisation de la configuration. La première option (généralement : 'Info') de ce menu 'Bureau' est réservée à la communication de certaines 'Informations' concernant le programme qui vient d'être chargé.

Il est très facile de confectionner une barre de menus, puisqu'elle ne se compose que de peu d'éléments :

- TITLE** intitulés des menus contenus dans la barre
- ENTRY** intitulés (options) contenus dans les menus déroulants.

On peut ajouter une ligne de pointillés, pour séparer éventuellement des groupes d'options dans un menu, ainsi qu'une boîte qui sert à réserver la place pour une autre barre de menus (lorsque le programme sera en train de tourner). Pour insérer une nouvelle ligne dans un menu, il faut d'abord agrandir son cadre puis y transporter les éléments voulus.

Il faut penser aussi à réserver une ou deux positions avant l'intitulé de l'option, de façon à pouvoir y insérer un petit crochet signalant éventuellement que l'option est activée.

Je voudrais maintenant vous montrer comment insérer une telle barre de menus dans vos programmes à l'aide d'un exemple concret.

Commencez par charger votre RCS pour construire une arborescence en suivant les indications qui vont suivre. Amenez dans la surface de travail l'icône 'Menu' et nommez la nouvelle arborescence 'Barre' ; nous appellerons 'Info' la première option du premier menu (nommé 'Bureau'). En cliquant sur 'Bureau' vous devriez voir s'ouvrir le menu correspondant, qui contient en principe les options suivantes : 'your message here' (votre message ici) et 'Accessory 1 à 6' (accessoires numéro 1 à 6). 'Bureau' appartient à la catégorie d'objets TITLE, alors que les options (your message here et accessory 1 etc.) appartiennent à la catégorie ENTRY.

Ouvrez l'objet 'your message here' en effectuant un double-clic dessus, et rebaptisez-le 'Informations' ; n'oubliez pas de laisser deux caractères vides. Il convient aussi de donner un nom à cette option : MINFO. Vous pouvez ensuite procéder de même pour transformer 'FILE' en 'Fichiers'. Si vous ouvrez ce dernier menu, vous constatez qu'il ne contient pour l'instant que l'option 'Quit'.

Il faut d'abord agrandir le cadre de ce menu pour pouvoir y insérer d'autres options : un simple clic sur le coin inférieur droit devrait nous y aider, mais en fait vous effleurez constamment le coin inférieur droit de l'option et non celui du cadre. Voilà une petite astuce pour vous en sortir : commencez par rétrécir la dimension de l'intitulé de l'option, après quoi vous pourrez sans difficulté vous emparer du coin inférieur droit du cadre pour l'agrandir. Renommez l'option existante en 'Charger' et appelez-la Mload. Après quoi vous prenez dans la fenêtre d'objets autant d'objets ENTRY que nécessaire pour reconstituer le menu suivant :

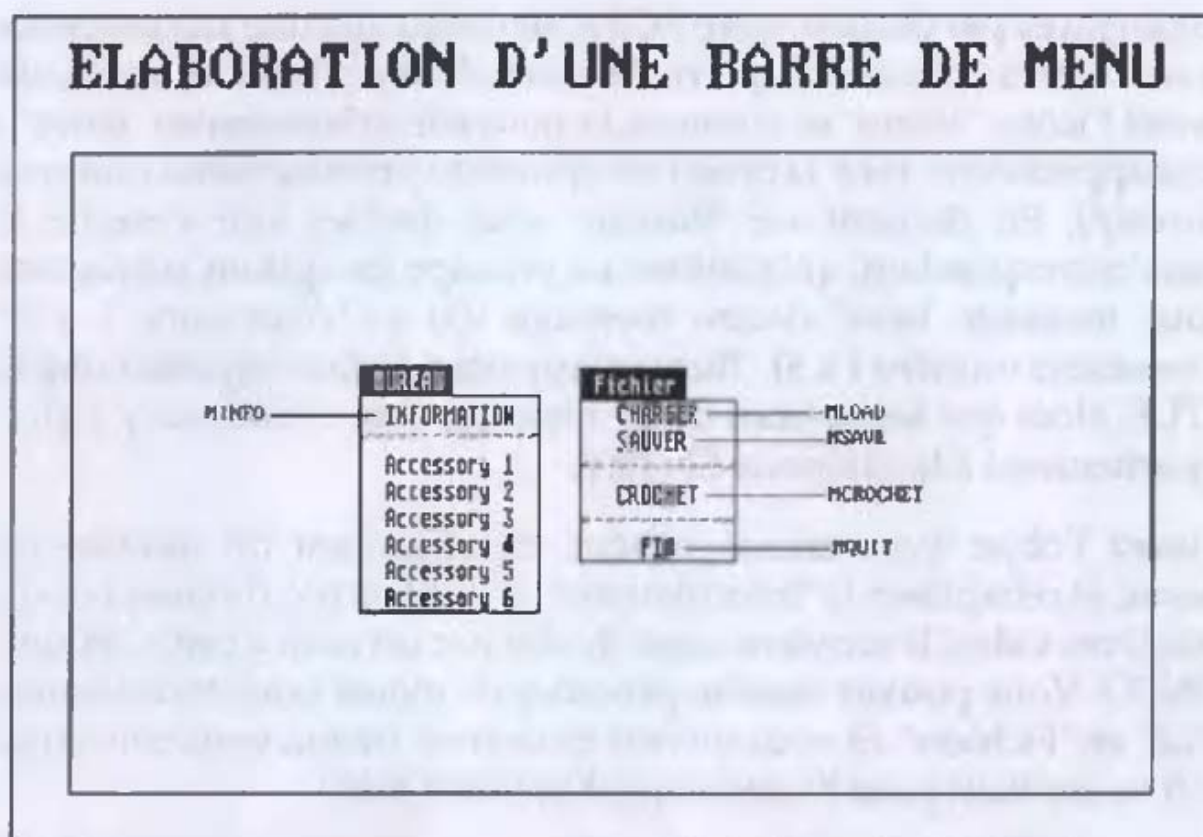


Figure 6.5 : la barre des menus

Dès que vous avez terminé de confectionner la barre des menus à l'aide du RCS, et que vous avez également nommé toutes les options dont vous souhaitez garnir vos menus, vous sauvegardez le tout sur disque(tte). Vous pourrez recharger ce fichier par la suite lorsque ce sera nécessaire, grâce à l'instruction (que vous connaissez déjà) :

```
Rsrc_Load(Fichier$.Retour)
```

Attention, vous ne lancez cette instruction qu'après avoir déclaré votre application auprès du GEM au moyen de

```
App1_init
```

n'oubliez pas non plus le petit test (par lequel devrait commencer tout bon programme sous GEM) contrôlant si le fichier RCS a bien été chargé. Nous en venons maintenant aux instructions permettant de gérer la barre des menus.

Pour que la barre des menus puisse s'afficher, il faut que la fonction concernée dispose d'un pointeur dirigé vers l'arborescence incarnant la barre de menus, arborescence qu'exceptionnellement nous ne nommerons pas Tree mais Menu_Adr. En principe, vous connaissez déjà, depuis le chapitre deux, la fonction servant à calculer cette adresse :

```
Rscr_Gaddr()
```

Maintenant que nous disposons de l'adresse de l'arborescence, il nous manque la routine amenant notre barre de menus sur notre écran :

```
Menu_Bar(Menu_Adr)
```

Voilà que cela se complique un peu. Vous savez que le GEM se sert de la routine FORM_DO, implémentée dans l'AES, pour gérer un formulaire ; mais cette routine n'est pas utilisable pour gérer une barre de menus. En effet, le système d'exploitation de l'Atari, et plus précisément le GEM, contient des fonctions qui attendent qu'un événement (event) donné se produise. Dès que l'événement attendu survient, le GEM informe l'application en cours de l'événement qui vient de se produire et l'application en tire les conséquences voulues, selon le programme. Répétons-le, il existe une foule de fonctions dont le rôle consiste ainsi à guetter la survenue d'un événement. Pour gérer notre barre de menus, nous allons utiliser l'une de ces fonctions 'event', laquelle retourne un string informant de la nature de l'événement survenu. Cette fonction s'écrit :

```
Evt_Mesag(Information$)
```

Dès que l'événement survient, l'AES dépose une information en rendant compte, dans le paramètre de retour 'Information\$' (message pipe), qui possède alors la structure suivante :

Octet	Contenu
1 + 2	Valeur '10' en tant que numéro d'identification pour un évènement survenu dans le menu (Mn_Selected)
7 + 8	Numéro d'objet du menu
9 + 10	Numéro d'objet de l'option sélectionnée

Pour plus de simplicité, nous transformons la chaîne, retournée après que l'évènement soit survenu, en un tableau array contenant les différentes informations délivrées par le message. Cette transformation ne pose aucun problème, car le format de ce message est comme fait pour CVI(). Une boucle FOR...NEXT nous permet de transformer cet ensemble de données occupant deux octets en un integer (nombre entier) :

```
FOR T%= 0 TO 4
  Evenement(T%)= CVI( MID$(Information$,T%*2+1,2))
NEXT T%
```

Passons maintenant à la comparaison : il faut d'abord vérifier si un évènement s'est bien produit au niveau du menu, car cette fonction peut réagir à bien d'autres évènements. C'est la valeur 10 qui sert à identifier un évènement comme étant survenu dans un menu, lequel sera enregistré dans le premier élément (indice 0) du tableau event-array :

```
IF Evenement(0) = 10 THEN ... (évènement dans le menu)
```

Une fois constaté que l'évènement survenu est bien celui qui était attendu, il faut vérifier l'état des différentes options contenues dans le menu concerné. Le numéro d'objet de l'option sélectionnée se trouve aussi dans le tableau event-array, en cinquième position (indice 4). Pour toutes ces vérifications, il vaut mieux intégrer le fichier header fourni par le RCS dans le programme puis lui passer les noms de variables pour contrôle :

```
IF Evenement(0) = 10 THEN
  IF Evenement(4) = Minfo Then
    <réaction déclenchée par cet évènement>
    <par ex : branchement dans un sous-programme>
  ENDIF
```



```

      IF Evenement(4) = Mquit THEN
      ....
      ....
    ENDIF
  ENDIF

```

Rappelons-nous que, une fois la réaction déclenchée et la fonction lancée, l'intitulé du menu est resté activé et qu'il convient de le ramener à son état originel ; l'Omikron Basic dispose pour cela d'un appel au GEM :

```
Menu_Tnormal(Numero_d'objet,1)
```

qui ramène le menu à son état normal, non-activé. Le numéro d'objet de l'intitulé du menu correspondant à l'option activée se trouve en quatrième position (indice 3) dans le tableau event-arrayu :

```
Menu_Tnormal(Evenement(3),1)
```

La gestion, y compris la sélection du menu, est à nouveau intégrée dans une boucle, de façon à ce qu'il soit possible d'appeler plusieurs fois le même menu. Nous utilisons une boucle REPEAT...UNTIL pour qu'il soit possible d'appeler au moins une fois le menu ; la condition de sortie de la boucle sera la sélection de l'option 'FIN' ou 'Sortie du programme' :

```

Fin = 0
REPEAT
  ....
  <attendre qu'une information arrive dans le buffer>
  ....
  IF Evenement(0) = 10 'sélection d'une option du menu ?
  ....
  ....
  IF Evenement(4) = Mquit 'sortir de la boucle
    Fin = 1
  ENDIF
  'remettre l'intitulé du menu dans son état normal
  Menu_Tnormal(Evenement(3),1)
  ENDIF
UNTIL Fin 'répéter jusqu'à ce que Fin = '1'

```

En règle générale, on prend soin d'envoyer, avant la sortie définitive du programme, un message d'avertissement ou tout au moins une demande de confirmation, ceci pour éviter que l'utilisateur ne ressorte involontairement du programme sans avoir auparavant sauvegardé ses données. On se sert pour cela de FORM_ALERT qui délivre un message comme :

Souhaitez-vous vraiment sortir du programme ? Oui / Non

après quoi une variable (par exemple Ret%) contiendra une valeur indiquant le bouton choisi par l'utilisateur ; la condition pourra s'écrire :

```
IF Ret% = 1 THEN
  Fin = 1
ELSE
  Fin = 0
ENDIF
```

Il est préférable et plus court d'écrire :

```
Fin = Ret% = 1
```

ce qui, pour le même résultat, ne nous coûte qu'une ligne d'écriture. Comment ? Examinons la partie droite de la formule : le signe '=' est utilisé comme un opérateur de comparaison ; la valeur de vérité résultant de la comparaison (Ret% = 1 ? 'faux' lorsque Ret% est différent de 1 et 'vrai' lorsque Ret% est 'égal' à 1) est passée à la variable 'Fin', ce qui tombe bien vu que son contenu sert précisément à ressortir de la boucle. On peut certes solutionner un même problème de diverses façons, mais il faut convenir qu'ici la deuxième méthode est plus élégante.

Dans un menu, il est possible de faire apparaître un crochet devant une option lorsqu'elle est activée. Cela permet à l'utilisateur de savoir immédiatement s'il a activé telle ou telle fonction (par exemple le mode 'insérer' dans un traitement de texte). Il est facile d'ajouter ainsi un crochet devant une option, puisqu'il existe une fonction prévue pour :

```
Menu_Icheck(Numero%, Flag)
```


<Numero%> désigne le numéro du menu dans l'arborescence, tandis que <flag> peut prendre deux valeurs :

0	le crochet présent est effacé
1	inscription d'un crochet devant l'option concernée

Il faut aussi penser à verrouiller certaines options des menus (Disabled), pour empêcher que l'utilisateur ne puisse les sélectionner avant que les conditions de leur utilisation ne soient remplies. Cela n'a par exemple aucun sens de donner accès à des options de traitement d'un fichier tant qu'aucun fichier n'est chargé. Nous utilisons la fonction :

Menu_Ienable(Numero%,Flag)

<Numero%> représentant encore une fois le numéro de l'option dans l'arborescence, et <flag> indique si l'option doit être activée ou désactivée :

0	verrouillage de l'option (affichage affaibli)
1	option accessible, non-verrouillée.

Voici maintenant notre programme de démonstration, illustrant la gestion des menus déroulants. Nous utilisons un fichier ressource nommé MENUE.RSC, contenant l'arborescence représentant tous les menus :

```

0  * ****
1  *                                     *
2  * ----- *
3  * Auteur : Michael Maier Version : 1.00 Date : 16.07.1990 *
4  * Programme joint au Livre de l'Omikron Basic *
5  * (C) 1988 by MICROAPPLICATION *
6  * ****
7  *
8  * Esquisse de programme de démonstration:
9  * programmation d'une barre de menus
10 * en Omikron Basic
11 * ****
12 *
13 *
14 * d'abord donner des valeurs aux variables
15 *
16 Barre%L=0

```

```

17 MInfo%L=7      ' STRING dans tree BARRE
18 Mload%L=16     ' STRING dans tree BARRE
19 Msave%L=17     ' STRING dans tree BARRE
20 Mcrochet%L=19  ' STRING dans tree BARRE
21 Mquit%L=21     ' STRING dans tree BARRE
22 '
23 ' Mettre sur zéro la variable servant à sortir de la boucle
24 ' et dimensionner le tableau event-array
25 '
26 Fin%L=0
27 DIM Evenement%L(4)
28 '
29 ' puis procéder à la cérémonie usuelle sous GEM:
30 '
31 Appl_Init
32 Rsrc_Load("MENUE.RSC",Ret%)
33 IF Ret%<0 THEN
34   FORM_ALERT (1."[3][Fatal Error!][ Cancel ]")
35   Appl_Exit
36   END
37 ENDIF
38 '
39 ' déterminer l'adresse racine de l'arborescence (Menu)
40 '
41 Rsrc_Gaddr(0,0,Menu_Adr%L)
42 '
43 ' puis dessiner la barre de menu
44 '
45 Menu_Bar(Menu_Adr%L)
46 Graf_Mouse(0) ' Souris en forme de flèche (mieux vaut s'en assurer!)
47 '
48 ' La boucle ci-dessous assure la gestion de la
49 ' barre de menus que nous venons de créer
50 '
51 REPEAT
52 '
53 ' attendre jusqu'à ce qu'un événement soit consigné dans
54 ' le 'Message-Buffer': une fois la barre de menus activée d'une
55 ' façon ou d'une autre, il en résulte un string ayant la
56 ' structure suivante:
57 '       Mot 0 => '10' (Format LOW-HIGHbyte)
58 '       Mot 3 => Numéro de la barre contenant l'option qui
59 '               a été sélectionnée
60 '       Mot 4 => Numéro d'objet de l'option sélectionnée
61 '
62   Evnt_Mesag(Me$)
63 '
64 ' Tirer maintenant les conséquences de l'évènement survenu
65 '
66   FOR Tx=0 TO 4
67     Evenement%L(Tx)= CVI( MID$(Me$,Tx*2+1,2))
68   NEXT Tx
69 '
70 ' Évènement dans la barre de menus ?
71 '
72   IF Evenement%L(0)=10 THEN
73     '
74     ' Le numéro d'objet se trouve dans 'Evenement(4)'
75     '

```



```

76 IF Evenement%L(4)=Minfo%L THEN
77     FORM_ALERT (1."[1][ Ce n'était qu'une démonstration ] [ OK ]")
78 ENDIF
79 '
80 IF Evenement%L(4)=Mload%L THEN
81     FORM_ALERT (1."[1][Eh bien! allez-y donc!][ OK ]")
82 ENDIF
83 '
84 IF Evenement%L(4)=Msave%L THEN
85     FORM_ALERT (1."[2][Quel fichier?][ Aucun! ]")
86 ENDIF
87 '
88 IF Evenement%L(4)=Mcrochet%L THEN
89     ' Inverser le contenu de la variable
90     IF Crochet%-1 THEN
91         Crochet%=0
92     ELSE
93         Crochet%=1
94     ENDIF
95     '
96     ' et activer ou désactiver le crochet
97     '
98     Menu_1check(Mcrochet%L.Crochet%)
99     '
100 ENDIF
101 '
102 IF Evenement%L(4)=Mquit%L THEN
103     FORM_ALERT (2."[2][Sortir du programme?][Oui|Non]".Ret%)
104     ' un peu de magie maintenant...
105     ' nous vérifions d'abord si Ret% contient la valeur
106     ' '1' ; la valeur de vérité résultant de cette opération
107     ' est passée à la variable 'Fin', servant de critère
108     ' de sortie de la boucle.
109     Fin%L=Ret%-1
110 ENDIF
111 '
112 ' vous pouvez ici insérer d'autres contrôles concernant la
113 ' sélection éventuelle d'autres options...
114 '
115 ' Ramener l'intitulé du menu à son état normal
116 '
117 Menu_Tnormal(Evenement%L(3),1)
118 '
119 ENDIF
120 UNTIL Fin%L
121 '
122 ' Sortir du GEM et du programme
123 '
124 Appl_Exit
125 END
126 '
127 *****
128 **          insérer ici la librairie GEMLIB.BAS          **
129 *****
130 '
131 ' .....
132 ' .....
133 ' .....

```

6.5. Comment créer votre boîte de sélection ?

Bien que la boîte de sélection d'objet du GEM offre une foule de possibilités, elle s'avère quelque peu fastidieuse à l'usage. En effet et si vous possédez les anciennes ROM, lorsque vous souhaitez changer d'unité de disque(tte), il vous faut d'abord changer l'identificateur au tout début du chemin d'accès, après quoi il faut encore cliquer sur la case de fermeture de la petite fenêtre se trouvant en-dessous pour valider cette modification. Il en va de même lorsque vous souhaitez modifier l'extension servant de critère de sélection pour l'affichage des noms de fichiers.

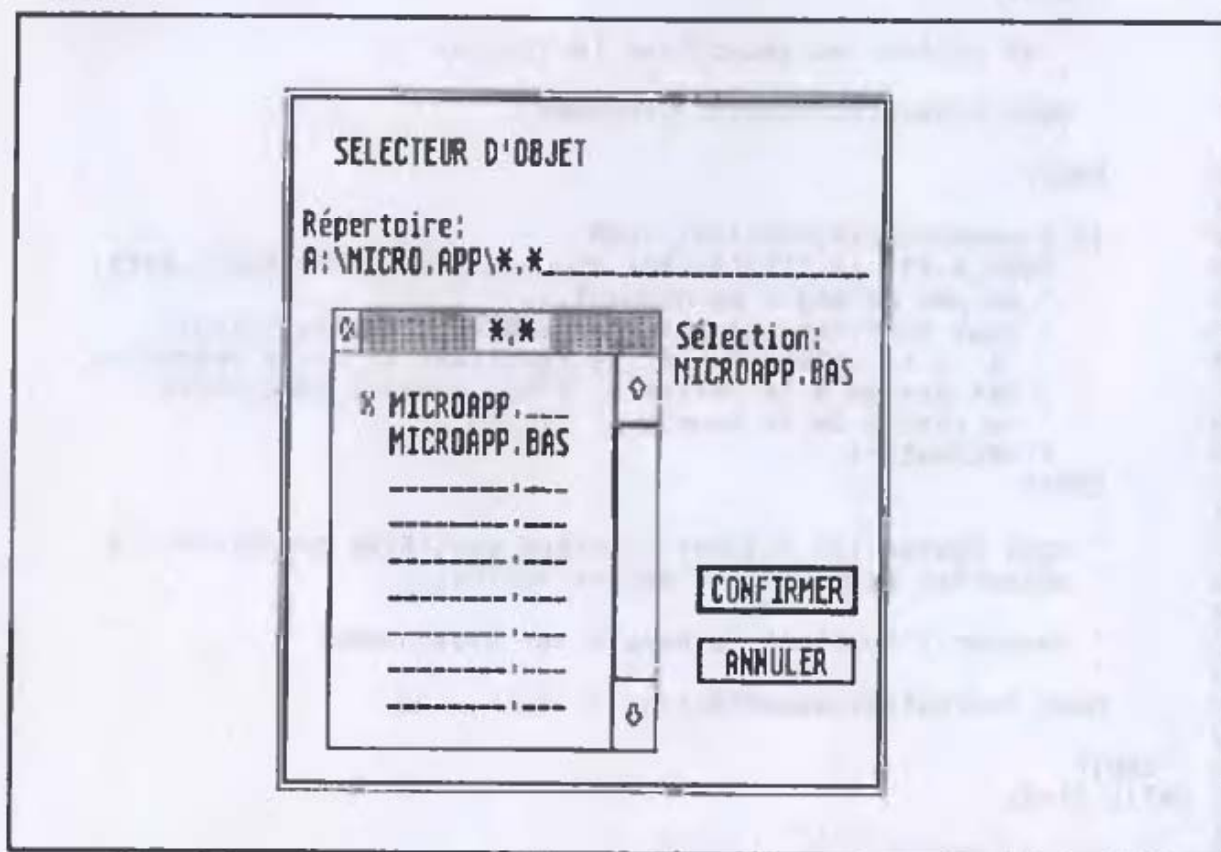


Figure 6.6 : Ffle selector box = boîte de sélection d'objet

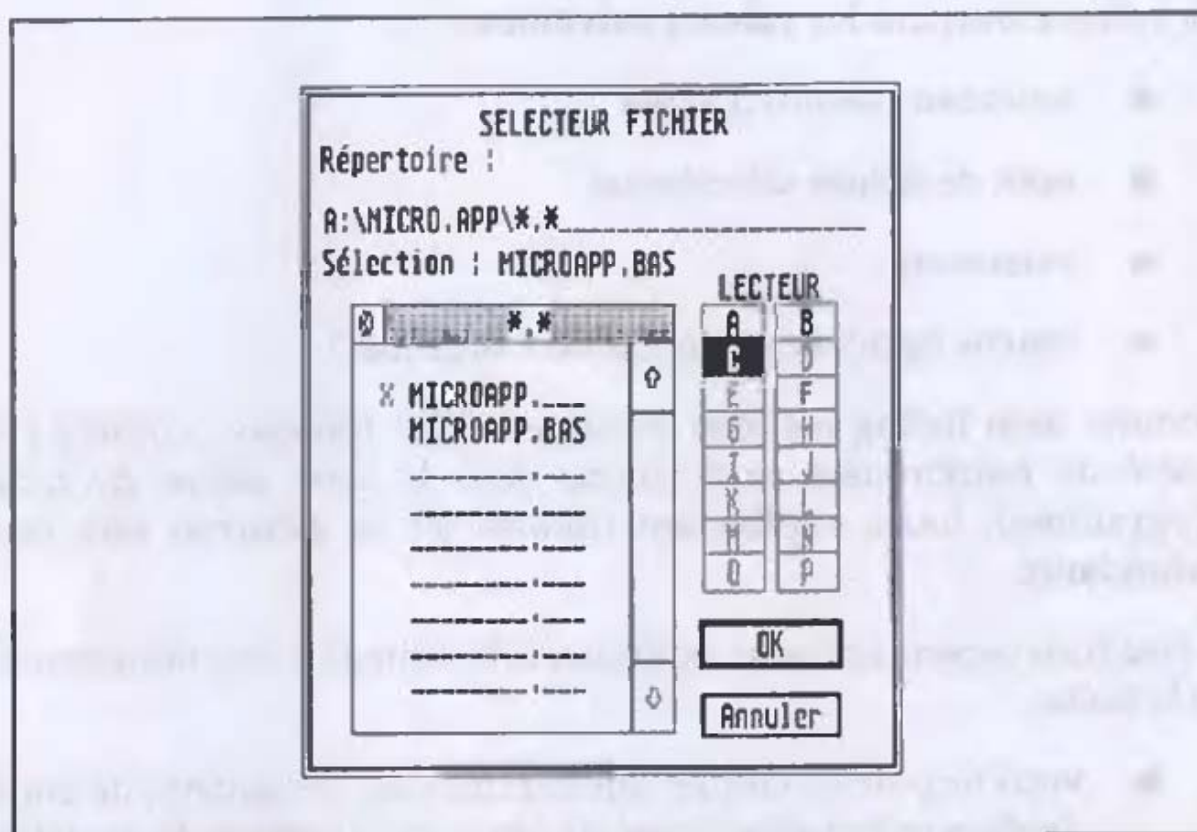


Figure 6.7 : boîte de sélection d'objet du STE (TOS 1.4)

Il serait également bien utile de disposer d'une ligne d'information indiquant à l'utilisateur pourquoi il est en train de sélectionner un fichier : pour le charger ou pour l'enregistrer ? Nous allons tenir compte de tout cela pour élaborer notre propre 'file selector box'. Il convient de passer les paramètres suivants lorsqu'on appelle la fonction :

- le chemin d'accès, précédé de l'identificateur d'unité de disque(tte)
- le nom de fichier venant s'afficher par défaut
- la ligne d'information (chargement d'un fichier etc.)

La boîte retournera les valeurs suivantes :

- nouveau chemin d'accès
- nom de fichier sélectionné
- extension
- touche appuyée (Annuler ou Confirmer)

Comme mon listing est bien documenté (en français : comme j'ai inséré de nombreuses explications dans le texte même de mon programme), toute explication donnée ici ne pourrait être que redondante.

Je voudrais cependant vous expliquer brièvement le fonctionnement de la boîte :

- vous ne pouvez cliquer que sur une case désignant une unité de disque(tte) effectivement connectée (routine du système d'exploitation), après quoi la formulation du chemin d'accès est automatiquement corrigée y compris à l'affichage
- vous pouvez changer d'extension en cliquant sur l'une des huit cases réservées à cet effet (voir figure 6.4), ce qui provoque, dans la fenêtre, l'affichage des noms de fichiers arborant cette extension. Lorsque vous souhaitez vous servir d'une extension qui n'existe pas encore, veuillez commencer par effacer l'ancienne extension dans le chemin d'accès et y inscrire la nouvelle ; cliquez ensuite dans la barre grisée de la fenêtre d'affichage des fichiers, qui contient l'extension actuelle. Le système reprend cette nouvelle extension et l'insère immédiatement dans le dernier des huit boutons d'extension, où elle sera mémorisée pour les prochains appels. Naturellement, cela ne fonctionne qu'avec des extensions comptant au total moins de cinq caractères, comme par exemple *.BAK, *.DOC ou *.H.


```

30 F$FILE3%L=14' TEXT in tree FILSEL
31 F$FILE4%L=15' TEXT in tree FILSEL
32 F$FILE5%L=16' TEXT in tree FILSEL
33 F$FILE6%L=17' TEXT in tree FILSEL
34 F$FILE7%L=18' TEXT in tree FILSEL
35 F$FILE8%L=19' TEXT in tree FILSEL
36 F$FILE9%L=20' TEXT in tree FILSEL
37 F$NAME%L=22' TEXT in tree FILSEL
38 F$DRVA%L=24' BOXCHAR in tree FILSEL
39 F$DRVB%L=25' BOXCHAR in tree FILSEL
40 F$DRVC%L=26' BOXCHAR in tree FILSEL
41 F$DRVD%L=27' BOXCHAR in tree FILSEL
42 F$DRVE%L=28' BOXCHAR in tree FILSEL
43 F$DRVF%L=29' BOXCHAR in tree FILSEL
44 EXT1%L=31' BOXTEXT in tree FILSEL
45 EXT2%L=32' BOXTEXT in tree FILSEL
46 EXT3%L=33' BOXTEXT in tree FILSEL
47 EXT4%L=34' BOXTEXT in tree FILSEL
48 EXT5%L=35' BOXTEXT in tree FILSEL
49 EXT6%L=36' BOXTEXT in tree FILSEL
50 EXT7%L=37' BOXTEXT in tree FILSEL
51 EXT8%L=38' BOXTEXT in tree FILSEL
52 F$ANNULE%L=44' BUTTON in tree FILSEL
53 F$RETURN%L=45' BUTTON in tree FILSEL
54 '
55 '
56 Appl_Init
57 PRINT @(0,30);"File-Selector-Box"
58 ' charger le fichier ressource
59 Rsrc_Load("FSELECT.RSC",Dummy%)
60 ' tout s'est bien passé ?
61 IF Dummy%=0 THEN
62     FORM_ALERT (1,"[3][Impossible de charger le fichier RSC !!][FIN]")
63     Appl_Exit
64     END
65 ENDIF
66 ' calculer l'adresse
67 Rsrc_Gaddr(0,0,Tree%L)
68 '
69 Info$=" Charger un fichier ... "
70 Fil_Sel("C:\","TEST.PRГ",Info$,Path$,Name$,Post$,Touche%)
71 '
72 Appl_Exit
73 END
74 '
75 '
76 DEF PROC Fil_Sel(Path$,Name$,Inf$,R Path$,R Na$,R Ext$,R Key%)
77 LOCAL T%,Quantite%,Ancre%,Ret%
78 Ext$="*.*)"
79 '
80 Form_Center(Tree%L,Xobj%L,Yobj%L,Wobj%L,Hobj%L)
81 Form_Dial(0,Xobj%L,Yobj%L,Wobj%L,Hobj%L)
82 Form_Dial(0,Xobj%L,Yobj%L,Wobj%L,Hobj%L)
83 '
84 Put_Text(F$INFO%L,Info$+ CHR$(0))
85 Put_Text(F$PATH%L,Path$+Ext$+ CHR$(0))
86 CHDIR Path$
87 Pos%= INSTR(Name$,".")
88 IF Pos%<>0 AND Pos%<9 THEN

```



```

89  Name$= LEFT$(Name$.Pos%-1)+ SPACE$(9-Pos%)+ MID$(Name$.Pos%+1)
90 ENDIF
91 Put_Text(Fsname%L,Name$+ CHR$(0))
92 Put_Text(Fsextend%L, CHR$(32)+Ext$+ CHR$(32)+ CHR$(0))
93
94 FOR T%=0 TO 7
95   IF FN Proof_Sel%L(Ext1%L+T%) THEN
96     Get_Text(Ext1%L+T%,Ext$)
97     Put_Text(Fspath%L,Path$+Ext$+ CHR$(0))
98     EXIT
99   ENDIF
100 NEXT T%
101
102 ' lire le répertoire et retourner le numéro du fichier
103
104 Ancre%-1
105 Directory(Ext$,Quantite%)
106 Contenu(Ancre%,Quantite%)
107
108 Objc_Draw(0,8,Xobj%L,Yobj%L,Wobj%L,Hobj%L,Tree%L)
109
110 ' Verrouiller les cases correspondant à des unités
111 ' non-connectées
112 BIOS (Dummy%,10)
113 FOR T%=0 TO 5
114   ' Bit mis dans Drive-Map ? non -> BIT(..) = 0
115   IF BIT(T%,Dummy%)=0 THEN
116     ' verrouiller cette case
117     Enable(Fsdrva%L+T%,0)
118     Disable(Fsdrva%L+T%,0)
119   ELSE
120     ' déverrouiller (autant en être certain!)
121     Enable(Fsdrva%L+T%,1)
122     Disable(Fsdrva%L+T%,1)
123   ENDIF
124 NEXT T%
125 ' d'abord dé-sélectionner toutes les unités, c'est plus sûr
126 FOR T%=0 TO 5
127   Select(Fsdrva%L+T%,0)
128 NEXT T%
129 ' puis activer l'unité actuelle indiquée par le chemin d'accès
130 Select(Fsdrva%L+ ASC(Path$)-65,1)
131 CHDIR Path$
132
133 ' cette fonction GEMDOS permettrait aussi de changer d'unité de
134 ' disque, mais n'oublions pas le reste du chemin d'accès ...
135 ' GEMDOS (,14, ASC(Path$)-65)
136
137 REPEAT
138   ' donner au GEM le contrôle du formulaire
139   Form_Do(Fsname%L,Tree%L,Ret%)
140   ' interdire le double-clic
141   Ret%=Ret% AND $7FFF
142
143   ' fichier sélectionné par un clic
144
145   IF Ret%=-Fsfile0%L AND Ret%<=-Fsfile9%L THEN
146     Get_Text(Ret%,File$)
147     IF File$<>"" THEN

```

```

148 IF ASC(File$)=7 THEN
149   ' Traiter le dossier en conséquence
150   Get_Text(Fspath%L,Path$)
151   Pos%= INSTR( MIRROR$(Path$),"\")
152   Path$= LEFT$(Path$, LEN(Path$)-Pos%+1)
153   File$= MID$(File$,3, INSTR(File$, CHR$(0))-3)
154   ' effacer les espaces vides dans le nom de dossier
155   WHILE INSTR(File$, CHR$(32))
156     Pos%= INSTR(File$, CHR$(32))
157     File$= LEFT$(File$,Pos%-1)+ MID$(File$,Pos%+1)
158   WEND
159   ' le chemin se termine par un backslash: '\'
160   Path$=Path$+File$+"\\"
161   ' Reformuler le chemin et l'inscrire dans la boîte
162   CHDIR Path$
163   Put_Text(Fspath%L,Path$+Ext$+ CHR$(0))
164   ' lire le nouveau répertoire
165   Directory(Ext$,Quantite%)
166   Contenu(Ancre%,Quantite%)
167   Objc_Draw(0,8,Xobj%L,Yobj%L,Wobj%L,Hobj%L,Iree%L)
168
169 ELSE
170   ' Fichier normal => copier dans le champ 'Filename'
171   Pos%= INSTR(File$,".")
172   ' le point nous fait des misères, détruisons-le
173   IF Pos%>0 THEN
174     File$= LEFT$(File$,Pos%-1)+ MID$(File$,Pos%+1)
175   ENDIF
176   Put_Text(Fsname%L, MID$(File$,3))
177   Objc_Draw(Fsname%L,0,Xobj%L,Yobj%L,Wobj%L,Hobj%L,Tree%L)
178 ENDIF
179
180 ENDIF
181
182 ' Changer d'unité de disque(tte)
183
184 IF Ret%>=Fsdrva%L AND Ret%<=Fsdrv%L THEN
185   Get_Text(Fspath%L,Path$)
186   Path$= LEFT$(Path$, LEN(Path$)- INSTR( MIRROR$(Path$),"\")+1)
187   Path$= CHR$(65+Ret%-Fsdrva%L)+ MID$(Path$,2)
188   Put_Text(Fspath%L,Path$+Ext$+ CHR$(0))
189   CHDIR Path$
190   ' à la place de 'CHDIR Path$' on peut utiliser une fonction
191   ' GEMDOS: Dsetdrv en lui donnant comme argument le numéro
192   ' de la nouvelle unité de disque(tte):
193   ' GEMDOS (,14, ASC(Path$)-65)
194   Directory(Ext$,Quantite%)
195   Ancre%=1
196   Contenu(Ancre%,Quantite%)
197   Objc_Draw(0,8,Xobj%L,Yobj%L,Wobj%L,Hobj%L,Tree%L)
198 ENDIF
199
200 ' Nouvelle extension
201
202 IF Ret%>=Ext1%L AND Ret%<=Ext8%L THEN
203   Get_Text(Ret%,Ext$)
204   Ext$= LEFT$(Ext$, LEN(Ext$)-1)
205   Get_Text(Fspath%L,Path$)
206   Path$= LEFT$(Path$, LEN(Path$)- INSTR( MIRROR$(Path$),"\")+1)

```



```

207 Put_Text(Fspath%L.Path$+Ext$+ CHR$(0))
208 Put_Text(Fsextend%L. CHR$(32)+Ext$+ CHR$(32)+ CHR$(0))
209 CHDIR Path$
210 Directory(Ext$.Quantite%)
211 Ancre%=1
212 Contenu(Ancre%.Quantite%)
213 Objc_Draw(0.8,Xobj%L.Yobj%L.Wobj%L.Hobj%L.Tree%L)
214 ENDIF
215 *
216 * un clic sur le coulisseau (slider) ?
217 *
218 IF Ret%=Fsslidebar%L THEN
219   Graf_Slidebox(Fscoulisse%L.Fsslidebar%L.1.Tree%L.Pos%)
220   Pos%=Pos%+500\((Quantite%-10)
221   Ancre%=((Quantite%-10)*Pos%\1000)+1
222   Contenu(Ancre%.Quantite%)
223   Objc_Draw(0.8,Xobj%L.Yobj%L.Wobj%L.Hobj%L.Tree%L)
224 ENDIF
225 *
226 * Scrolling du répertoire (flèche vers le haut)
227 *
228 IF Ret%=Fsup%L THEN
229   IF Ancre%>1 THEN
230     Ancre%=Ancre%-1
231     Contenu(Ancre%.Quantite%)
232     Objc_Draw(0.8,Xobj%L.Yobj%L.Wobj%L.Hobj%L.Tree%L)
233   ENDIF
234 ENDIF
235 *
236 * Scrolling du répertoire (flèche vers le bas)
237 *
238 IF Ret%=Fsdowndown%L THEN
239   IF Ancre%<Quantite%-9 THEN
240     Ancre%=Ancre%+1
241     Contenu(Ancre%.Quantite%)
242     Objc_Draw(0.8,Xobj%L.Yobj%L.Wobj%L.Hobj%L.Tree%L)
243   ENDIF
244 ENDIF
245 *
246 * clic sur la coulisse grisée => déplacer le coulisseau
247 *
248 IF Ret%=Fscoulisse%L THEN
249   Graf_Mkstate(Dummy%.Pos%,Dummy%,Dummy%)
250   Objc_Offset(Fsslidebar%L.Tree%L.Db_ofx%L.Db_ofy%L)
251   IF Pos%>Db_ofy%L THEN
252     Ancre%=Ancre%+10
253     IF Ancre%>Quantite%-9 THEN
254       Ancre%=Quantite%-9
255     ENDIF
256   ELSE
257     Ancre%=Ancre%-10
258     IF Ancre%<1 THEN
259       Ancre%=1
260     ENDIF
261   ENDIF
262   Contenu(Ancre%.Quantite%)
263   Objc_Draw(0.8,Xobj%L.Yobj%L.Wobj%L.Hobj%L.Tree%L)
264 ENDIF
265

```

```

266 ' quitter le subdirectory (si possible)
267
268 IF Ret%-Fscloser%L THEN
269   Get_Text(Fspath%L,Path%)
270   Pos%- INSTR( MIRROR$(Path%),"\")
271   IF Pos%<>0 THEN
272     Path%- LEFT$(Path%, LEN(Path%)-Pos%)
273     Pos%- INSTR( MIRROR$(Path%),"\")
274     IF Pos%<>0 AND Pos%<>3 THEN
275       Path%- LEFT$(Path%, LEN(Path%)-Pos%)+"\ "
276       CHDIR Path%
277       Put_Text(Fspath%L,Path%+Ext%+ CHR$(0))
278       Ancre%-1
279       Directory(Ext%.Quantite%)
280       Contenu(Ancre%.Quantite%)
281       Objc_Draw(0.8,Xobj%L,Yobj%L,Wobj%L,Hobj%L,Tree%L)
282     ENDIF
283   ENDIF
284 ENDIF
285
286 ' insérer la nouvelle extension dans la dernière extender-box
287
288 IF Ret%-Fsextend%L THEN
289   Get_Text(Fspath%L,Path%)
290   Path%- LEFT$(Path%, LEN(Path%)- INSTR( MIRROR$(Path%),CHR$(0)))
291   Pos%- INSTR( MIRROR$(Path%),"\")
292   ' surtout pas de backslash à la fin du chemin!
293   IF Pos%< LEN(Path%) THEN
294     Ext%- MID$(Path%, LEN(Path%)-Pos%+2)
295     FOR TX=0 TO 7
296       Select(Ext1%L+TX,0)
297     NEXT TX
298     IF LEN(Ext%)<6 THEN
299       Put_Text(Fsextend%L, CHR$(32)+Ext%+ CHR$(32)+ CHR$(0))
300       Put_Text(Ext8%L,Ext%+ CHR$(0))
301       Select(Ext8%L,1)
302       Directory(Ext%.Quantite%)
303       Ancre%-1
304       Contenu(Ancre%.Quantite%)
305     ELSE
306       Ext%="*,*"
307       Path%- LEFT$(Path%, LEN(Path%)-Pos%+1)
308       Put_Text(Fspath%L,Path%+Ext%+ CHR$(0))
309       Put_Text(Fsextend%L, CHR$(32)+Ext%+ CHR$(32)+ CHR$(0))
310       Select(Ext1%L,1)
311     ENDIF
312   ENDIF
313   Objc_Draw(0.8,Xobj%L,Yobj%L,Wobj%L,Hobj%L,Tree%L)
314 ENDIF
315
316 UNTIL Ret%-Fsannule%L OR Ret%-Fsreturn%L
317
318 Get_Text(Fspath%L,Path%)
319 Path%- LEFT$(Path%, LEN(Path%)- INSTR( MIRROR$(Path%), CHR$(0)))
320
321 IF Ret%-Fsannule%L THEN
322   Na%=""
323   Key%=0
324 ELSE

```



```

325 Key%-1
326 Get_Text(Fsname%L,Na$)
327 Pos%- INSTR(Na$, CHR$(0))
328 ' si aucun nom => retourner un string vide
329 IF Pos%-1 THEN
330   Na$=""
331 ELSE
332   Na$= LEFT$(Na$, LEN(Na$)-1)
333   ' ne pas oublier le point de séparation devant l'extension
334   IF LEN(Na$)>8 THEN
335     Na$= LEFT$(Na$,8)+". "+ MID$(Na$,9)
336   ENDIF
337   WHILE INSTR(Na$, CHR$(32))
338     Pos%- INSTR(Na$, CHR$(32))
339     Na$= LEFT$(Na$,Pos%-1)+ MID$(Na$,Pos%+1)
340   WEND
341 ENDIF
342 ENDIF
343 '
344 RETURN
345 '
346 '
347 DEF PRDC Directory(Extender$,R Quantite%)
348 LOCAL TX,Buffer$= SPACES(44),Buf_Adr%L,Ext_Adr%L,Dummy%
349 LOCAL Pos%,Post$
350 FOR TX=0 TO 125
351   Name$(TX)=""
352 NEXT TX
353 TX=0
354 ' Adresse du buffer DTA
355 Buf_Adr%L= LPEEK( VARPTR(Buffer$))+ LPEEK( SEGPTR +28)
356 GEMDOS (,26, HIGH(Buf_Adr%L), LOW(Buf_Adr%L))
357 Ext_Adr%L= LPEEK( VARPTR(Extender$))+ LPEEK( SEGPTR +28)
358 '
359 ' lire maintenant le premier intitulé du répertoire
360 '
361 GEMDOS (Dummy%,78, HIGH(Ext_Adr%L), LOW(Ext_Adr%L),16)
362 IF Dummy%<0 THEN ' aucun intitulé ?
363   Quantite%=0
364   RETURN
365 ENDIF
366 ' déterminer le type du fichier et le consigner dans array
367 IF ASC( MID$(Buffer$,22,1))=16 THEN ' identifier le dossier
368   Name$(TX)= CHR$(7)+ CHR$(32)+ MID$(Buffer$,31,13)
369 ELSE
370   Name$(TX)= CHR$(32)*2+ MID$(Buffer$,31,13)
371 ENDIF
372 ' lire jusqu'à l'octet nul ... (pour le GEM III)
373 Name$(TX)= LEFT$(Name$(TX), INSTR(Name$(TX), CHR$(0)))
374 '
375 REPEAT
376   TX=TX+1
377   GEMDOS (Dummy%,79)
378   ' plus aucun intitulé => sortir de la boucle
379   IF Dummy%<0 THEN
380     EXIT
381   ENDIF
382   IF ASC( MID$(Buffer$,22,1))=16 THEN ' identifier le dossier
383     Name$(TX)= CHR$(7)+ CHR$(32)+ MID$(Buffer$,31,13)

```

```

384 ELSE
385   Name$(TX)= CHR$(32)*2+ MID$(Buffer$,31,13)
386 ENDIF
387 ' lire jusqu'à l'octet nul (pour le GEM III)
388 Name$(TX)= LEFT$(Name$(TX), INSTR(Name$(TX), CHR$(0)))
389 UNTIL 0
390 Quantite%-TX
391 ' ajuster la longueur de tous les intitulés
392 FOR TX=0 TO Quantite%-1
393   Pos%= INSTR(Name$(TX), ".")
394   ' un point, mais pas à la bonne position ?
395   IF Pos%<>0 AND Pos%<11 AND Pos%>3 THEN
396     Post%= MID$(Name$(TX),Pos%)
397     ' insérer des espaces vides entre le nom et l'extension
398     Name$(TX)= LEFT$(Name$(TX),Pos%-1)+ SPACE$(10-Pos%+1)
399     Name$(TX)=Name$(TX)+Post%
400   ENDIF
401 NEXT TX
402 RETURN
403 '
404 '
405 DEF PROC Select(Numero%,Flag%)
406 LOCAL ADR%L=Tree%L+24*Numero%
407 IF Flag%=-1 THEN
408   LOCAL Ob_State%L= WPEEK(ADR%L+10) OR 1
409 ELSE
410   LOCAL Ob_State%L= WPEEK(ADR%L+10) AND 254
411 ENDIF
412 Objc_Change(Numero%,Ob_State%L,Xobj%L,Yobj%L,Wobj%L,Hobj%L,Tree%L)
413 RETURN
414 '
415 '
416 DEF PROC Enable(Numero%,Flag%)
417 LOCAL ADR%L=Tree%L+24*Numero%
418 IF Flag%=-1 THEN
419   WPOKE ADR%L+8, WPEEK(ADR%L+8) OR 65
420 ELSE
421   WPOKE ADR%L+8, WPEEK(ADR%L+8) AND 190
422 ENDIF
423 RETURN
424 '
425 '
426 DEF PROC Disable(Nu%,Flag%)
427 LOCAL ADR%L=(Tree%L+24*Nu%)+10
428 IF Flag%=-1 THEN
429   Objc_Change(Nu%, WPEEK(ADR%L) AND 247,Xobj%L,Yobj%L,Wobj%L,Hobj%L,
Tree%L)
430 ELSE
431   Objc_Change(Nu%, WPEEK(ADR%L) OR 8,Xobj%L,Yobj%L,Wobj%L,Hobj%L,
Tree%L)
432 ENDIF
433 RETURN
434 '
435 '
436 DEF PROC Put_Text(Numero%,Text$)
437 ' d'abord lire l'adresse du texte ...
438 LOCAL ADR%L=Tree%L+24*Numero%
439 LOCAL Tedinfo%L= LPEEK(ADR%L+12)
440 LOCAL Text_ADR%L= LPEEK(Tedinfo%L)

```



```

441 LOCAL TX=0
442 ' puis l'insérer caractère par caractère ...
443 WHILE TX< LEN(Text$)
444   POKE Text_Adr%L+TX, ASC( MID$(Text$,TX+1,1))
445   TX=TX+1
446 WEND
447 RETURN
448 '
449 '
450 DEF PROC Get_Text(Numero%,R Text$)
451   LOCAL Adr%L=Tree%L+24*Numero%
452   LOCAL Tedinfo%L= LPEEK(Adr%L+12)
453   LOCAL Text_Adr%L= LPEEK(Tedinfo%L)
454   LOCAL TX=-1
455   Text$=""
456   ' prendre tous les caractères jusqu'à l'octet nul
457   ' et les inscrire dans le string 'Text$' ...
458   REPEAT
459     TX=TX+1
460     Text$=Text$+ CHR$( PEEK(Text_Adr%L+TX))
461   UNTIL PEEK(Text_Adr%L+TX)=0
462   '
463   RETURN
464   '
465   '
466 DEF PROC Contenu(Start%.Nombre%)
467   LOCAL TX.Note_1%L
468   IF Nombre%<=10 THEN
469     WPOKE (Tree%L+24*Fsslider%L)+18,0
470     WPOKE Tree%L+24*Fsslider%L+22, WPEEK(Tree%L+24*Fscoulis%L+22)
471     FOR TX=0 TO Nombre%-1
472       Put_Text(Fsfile0%L+TX,Name$(TX))
473       IF MID$(Name$(TX),3,1)=". " THEN
474         Enable(Fsfile0%L+TX,0)
475       ELSE
476         Enable(Fsfile0%L+TX,1)
477       ENDIF
478       Select(Fsfile0%L+TX,0)
479     NEXT TX
480     FOR TX=Nombre% TO 9
481       Put_Text(Fsfile0%L+TX, CHR$(0))
482       Enable(Fsfile0%L+TX,0)
483       Select(Fsfile0%L+TX,0)
484     NEXT TX
485   ELSE
486     Note_1%L= WPEEK(Tree%L+24*Fscoulis%L+22)
487     WPDKE Tree%L+24*Fsslider%L+18,(Note_1%L*(Start%-1))/Nombre%
488     WPOKE Tree%L+24*Fsslider%L+22,(Note_1%L*10)/Nombre%
489     FOR TX=0 TO 9
490       IF Name$(Start%+TX)="" THEN
491         Put_Text(Fsfile0%L+TX, CHR$(0))
492         Enable(Fsfile0%L+TX,0)
493         Select(Fsfile0%L+TX,0)
494       ELSE
495         Put_Text(Fsfile0%L+TX,Name$(Start%+TX))
496         IF MID$(Name$(Start%+TX),3,1)=". " THEN
497           Enable(Fsfile0%L+TX,0)
498         ELSE
499           Enable(Fsfile0%L+TX,1)

```

```

500         ENDIF
501         Select(Fsfile0%L+T%,0)
502     ENDIF
503 NEXT T%
504 ENDIF
505 RETURN
506 '
507 '
508 DEF FN Proof_Sel%L(Numero%)
509 LOCAL ADR%L=Tree%L+24*Numero%
510 IF WPEEK(ADR%L+10) AND 1=1 THEN
511     RETURN (-1)
512 ENDIF
513 RETURN (0)
514 '

```

6.6. La technique des fenêtres

L'une des plus grandes innovations du GEM fut sans conteste la possibilité de se servir de fenêtres (anglais : windows) : que vous vous serviez d'un logiciel de traitement de texte ou d'un tableur, vous pouvez constamment écrire plusieurs données différentes dans la mémoire et les afficher à l'écran.

Bien sûr, vous vous doutez déjà que nous allons recourir à la bibliothèque GEMLIB.BAS.

Une fenêtre se compose de plusieurs éléments : deux 'ascenseurs' (un horizontal, un vertical) et trois cases réparties dans les trois coins à partir de celui qui se trouve en haut à gauche : une case de fermeture, une case d'ouverture maximale, une case de modification de la taille de la fenêtre (voir figure 6.8). Tout programme faisant appel aux bons et loyaux services d'une fenêtre doit être à même d'enregistrer plusieurs événement (voilà un mot qui ne nous est pas inconnu...) et de réagir en conséquence.

Nous utilisons pour cela une routine GEM avec laquelle nous avons déjà fait connaissance lorsqu'il s'agissait de gérer la barre des menus :

```
Evnt_Mesag(Evénement$)
```

Aussi modeste qu'elle puisse paraître, cette fonction permet bel et bien (heureusement !) de transmettre à l'application en cours d'utilisation des informations décrivant les événement les plus

divers (exemple : l'utilisateur a déplacé la fenêtre sur l'écran). Mais avant qu'elle ne puisse intervenir aussi brillamment, c'est au programmeur de se soucier de créer sa fenêtre.

❑ Les éléments composant une fenêtre

Une fenêtre se compose de nombreux éléments, comme vous le voyez sur la figure ci-dessous :

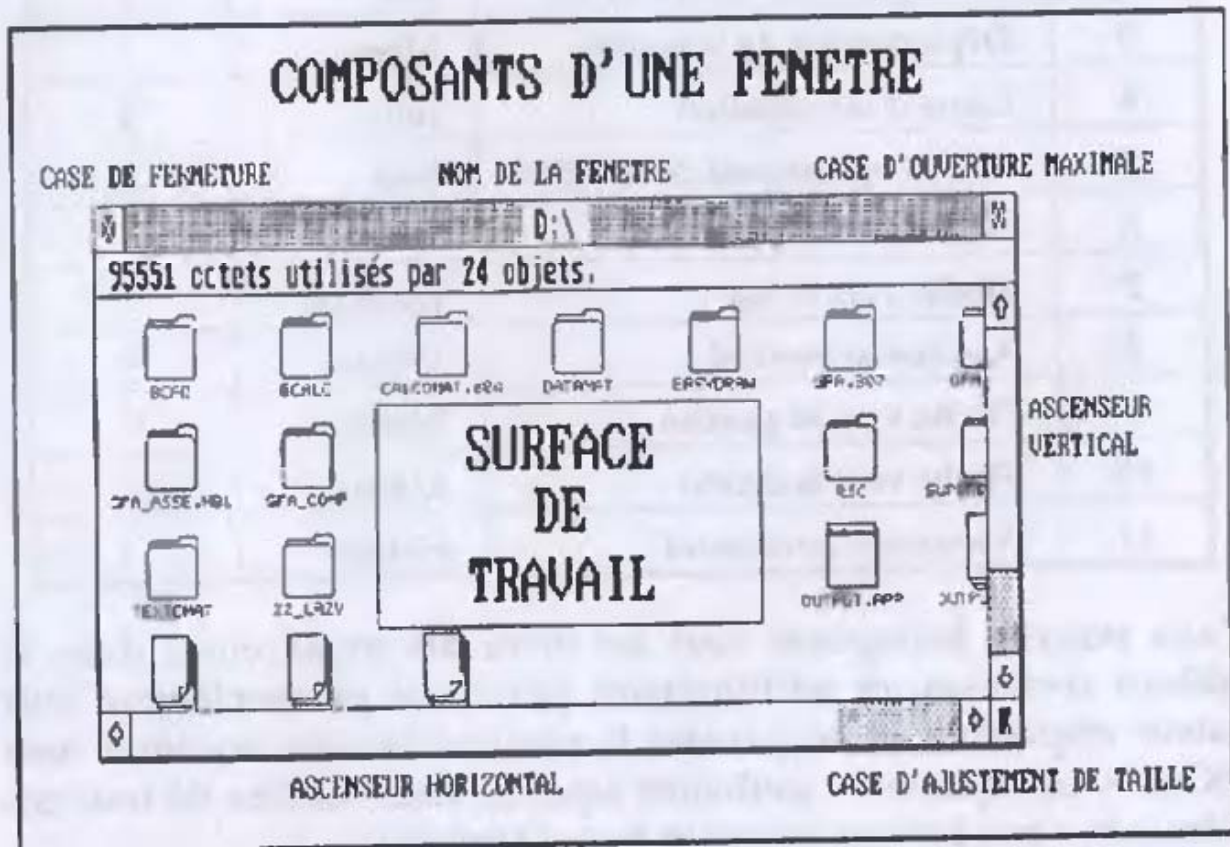


Figure 6.8 : les éléments composant une fenêtre GEM.

Il n'est pas toujours indispensable qu'une fenêtre soit équipée de tous ces éléments (elle peut par exemple être dépourvue de case de fermeture ou de case d'ajustement de sa taille) : c'est pourquoi il est nécessaire de la décrire précisément avant qu'elle ne s'affiche à l'écran. On se sert pour cela de l'instruction :

```
Wind_Create(kind,xmax,ymax,wmax,hmax,handle)
```

'Kind' sert à lister les éléments contenus ou non dans la fenêtre : un bit est en effet réservé pour chaque élément. Selon que ce bit est mis ou non, l'élément est présent ou non dans la fenêtre :

Bit	Élément correspondant	Nom	Valeur
0	Barre de titre	Name	0
1	Case de fermeture	Close	1
2	Case d'ouverture maximale	Full	2
3	Déplacement de la fenêtre	Move	3
4	Ligne d'information	Info	4
5	Case d'ajustement de la taille	Size	5
6	Flèche vers le haut	Uparrow	6
7	Flèche vers le bas	Dnarrow	7
8	Ascenseur vertical	Vslide	8
9	Flèche vers la gauche	Ffarrow	9
10	Flèche vers la droite	Rtarrow	10
11	Ascenseur horizontal	Hslide	11

Vous pouvez juxtaposer tous les éléments mentionnés dans le tableau ci-dessus en additionnant purement et simplement leur valeur respective, et en passant le résultat de cette addition sous <Kind>. Lorsque vous souhaitez équiper votre fenêtre de tous ces éléments, vous écrivez (notation hexadécimale) :

\$FFF

ce qui fait que tous les bits (et même quelques uns de plus, mais ça ne fait rien) sont mis.

Les quatre paramètres suivants (xmax, ymax, wmax et hmax) indiquent la taille maximale que la fenêtre peut prendre. Ces paramètres varient selon la résolution de l'écran, c'est pourquoi il est conseillé de s'enquérir d'abord de la taille maximale de l'écran entier. Comme le moniteur Atari-ST est alors considéré comme une fenêtre, nous pouvons retrouver sa taille à l'aide de l'instruction GEM :


```
Wind_Get(Handle, Gfield, xmax, ymax, wmax, hmax)
```

Donnez la valeur 0 au paramètre 'handle' (sur lequel nous reviendrons plus longuement ci-après) lorsque vous souhaitez obtenir en retour la taille maximale de l'écran entier (qui est considéré comme la fenêtre 0, d'où ce handle). Cette fonction vous permet aussi de connaître les autres paramètres, en donnant à Gfield une des valeurs suivantes :

Gfield	donne en retour les coordonnées
4	de la surface de travail sur l'écran
5	de la surface globale de la fenêtre
6	de la fenêtre précédente
7	de la taille maximale de la fenêtre
11	du premier rectangle dans la fenêtre
12	du rectangle suivant

Comme nous souhaitons pour l'instant connaître les coordonnées de la surface de travail maximale sur l'écran, nous donnons la valeur 4 au paramètre Gfield; en respectant la syntaxe ci-dessus, nous écrivons :

```
Wind_Get(0,4,xmax,ymax,wmax,hmax)
```

Une fois exécutée, la fonction retourne la taille maximale de la fenêtre (qui dépend, comme nous l'avons déjà dit, de la résolution de l'écran) dans les quatre dernières variables, que nous utilisons avec Wind_Create().

Le premier paramètre était donc le 'windows-handle' nommé plus simplement 'handle'. De quoi s'agit-il ? Le GEM doit pouvoir identifier chacune des fenêtres, puisqu'il vous permet d'en ouvrir plusieurs à la fois sur l'écran. C'est pourquoi chaque fenêtre reçoit un numéro d'identification, le handle, à l'aide duquel on peut ensuite la manipuler individuellement. Le GEM transmet ce handle au programme en cours après le lancement de la fonction Wind_Create.

Nous avons presque terminé de créer notre fenêtre, mais il manque encore un petit détail : toute fenêtre peut contenir un nom et une ligne d'information. Les instructions étudiées jusqu'ici ne nous permettent pas d'inscrire ces données dans la fenêtre, il nous faut recourir à

`Wind_Set(Handle,Sfield,Sline$,Adresse)`

Handle représente le numéro d'identification de la fenêtre dans laquelle les données viendront s'insérer, et Sfield sert à préciser le paramètre dont il s'agit :

Sfield	pour le paramètre
2	nom de la fenêtre
3	ligne d'information

Le contenu de la ligne d'information ou du nom de la fenêtre se compose d'une chaîne de caractères devant se trouver sous Sline\$. Particularité du GEM : il a besoin de plus de l'adresse à laquelle il peut déposer le nom de la fenêtre ou le contenu de la ligne d'information. Il est conseillé de réserver un espace suffisant en mémoire vive à l'aide de l'instruction :

Adresse - MEMORY(Quantité-Octets)

L'ordinateur ne touche plus à l'espace ainsi réservé ; qui plus est, la fonction retourne l'adresse à partir de laquelle on trouve l'espace ainsi réservé. Pour l'instant, je pense que 70 octets suffiront amplement pour ce que nous voulons faire :

Adresse - MEMORY(70)

`Wind_Set(Handle,2,"Nom_de_la_fenêtre",Adresse)`

Ces deux lignes servent d'abord à réserver un espace mémoire de 70 octets, puis à y inscrire le nom de la fenêtre, qui est ainsi 'baptisée'. Wind_Set s'utilise encore pour bien d'autres usages en Omikron Basic, mais avec d'autres paramètres : ainsi par exemple

`Wind_Set(Handle,Sfield,Sw1)`

sert à préciser les paramètres de la fenêtre de la façon suivante :

Sfield	pour modifier le paramètre
1	redétermine les composants entrés par 'Wind_Create'
8	modifie la position de Hslide
9	modifie la position de Vslide
15	modifie la taille relative de Hslide
16	modifie la taille relative de Vslide

Les données nécessaires sont passées à la variable Sw1. Comme le GEM ne peut gérer qu'une fenêtre à la fois,

`Wind_Set(Handle)`

permet d'activer une autre fenêtre, tout en désactivant la fenêtre qui était active auparavant.

☐ Comment exploiter les événement

Assez de préliminaires, venons-en à l'utilisation concrète de l'instruction `Evnt_Mesag()`, que nous utilisons à nouveau dans une boucle :

```

REPEAT
  Evnt_Mesag(Evénement$)
  .....
  .....
  <vérifier la nature des événement survenus>
  .....
  .....
UNTIL <condition d'interruption remplie>

```

Une fois qu'il a reçu l'instruction `Evnt_Mesag`, le GEM attend qu'un événement se produise, et le passe sous `<Evénement$>`. Mais comme de nombreux événement très divers peuvent se produire, la répartition entre 5 indices dans le buffer 'message' ne suffit plus : il faut le découper pour qu'il puisse accueillir jusqu'à 8 mots longs (indice 7) :

```

DIM Evénement(7)
REPEAT
  Evnt_Mesag(Me$)
  FOR T%- 0 TO 7
    Evénement(T%)= CVI( MID$(Me$,T%*2+1,2))
  NEXT T%
  ....
  ....
UNTIL <condition d'interruption remplie>

```

Le premier élément (indice 0) contient le code d'identification de l'événement, selon la nomenclature suivante :

Code	Nom	Nature de l'événement survenu (indice=>)
10	Mn_Selected	clic sur une option d'un menu
20	Mw_Redraw	redéfinition de la surface 3 => window-handle 4 => coordonnées X de la surface 5 => coordonnées Y de la surface 6 => Largeur de la surface 7 => Hauteur de la surface
21	Wm_Topped	fenêtre à activer 3 => window-handle
22	Wm_Closed	clic sur la case de fermeture 3 => window-handle
23	Wm_Fulled	clic sur la case d'ouverture maximale
24	Wm_Arrowed	clic sur une des flèches 3 => window-handle 4 => objet sélectionné par ce clic : 0 = retour d'une page en arrière 1 = avance d'une page 2 = retour d'une ligne en arrière 3 = avance d'une ligne 4 = décalage d'une page vers la gauche 5 = décalage d'une page vers la droite 6 = décalage d'une colonne vers la gauche 7 = décalage d'une colonne vers la droite

Code	Nom	Nature de l'événement survenu (indice=>)
25	Wm_Hslid	déplacement du coulisseau de l'ascenseur horizontal 3 => window-handle 4 => position relative du coulisseau 0 = à l'extrême gauche 1000 = à l'extrême droite
26	Wm_Vslid	déplacement du coulisseau de l'ascenseur vertical 3 => window-handle 4 => position relative du coulisseau 0 = tout en haut 1000 = tout en bas
27	Wm_Sized	modification de la taille de la fenêtre 3 => window-handle 4 => nouvelles coordonnées X 5 => nouvelles coordonnées Y 6 => nouvelle largeur 7 => nouvelle hauteur
28	Wm_Moved	déplacement de la fenêtre 3 => window-handle 4 => nouvelles coordonnées X 5 => nouvelles coordonnées Y 6 => nouvelle largeur de la fenêtre 7 => nouvelle hauteur de la fenêtre
29	Wm_Topped	une nouvelle fenêtre a été activée 3 => window-handle
30	Ac_Open	clic sur un accessoire 3 => numéro d'identification du menu
31	Ac_Close	refermer un accessoire 3 => numéro d'identification du menu.

Lorsque votre programme exige par exemple l'ouverture d'une fenêtre (que vous avez définie auparavant, faute de quoi c'est évidemment impossible !) il vous suffira d'appeler la procédure en écrivant :

```
Wind_Open(Handle,xmax,ymax,wmax,hmax)
```

ce qui fera apparaître la fenêtre à l'écran. Une petite instruction (consistant à vérifier si l'utilisateur a ou non cliqué sur la case de fermeture) suffira ensuite pour refermer cette même fenêtre :

`Wind_Close(Handle)`

Ceci permet certes de faire disparaître la fenêtre de la surface de l'écran, mais tout en la conservant au niveau interne à l'ordinateur. En effet, pour la refermer définitivement, il faut écrire :

`Wind_Delete(Handle)`

Une fois muni de ces connaissances de base, vous ne devriez plus guère rencontrer de difficulté dans la programmation de vos fenêtres. Vous trouverez un programme de démonstration complet sur la disquette jointe à l'Omikron Basic, dans le fichier GEMDEMO.BAS : toutes les manoeuvres que nous venons d'expliquer y sont tour à tour mises en oeuvre.

Chapitre 7

Multitasking

Le multitasking (fonctionnement multitâches) : un progrès décisif qui permet de charger simultanément en mémoire vive plusieurs programmes différents, et de les utiliser apparemment en même temps. Les possibilités de l'Atari sont assez limitées en ce domaine, ce qui ne tient pas tant au microprocesseur qu'au système d'exploitation qui ne prévoyait pas ce type d'usage.

En fait, lorsque vous avez l'impression (trompeuse) qu'un ordinateur fait tourner deux programmes en même temps, c'est tout simplement qu'il passe sans arrêt de l'un à l'autre. Aucun être humain, et à plus forte raison aucun ordinateur, ne peut faire deux choses à la fois. Mais il suffit d'exécuter un bout d'un programme puis de passer rapidement à l'exécution d'un bout de l'autre programme pour que l'utilisateur ait l'impression que l'ordinateur traite deux programmes à la fois.

L'une des possibilités de travail en multitâches avec l'Atari réside justement dans le recours à la fonction Event du GEM : celle-ci permet de guetter la survenance de plusieurs événements et de lancer les réactions y afférentes. Mais nous n'allons pas revenir là-dessus dans ce chapitre.

Nous allons plutôt nous intéresser aux instructions comprises dans l'Omikron Basic, qui permettent de guetter certains événements (appui sur certaines touches par exemple) durant le déroulement du programme. Ces instructions se déroulent toujours selon le même schéma :

- au début du programme, on entre une instruction indiquant à l'Omikron Basic qu'il doit se brancher sur un sous-programme particulier lorsque survient tel ou tel événement ;
- le programme se déroule normalement ;
- lorsque l'événement déclaré survient, l'Omikron Basic passe, comme on le lui a indiqué, dans le sous-programme et l'exécute, après quoi il revient dans le programme principal, à l'endroit où il en était sorti.

Voici les diverses instructions de multitasking en Omikron Basic :

□ ON KEY GOSUB<cible>

Sert à se brancher sur la routine <cible> dès que l'utilisateur appuie sur telle touche ; il convient d'annuler la condition de sortie (la touche appuyée) du programme principal dès la première ligne de la routine

```
ON KEY GOSUB 0
```

avant l'exécution de cette dernière

```
ON KEY GOSUB Saisie
```

```
.....
```

```
.....
```

```
.....
```

```
-Saisie
```

```
ON KEY GOSUB 0
```

```
.....
```

```
<insérer ici la routine>
```

```
.....
```

```
.....
```

```
RETURN
```


Lorsque la condition de sortie du programme principal consiste à appuyer sur une touche de la souris, vous écrivez :

```
ON MOUSEBUT GOSUB <Cible>
```

pour annuler cette instruction, on remplace également <cible> par zéro ; par exemple,

```
ON MOUSEBUT GOSUB 0  
ON HELP GOSUB <Cible>
```

provoque un branchement sur la routine dès que l'utilisateur appuie sur la touche <HELP> (se trouvant entre le clavier et le pavé numérique) ; pour y mettre fin, vous écrivez :

```
ON HELP GOSUB 0
```

Notez que le code-touche de HELP est alors effacé du buffer-clavier.

□ ON TIMER <Délai> GOSUB <Cible>

Cette instruction vous permet de passer dans un autre programme <cible> dès qu'un certain <délai> (indiqué en secondes) est écoulé. Le délai se découpe en tranches de 1/200ème de seconde, mais l'Omikron Basic se charge si nécessaire d'arrondir les valeurs entrées. Vous annulez cette instruction en écrivant :

```
ON TIMER GOSUB #0
```

La première partie de ce livre est consacrée à la description des principes de base de la programmation en langage BASIC.

La deuxième partie est consacrée à la description des principes de base de la programmation en langage BASIC.

La troisième partie est consacrée à la description des principes de base de la programmation en langage BASIC.

La quatrième partie est consacrée à la description des principes de base de la programmation en langage BASIC.

La cinquième partie est consacrée à la description des principes de base de la programmation en langage BASIC.

La sixième partie est consacrée à la description des principes de base de la programmation en langage BASIC.

La septième partie est consacrée à la description des principes de base de la programmation en langage BASIC.

La huitième partie est consacrée à la description des principes de base de la programmation en langage BASIC.

La neuvième partie est consacrée à la description des principes de base de la programmation en langage BASIC.

La dixième partie est consacrée à la description des principes de base de la programmation en langage BASIC.

Chapitre 8

Le compilateur

Tout au long de cet ouvrage, j'ai déjà mentionné plusieurs fois les mots 'interpréteur' et compilateur' (en anglais : 'interpréter' et 'compiler') sans toutefois les définir : il est maintenant amplement temps de combler cette lacune.

Voyons d'abord ce qu'est un 'interpréter'. Le processeur de votre ordinateur, l'Atari, parle en quelque sorte sa propre langue, laquelle ne se compose que de 0 et de 1 : il ne comprend absolument pas le Basic. S'il est cependant possible, comme vous le constatez, de lui transmettre des instructions en Omikron Basic c'est parce qu'il existe un intermédiaire, un interprète, entre votre programme écrit en Basic et le langage (dit 'langage machine') du processeur. C'est justement ce qu'on appelle l'interpréteur, le programme Omikron Basic que vous devez charger avant chaque session de travail pour pouvoir programmer en Basic.

Pour préciser le rôle de cet interpréteur, disons qu'il fait exactement de l'interprétariat simultané : cela signifie qu'il prend une à une les instructions en Basic (qu'il trouve dans le texte du programme), et qu'il les traduit pour le processeur. Ce mode de fonctionnement entraîne évidemment un énorme inconvénient : le processus d'interprétariat demande du temps, et le code traduit n'est jamais

pleinement satisfaisant (rappelons-nous ce bon précepte italien : toute traduction est trahison, ou plus simplement : avez-vous déjà écouté une traduction simultanée à la radio ou à la télévision ?). Bref, tout cela ralentit le déroulement du programme.

Le compilateur fait lui plutôt le travail du traducteur : il consigne le résultat de son activité une fois pour toute 'par écrit', c'est-à-dire dans un fichier, si bien qu'il devient inutile ensuite. On obtient en quelque sorte un programme autonome. L'interprète étant ainsi supprimé, les programmes compilés tournent beaucoup plus rapidement que les programmes interprétés. Si l'on en croit les indications du producteur, un programme écrit en Omikron Basic puis compilé tournerait 9,5 fois plus vite que le même programme interprété.

8.1. Comment utiliser le compilateur

Le mode d'emploi est très simple :

- lancez le compilateur depuis la disquette en effectuant un double-clic sur le fichier COMPILER.PRG
- le compilateur vous envoie une boîte de sélection d'objet, dans laquelle vous entrez le nom du programme que vous souhaitez compiler ; ce programme BASIC doit se présenter sous sa forme codée (et non sous la forme d'un fichier ASCII sauvegardé par SAVE,A ou <F8>) ; si tel n'est pas le cas, rechargez l'éditeur Omikron Basic et le programme, puis sauvegardez ce dernier correctement ;
- on dit que le compilateur exécute son oeuvre en trois 'passes' ; lors des deuxième et troisième passes, vous pouvez en suivre le déroulement à l'écran, puisque vous y voyez défiler les numéros des lignes du programme en cours de compilation ;

- une fois le texte compilé, vous avez sur votre disque un programme directement exécutable, se trouvant dans un fichier de même nom mais dont l'extension est devenue 'prg' (exemple : DEMO.BAS devient DEMO.PRG) ;
- ⚠ **Attention :** Avant toute compilation, vérifiez qu'il vous reste assez de place sur votre disque, car un programme compilé occupe jusqu'à deux fois plus de place que l'original en Basic ; ceci étant fait, le compilateur de l'Omikron Basic vous envoie à nouveau une boîte de sélection d'objet, à l'aide de laquelle vous pouvez éventuellement compiler un deuxième programme etc. ;
- vous devez enregistrer le fichier BASLIB sur toute disquette contenant un programme ainsi compilé, car l'ordinateur en a absolument besoin (exception : CUTILIB) ;
 - vous pouvez ensuite lancer votre programme, sans plus recourir à l'interpréteur.

8.2. Les commandes

Le compilateur comprend un certain nombre de commandes, dont il tient compte lors du traitement d'un programme. Pour éviter que l'interpréteur ne 'trébuche' là-dessus, je vous conseille d'intégrer discrètement ces codes de commande dans le texte de votre programme en les entrant comme des procédures :

```
DEF PROC <Code de commande>: RETURN
```

L'interpréteur ne se formalisera pas de rencontrer ainsi un code de commande qu'il ne comprend pas : il 'pensera' qu'il s'agit là d'une définition comme une autre. Quant au compilateur, ne vous faites pas de souci, il se débrouillera bien. Les deux instructions TRACE peuvent elles-aussi être intégrées dans une condition IF...THEN, mais pas les deux instructions suivantes :

MA (MULTITASKING_ALWAYS)
MBS (MULTITASKING_BETWEEN_STATEMENTS)

S'il est possible de faire dépendre l'exécution des instructions TRACE d'une condition à vérifier, ceci est impossible dans le cas des deux instructions de multitasking.

TRACE_ON

Sert à surveiller :

- Ctrl-C (break)
- les interruptifs
- le clavier

avant de passer à l'exécution d'une instruction comme ON_MOUSEBUT ou ON_KEY. Le pointeur de pile, important pour les instructions ON_ERROR et RESUM, est sauvegardé.

TRACE_OFF

permet d'annuler ce contrôle préalable de Ctrl-C, des interrupteurs et du clavier, ce qui rend inopérantes les instructions de multitasking. La variable système ERL reçoit constamment la valeur 0, et RESUME devient impossible.

MULTITASKING_ALWAYS (MA)

Même lorsqu'il ne trouve pas de commande TRACE_ON dans le texte source, le programme compilé vérifie toujours si la touche CTRL-C n'a pas été appuyée ou si une condition de multitasking n'a pas été remplie. Il est du reste possible d'interrompre l'exécution du programme en plein milieu d'une instruction (par exemple SORT) dans la mesure où l'ordinateur n'est pas précisément en train d'exécuter une routine du système d'exploitation. Les routines appelées par des instructions de multitasking ne doivent exécuter aucune manipulation de string.

MULTITASKING_BETWEEN_STATEMENTS (MBS)

formule d'interruption du traitement d'un programme (pour passer dans un autre), compatible avec l'Omikron Basic : les interruptions ne peuvent se produire (lorsqu'elles sont autorisées par TRACE_ON) qu'à la limite des instructions.

Après un CLEAR, il faut relancer ces deux instructions ; elles ne doivent par contre jamais figurer dans une condition IF...THEN.

Sachez encore que vous pouvez combiner ces codes de commande.

MA & TRACE_OFF

c'est là le paramétrage standard : RESUME ne fonctionne pas, ERL contient toujours la valeur 0 ; les routines de multitasking ne doivent jamais contenir d'instruction de manipulation de string (string-handling), ni de dimensionnement d'un tableau (DIM) ni d'ouverture d'un fichier (OPEN).

MBS & TRACE_OFF

RESUME ne fonctionne pas ; ERL contient toujours la valeur 0 et vous ne pouvez pas travailler en multitâche !

MA & TRACE_ON

les routines de multitasking ne peuvent contenir des instructions de manipulation de string, ni OPEN ni DIM.

MBS & TRACE_ON

permet de conserver une totale compatibilité avec l'interpréteur.

8.3. Les programmes en BASIC et le compilateur

En règle générale, le compilateur est bien plus 'pinailleur' que l'interpréteur, c'est pourquoi vous n'échapperez pas à certains ajustements :

- contrairement à l'interpréteur, le compilateur ne dimensionne pas automatiquement : vous devez dimensionner vos tableaux arrays en début de programme et les redimensionner après un CLEAR ; faute de quoi, si vous tentez d'accéder à une de ces variables, vous recevrez le message : "? BUS ERROR" ; il est interdit de dimensionner dans une procédure et une fonction lorsque cette procédure recourt à des variables ou paramètres locaux ;
- le compilateur peut réduire un tableau de variables lors de son redimensionnement, alors que l'interpréteur ne peut que le restructurer ;
- il est impossible d'utiliser localement des flags (%F), ils ne peuvent servir non plus de paramètres de retour dans les fonctions et procédures ;
- il faut faire bien attention lorsqu'on utilise des variables locales qui ne sont pas conservées dans le segment GARBAGE mais dans un segment de string et dans la pile du processeur ; il faudra si nécessaire augmenter la pile à l'aide de CLEAR,X ;
- il faut définir les fonctions de telle sorte qu'elles arborent toujours le suffixe des variables qu'elles retournent ; par exemple, lorsqu'une fonction retourne une variable float, elle doit se terminer par "!" ou "#", selon le degré d'exactitude souhaité ; CSNG et CINT permettent de modifier ce format à l'intérieur de la fonction ;

- les variables string se trouvant dans des attributs de FIELD ne peuvent s'utiliser localement tant que l'attribut de FIELD n'a pas été supprimé par l'attribution d'une valeur par LET ; l'attribut de FIELD est annulé lorsque survient la fermeture du canal correspondant par CLOSE ;
- la division de deux integers (nombres entiers) donne toujours comme résultat une valeur décimale, même si les positions au-delà de la virgule ne servent à rien ; il est donc préférable de recourir à la division de nombre entier "\" ;
- le type de variables dans une ligne DATA doit concorder avec le READ correspondant ; par exemple, pour reprendre dans la variable décimale "G!" le nombre entier "1", il faut déclarer le nombre entier comme un nombre décimal dans la ligne DATA, ce que l'on fait en lui ajoutant un point décimal : "1." ;
- lorsque le texte du programme contient des instructions de multitasking, il faut qu'il contienne aussi des codes de commande TRACE_ON et MSB du compilateur ; INPUT\$ et INPUT USING sont eux-aussi interrompus par ON TIME GOSUB si le code de commande MSB se trouve quelque part dans le texte du programme ;
- le branchement dans une routine de traitement d'erreur peut entraîner la perte de la valeur finale d'une boucle integer-FOR-NEXT ; la routine d'erreur doit donc restaurer le compteur de boucle, et le manuel de l'Omikron Basic contient une bonne idée à ce sujet :


```

100 FOR M=0 TO Numéro
110     <insérer ici le contenu de la boucle>
120 NEXT M
...
1000 Routine_d'erreur
1010 IF ERL=110 THEN FOR M=0 TO Numéro:RESUME NEXT:
NEXT M
1020 RESUME NEXT

```
- la fonction ASC("") retourne la valeur 0 ;

- l'arithmétique des nombres entiers ne tient pas compte des débordements OVERFLOW, ce qui peut amener des résultats faux si le nombre ne peut plus contenir la valeur véritable ; alors que l'interpréteur procède dans ce cas automatiquement à une transformation de type, le compilateur ne le fait pas ;
- dans le compilateur, EXIT ne peut s'utiliser que sans indication de cible, c'est-à-dire EXIT -1 ; EXIT TO ne s'emploie que pour sortir d'une boucle.

8.4. Comment optimiser un programme

Je vous conseille de lire attentivement les petits conseils qui suivent si vous souhaitez vraiment tirer un maximum d'efficacité de vos programmes :

- les opérations sur les integers (entiers) étant les plus rapides à traduire, essayez de formuler le plus possible de vos calculs en utilisant ce type de nombres ;
- évitez de stocker temporairement des chaînes de caractères : les fonctions appliquées aux strings consomment beaucoup de temps ;
- plus vous insérez de parenthèses dans une condition IF...THEN, plus vous ralentissez le programme compilé ; les conditions IF...THEN vont très bien à traduire, mais plus vous en imbriquez l'une dans l'autre, plus il faudra de temps pour leur exécution ; lorsque vous imbriquez des boucles les unes dans les autres, tâchez de placer la boucle la plus parcourue au centre de ce montage ;

- de nombreuses opérations de calcul (division, racine carrée etc) exigent le recours aux nombres décimaux, même si l'opération ne concernent au départ que des nombres entiers ; à chaque fois que les chiffres après la virgule ne vous servent à rien, éliminez-les, soit en procédant à la division par le symbole "\ " soit en reconvertissant le résultat obtenu en un nombre entier grâce à CINT.

8.5. Les messages d'erreur envoyés par le compilateur

Bad exit

Le compilateur ne tolère l'utilisation de EXIT que sous la forme EXIT ou EXIT -1 ; pour sortir d'une boucle, vous devez utiliser EXIT TO.

Out of memory

il n'y a plus assez de place dans la mémoire vive ; pour vous sortir de ce mauvais pas, vous pouvez réduire le disque RAM ou découper le programme en morceaux que vous rechargerez l'un à la suite de l'autre.

Structure too long

vous venez d'écrire une structure trop longue, c'est-à-dire dépassant 32 Ko compilée, dans une condition IF...THEN ou une boucle ; essayez de passer une partie de la boucle dans une routine.

Too many variables

l'ensemble des variables (en dehors des tableaux arrays) ne doit pas occuper plus de 64 Ko en mémoire vive.

Type mismatch

vous avez transmis un type de variable erroné à une fonction ou une procédure.

Undefined statement(s) or DIM

Causes possibles :

- ① les branchements renvoient à des lignes qui n'existent plus
- ② vous avez oublié de dimensionner un tableau

Warning : RETURN type mismatch

rappelez-vous qu'une fonction s'étendant sur plusieurs lignes doit renvoyer un type de variable identique à celui qui figure dans son nom ; ça n'est apparemment pas ici le cas !

Warning : unused statement(s)

le compilateur a rencontré des définitions d'étiquettes (labels) ou des procédures qui ne sont pas utilisées par la suite ; détruisez-les pour gagner de la place en mémoire vive.

8.6. Les types de variables retournés par les fonctions

La liste ci-dessous vous indique les types de variables retournés par les différentes fonctions ; consultez ce tableau lorsque vous recevez le message 'type mismatch error' :

Integer :

AND, ASC, BIT, CINT, CINTL, CRSLIN, CVI, CVIL, EOS, EQV, ERL, ERR, FRE, HIGH, IMP, INSTR, LEN, LOC, LOF, LOW, LPEEK, LPOS, MEMORY, MOUSEBUT, MOUSEX, MOUSEY, NAND, NOR, NOT, OR, PEEK, POINT, POS, SEGPtr, SGN, SHL, SHR, TIMER, USR, VARPTR, WPEEK, XOR, =, >, >=, <=, <>

Float (Single et Double) :

ARCCOS, ARCCOT, ARCCOTH, ARCSIN, ARCSINH, ATN, COS, COSEC, COSH, COT, COTH, DET, EXP, FACT, LN, LOG, SEC, SECH, SIN, SINH, SQR, TAN, TANH, ^, /

Single-Float :

CSNG, CVS, RND

Double-Float :

CDBL, CVS, PI, VAL

String :

BINS, CHR\$, DATE\$, ERR\$, HEX\$, INKEY\$, INPUT\$, LEFT\$, LOWER\$, MID\$, MIRRORS, MKD\$, MKIL\$, MKSS\$, OCT\$, RIGHT\$, SPACE\$, SPC, STR\$, STRING\$, TIMES\$, UPPER\$, @

En fonction du type d'argument :

ABS, FIX, FRAC, INT, MAX, MIN, MOD, +, -, *, \, 0, +1, -1, *2

8.7. Les autres programmes utilitaires sur la disquette du compilateur

☐ CUTLIB.PRG ;

Le programme CUTLIB.PRG sert à réduire la bibliothèque BASLIB et à l'ajouter à la fin du programme, ce qui le rend totalement autonome.

Une fois chargé, CUTLIB.PRG vous envoie une boîte de sélection d'objet, dans laquelle vous entrez le nom du fichier de programme pour lequel vous souhaitez réduire BASLIB ; vous pouvez aussi choisir entre les options suivantes :

- autoriser CTRL-C (oui/non)
- saisie de caractères sous Omikron Basic (oui/non)
- sortie de caractères sous Omikron Basic (oui/non)

Si vous répondez par 'oui' aux deux dernières questions, vous ralentissez le programme mais vous bénéficiez de toutes les possibilités d'entrée et de sortie de données offertes par Omikron Basic. Les routines nécessaires au programme sont de toute façon intégrées dans le texte. CUTLIB est aussi exécutable en mode batch, en respectant la syntaxe :

CUTLIB Nom[,Nom.Nom][-Réponse]

sous "Nom", vous indiquez le programme à traiter ; si vous voulez y adjoindre d'autres morceaux de programmes, indiquez leur nom en les faisant précéder d'une virgule. Sous "Réponse" vous répondez aux trois questions posées par CUTLIB : un 'O' majuscule pour 'Oui' et un 'N' majuscule pour 'Non' ; vous pouvez aussi entrer 999, pour insérer toute la bibliothèque BASLIB dans votre programme.

❑ SHELL.PRg ;

Il s'agit d'un interpréteur orienté texte, qui comprend les instructions suivantes :

Instruction	Effet
DIR	Affichage du répertoire
CHDIR Path	Changement de répertoire
MKDIR Path	Création d'un nouveau dossier
RMDIR Path	Destruction d'un dossier
COPY Source Cible	Copie d'un fichier, de 'Source' dans 'Cible'
REN, Ancien Nouveau	Renommer un fichier
DEL, Nom	Destruction du fichier
DATE, Date	Mise à jour de la date
TIME, Heure	Mise à l'heure
PAUSE	Attendre l'appui sur une touche
REM	Remarque (REMark)

Instruction	Effet
PROMPT Chaîne de caractères \$p \$d \$g \$n	Déterminer le prompt : affichage du chemin d'accès actuel affichage de la date >-Caractères Unité de disque actuelle
TYPE Nom	Afficher le contenu d'un fichier à l'écran
VER	Retourne le numéro de la version du TOS
Nom_du_programme [Paramètre]	Permet de lancer le programme
Batchname [Paramètre]	Permet de lancer un fichier BATCH
ECHO ON	Activer affichage écran avec fichier batch
ECHO OFF	Désactiver affichage écran avec fichier batch
EXIT	Sortir de l'interpréteur

Il ne faut pas préciser les types (extensions) des noms de fichier du genre .BAT ou .PRG.

Annexe A

Tableau des codes ASCII

Le tableau ci-après vous donne le code ASCII de chaque caractère, en notation décimale et hexadécimale :

Valeur décimale	Valeur hexadécimale	Atari
0	00	
1	01	␣
2	02	␣
3	03	␣
4	04	␣
5	05	␣
6	06	␣
7	07	␣
8	08	✓
9	09	␣
10	0A	␣
11	0B	␣
12	0C	␣
13	0D	␣
14	0E	␣
15	0F	␣

Valeur décimale	Valeur hexadécimale	Atari
16	10	0
17	11	1
18	12	2
19	13	3
20	14	4
21	15	5
22	16	6
23	17	7
24	18	8
25	19	9
26	1A	a
27	1B	b
28	1C	c
29	1D	d
30	1E	e
31	1F	f
32	20	
33	21	!
34	22	"
35	23	#
36	24	\$
37	25	%
38	26	&
39	27	'
40	28	(
41	29)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0

Valeur décimale	Valeur hexadécimale	Atari
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	@
65	41	A
66	42	B
67	43	C
68	44	D
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

Valeur décimale	Valeur hexadécimale	Atari
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	[
92	5C	\
93	5D]
94	5E	^
95	5F	_
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	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

Valeur décimale	Valeur hexadécimale	Atari
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	~
127	7F	Δ
128	80	Ç
129	81	Ü
130	82	é
131	83	à
132	84	ä
133	85	å
134	86	ä
135	87	ç
136	88	ê
137	89	ë
138	8A	è
139	8B	ï
140	8C	î
141	8D	í
142	8E	ñ
143	8F	ñ
144	90	É
145	91	æ
146	92	Æ
147	93	ô

Valeur décimale	Valeur hexadécimale	Atari
148	94	ø
149	95	ò
150	96	ô
151	97	ù
152	98	ÿ
153	99	ö
154	9A	ü
155	9B	ç
156	9C	£
157	9D	¥
158	9E	ß
159	9F	f
160	A0	à
161	A1	í
162	A2	ó
163	A3	ú
164	A4	ñ
165	A5	ñ
166	A6	ä
167	A7	o
168	A8	¿
169	A9	¬
170	AA	¬
171	AB	½
172	AC	¼
173	AD	¡
174	AE	«
175	AF	»
176	B0	ä
177	B1	ö
178	B2	ø
179	B3	ø
180	B4	œ

Valeur décimale	Valeur hexadécimale	Atari
181	B5	€
182	B6	À
183	B7	Â
184	B8	Ö
185	B9	.
186	BA	.
187	BB	.
188	BC	9
189	BD	@
190	BE	@
191	BF	2
192	C0	ij
193	C1	ij
194	C2	x
195	C3	1
196	C4	λ
197	C5	T
198	C6	η
199	C7	l
200	C8	7
201	C9	η
202	CA	u
203	CB	,
204	CC	3
205	CD	7
206	CE	η
207	CF	3
208	D0	0
209	D1	u
210	D2	9
211	D3	z
212	D4	η
213	D5	7

Valeur décimale	Valeur hexadécimale	Atari
214	D6	Ⓜ
215	D7	Ⓝ
216	D8	Ⓛ
217	D9	Ⓜ
218	DA	Ⓞ
219	DB	Ⓟ
220	DC	Ⓠ
221	DD	Ⓡ
222	DE	Ⓢ
223	DF	Ⓣ
224	E0	Ⓤ
225	E1	Ⓥ
226	E2	Ⓦ
227	E3	Ⓧ
228	E4	Ⓨ
229	E5	Ⓩ
230	E6	ⓐ
231	E7	ⓑ
232	E8	ⓒ
233	E9	ⓓ
234	EA	ⓔ
235	EB	ⓖ
236	EC	ⓗ
237	ED	Ⓣ
238	EE	Ⓤ
239	EF	Ⓥ
240	F0	Ⓦ
241	F1	Ⓧ
242	F2	Ⓨ
243	F3	Ⓩ
244	F4	ⓐ
245	F5	ⓑ
246	F6	ⓒ

Valeur décimale	Valeur hexadécimale	Atari
247	F7	≡
248	F8	◊
249	F9	•
250	FA	•
251	FB	√
252	FC	∩
253	FD	∩
254	FE	∩
255	FF	—

Annexe C

Liste des fonctions du standard VT-52

Dans les lignes ci-dessous, nous avons remplacé CHR\$(27) par ESC :

Fonction	Effet
ESC A	Curseur une ligne vers le haut
ESC B	curseur une ligne plus bas
ESC C	Curseur une colonne vers la droite
ESC D	Curseur une colonne vers la gauche
ESC E	Vider l'écran (CLS)
ESC H	Curseur dans le coin supérieur gauche de l'écran
ESC I	Curseur haut
ESC J	Effacer le reste de l'écran
ESC K	Effacer le reste de la ligne
ESC L	Insérer une ligne vide
ESC M	Effacer une ligne
ESC Y c l	PRINT @(Colonne,Ligne) c = CHR\$(Colonne+32) l = CHR\$(Ligne+32)
ESC b col	Choix de la couleur de l'écriture <col>

Fonction	Effet
ESC c col	Choix de l'arrière plan <col>
ESC d	Effacer l'écran
ESC e	Activer le curseur
ESC f	Désactiver le curseur
ESC j	Sauvegarder la position du curseur
ESC k	Placer le curseur à la position sauvegardée
ESC l	Effacer une ligne
ESC o	Effacer une ligne
ESC p	Passer en inversion vidéo
ESC q	Revenir de l'inversion-vidéo à l'affichage normal
ESC v	Activer le suivi de ligne automatique du curseur, le curseur saute automatiquement à la colonne 0 de la ligne suivante lorsqu'il franchit la colonne 79 d'une ligne.
ESC w	Désactiver le suivi de ligne automatique du curseur

Annexe D

Messages d'erreur envoyés par l'interpréteur

1 *Structure trop longue (Structure too long)*

Il y a plus de 64 Ko de programme entre deux mots d'une même structure (FORNEXT, WHILE...WEND, REPEAT UNTIL) ; veuillez faire passer une partie de ce contenu dans un sous-programme.

2 *Erreur de syntaxe (Syntax Error)*

Vous venez d'entrer une instruction que l'Omikron Basic ne peut absolument pas exploiter : ne s'agit-il pas d'une faute de dactylographie ?

3 *RETURN sans GOSUB (RETURN without GOSUB)*

L'interpréteur vient de rencontrer un RETURN sans s'être branché auparavant sur une routine par un GOSUB.

4 *Plus de DATA (Out of DATA)*

Le pointeur DATA renvoie à une position située au-delà du dernier élément, si bien que l'instruction READ ne peut lire de données ; seul remède : corriger le nombre des DATA.

- 5 ***Appel illégal de fonction (Illégal fonction call)***
Vous utilisez une instruction ou une fonction d'une façon qui est incorrecte.
- 6 ***Dépassement (Overflow)***
Vous avez dépassé le champ de calcul couvert par tel ou tel type de variable numérique.
- 7 ***Plus de mémoire (Out of memory)***
Soit il n'y a plus de place disponible en mémoire vive pour les variables à venir, soit il n'y a plus de place sur la pile du processeur (Stack).
- 8 ***Commande inconnue (Undefined Statement)***
La ligne ou l'étiquette de branchement n'existe pas.
- 9 ***Accès hors limite (Subscript out of range)***
Vous tentez d'accéder à une variable dont l'indice est supérieur aux limites du tableau telles qu'elles sont indiquées dans DIM.
- 10 ***Définition multiple (Duplicata définition)***
Vous tentez de donner le même nom à deux procédures différentes.
- 11 ***Division par zéro (Division by zéro)***
Vous avez programmé une division par zéro !
- 12 ***Accès direct illégal (Illégal direct)***
Cette instruction n'est pas exécutable en mode direct.
- 13 ***Confusion de type (Type mismatch)***
Vous passez à une variable un contenu incompatible avec son type.

-
- 14** *RETURN sans fonction (RETURN without fonction)*
L'interpréteur a rencontré un RETURN sans qu'une fonction ait été appelée auparavant.
- 15** *Chaîne trop longue (String too long)*
Une chaîne de caractères ne peut contenir plus de 32766 caractères.
- 16** *Formule trop complexe (Formula too complex)*
Vous avez imbriqué trop de calculs les uns dans les autres dans une même formule (franchement, vous exagérez !), ce qui dépasse les capacités de la pile (stack).
- 17** *Impossible de continuer (Can't continue)*
Impossible d'exécuter CONT.
- 18** *Fonction utilisateur non définie (Undefined user fonction)*
Aucune fonction ne répond à ce nom.
- 19** *Pas de RESUME (No RESUME)*
Il manque un RESUME dans une routine d'erreur (ON ERROR).
- 20** *RESUME sans erreur (RESUME without error)*
L'interpréteur devrait exécuter un RESUME, sans pour autant s'être branché sur la routine de traitement d'erreur par un ON ERROR.
- 21** *Utilisez EXIT (Use EXIT)*
Vous ne pouvez sortir d'une boucle que par EXIT.
- 22** *Opérande manquant (Missing operand)*
Vous avez oublié d'entrer une opérande.

- 23 Ligne pleine (Line buffer overflow)**
Vous avez écrit une ligne de programme trop longue (en saisie : maximum 255 caractères, en affichage 512 caractères maxi).
- 24 REPEAT sans UNTIL (REPEAT without UNTIL)**
Vous avez ouvert une boucle par un REPEAT sans la faire suivre par UNTIL.
- 25 UNTIL sans REPEAT (UNTIL without REPEAT)**
Présence d'un UNTIL non précédé de son REPEAT.
- 26 FOR sans NEXT (FOR without NEXT)**
Vous avez ouvert une boucle par un FOR, sans la faire suivre d'un NEXT.
- 27 NEXT sans FOR (NEXT without FOR)**
Présence d'un NEXT non précédé de son FOR.
- 28 IF sans THEN ou ENDIF (IF without THEN or ENDIF)**
Soit vous avez oublié un THEN derrière un IF, soit il vous manque un ENDIF (à moins qu'il n'y en ait un de trop !).
- 29 WHILE sans WEND (WHILE without WEND)**
Présence d'un WHILE non suivi de son WEND.
- 30 WEND sans WHILE (WEND without WHILE)**
Présence d'un WEND non précédé de son WHILE.
- 31 THEN, ELSE ou ENDIF sans IF ou THEN (THEN, ELSE or ENDIF without IF or THEN)**
Soit il manque un IF devant THEN, ELSE ou ENDIF, soit il manque un THEN accompagnant un ELSE ou un ENDIF.

- 33 *Reset (Reset)***
Vous avez appuyé sur le bouton de réinitialisation RESET ; le contenu des variables est perdu, mais le programme est toujours en mémoire.
- 34 *Erreur Bus (Bus Error)***
Un programme écrit en langage machine et appelé par CALL ou USR vient de provoquer une erreur de BUS.
- 35 *Erreur d'adressage (Adress error)***
Un programme en langage machine vient de provoquer une erreur d'adressage.
- 36 *Opcode inconnu (Unknown opcode)***
Un programme appelé par CALL ou USR contient une instruction machine inconnue.
- 45 *EXIT sans structure (EXIT without Structure)***
L'interpréteur rencontre un EXIT, mais il n'existe aucun sous-programme ni boucle dont il puisse ressortir !
- 46 *Utilisez EXIT TO dans les fonctions (USE EXIT TO in functions)***
Seul EXIT TO est autorisé dans les fonctions de plusieurs lignes.
- 47 *Matrice irrégulière (Not regular Matrix)***
La matrice en question ne peut être inversée (contre épreuve : lorsque le déterminant est nul => pas d'inverse)
- 50 *Dépassement FIELD (Field overflow)***
Trop de données pour le FIELD concerné.
- 52 *Mauvais numéro de fichier (Bad file number)***
Les numéros des canaux doivent être compris entre 1 et 16.

53 *Fichier introuvable (File not found)*

Le fichier recherché n'existe pas, tout au moins sur la disquette (le disque) ou dans le répertoire actuel.

54 *Mauvais mode d'accès (Bad file mode)*

Vous demandez l'exécution d'une opération illégale sur le fichier.

55 *Fichier déjà ouvert (File already open)*

Vous avez déjà ouvert par ailleurs le fichier en question ; solution : utiliser un autre numéro de canal ou commencer par fermer le fichier ouvert.

56 *Fichier fermé (File not open)*

Vous avez oublié d'ouvrir le fichier auquel vous tentez d'accéder.

57 *Erreur TOS #XX (TOS error #XX)*

Le système vous avertit d'une erreur TOS (voir annexe E).

61 *Disque plein (Disk full)*

Il n'y a plus de place disponible sur le disque ou la disquette.

62 *INPUT après fin (Input past end)*

Dans un fichier séquentiel, le programme tente de lire des données se trouvant au-delà de la fin de fichier ; demandez la position de la fin de fichier par EOF.

63 *Mauvais numéro d'enregistrement (Bad record number)*

Dans un fichier relatif, vous essayez d'accéder à un enregistrement qui n'existe pas.

64 *Mauvais nom de fichier (Bad file name)*

Votre nom de fichier contient des caractères interdits : le point, la virgule, le double-point, le point-virgule etc.

65 *Chemin introuvable (path not found)*

Le chemin spécifié n'est pas valide ; vérifiez les noms des répertoires.

66 *Pas de numérotation des lignes (Direct statement in file)*

Les lignes du programme que vous souhaitez charger ne sont pas numérotées ; solution : utiliser LOAD BLOCK *.*

67 *Trop de fichiers (Too many files)*

Le répertoire n'accepte plus de nom de fichier supplémentaire ; solution : détruire des fichiers inutiles (KILL), ou enregistrer sur une autre disquette.

64. Les données de base (base de données) sont les données de base de la base de données. Elles sont les données de base de la base de données.
65. Les données de base (base de données) sont les données de base de la base de données. Elles sont les données de base de la base de données.
66. Les données de base (base de données) sont les données de base de la base de données. Elles sont les données de base de la base de données.

Annexe E

Les messages d'erreur envoyés par le TOS

Numéro d'erreur	Signification
1	Erreur générale
2	Lecteur de disquette pas prêt
3	Instruction inconnue
4	Erreur de checksum
5	Exécution impossible (Instruction illégale)
6	Secteur introuvable
7	Boot-sector défectueux
8	Secteur non trouvé
9	Plus de papier (Paper out)
10	Erreur d'écriture
11	Erreur de lecture
12	Erreur générale
13	Disquette protégée
14	Changement de disquette
15	Périphérique inconnu
16	Erreur de vérification

Numéro d'erreur	Signification
17	Pas de disquette dans le lecteur
32	Numéro de fonction non-valide
33	Fichier non trouvé
34	Chemin d'accès non trouvé
35	Trop de fichiers ouverts
36	Accès impossible
37	Numéro de fichier non-valide
39	Plus assez de mémoire
40	Aucun bloc mémoire à cette adresse
46	Identificateur du lecteur non-valide
49	Plus d'autres fichiers

Annexe F

Conversion des listings en GFA Basic

Commencez par sauvegarder votre programme écrit en GFA Basic sous forme de fichier ASCII ; chargez ensuite l'interpréteur Omikron Basic puis le programme en question, à l'aide de `LOAD BLOCK *.*` et non de `LOAD` car vous recevriez un message d'erreur. Procédez ligne par ligne, et modifiez les instructions différentes en GFA Basic pour qu'elles respectent la syntaxe de l'Omikron Basic.

N'oubliez pas qu'en GFABasic, les variables sont d'emblée considérées comme des variables `FLOAT` s'il n'y a pas d'indication contraire, alors que l'Omikron Basic les considère comme des long-integers ; vous pouvez vous en tirer en entrant l'instruction `DEFSNG A-Z` dans la première ligne de votre programme.

Pour la conversion de vos programmes, je vous recommande vivement de vous procurer le livre du GFA Basic publié chez MICRO APPLICATION, guide qui vous permettra de repérer rapidement la syntaxe propre à telle ou telle instruction.

Index

A

Accessory	1-29
AES	4-252
Alertbox	6-310
AND	2-162
Arborescence	6-304
ASC	2-70

B

BACKUP	3-214
BIN\$	2-92
BIOS	4-250, 4-263
Bit	2-42
BITBLT	5-295
BLOAD	5-298
Bloc	1-32
Block	1-23
Boîte de sélection d'objet	1-18, 3-200
BOX	5-293, 6-311
Boxchar	6-312
Boxtext	6-312
BSAVE	5-298
Button	6-311
Byte	2-42

C

Canal	3-189
Catalogue	1-20
Chaîne de caractères	2-45, 2-68
Change size	1-26
Char	2-45
Charger un bloc	1-19, 1-24
CHDIR	3-218
CHECKED	6-326
CHR\$	2-70
CLEAR	2-100
CLOSE	3-193
CMD	3-242
Combinaison de touches	1-30
Comparaison	2-105
Compilateur	1-29, 8-381
COMPILER.PRG	8-382
Concaténation	2-72
COPY	3-196, 3-214
CROSSED	6-326
CSRLIN	4-250
CUTLIB.PRG	8-391
CVD	3-223
CVI	3-223, 6-350
CVIL	3-223
CVS	3-223

D

DATA	2-142
DATE\$	4-249
DEF FN	2-138
DEFAULT	6-324
DEFDBL	2-47
DEFINT	2-47
DEFINTL	2-47
DEFSNG	2-47

DEFSTR	2-47
Dialogue	6-310
Directory	1-20
DISABLED	6-327
Division entière	2-54
Drapeaux	2-48
DRAW	5-293
DUMP	2-99

E

EDIT	1-31
EDITABLE	6-325
Editeur pleine page	1-17, 1-31
Emulateur VT-52	1-15, 2-129
ENDIF	2-107
Enregistrement	3-186
Enregistrer	1-18
→ le bloc	1-19, 1-24
Entier Court	2-43
EOF()	3-194
ERL	2-98, 2-101, 3-211
ERR	3-210
ERR\$	2-101, 3-210
ERR, ERL, ERR\$	4-250
ERROR	3-212
Evnt_Mesag	6-349, 6-368
Exec *.Prg	1-29
EXIT	2-121, 6-324

F

Factorielle	2-140
Fboxtext	6-311
Fichier	
→ à accès direct	3-186
→ séquentiel	3-186

FICHER.BAS	2-179
Fichiers ISAM	3-187
FIELD	3-220
File	1-18
FILESELECT	3-204
FILL	5-294
FILL COLOR	5-294
FILL STYLE	5-294
Find	1-20, 1-22
Find error	1-28
Find next	1-22
Find token	1-22
FIX	2-65
Flag	2-48, 2-106, 6-323
FN	2-138
Fonction	2-137
Fonctions mathématiques	2-62
FOR..NEXT	2-114
FORM_ALERT	2-177
FORM_CENTER	6-337
FORM_DIAL	6-338
FORM_DO	6-339
Formatage	4-281
FRAC	2-66
Free	6-309

G

GEM	2-176, 6-303
GEM-AES	4-251
GEM-VDI	4-251
GEMDOS	4-250, 4-253
GET	3-222
GO	1-27, 1-34
GOSUB	2-126
GOTO	2-110
Graphisme	5-291

H

HEX\$	2-92
Hide	1-25
HIDETREE	6-325

I

Ibox	6-311
Icon	6-312
IF...THEN...ELSE	2-104
Image	6-312
Impression	3-240
Imprimer un bloc	1-25
INDIRECT	6-326
INKEY\$	2-151
INPUT	2-59
INPUT #	3-193
INPUT USING	2-155
INPUT\$	3-197
Insert	1-24 - 1-25
Insertion	1-25
INSTR	2-85
INT	2-65
Interpréter	8-381

K

KEY	2-96
KEY LIST	2-97
Kill	1-24, 3-213

L

Lancer un programme	1-28
LASTOB	6-325
LDUMP	2-100
LEFT\$	2-77
LEN	2-80
LET	2-39
Liaison des lignes	1-15
LINE COLOR	5-293
LINE INPUT	2-61
LINE INPUT#	3-193
Line numbers	1-26
LINE STYLE	5-293
Line to bottom	1-28
Line to top	1-27
List	1-22, 2-97
List to printer	1-23
List token	1-23
Lister	1-22
LLIST	3-242
LOAD	2-57
Load **	1-19
Load block **	1-20, 1-24
LOF	3-199
LOWER\$	2-84
LPEEK	6-322
LPOKE	6-322
LPRINT	3-240
LSET	3-221

M

MA	8-384
Mark block end	1-23 - 1-24
Mark block start	1-23 - 1-24
MBS	8-384
MEMORY	5-299

Menu	6-310
Menu_Bar	6-349
Messages d'erreur	8-389
MID\$	2-78
MIRROR\$	2-82
MKD\$	3-222
MKDIR	3-217
MKI\$	3-222
MKIL\$	3-222
MKS\$	3-222
MOD	2-54
Mode	1-25, 2-84
MODE	5-293
Mode	
☞ d'affichage écran	1-25
☞ direct	1-17
☞ insertion	1-25
MODE LPRINT	3-241
MODULO	2-54
Mot	2-43
Mots-clés	1-20
MOUSEBUT	4-246
MOUSEOFF	2-179
MOUSEON	2-179
MOUSEX	4-246
MOUSEY	4-246
Move	1-24
Multitâches	7-377

N

NAME ... AS	3-213
NEW	1-20, 2-57
Nombres entiers	2-41
NON logique	2-165
NOT	2-165
Nouveau	1-20
Numérotation des lignes	1-26, 1-35

O

Objc_Draw	6-338
Object	6-326
OCTS	2-93
OMIKRON.INF	1-26
ON HELP GOSUB	7-379
ON KEY GOSUB	7-378
ON MOUSEBUT GOSUB	7-379
ON...GOSUB	2-126
ON...GOTO.....	2-113
ON...RESTORE	2-144
Opérateur	
⇨ d'attribution	2-38
⇨ d'arithmétique	2-53
OR	2-164
OU exclusif	2-166
OU logique	2-164
OUTLINED	6-327

P

Partager l'écran en deux parties	1-25
PBOX	5-294
PEEK	6-321
PI	4-249
POKE	6-321
PRINT	2-37
PRINT AT	2-149
Print block	1-25
PRINT USING	2-167
PRINT#	3-191
Procédure	2-127
Programmation structurée	2-102
PUT	3-222

Q

Query replace	1-22
Quit Edit	1-20

R

RADIO-BUTTON	6-325
RBOX	5-294
RCS	6-304
READ	2-142
Recherche	1-20, 1-22, 1-33
Rechercher une erreur	1-28
Record	3-186
Récurtivité	2-131
Remplacement	1-33
Remplacer	1-22
Rename Token	1-23
RENUM	2-102
RENUMBER	2-101
Renommer	2-101
Repeat	1-35
REPEAT...UNTIL	2-119
Répertoire	1-20
Répéter	1-35
Replace all	1-22
Résolution graphique	5-292
Ressource	6-305
RESTORE	2-142
RESUME	3-212
RIGHT\$	2-77
RMDIR	3-217
Routine	2-124
Rscr_Gaddr	6-337
Rsrc_Load	6-336
RSET	3-221

RUN	2-58
RUN	1-28
Run *.Bas	1-29

S

Sauvegarder	1-18
SAVE	2-55
Save & compile	1-29
Save & Run	1-28
Save **	1-19
Save block **	1-19, 1-24
Save settings	1-26
SCAN	2-158
Scancode	2-158
SELECTABLE	6-324
SELECTED	6-326
Set mark #X	1-28
SHADOWED	6-327
SHELL	8-392
SHL	2-167
Show errors	1-26
SHR	2-166
Sortir de l'éditeur	1-20
SPACE\$	2-81
SPC	2-81
Split screen	1-25
SQR	2-62
Status	6-326
STEP	2-118
Str\$	2-75
String	2-45, 2-68, 6-311
STRING\$	2-82
Super-éditeur	1-17
SWAP	2-134
Switsch screen	1-25
SYSTEM	1-36

Système	
↻ binaire	2-42, 2-92
↻ d'exploitation	4-245
↻ décimal	2-42
↻ hexadécimal	2-91
↻ octal	2-91

T

Tableau	2-94
Taille de caractères	1-25
te_color	6-330
te_font	6-329
te_just	6-329
te_ptext	6-328
te_ptmplt	6-328
te_pvalid	6-329
te_thickness	6-331
te_tmplen	6-331
te_txtlen	6-331
TEDINFO	6-327
Text	6-311
THEN..ELSE	2-104
TIME\$	4-248
TIMER	4-248
Title	6-312
To last mark	1-27
To line	1-27
To mark #X	1-28
Token	1-20
TOS	4-250
Touche de fonction	2-96
TOUCHEXIT	6-325
TRACE	1-28
TRACE_OFF	8-384
TRACE_ON	8-384
TROFF	2-101

TRON	2-100
Tron & Run	1-28
Types de variables	2-41

U

UPPER\$	2-83
USING	2-156, 2-170

V

VAL	2-75, 2-93
Variable	2-38
☞ globale	2-129
☞ locale	2-129
☞ 'integer'	2-41
☞ booléenne	2-48
VDI	4-251

W

WHILE...WEND	2-122
Wind_Create	6-369
Wind_Get	6-371
Wind_Set	6-372
Word	2-43
WPEEK	6-322
WPOKE	6-322
WRITE#	3-192

X

XBIOS	4-250
XOR	2-166

Dans la même collection...

Micro Application



Réf.	Titre	Prix T.T.C.
ML156	Bien débiter avec l'ATARI ST et STE	129.00
ML527	Bien débiter en GFA BASIC 2.0 et 3.0	129.00
ML561	Bien débiter Le Rédacteur	129.00
ML717	Bien débiter STOS	129.00
ML631	Boîte à Outils ST	disquette incluse 299.00
ML688	Dévelop. sous Superbase PRO/PRO III	disquette incluse 299.00
ML172	Disquette et Disque Dur	179.00
ML272	Disquette et Disque Dur	disquette incluse 279.00
GL102S	Guide SOS GFA BASIC 2.0 à 3.0	99.00
ML530	Le Grand Livre de l'ATARI ST	199.00
ML530 OS	Pack Le Grd Liv. de l'At. ST + Ad. STE + freeware (2 disquettes)	199.00
ML556	Le Livre de CALAMUS	199.00
ML573	Le Livre de SUPERBASE (versions II, PRO, PRO III)	169.00
ML692	Le Livre des Imprimantes sur ATARI ST	disquette incluse 249.00
ML550	Le Livre du Développeur sur ATARI ST (T1)	299.00
ML589	Le Livre du Développeur sur ATARI ST (T2)	199.00
ML689	Le Livre du Dévelop. sur Atari ST (T2)	2 disquettes incluses 299.00
ML502	Le Livre du Graphisme	199.00
ML602	Le Livre du Graphisme	2 disquettes incluses 299.00
ML141	Le Livre du Langage Machine	149.00
ML193	Le Livre de l'Intelligence Artificielle	179.00
ML616	Le Livre de 1ST WORD PLUS	disquette incluse 299.00
ML185	Le Livre du GFA BASIC 2.0	199.00
ML285	Le Livre du GFA BASIC 2.0	disquette incluse 299.00
ML571	Le Livre du GFA BASIC 3.0 / 3.5	199.00
ML671	Le Livre du GFA BASIC 3.0 / 3.5	disquette incluse 265.00
ML716	L'Histoire de LARRY - Lelure Suit I, II et III	79.00
ML591	Musique, Midi et Séquenceurs	99.00
ML598	TOS 1.4 ET TOS STE	99.00
ML299	Trucs et Astuces en GFA 2.0	disquette incluse 269.00
ML651	Trucs et Astuces ATARI ST	disquette incluse 299.00

ATARI
ST+STE

LE LIVRE OMIKRON[®] BASIC

Michael MAIER

Langage structuré, de haut niveau et possédant un compilateur, l'Omikron Basic tire le meilleur parti des capacités des Atari ST et STE. Pour réussir vos premiers pas, découvrez avec cet ouvrage les éditeurs intégrés, les règles fondamentales de la programmation structurée et les concepts s'y rattachant : variables locales, procédures, fonctions...

Puis, à l'aide des applications proposées, apprenez à réaliser des programmes de qualité professionnelle en utilisant les routines du système d'exploitation ou en créant vos propres fichiers Ressource. Enfin, bénéficiez de nombreux utilitaires, routines et bibliothèques de fonctions très pratiques pour vos futurs programmes.

Principaux sujets traités :

- Installation, découverte des éditeurs, types de variables (entières, booléennes, chaînes de caractères...).
- Éléments de programmation : boucles conditionnelles, test, répétitions...
- Entrée et sortie des données : clavier, écran, souris...
- Création de fichiers séquentiels et à accès direct...
- Programmer avec les routines TOS, GEMDOS, BIOS et XBIOS.
- Programmer le GEM : boîte d'alerte, ascenseurs, menus déroulants...
- Applications prêtes à l'emploi (avec les sources) : programme de gestion d'adresses et de formatage d'une disquette, boîte de sélection de fichiers complète...



9 782868 993403

Réf: ML 728 / Prix : 165 F.
ISBN: 2-86899-340-0 / ISSN: 0980-1928

EDITIONS MICRO APPLICATION

58 RUE DU FAUBOURG POISSONNIERE
75010 PARIS TEL (1) 47 70 32 44