MASTER EN SCIENCES INFORMATIQUES

Analyse des structures du système d'exploitation U.C.S.D.

Lahaye, Michel

Award date:
1982

Awarding institution:
Universite de Namur

ANALYSE DES STRUCTURES DU

SYSTEME D'EXPLOITATION

U. C. S. D. PASCAL.

Mémoire présenté par

Michel Lahaye

en vue de l'obtention
du titre de

Licencié et Maître en Informatique.

Année académique 1981-1982

Remerciements.

Qu'il nous soit permis de remercier tout particulièrement Monsieur le Professeur Ramaekers qui nous a fait bénéficier de ses connaissances et qui nous a témoigné beaucoup d'intérêt et d'attention, tant pendant les recherches préliminaires que pendant la rédaction de ce mémoire.

Que Monsieur R. Verhaeghe trouve également ici l'expression de nos plus vifs remerciements.

Table des matières.

Remerciements.

Introduction. Page 1

Chapitre 1. Présentation du système d'exploitation. 4
---

1. Introduction. 4

2. Un bref historique des systèmes d'exploitation pour micro-ordinateurs. 6

3. Caractéristiques du système d'exploitation étudié. 8

4. Présentation du jeu d'instructions (CODE-P). 14

5. Les réalisations programmées. 2Ø

6. Les réalisations microprogrammées. 28

7. Présentation des utilitaires du système. 32

Chapitre 4. La gestion des fichiers sous UCSD-Pascal. Page 98
---

1. Introduction. 98

2. Organisation d'une disquette. 100

3. Utilisation d'une mini-disquette par UCSD Pascal. 101

4. Les différents types de fichiers. 110

5. Les primitives d'accès. 114

6. L'accès au catalogue. 120

7. La gestion des tampons. 123

8. Les protections prévues contre les erreurs. 126

Chapitre 5. Vérification de la portabilité des programmes. 128
---

1. La portabilité au niveau du langage source. 128

2. La portabilité au niveau du code objet. 129

Conclusions. 131

INTRODUCTION.

Le microprocesseur, fruit de l'évolution technologique, constitue actuellement le composant principal d'un nombre croissant de réalisations faisant appel à l'électronique.

D'abord utilisé en remplacement de composants existants (remplacement de circuits en logique câblée par une logique programmée), ce composant a permis une amélioration sensible des performances tout en abaissant les coûts de production et de maintenance.

On a vu ensuite apparaître de " petits ordinateurs " dont l'unité centrale était composée d'un microprocesseur (micro-ordinateur) .
Ceux-ci ont évolué rapidement vers des machines relativement performantes, dotées d'une mémoire centrale de taille importante et gérant un grand nombre de périphériques (imprimantes , disquettes, ...) .
Pour faciliter la tâche du programmeur, un ensemble de programmes est mis à sa disposition. Ces programmes (système d'exploitation et quelques utilitaires) permettent une utilisation optimale de toutes les ressources du micro-ordinateur tout en dégageant le programmeur de toute connaissance concernant la machine physique.

A l'origine, presque tous les constructeurs de micro-ordinateurs (et ils sont nombreux) proposaient un système d'exploitation spécifique. Cela ne manquait pas de provoquer des difficultés lors de l'adaptation sur une machine d'un programme écrit pour un autre type de matériel.
Ces difficultés existaient, dans une moindre mesure, il est vrai, lorsque les microprocesseurs des deux micro-ordinateurs étaient identiques.

Quelques firmes, indépendantes de tout constructeur, ont alors proposé des systèmes d'exploitation adaptables sur des machines de différentes marques. Nous pensons surtout ici à CP/M, qui parmi les systèmes proposés, est largement le plus populaire, mais d'autres systèmes d'exploitation de ce type existent.
Une certaine normalisation de fait a donc été observée, pour le plus grand bien des utilisateurs qui se sont vu, de la sorte, offrir un choix infiniment plus grand de programmes.

Ce mémoire a pour but l'analyse d'un système d'exploitation pour micro-ordinateurs, relativement répandu et possédant quelques caractéristiques originales.

Le système d'exploitation UCSD Pascal utilise une machine virtuelle dont le langage machine est un Pseudo-code créé de toutes pièces.
La machine virtuelle est étudiée dans le premier chapitre de ce travail.

L'utilisation d'une machine virtuelle permet une gestion efficace de la mémoire centrale.
Le chapitre 2 décrit le système de gestion de la mémoire.

Le chapitre 3 étudie la gestion des entrées-sorties et la gestion des fichiers est analysée dans le chapitre 4.

Enfin le chapitre 5 analyse certains essais réalisés en vue de vérifier la portabilité des programmes , d'une machine à l'autre et d'une version du système d'exploitation à l'autre.

Chapitre 1.

Présentation du système d'exploitation.

1 . Introduction.

Depuis la naissance de l'informatique, on assiste à une multiplication des fonctions des systèmes d'exploitation (d'abord travail en monoprogrammation avec gestion de la prise en charge des travaux, ensuite travail en multiprogrammation, soit en traitement par lot, soit en temps partagé).

Ces perfectionnements avaient pour but de diminuer les périodes d'inactivité de certains composants coûteux de la machine, comme le processeur, la mémoire centrale et certains périphériques.

Les progrès technologiques ont permis la fabrication d'un nouveau type de composants : les microprocesseurs.
Ceux-ci ont permis la fabrication d'un nouveau type d'ordinateurs, les micro-ordinateurs. Ceux-ci possèdent une structure semblable à celle des ordinateurs classiques mais les éléments qui les composent sont en général moins performants, moins coûteux et moins encombrants.
Une des conséquences est qu'il n'est dès lors plus nécessaire de partager ni le processeur principal, ni la mémoire centrale entre plusieurs utilisateurs.

En contrepartie, les possibilités d'échange d'informations devront être très étendues. L'évolution actuelle conduit même à la réalisation de réseaux de micro-ordinateurs.

Par rapport aux systèmes d'exploitation classiques, il apparaît qu'un retour vers des concepts moins évolués soit nécessaire pour la réalisation des systèmes d'exploitation de ces micro-ordinateurs.
Il ne s'agit pas d'un retour en arrière, car s'il s'agit de supprimer la multiprogrammation, certaines parties de la gestion de la mémoire et peut-être d'autres possibilités, la gestion des entrées-sorties devra être très soignée de manière à permettre l'utilisation de ces machines en tant que terminaux intelligents ou le raccordement à un réseau.

2. Un bref historique des systèmes d'exploitation pour micro-ordinateurs.

Les premiers micro-ordinateurs [i] n'étaient que de petits systèmes d'évaluation ne possèdant qu'une mémoire centrale de taille très limitée (généralement moins de 4 K. octets).
Les périphériques de communication avec l'utilisateur étaient le plus souvent un clavier (octal ou hexadécimal plus quelques touches de fonctions spéciales) et un ensemble (le plus souvent 6) d'afficheurs 7 segments permettant l'affichage de chiffres hexadécimaux [ii] .

L'accès à ces machines était facilité par la présence d'un petit moniteur (1 ou 2 K. octets) situé en mémoire morte et possèdant quelques fonctions de base telles que :
- Accès à une case mémoire préalablement choisie.
- Lancement de l'exécution d'un programme à une adresse choisie.
- Sauvetage et récupération d'une zone de la mémoire centrale sur cassette magnétique ou sur ruban perforé.

Les facilités de développement de programme étaient le plus souvent très réduites : le programme devait être entièrement écrit en langage machine et la seule possibilité de mise au point consistait en une exécution en PAS à PAS du programme.

---

[i] Exemple : MEK D1 et MEK D2 de Motorola.
SDK 8Ø et SDK 85 de INTEL.

[ii] Deux afficheurs pour les adresses.
Quatre afficheurs pour les données.

Le but de ces cartes d'évaluation, comme leur nom l'indique était de permettre la découverte d'un nouveau type de composants : les microprocesseurs et leurs familles de circuits annexes.

L'étape suivante a été la réalisation de micro-ordinateurs de taille de mémoire centrale plus importante et comportant comme moyen de communication avec l'utilisateur un clavier semblable à celui d'une machine à écrire et un affichage sur écran cathodique.
Ces micro-ordinateurs sont généralement livrés avec un interpréteur BASIC en mémoire morte, quelquefois accompagné par un petit moniteur permettant quelques fonctions de base, telles celles décrites précédemment. La mémoire de masse est le plus souvent constituée de cassettes magnétiques.
La destination de ces appareils étant le grand public, la majorité des programmes développés le sont en langage BASIC.

La présence de cet interpréteur BASIC en mémoire morte facilite grandement la phase d'initialisation du micro-ordinateur, puisqu'il n'y a pas de programme à charger en mémoire centrale, mais constitue un frein important lorsqu'un autre langage doit être utilisé (un problème se pose également lorsqu'une erreur est découverte dans le programme interpréteur, car dans ce cas un changement des mémoires mortes est nécessaire).

Sur ces micro-ordinateurs sont apparues des mémoires de masse à accès direct, constituées de disquettes ou de minidisquettes magnétiques, accompagnées d'un programme de gestion de ces mémoires (DOS : Disk Operating System), mais ce programme de gestion ne permet en général que le développement de programmes écrits en langage BASIC.

La dernière étape a été la réalisation de micro-ordinateurs dont la mémoire était constituée uniquement de mémoire vive, mis à part un tout petit programme situé en mémoire morte et servant de chargeur au système d'exploitation situé sur support magnétique. C'est pour cette catégorie de matériel que les systèmes d'exploitation pour micro-ordinateurs sont les plus nombreux et les plus complets.

3. Caractéristiques du système d'exploitation étudié.

Structure.

Ce mémoire étudie un système d'exploitation pour micro-ordinateur : le système d'exploitation UCSD-Pascal.
Il s'agit d'un système d'exploitation

- Mono-utilisateur.
- Mono-programmé.

Implémentation.

Il est implémenté sur une large gamme de matériel, miniordinateur comme le PDP 11 de DIGITAL EQUIPMENT CORPORATION, ou micro-ordinateur utilisant les microprocesseurs Z8Ø, 8Ø8Ø, 8Ø85, 65Ø2, 68ØØ, 68Ø9, 99ØØ, LSI-11. Ce système d'exploitation fonctionne donc avec des processeurs différents, utilisant des mots de longueur différente (16 ou 8 bits) [i], [34], [35].

---

[i] Z8Ø : microprocesseur 8 bits de ZILOG, INC.
8Ø8Ø, 8Ø85 : microprocesseurs 8 bits de INTEL CORPORATION.
65Ø2 : microprocesseur 8 bits de MOS TECHNOLOGY, INC.
68ØØ : microprocesseur 8 bits de MOTOROLA INCORPORATED.
99ØØ : microprocesseur 16 bits de TEXAS INSTRUMENTS, INC.
LSI 11 : microprocesseur 16 bits de DIGITAL EQUIPMENT CO.

Origine.

Le système d'exploitation UCSD-Pascal a été créé à l'Université de Californie à San Diego (UCSD) sous la direction de Kenneth L. Bowles en 1977.
Pendant quelques années, la distribution et la maintenance des programmes furent assurées par l'Université, mais actuellement, les droits de distribution appartiennent à SOFTECH Microsystems, une filiale de SOFTECH, entreprise de développement de programmes située à San Diego.

Applications visées.

Le système d'exploitation UCSD-Pascal a été conçu pour permettre le développement et l'exécution de programmes écrits en langage évolué sur des petites machines.
La configuration minimale requise est un micro-ordinateur avec visualisation sur écran cathodique ou sur télé-imprimeur, et une mémoire de masse à accès direct d'une taille minimale d'une centaine de K. octets (la mémoire de masse sera le plus souvent constituée d'un ou plusieurs disques souples).
Pour développer des programmes importants, la taille de la mémoire centrale devra être au minimum de 48 k octets, mais pour simplement exécuter des programmes déjà compilés, une taille de mémoire centrale inférieure peut être suffisante.

UCSD-Pascal a, à l'origine, été développé pour l'utilisation exclusive du langage de programmation Pascal, mais depuis sa commercialisation, vu le succès rencontré (30.000 programmes ont été vendus) , des compilateurs pour d'autres langages de programmation ont vu le jour (compilateurs BASIC et FORTRAN).

Le système d'exploitation est constitué de 2 parties principales :

- Un programme principal et un ensemble de procédures (constituant le noyau du système d'exploitation) permettant l'exécution d'un programme et quelques fonctions de base.

- Un ensemble de programmes utilitaires tels que les compilateurs, les assembleurs, le programme de manipulation des fichiers, l'éditeur, etc..

Objectifs poursuivis.

Le but de ce système d'exploitation est de permettre une portabilité totale des programmes, d'un ordinateur à l'autre, même si ces machines utilisent des processeurs différents et même si les périphériques de ces ordinateurs n'ont que peu de points communs.
D'autres systèmes d'exploitation pour micro-ordinateurs offrent également une portabilité totale des programmes (exemple : CP/M), mais ils sont généralement écrits pour un type de processeurs (Exemple : 8Ø8Ø, 8Ø85 et Z8Ø pour CP/M).

L'originalité de ce système réside dans le fait que contrairement aux langages évolués qui réalisent la portabilité des programmes au niveau du langage source, le système UCSD-Pascal réalise cette portabilité au niveau du code objet.
Toutes les fonctions, y compris la gestion des mémoires de masse et des entrées-sorties, sont alors simulées et apparaissent donc identiques d'une machine à l'autre.

Techniques de réalisation.

Le système consiste à créer une machine virtuelle. Vues du niveau supérieur, les machines virtuelles apparaîtront donc toutes semblables, quelles que soient les machines physiques.

Un jeu d'instructions pour une machine imaginaire a été défini. Ce jeu d'instructions a été choisi de manière à simplifier la compilation de programmes écrits en langage évolué. Dans ce mémoire, nous appellerons ce jeu d'instructions, le CODE-P ou Pseudo-code.

Concevoir une machine virtuelle pour une machine physique donnée peut s'envisager comme suit [41] : (du niveau physique le plus bas vers un niveau plus élevé)

1 : Construction du BIOS (Basic Input-Output Subroutines).

La gestion des entrées-sorties est particulière à la machine physique utilisée, mais les fonctions de haut niveau du BIOS doivent être identiques d'une machine à l'autre. Les procédures du BIOS permettent le transfert dans les deux sens, d'un ensemble de bytes, entre la mémoire centrale et les différents périphériques. Les procédures d'entrées-sorties correspondent à des instructions exécutables de la machine virtuelle.

2 : Construction de l'interpréteur de CODE-P.

Ce CODE-P se compose des instructions permettant de gérer facilement des objets de types très différents (variables booléennes, entières, réelles; tableaux et chaînes de caractères; ensembles), permet les appels de procédures et de fonctions et gère de manière efficace le mécanisme de segmentation.
La plupart des instructions sont en fait relativement évoluées et seront dès lors simulées par un petit programme réalisé sur la machine hôte.

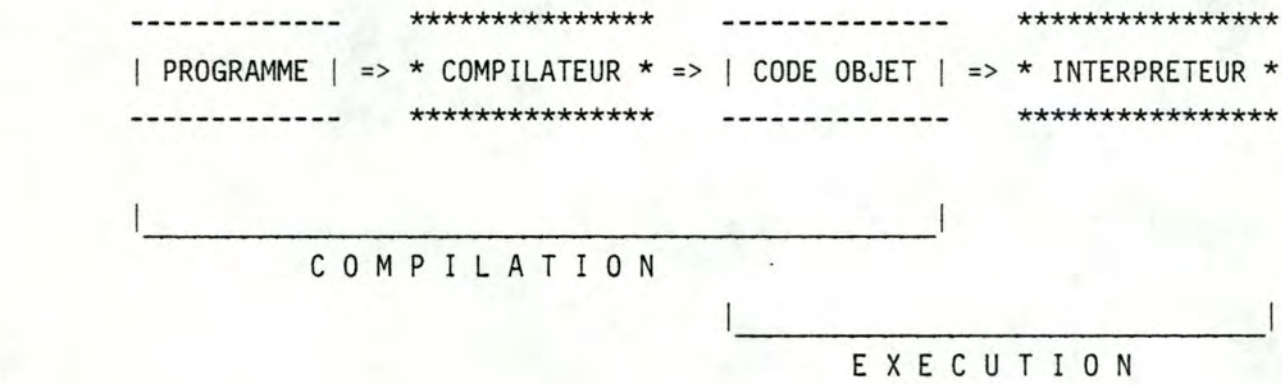
Dans le cadre de cette machine virtuelle, la compilation d'un programme consistera à traduire ce programme dans le CODE-P.
L'exécution du programme consistera en l'interprétation du fichier contenant les instructions du CODE-P générées par le compilateur.

On trouvera page suivante, schématisées, les différentes mises en oeuvre possibles d'un compilateur [i] :

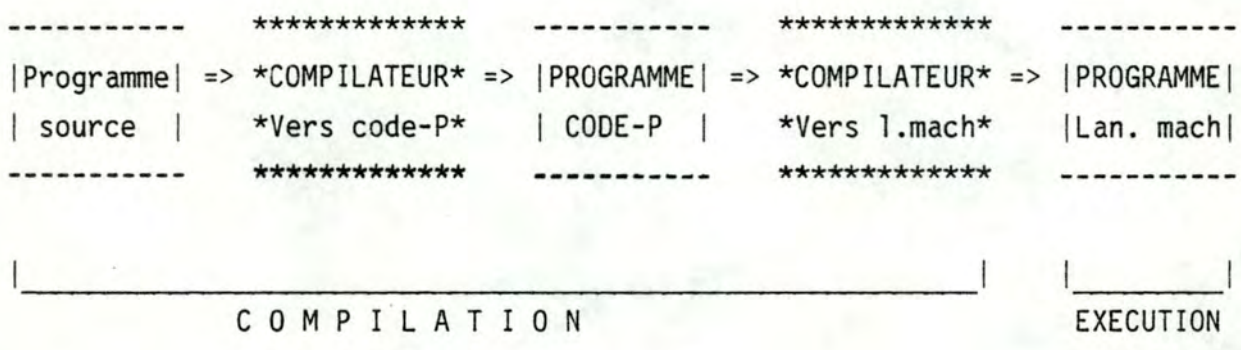
---

[i] Extraits de [24].

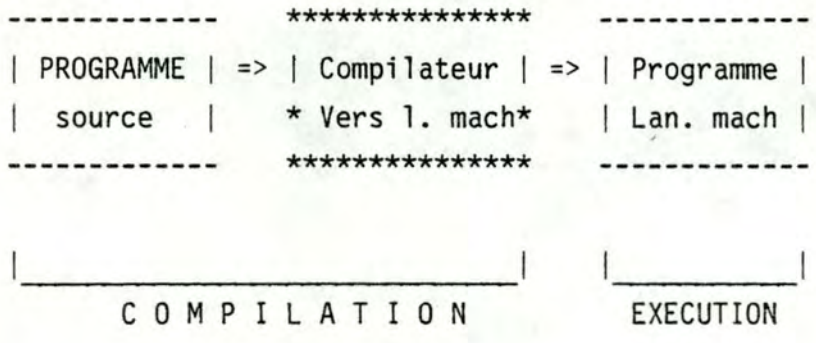
Compilateur + interpréteur de CODE-P.



Compilateur + compilateur de CODE-P.



Compilation directe.

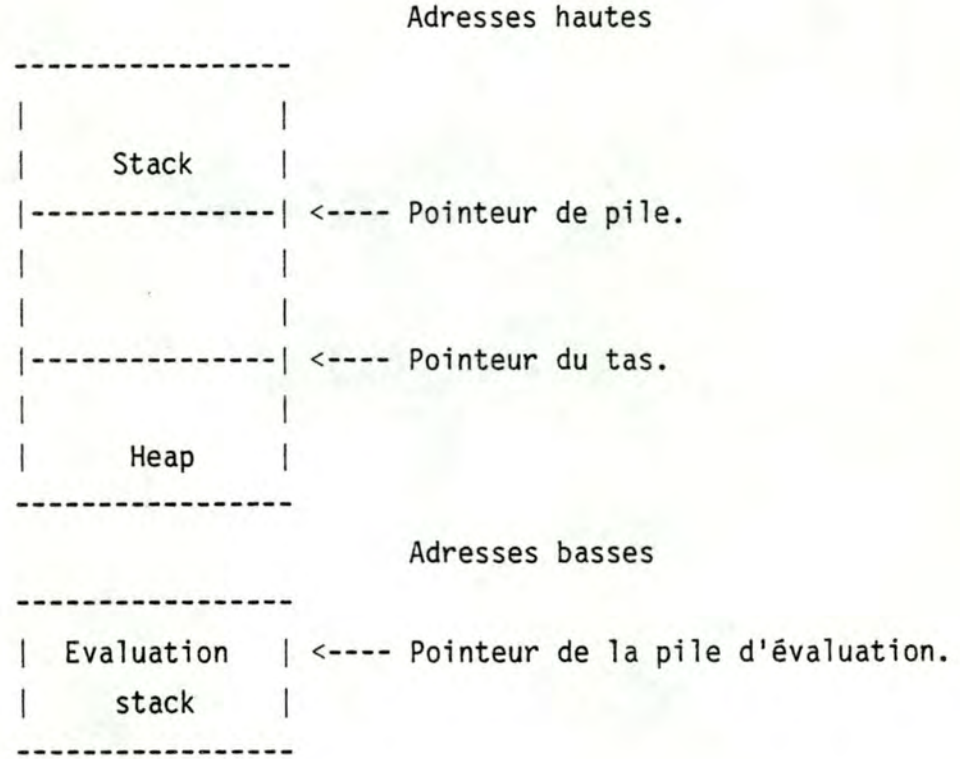


4. Présentation du jeu d'instructions (CODE-P).

Le but n'est pas ici de détailler toutes les instructions du CODE-P mais de décrire succinctement celles-ci et de détailler uniquement les instructions particulièrement importantes pour la compréhension du système d'exploitation.

Présentation de l'utilisation de la mémoire centrale.

La mémoire centrale disponible (la zone occupée par l'interpréteur de CODE-P n'intervient pas ici) de la machine virtuelle est divisée en trois parties, la pile ('stack') , le tas ('heap') et une petite zone réservée à une petite pile d'évaluation des expressions arithmétiques ('evaluation stack').



La pile d'évaluation d'expressions arithmétiques a une taille de 256 bytes et sert également pour le passage des paramètres, lors des appels de procédures (Sa situation physique dépend de contraintes matérielles, notamment du type de microprocesseur utilisé) .

La croissance de la pile s'opère vers les adresses basses, celle du tas vers les adresses hautes. L'espace de la mémoire centrale situé entre le pointeur de pile et le pointeur du tas représente l'espace mémoire disponible à un moment donné. Le fait de placer la pile et le tas aux deux extrémités de l'espace mémoire permet de ne pas devoir, à priori, réserver d'espace mémoire de taille fixe pour ces deux zones.

La pile contient le code objet et les variables statiques. Le tas contient les variables dynamiques (zones pointées).

Les différents types d'instructions.
---

1 : Opérations sur la pile d'évaluation.

a : Chargement d'une constante en sommet de la pile d'évaluation.

b : Opérations arithmétiques en sommet de la pile d'évaluation. Il existe des opérateurs agissant sur un, deux ou plusieurs opérantes. Le résultat est placé en sommet de pile.

2 : Transfert d'une variable.

Le jeu d'instructions a été créé pour exécuter des programmes écrits en langage de programmation Pascal. Cela explique que la terminologie utilisée par le constructeur s'apparente au vocabulaire "Pascal". C'est le cas de la notion de procédure.

Le langage de programmation Pascal autorise l'utilisation de variables locales à une procédure. Pour permettre l'accès à ces variables, le code objet de chaque procédure contiendra un pointeur de données.

a : Chargement du contenu d'une variable au sommet de la pile d'évaluation (accès relatif par rapport au pointeur de zone de données de la procédure) et inversément.

b : Transfert du contenu d'une variable dans une autre variable (même mode d'adressage que le point a).

3 : Gestion du tas.

4 : Branchements inconditionnels et conditionnels.

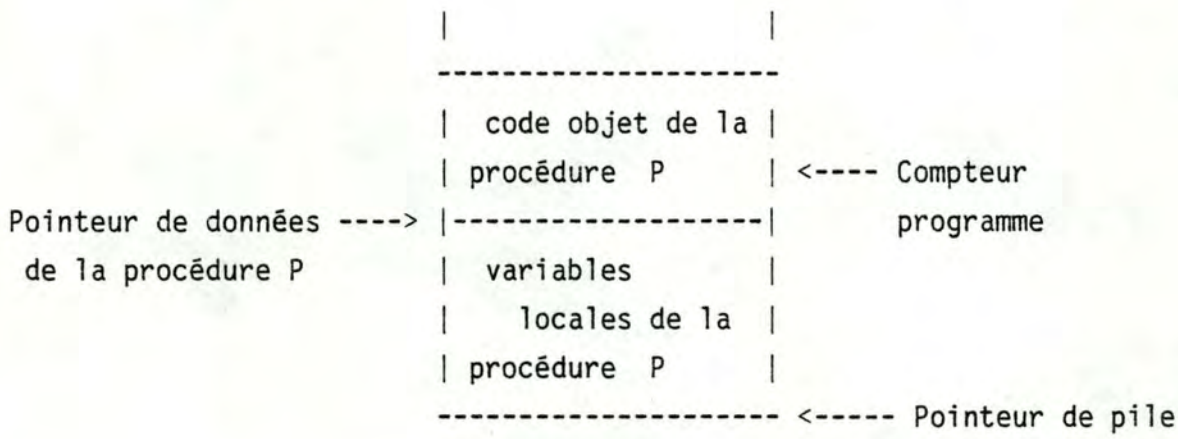
a : Sauts inconditionnels. L'adresse de branchement est spécifiée par un déplacement par rapport à l'adresse actuelle (déplacement positif ou négatif).
Dans le cas d'un déplacement négatif, le déplacement effectif se trouve dans une table de branchements située à la fin du code objet de la procédure.

b : Sauts conditionnels. Une comparaison a d'abord été effectuée et le résultat (variable booléenne) a été placé en sommet de pile.
Selon le résultat de cette comparaison et selon le type de branchement à effectuer, il y aura ou non branchement. L'adresse de branchement est obtenue comme en a.

5 : Appel de procédures.

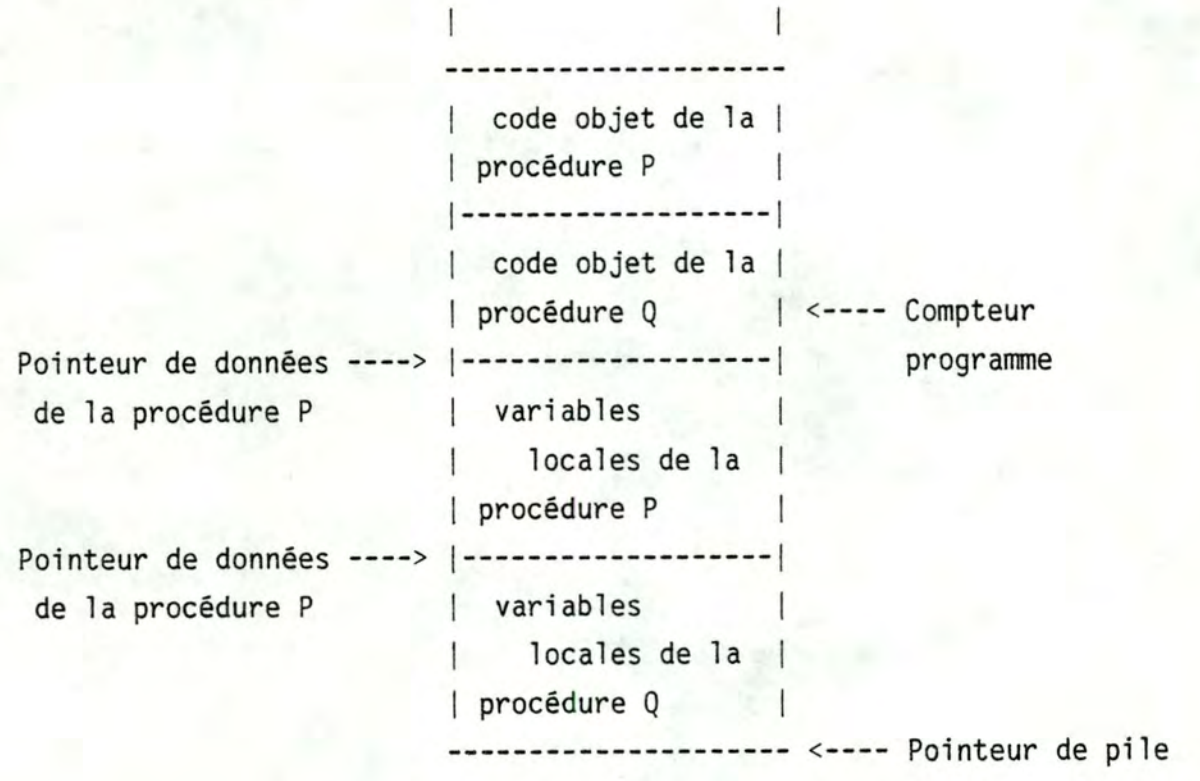
Gestion des procédures :

Lors de l'appel d'une procédure, il y a création en sommet de pile d'une zone contenant les variables locales à cette procédure.
L'adresse du début de cette zone est notée dans un pointeur indiquant le début de la zone de données.
Les intructions et les données se trouvent dans la même zone de la mémoire centrale (pile).

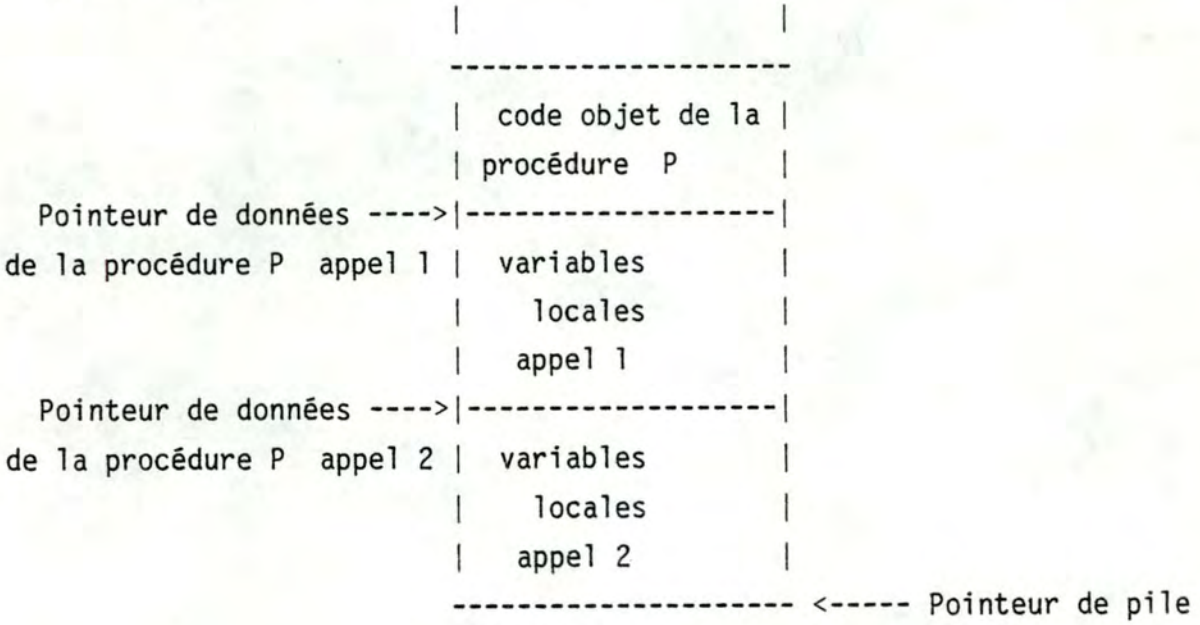


Lors de la terminaison d'une procédure, on enlève de la pile la zone contenant les variables locales.

Si une procédure (P) appelle une autre procédure (Q), en supposant que les codes objets de ces 2 procédures se trouvent déjà en mémoire centrale, il y a création en sommet de pile d'une zone destinée à contenir les variables locales de la nouvelle procédure.



Si une procédure s'appelle elle-même (appel récursif ou appel croisé), on crée une nouvelle zone de données et un nouveau pointeur.



Nous voyons que lors de chaque appel d'une procédure, le système doit mémoriser certaines informations nécessaires à l'adressage correct des variables d'une procédure.
Ces informations sont mémorisées à la suite de la zone contenant les variables locales de la procédure (Cette zone est appelée MARKSTACK, [4]).

Bien entendu, cette zone de sauvetage doit également contenir l'adresse de retour à la procédure appelante et l'adresse de la zone de variables locales précédentes.

A chaque procédure est affecté un numéro (selon l'ordre de rencontre des procédures par le compilateur).
Il existe une table (dictionnaire des procédures) donnant pour chaque procédure, son adresse, la taille de ses variables locales et de ses paramètres formels.
Lors de la compilation d'un programme, le compilateur remplace les noms symboliques des variables par des adresses relatives, (par rapport à des pointeurs de zones de données de procédures) et les noms symboliques des procédures par les numéros de celles-ci.

a : Appel à des procédures standard (ex: entrées-sorties de base, horloge temps réel...).
Ces procédures standard peuvent se concevoir comme une extension du jeu d'instructions de base.

b : Appel à une procédure locale.
Lors de l'appel, la procédure se trouve déjà en mémoire centrale. Un simple branchement, avec mise à jour des différents pointeurs, suffit.

c : Appel à une procédure externe.
Cette instruction (qui n'est pas apparentée à la notion de référence externe) permet une gestion efficace de la mémoire centrale.
Des informations supplémentaires étant nécessaires, nous exposerons en détail cette instruction dans le chapitre suivant.

5. Les réalisations programmées. (Exemple APPLE II).

Plusieurs systèmes [i] réalisent l'interprétation du CODE-P de manière programmée. Cette manière de procéder offre l'avantage de la simplicité, mais provoque les inconvénients suivants :

- Vitesse d'exécution plus lente d'un programme traduit en CODE-P par rapport à un programme écrit dans le langage machine de l'ordinateur utilisé.

- Nécessité de conserver en mémoire, en permanence, la totalité de l'interpréteur de CODE-P.

A titre d'exemple, voyons comment est réalisé l'interpréteur de CODE-P sur le micro-ordinateur APPLE II :

La mémoire centrale de l'APPLE II est partagée comme suit :

ØØØØ à ØØFF : Page zéro permettant un adressage court [45] sur 8 bits. Elle est utilisée par l'interpréteur de CODE-P.

Ø1ØØ à Ø1FF : Pile utilisée par le microprocesseur 65Ø2 pour les appels de sous-programmes et par l'interpréteur de CODE-P pour la pile d'évaluation. L'emplacement de la pile ne peut pas être choisi par le programmeur car, par construction, le microprocesseur 65Ø2 impose un pointeur de pile de 16 bits dont l'octet de poids fort contient toujours le chiffre hexadécimal 1.

---

[i] Citons North Star, TRS 8Ø, Heathkit basés sur des microprocesseurs Z8Ø.
APPLE II basé sur un microprocesseur 65Ø2.
Goupil basé sur un microprocesseur 68ØØ.

Ø2ØØ à Ø3FF : Cette partie de la mémoire contient certaines variables de l'interpréteur de CODE-P, ainsi que le buffer du clavier.

Ø4ØØ à ØBFF : Cette partie de la mémoire est affichée en permanence sur l'écran cathodique. Le rafraîchissement s'effectue en DMA (Direct Memory ACCESS ou ADM = Accès direct à la mémoire).
Si un terminal externe est utilisé, cette partie de la mémoire est utilisée à d'autres fins [i].

ØCØØ à BFFF : Il s'agit de la partie de la mémoire centrale affectée à la pile et au tas. De BDØØ à BFFF, on trouve les variables de l'interpréteur de CODE-P. Ces variables sont également connues du programme CODE-P se trouvant dans la pile.

CØØØ à CFFF : A ces adresses se trouvent les entrées-sorties. Le microprocesseur 65Ø2 possède les entrées-sorties projetées en mémoire.

DØØØ à FFFF : L'accès à cette partie de la mémoire centrale est subordonné au positionnement d'une bascule. Dans un cas, le microprocesseur aura accès à l'interpréteur de CODE-P, dans l'autre cas, le microprocesseur aura accès au BIOS. Cette bascule est positionnée par des lectures à certaines adresses [2] :

- CØ83 : On sélectionne la page 2, c'est-à-dire le BIOS.

- CØ8B : On sélectionne la page 1, c'est-à-dire l'interpréteur de CODE-P.

En réalité, seule la mémoire comprise entre DØØØ et DFFF est paginée (2 pages de 4 K octets). Il existe aux adresses comprises entre FFFØ et FFFF deux petits sous-programmes permettant les changements de pages.

---

[i] Ce point sera détaillé dans le chapitre 4.

On observe un très petit nombre de changements de pages.

En résumé :

- L'interpréteur de CODE-P et le BIOS occupent ensemble 16 K octets.
- La mémoire consacrée à l'affichage cathodique occupe une zone de 2 K octets.
- Les zones occupées par les variables de l'interpréteur de CODE-P et par la pile d'évaluation occupent au total 1 K octets.

L'utilisateur dispose donc d'une mémoire de 45 K octets ou de 43 K octets selon qu'il utilise ou pas un terminal externe.

Chaque instruction du CODE-P est simulée par un petit programme écrit dans le langage machine du microprocesseur utilisé (65Ø2 de MOS TECHNOLOGY). La dernière instruction de ces programmes sera un branchement inconditionnel [i] .

Tous les programmes sont placés séquentiellement en mémoire centrale et les adresses de début des programmes sont placées dans une table. Il y a 256 instructions, mais seules les 128 dernières, numérotées de 128 à 255 sont réalisées de la sorte et ont un point d'entrée dans la table. Une adresse occupant deux octets, une table de 128 adresses occupera 256 octets, nous verrons que cette longueur s'adapte bien au jeu d'instructions du microprocesseur utilisé.

---

[i] Ces morceaux de programmes ne peuvent pas être des sous-programmes au sens habituel du terme, car la pile qui aurait dû contenir les adresses de retour est utilisée pour l'évaluation des expressions arithmétiques.
Il n'est possible d'utiliser des sous-programmes que lorsque l'on est certain de revenir au programme principal avant toute évaluation d'expression arithmétique.

```
                    instruction 128 ....
                                    ....
                                    ....
                    instruction 129 ....
                                    ....
                                    ....
                                    ....
                    instruction 130 ....
                                    ....
                    instruction 131 ....
                                    ....


                    instruction 254 ....
                                    ....
                    instruction 255 ....
                                    ....
                                    ....
```

Table de branchements :

1	poids fort de l'adresse de l'instruction	128
2	faible	128
3	fort	129
4	faible	129
5	fort	130
6	faible	130
252	fort	254
253	faible	254
254	fort	255
255	faible	255

Fetch de la machine virtuelle.

Désassemblage de l'interpréteur de CODE-P entre les adresses D24D et D25E (page 1)

IPCfaible : compteur programme poids faible.
IPCfort : compteur programme poids fort.
SLDC : programme exécutant les 128 instructions.
TABLE : adresse du début de la table poids faible.
TABLE + 1 : adresse du début de la table poids fort.

```
INCRE :  INC IPCfaible    ;Incrémenter le compteur programme.
         BNE FETCH
         INC IPCfort
FETCH :  LDY #ØØ          ;Chargement du code opératoire.
         LDA @58,Y
         BPL SLDC         ;Test si le code opératoire est > 128.
         ASL A            ;Calcul de l'adresse dans la table de
                               branchements.
         STA TABLE
         JMP (TABLE)      ;Branchement indirect.
```

Explication de ce programme :

1 : Incrémenter le PC (compteur programme de la machine virtuelle).

Le microprocesseur de l' APPLE II ne possède que des registres de 8 bits.
Le compteur programme de la machine virtuelle ayant 16 bits, ce compteur sera constitué de 2 octets en mémoire centrale.
Pour l'incrémentation du compteur programme, on ajoutera 1 au premier octet et si celui-ci est égal à Ø, on ajoutera 1 au deuxième octet.

2 . Chargement du code opératoire.

Le microprocesseur de l' APPLE II ne possède pas l'adressage indirect simple mais uniquement deux formes plus complexes de celui-ci

- l'adressage indirect préindexé.

- l'adressage indirect postindexé.

On utilise ici l'adressage indirect postindexé en ayant soin d'initialiser l'index à Ø.

3 . Test si le code opératoire est < 128.

Les codes opératoires inférieurs à 128 (8Ø en hexadécimal) impliquent le chargement d'une constante égale au code opératoire dans la pile.
Pour ces instructions de chargement, il n'est pas nécessaire de passer par la table de branchements.

4 . Calcul de l'adresse dans la table de branchements.

On multiplie le code opératoire par 2, par un décalage de 1 position vers la gauche (car une adresse occupe deux octets dans la table de branchements). De cette manière, on obtient le déplacement, par rapport au début de la table de branchements, de l'adresse du programme correspondant à l'instruction à exécuter.
L'adresse du début de la table de branchements est DØØØ. Le déplacement est de maximum 256 (FF en hexadécimal).
On place le déplacement dans l'octet de poids faible de l'opérante de l'instruction de saut indirect. Seul l'octet de poids faible de l'adresse devra être modifié, l'octet de poids fort restant toujours égal à DØ.

Voici les adresses du début et de la fin de la table de branchements :

Début : DØØØ.

FIN : DØFF.

Conclusions.

Dans le micro-ordinateur APPLE II, (microprocesseur 65Ø2 avec une horloge à 1 MHz.), le fetch dure 29 ms ou 34 ms selon que l'on franchit ou pas une limite de page de 256 octets.
On voit donc le ralentissement que peut provoquer l'interprétation du CODE-P, mais ici trois commentaires peuvent être faits :

1 . Une version interprétée de SPITBOL (SNOBOL4) a été implémentée à l' ILLINOIS INSTITUTE TECHNOLOGY [17]. Il est apparu que l'espace requis et le temps d'exécution avaient diminué par rapport au code généré par un compilateur FORTRAN habituel.

Une autre réalisation [8] a donné un temps d'exécution semblable, mais avec un espace mémoire réduit de 1Ø à 2Ø % par rapport au code généré par un compilateur classique.

2 . Il est évident que, dans le cas d'une interprétation telle qu'elle est conçue dans le micro-ordinateur APPLE II, le temps perdu l'est lors du FETCH du code opératoire.
Il faut donc diminuer, autant que possible, le nombre d'instructions CODE-P générées par les compilateurs. Pour ce faire, les instructions CODE-P doivent être adaptées à la compilation de langage de haut niveau tel que Pascal.

3 . Le pseudo-code devient de plus en plus intéressant au fur et à mesure que la taille du programme s'accroît. En effet le coût de l'interpréteur de CODE-P s'amortit sur un plus grand nombre d'appels.
De plus, la taille de l'interpréteur de CODE-P est fixe et constituera la plus grande partie de la mémoire occupée dans les cas d'exécution de petits programmes [i] .

---

[i] Il est également possible de ne charger en mémoire centrale que les procédures de l'interpréteur de CODE-P nécessaires à l'interprétation du programme courant [8]. Cette pratique semble peu intéressante pour la conception d'une machine virtuelle telle que celle décrite ici.

6. Les réalisations microprogrammées.

Il existe à l'heure actuelle 2 micro-ordinateurs, basés sur des microprocesseurs dont le jeu d'instructions est le CODE-P. Ces 2 micro-ordinateurs sont :

- Pascal 1ØØ.

- Pascal Microengine.

Il s'agit dans les deux cas du même type de microprocesseur, à savoir un microprocesseur 16 bits de WESTERN DIGITAL, mais les possibilités des deux appareils sont différentes. En effet, le micro-ordinateur Pascal 1ØØ comporte deux microprocesseurs : le microprocesseur de WESTERN DIGITAL microprogrammé avec le CODE-P et un microprocesseur 8 bits (Z8Ø de ZILOG).
Le micro-ordinateur Pascal Microengine, quant à lui, ne comporte qu'un seul microprocesseur.

Cette différence de conception entre les deux micro-ordinateurs apparaît largement dans les avantages et inconvénients résumés ci-dessous

Avantages des systèmes microprogrammés :

- Grâce à la microprogrammation du CODE-P, l'exécution d'un programme est beaucoup plus rapide qu'avec une interprétation du CODE-P (de 4 à 9 fois plus rapide).

- Un interpréteur de CODE-P résident n'est plus nécessaire, ce qui permet de gagner une dizaine de K. octets. On peut donc, avec la même taille de mémoire centrale, exécuter de plus grands programmes.

- L'initialisation du système est plus rapide car il ne faut plus charger l'interpréteur de CODE-P en mémoire centrale.

Inconvénients des systèmes microprogrammés :

- Il n'existe pas d'assembleur pour le CODE-P. Il est donc très difficile d'écrire des programmes directement en CODE-P.
De plus le choix des langages est actuellement assez restreint. Cet inconvénient n'existe que partiellement pour le micro-ordinateur Pascal 1ØØ, car celui-ci peut parfaitement fonctionner avec le Z8Ø et, de ce fait, utiliser tous les logiciels écrits pour ce microprocesseur.

- Il est possible que le CODE-P utilisé évolue. Une réalisation microprogrammée sera, elle, très peu adaptable.

Conclusions.

Voici un tableau reprenant les temps de compilation et d'exécution d'un programme, ceci avec différents types de matériels [Byte sptembre 1981] :

Exécution sur des micro-ordinateurs 8 bits et 16 bits :

Compilateur	matériel	temps de compilation et de chargement	temps d'exécution
NBS Pascal	PDP 11/7Ø	2.68	2.6
RSI Pascal	68ØØØ 4MHz		1Ø.2
UCSD Pascal	Pascal 1ØØ	12	54
Ithaca Pascal	Z8Ø 4MHz	124	1Ø9
UCSD Pascal	Z8Ø 4MHz	14	239
Pascal M	Z8Ø 4MHz	5Ø	45Ø
UCSD Pascal	APPLE II	43	516

Les temps de compilation et d'exécution sont exprimés en secondes.

Ce tableau appelle les commentaires suivants :

1 . Le compilateur étant un programme comme les autres, il est remarquable que certains systèmes compilent rapidement et exécutent lentement, et que d'autres systèmes compilent lentement et exécutent rapidement.
Bien sûr, le matériel utilisé influence le temps de chargement et de compilation, en raison des nombreux accès aux mémoires de masse nécessaires, mais ici les différences sont énormes. Il semble que certains réalisateurs de compilateurs ont mis plus de soin à la réalisation de leur programme que d'autres.

2 . La version standard du Z8Ø possède une horloge de 2MHz. Pour pouvoir comparer valablement les temps d'exécution des micro-ordinateurs basés sur des microprocesseurs Z8Ø avec les autres machines reprises au tableau, il convient de multiplier par 2 les temps observés pour les micro-ordinateurs Z8Ø.

La réalisation de machines virtuelles possédant un jeu d'instructions répandu et bien adapté à la compilation des langages de haut niveau paraît être une bonne solution. Cependant les progrès technologiques vont très vite et on trouve déjà actuellement sur le marché, un microprocesseur 8 bits possédant l'adressage relatif sur toute l'étendue de la mémoire, la multiplication câblée de 2 nombres de 8 bits, etc ... (microprocesseur 68Ø9 MOTOROLA).
Les micro-processeurs 16 bits vont encore beaucoup plus loin. Dès lors, il est permis de se demander si l'utilisation d'un CODE-P se justifie encore pour d'autres raisons que pour des raisons purement commerciales .

7. Présentation des utilitaires du système et de leur utilisation.

Les utilitaires les plus importants et les plus fréquemment appelés peuvent être invoqués par l'appui sur une seule touche :

"E" : "EDITOR". Il s'agit d'un éditeur d'écran relativement puissant.
Cet éditeur peut être paramétré de manière à pouvoir créer soit des programmes, soit du texte courant [i].

"F" : "FILER". Il s'agit du programme utilitaire manipulant les fichiers. Il est possible de créer, copier, changer de nom, supprimer des fichiers.

"C" : "COMPILER". Il s'agit d'un appel au compilateur se trouvant sur la disquette qui a servi au chargement du système.

"L" : "LINKER". Il s'agit d'un appel à l'éditeur de liens.

"A" : "ASSEMBLER". Il s'agit d'un appel à l'assembleur se trouvant sur la disquette qui a servi au chargement du système.

"D" : "DEBUGGER". Il s'agit d'un appel au 'debugger'. Normalement celui-ci est absent et lors de l'appel, le système répond par :

"No debugger in system".

Un exemple d'implémentation d'un debugger sera donné à la fin du chapitre 2.

---

[i] Le texte de ce mémoire a été composé à l'aide de cet éditeur.

Au niveau de base, l'interpréteur de commandes présente la ligne suivante :

```
Command:  E(dit,R(un,F(ile,C(omp,L(ink,X(ecute,A(ssem,D(ebug, ? [1.1]
```

ou

```
Command:   U(ser restart,I(nitialise,H(alt,S(wap,M(ake exec
```

La seconde ligne est visible si l'utilisateur a demandé un complément d'information au sujet des commandes possibles (option '?').

La lettre majuscule indique l'abréviation utilisée pour désigner la commande. Après la parenthèse se trouve, en minuscules, le nom complet de la commande.
Par exemple :
E = Exécution de l'éditeur
C = Exécution du compilateur etc...

Il faut noter que les deux lignes ci-dessus peuvent se présenter de manière abrégée lorsque l'utilisateur a précisé l'option "SLOWTERM" lors de la configuration du système (voir l'utilitaire "SETUP.CODE" présenté ci-dessous) .

Il existe également une commande combinée permettant la compilation, l'édition de liens et l'exécution à l'aide d'une seule commande : "R" = "RUN".

Les cinq commandes simples sont :

"U" : "USER RESTART". On exécute à nouveau le programme qui vient de se terminer.

"I" : "INITIALISE". On effectue un 'WARM BOOT' du système, c'est-à-dire une initialisation sans recharger l'interpréteur de CODE-P.

"H" : "HALT". On effectue un 'COLD BOOT' du système, c'est-à-dire une initialisation avec rechargement du système d'exploitation.
Si avant d'exécuter la commande "H", on a placé une disquette contenant un autre système d'exploitation dans le lecteur servant au chargement, il y a initialisation de ce nouveau système d'exploitation.

"S" : "SWAP". Cette commande permet l'exécution de plus grands programmes. En effet, on obligera le système à éliminer à tout moment les segments non utilisés.

"M" : "MAKE EXEC". Cette commande permet de créer des fichiers de commandes.

L'exécution d'un programme utilisateur ou d'un utilitaire non repris ci-dessus se fera via la commande "X" = "EXECUTE".

Voici la liste des utilitaires principaux :

LIBRARY.CODE : Cet utilitaire permet de créer des librairies. Cependant, comme les programmes compilés et les librairies ont la même structure, cet utilitaire permettra également de substituer à l'intérieur d'un programme un segment par un autre. Cette astuce a été utilisée lors de la conception du 'debugger'.

LIBMAP.CODE : Cet utilitaire permet de lister tous les segments se trouvant dans un programme ou dans une librairie.

SETUP.CODE : Cet utilitaire permet de changer la configuration du système en fonction de l'utilisation d'un certain type de périphérique. Il est ainsi possible de définir les caractères de contrôle nécessaires pour un terminal donné, de définir les dimensions de l'écran (nombre de lignes et de colonnes) et de préciser si l'on travaille sur écran cathodique ou sur terminal imprimant. L'option "SLOWTERM" (pour terminal imprimant) permet de réduire la taille des messages du système et, de ce fait, de ne pas trop ralentir le travail à la console.

Chapitre 2.

Etude du système de segmentation.

1. Raisons de la segmentation.

Les micro-ordinateurs ont une mémoire centrale de taille relativement réduite, généralement inférieure à 64 K octets.
Ceci est dû à la structure même des microprocesseurs 8 bits actuels qui ont un bus d'adresses de 16 fils.

Pour permettre l'exécution de très gros programmes, deux solutions sont possibles :

a : Certains micro-ordinateurs ont une taille de mémoire centrale supérieure à 64 K. octets, mais cette mémoire centrale est alors constituée d'un certain nombre de 'blocs'; un système d'adressage spécial doit être mis en oeuvre pour passer d'un bloc à un autre .

b : Pour permettre l'exécution de programmes d'une taille supérieure à la taille de la mémoire centrale, un système de segmentation des programmes peut être utilisé. Ce système permet à l'utilisateur de découper son programme en différentes parties, appelées segments. Ces différentes parties seront chargées en mémoire centrale au fur et à mesure de leur activation. Dans ce cas, la taille maximale d'un programme n'est limitée que par le nombre maximum de segments autorisés.

Dans le cas des micro-ordinateurs, les mémoires de masse sont généralement des disquettes souples (8 pouces de diamètre) ou des mini-disquettes souples (5.25 pouces de diamètre) magnétiques. Les temps d'accès de ces périphériques varient en général de 1ØØ a 5ØØ millisecondes [36], et sont beaucoup trop longs pour envisager la gestion d'une mémoire virtuelle. Il faut donc réduire au strict minimum le nombre des accès physiques aux disques ou disquettes souples. De plus, les micro-ordinateurs ne possèdent généralement pas les possibilités 'HARDWARE' nécessaires à une bonne gestion de la mémoire par pagination [i].

Cependant il n'est pas impossible que, dans un proche avenir, les données du problème soient complètement modifiées, pour deux raisons :

- d'abord les microprocesseurs 16 bits de la nouvelle génération (68ØØØ de Motorola et Z8ØØØ de Zilog [35]) ont une capacité d'adressage beaucoup plus grande (jusqu'à 16 Méga octets; 24 fils d'adresses);

- ensuite l'apparition de disques durs à prix compétitif, et peut-être de mémoires à bulles magnétiques de grande capacité et bon marché.

---

[i] Sur les microprocesseurs 8 bits actuels, une interruption est toujours prise en compte entre 2 instructions, une instruction est toujours exécutée d'un seul trait. Il n'est donc pas possible d'interrompre l'exécution au milieu d'une instruction parce que les données nécessaires à l'exécution de cette instruction ne sont pas présentes en mémoire centrale (défaut de page).

2. Principe de la segmentation.

Le système d'exploitation UCSD Pascal permet la découpe d'un programme en plusieurs segments. Pendant l'exécution d'un programme, il n'y aura simultanément en mémoire centrale qu'un nombre réduit de segments; les changements de segments (chargement et déchargement) sont effectués automatiquement par le système d'exploitation.

La segmentation adoptée, oblige le programmeur à un découpage des programmes en segments.
Il est bien évident que si 2 ou plusieurs segments s'appellent mutuellement un grand nombre de fois, il y aura un grand nombre de chargements et déchargements, ce qui entraînera un grand nombre d'accès physiques aux disques, ce qui dans certains cas entraînera un temps d'exécution exagérément long. On retrouve ici les techniques d'overlay des anciens systèmes (DOS, etc...).

C'est donc l'utilisateur qui doit effectuer une découpe intelligente de son programme.

3. Exemples.

Nous présentons ci-dessous un exemple de segmentation.
Cet exemple n'a qu'une valeur pédagogique, et ce, pour deux raisons :

- Le programme est très court. Il n'y a donc pas de problème d'encombrement de la mémoire centrale.

- Ce programme ne permet pas de découpe en segments satisfaisante.

```
(* exemple 1 :  Ce programme trie un tableau de 1ØØ nombres par la
méthode du tri SHELL sans utiliser la segmentation *)

PROGRAM ESSAI1;
const nb=1ØØ;
var ech,ecart,i:integer;
tab:array[1..nb] of integer;

procedure echange;
var x:integer;
begin
  x:=tab[i];
  tab[i]:=tab[i+ecart];
  tab[i+ecart]:=x
end;

begin
   for i:=1 to nb do
      read(tab[i]);
   writeln('tableau non classe');
   writeln;
   writeln;
   ecart:=nb;
   repeat
      ecart:=ecart div 2;
      repeat
         ech:=Ø;
         for i:=1 to nb-ecart do
         begin
            if tab[i]>tab[i+ecart] then
            begin
               ech:=1;
               echange
            end
         end
      until ech=Ø
   until ecart=1;
   writeln('tableau classe');
end.
```

Le programme est constitué de 2 parties.

- Une procédure "ECHANGE" qui intervertit 2 éléments du tableau de départ.

- Le programme proprement dit, qui est composé de 3 boucles imbriquées. C'est à l'intérieur de la troisième boucle que l'on fera appel si nécessaire à la procédure échange.

La compilation de ce programme génère un fichier exécutable enregistré sur la mini-disquette. Lors de l'exécution de ce fichier exécutable, le code objet est tout d'abord placé entièrement en mémoire centrale, ensuite l'exécution est lancée.

Il s'agit du cas le plus classique où le programme principal et la procédure "ECHANGE" se trouvent simultanément en mémoire centrale. Dans ce cas, un appel à la procédure "ECHANGE" se traduira par un simple branchement (il faudra néanmoins mémoriser l'adresse de retour et mettre à jour un certain nombre de pointeurs permettant l'accès aux variables locales (mise à jour du "display" [25])). Le programme ESSAI1 constitué d'un seul segment s'exécute en 3 secondes environ.

Il sera nécessaire si la taille du programme devient importante de scinder le programme en 2 ou plusieurs segments. Dans notre exemple, la procédure "ECHANGE" et le programme principal constitueront chacun un segment. Ces 2 segments se trouveront alternativement en mémoire centrale. Le programme ESSAI2 donne un exemple de cette possibilité.

```
(* exemple 2 :  Ce programme trie un tableau de 1ØØ nombres par la
méthode du tri SHELL ( programme segmenté ) *)

PROGRAM ESSAI2;
const nb=1ØØ;
var ech,ecart,i:integer;
tab:array[1..nb] of integer;

segment procedure echange;
var x:integer;
begin
   x:=tab[i];
   tab[i]:=tab[i+ecart];
   tab[i+ecart]:=x
end;

begin
   for i:=1 to nb do
      read(tab[i]);
   writeln('tableau non classe');
   writeln;
   writeln;
   ecart:=nb;
   repeat
      ecart:=ecart div 2;
      repeat
         ech:=Ø;
         for i:=1 to nb-ecart do
         begin
            if tab[i]>tab[i+ecart] then
            begin
               ech:=1;
               echange
            end
         end
      until ech=Ø
   until ecart=1;
   writeln('tableau classe');
end.
```

Ce programme constitué de 2 segments s'exécute en environ 5Ø secondes, non compris le temps de lecture des données. Bien entendu, il s'agit d'un découpage en 2 segments très mal choisi, (un programme comme celui de cet exemple, qui ne contient pas de procédures peu fréquemment appelées, ne permet pas de découpe satisfaisante en segments), mais cet exemple illustre bien la perte de temps que peut entraîner une mauvaise perception du problème.
L'utilisateur doit donc pouvoir évaluer le nombre de fois que chaque procédure sera appelée, et par quelle autre procédure elle sera appelée. Cette contrainte peut devenir très gênante dans le cas d'une modification d'un programme existant.

La découpe en segments peut s'avérer très utile, et même efficace, pour des procédures d'initialisation ou de récupération d'erreurs, car dans ce cas, ces procédures n'encombreront pas la mémoire centrale pendant l'exécution du corps du programme.

4. Les réalisations possibles.

Sur les ordinateurs classiques, généralement, un mécanisme câblé établit la correspondance entre adresses virtuelles et adresses physiques.

Le temps de réaction de ce dispositif câblé est négligeable par rapport au temps d'accès à la mémoire centrale (1Ø à 5Ø nanosecondes pour le dispositif câblé, de l'ordre de 1 microseconde pour la mémoire centrale).

Lors de l'exécution d'une instruction, si l'opérante n'est pas accessible (défaut de page par exemple), le programme est interrompu et passe la main à une routine du gérant des interruptions qui se chargera de placer en mémoire la zone de code ou de données désirée (actuellement, sur les configurations importantes, les entrées-sorties sont gérées par des processeurs indépendants du processeur principal).
La mémoire centrale est physiquement constituée de plusieurs parties et deux processeurs différents peuvent accéder simultanément à la mémoire centrale, à condition que les zones adressées soient physiquement distinctes. Dans ces conditions, les échanges de données avec les périphériques rapides ne monopoliseront pas le processeur principal) . A la fin d'une opération d'entrée-sortie, lorsque le programme utilisateur reprendra son exécution, toutes les références pour l'instruction interrompue seront satisfaites et l'exécution du programme pourra se poursuivre.

Sur les micro-ordinateurs, ce mécanisme câblé n'existe pas (sauf pour des appareils de haut de gamme, mais ceux-ci sont plutôt à placer parmi les miniordinateurs, du moins par leur prix [i]).
Toutes les opérations de gestion des segments doivent être réalisées par logiciel, avec tous les inconvénients que cela apporte (augmentation du temps d'exécution par exemple).

---

[i] De plus, ce mécanisme câblé n'est pas intégré sur la puce du microprocesseur, mais est réalisé en circuits intégrés classiques à moindre intégration.

C'est une des raisons qui ont plaidé en faveur du choix d'un P-CODE et de son interprétation, malgré les inconvénients. Les systèmes d'exploitation pour micro-ordinateurs n'offrent en général que peu de possibilités de gestion de la mémoire, le plus souvent seul un chaînage des programmes avec une zone de données préservée est possible.

Une deuxième raison est la difficulté d'écrire des programmes relogeables, du moins pour la majorité des microprocesseurs actuels; la nouvelle génération de microprocesseurs permet un adressage relatif sur toute l'étendue de la mémoire centrale, alors que les microprocesseurs actuels les plus courants (8080, Z80, 6800, 6502) ne le permettent que pour un déplacement de 128 octets (en avant ou en arrière, le déplacement étant codé sur 8 bits).

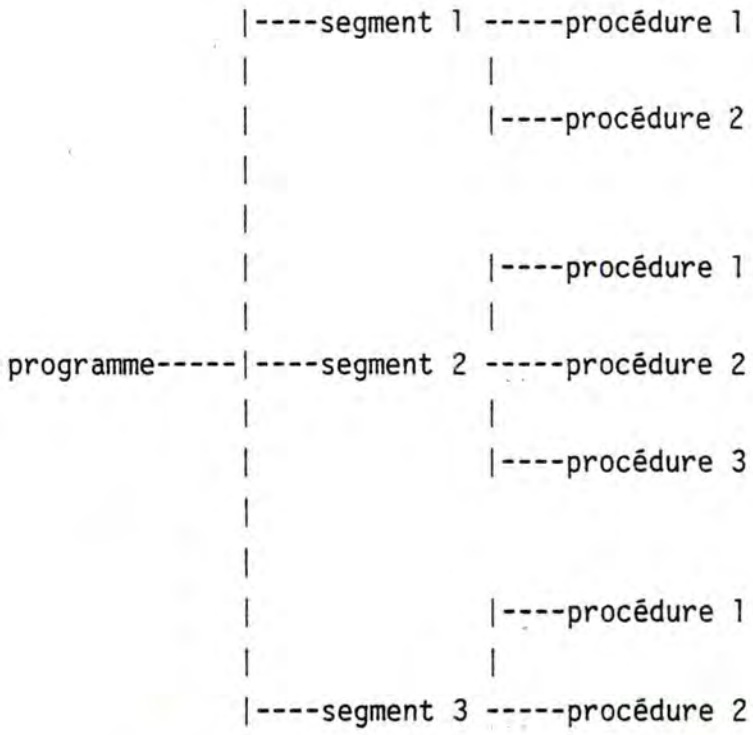
L'interprétation d'un P-CODE autorise tous les modes d'adressage possibles puisque les instructions de la machine virtuelle sont simulées sur l'ordinateur hôte.

5. La gestion des segments sous UCSD-Pascal.

Les possibilités offertes.

Le système d'exploitation UCSD-Pascal permet l'utilisation de 15 segments. Chaque segment porte un numéro logique (déterminé par le compilateur ou par l'éditeur de liens) et un numéro physique (déterminé par la position du segment sur le disque). Le système contient une table à 16 entrées contenant les adresses et les longueurs des différents segments sur le disque. Le système fait référence aux différents segments via leur numéro logique.

En Pascal, le découpage en segments est assez restrictif. On ne peut déclarer comme segments que des procédures et des fonctions (voir l'exemple ESSAI2). Une procédure ou une fonction doit se trouver dans un seul segment, mais un segment peut contenir plusieurs procédures ou fonctions (maximum 149). Il n'y a jamais de segments de données.



La réalisation.

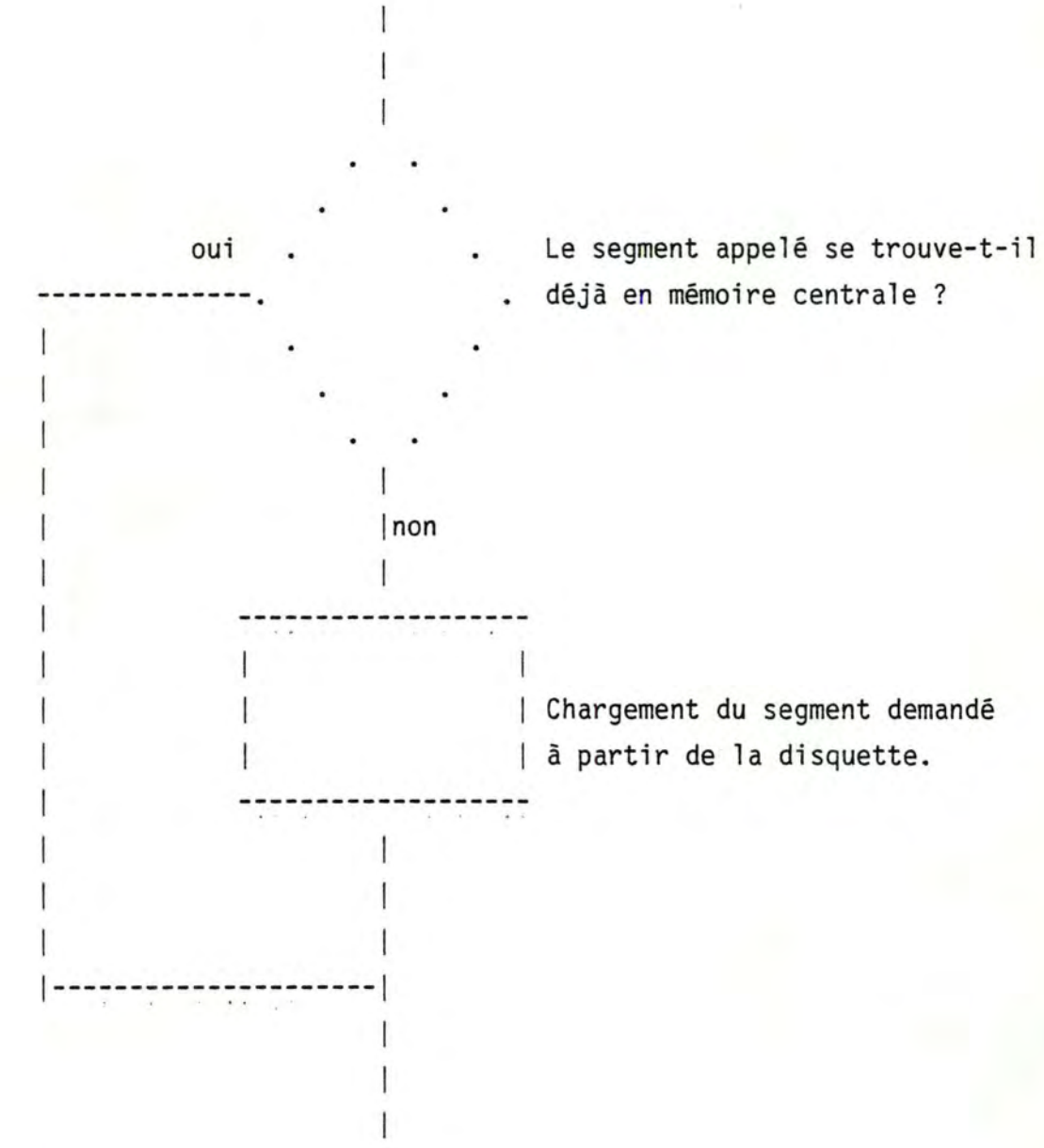
Il existe une instruction P-CODE qui manipule les segments.

CXP X,Y (CALL EXTERNAL PROCEDURE
Appel de la procédure Y se trouvant dans le segment X).

Appel à une procédure se trouvant dans un autre segment. Lors de l'appel, le système parcourt la liste des segments actifs (un segment est actif s'il se trouve déjà en mémoire centrale) . Si le segment appelé est déjà en mémoire centrale, il y a sauvetage de l'adresse de retour, calcul de l'adresse de la procédure et de différents pointeurs, et branchement. Si le segment appelé n'est pas en mémoire centrale, il y a chargement du segment [i], sauvetage de l'adresse de retour, calcul de l'adresse de la procédure et des différents pointeurs et branchement.

---

[i] Lors du chargement d'un segment en mémoire centrale, celui-ci est placé au sommet de la pile. L'utilisateur n'a donc pas la possibilité de choisir l'adresse physique de chargement.



Le fichier contenant le code généré.

La compilation d'un programme composé de plusieurs segments va générer un fichier 'CODE' qui sera enregistré sur disque. Celui-ci est découpé en blocs de 512 octets [i].

---

[i] Nous verrons au chapitre 4 la raison de la taille de ces blocs.

Le premier bloc contiendra le dictionnaire des segments, les blocs suivants contiendront le code objet proprement dit. Un segment devant toujours commencer en début de bloc, il en résulte une perte de place entre 2 segments, perte qui sera en moyenne d'un demi-bloc entre chaque segment, c'est-à-dire au total, un nombre de blocs égal au nombre de segments divisé par 2 [i].

Dans un segment, tous les objets sont adressables relativement à l'origine du segment. A chaque objet est associée une adresse relative R telle que Ø<=R<=L (L = longueur du segment).

Il s'agit d'une adresse virtuelle car elle est indépendante de toute localisation du segment dans la mémoire centrale et donc des adresses physiques [3Ø]. Seules les adresses virtuelles peuvent figurer dans les instructions; la correspondance entre les adresses virtuelles et les adresses physiques se fait par la simple addition de l'adresse virtuelle et d'un pointeur indiquant l'adresse physique du premier objet d'un segment.

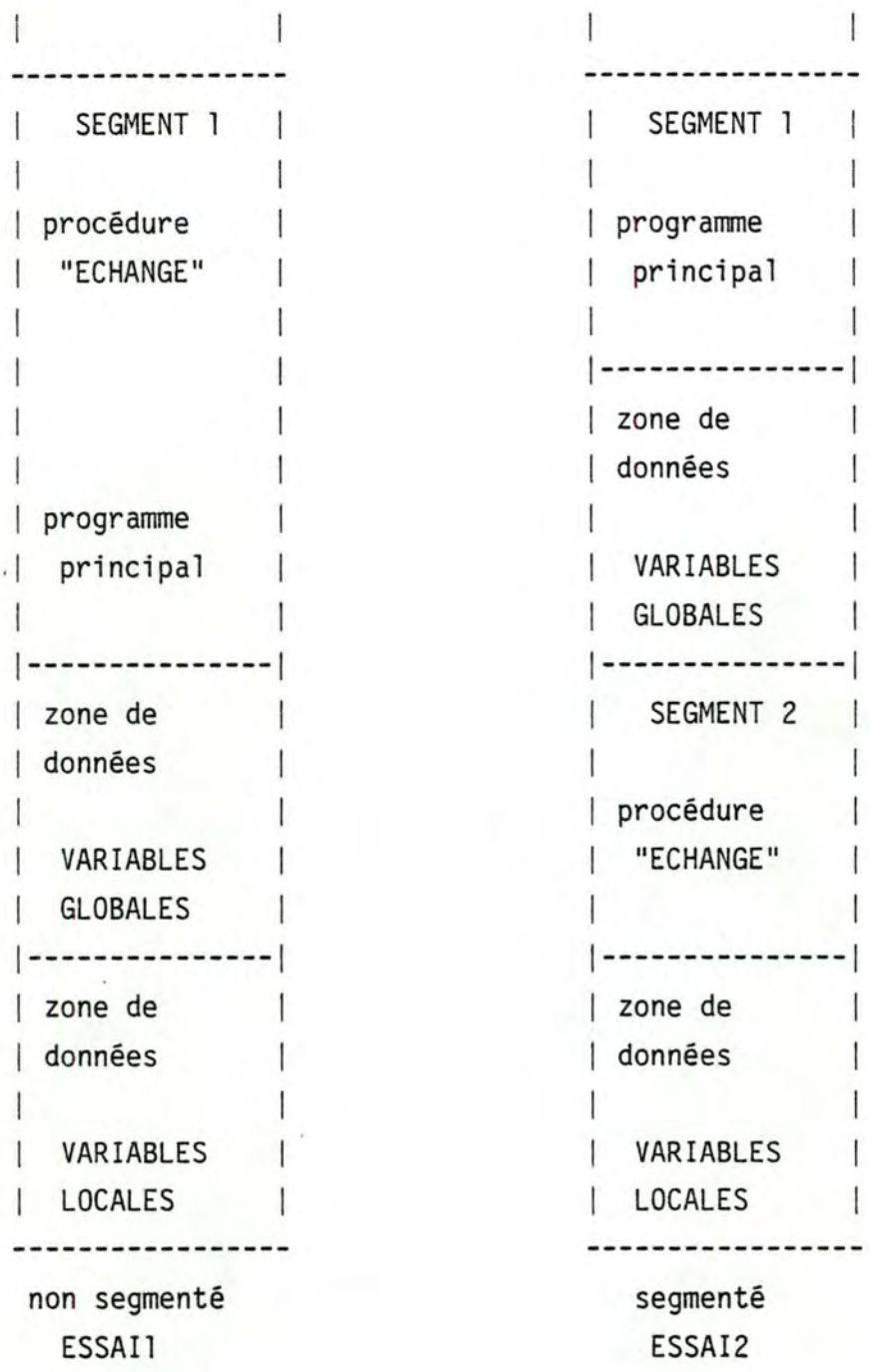
---

[i] Voir en annexes 2 et 5 les codes générés pour les programmes ESSAI1 et ESSAI2.

Lors de l'exécution d'un segment, la pile est étendue avec les codes des procédures constituant ce segment. Toutes les procédures d'un segment sont donc chargées en une seule fois. Il y aura création d'une zone de variables locales lors de chaque appel de procédure (La zone de variables locales contiendra également une série de pointeurs vers les zones de variables locales précédentes, etc...).

Exemple : La structure de la pile pendant l'exécution de la procédure "ECHANGE" est présentée sur la figure ci-dessous :

en a pour le programme ESSAI1 (non segmenté), et en b pour le programme ESSAI2 (segmenté).



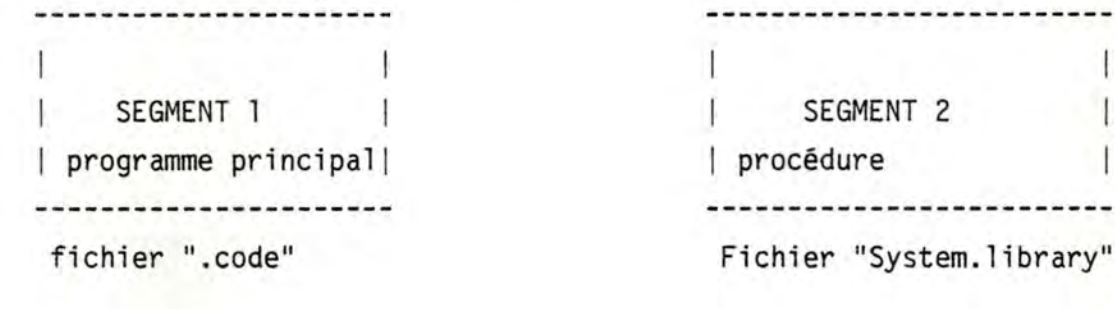
a. b.

Un segment se trouvera en mémoire aussi longtemps qu'il sera actif (aussi longtemps qu'une des procédures contenues dans ce segment se trouvera dans la chaîne d'exécution) .
Lorsqu'il deviendra inactif (retour à la procédure appelante) , l'espace mémoire qu'il occupait redeviendra libre pour charger un autre segment.

Il n'y a jamais sauvetage temporaire d'un segment, car à la fin de l'exécution d'un segment, les variables qui lui font suite dans la pile ne sont connues que des procédures et fonctions de ce segment (variables locales à ces procédures et fonctions) et ne doivent pas être récupérées lors d'un autre appel à ces procédures et fonctions.

S'il s'agit d'un appel récursif ou d'un appel croisé, le segment contenant la procédure désirée se trouvera déjà en mémoire centrale. Dans ce cas, il y aura seulement création, dans la pile, d'une zone de variables locales (voir au chapitre 1 les exemples donnés pour la gestion des appels de procédures) .

Un des gros avantages du découpage en segments, outre la possibilité de créer de très gros programmes, est la possibilité de créer des procédures et fonctions utilisables par plusieurs programmes ('Intrisic Unit').
Ces procédures peuvent être compilées séparément et placées en bibliothèque (Voir le listing en annexe 9) .
Il n'y a pas d'édition de liens, les références aux objets extérieurs étant faites par le compilateur à l'aide de directives spéciales. Dans ce cas, les différents segments sont placés dans des fichiers distincts (Voir figure ci-dessous.) .



Deux ou plusieurs programmes peuvent faire référence à une même procédure se trouvant dans la librairie.

Le système d'exploitation utilise cette possibilité pour les procédures de gestion des nombres réels. Ces procédures (fonctions trigonométriques, lecture et écriture de nombres réels, etc ...) sont compilées séparément et placées en bibliothèque.
Lors de l'utilisation, le programmeur doit simplement donner le nom du segment contenant les procédures (option 'USES' du compilateur Pacsal) et veiller pendant l'exécution à ce que le fichier contenant la bibliothèque se trouve en ligne.

Cette solution (Intrinsic Unit) devra être choisie si plusieurs programmes utilisent les mêmes procédures ou fonctions, car dans ce cas on économise l'espace sur les mémoires de masse.

Détails du dictionnaire de segments.

Le dictionnaire de segments est constitué de 5 parties.

1: une zone contenant 32 nombres entiers (nombres de 16 bits) représentant les adresses des premiers secteurs et les longueurs des différents segments.

2: Une zone contenant 16 chaînes de caractères, de 8 caractères chacune, représentant les noms des différents segments.

3: Une zone contenant diverses informations concernant les différents segments (programme principal, unit, segment déjà relié; adresse de la description des variables globales) .

4: Une zone contenant une chaîne de caractères représentant une note placée par le programmeur. Il s'agit le plus souvent d'un copyright. Celui-ci est placé par le compilateur (option "(*C *)" du compilateur) ou par l'utilitaire de création des librairies (par la commande "Notice") .

5: Une zone contenant les numéros des segments, la version du compilateur utilisé, etc ... .

Nous présentons ci-dessous deux exemples de dictionnaire de segments tels qu'ils sont donnés par l'utilitaire 'LIBRARY.CODE' .

a) Voici le dictionnaire des segments du système d'exploitation "SYSTEM.PASCAL" (les numéros entre parenthèses sont les numéros logiques) :

Ø-(Ø)	PASCALSY	2376	8-		Ø
1-(1)	USERPROG	56	9-		Ø
2-(2)	DEBUGGER	62	1Ø-		Ø
3-(3)	PRINTERR	1Ø34	11-		Ø
4-(4)	INITIALI	299Ø	12-		Ø
5-(5)	GETCMD	3182	13-		Ø
6-		Ø	14-		Ø
7-		Ø	15-		4636

Assez paradoxalement, les noms des segments ne sont jamais utilisés, mais certains utilitaires reconnaissent la présence ou l'absence d'un segment par la présence ou l'absence de son nom dans le dictionnaire des segments (exemple : l'utilitaire LIBRARY.CODE).

b) Certains réalisateurs de programmes utilisent les noms des segments pour placer un copyright. Exemple: Compilateur FORTRAN

Ø-		Ø	8-(8)	MPUTER I	2378
1-(1)	FORTRAN:	1844	9-(9)	NC. ALL	1666
2-		Ø	1Ø-(1Ø)	RIGHTSRE	53Ø4
3-	COPYRI	Ø	11-(11)	SERVED	438Ø
4-	GHT 198Ø	Ø	12-(12)	(R.WIGGI	13854
5-	BY	Ø	13-(13)	NTON WAS	6666
6-		Ø	14-(14)	HERE)	2114
7-(7)	APPLE CO	5162	15-		Ø

6. L'exécution d'un programme.

Introduction.

Supposons que l'on veuille exécuter un programme utilisateur, écrit en langage Pascal. Ce programme utilisateur sera d'abord compilé, ce qui aura pour effet de créer un fichier contenant le code objet. Le compilateur Pascal, fourni avec le système d'exploitation, a la particularité que les fichiers contenant les codes objets générés ne nécessitent pas d'édition de liens et peuvent être chargés tels quels en mémoire centrale [i] .

Si le programme utilisateur est écrit dans un autre langage ou si ce programme utilise des procédures et fonctions compilées séparément, une édition de liens sera nécessaire [ii]. L'éditeur de liens est un programme utilitaire qui rétablit les références manquantes et crée un fichier contenant le code objet exécutable.

A partir de ce moment, pour exécuter le programme, il suffit de charger celui-ci en mémoire centrale, à une adresse préalablement choisie (sur les micro-ordinateurs, cette adresse est

---

[i] Bien entendu, l'édition de liens sera nécessaire si le programme utilise des procédures et fonctions compilées séparément et se trouvant dans un autre fichier.

[ii] Le compilateur FORTRAN possède par exemple un ensemble de procédures précompilées qui se trouvent dans la librairie FORTRAN. Ces procédures sont indispensables pour permettre l'exécution d'un programme (c'est le cas par exemple de la procédure "EXIT") et constituent le "RUN TIME PACKAGE".

presque toujours la même), et d'effectuer un branchement à la première instruction de ce programme. C'est le rôle d'un autre programme utilitaire, le chargeur [i].

Dans le cas du système d'exploitation UCSD-Pascal, le chargeur est très réduit. Il s'agit d'une procédure qui modifie la table des segments (table située en mémoire centrale et contenant une partie des informations contenues dans le dictionnaire des segments du programme en train de s'exécuter) en y plaçant les nouveaux segments à charger en mémoire centrale. Les routines de gestion des segments feront le reste, c'est-à-dire, placer le code objet en mémoire centrale au moment voulu.

Analyse du programme "PASCALSYSTEM".

Le noyau [ii] du système d'exploitation est écrit en Pascal (Programme "PASCALSYSTEM"). Il est constitué de 8 segments (les numéros logiques de ces segments sont compris entre Ø et 6 pour les 7 premiers segments, quant au dernier segment, il porte le numéro 15).

Voici la liste des segments du programme "PASCALSYSTEM"(listing fourni par l'utilitaire "LIBMAP.CODE") .

---

[i] Il existe d'autres méthodes, par exemple "CODE AND GO". De plus certains systèmes intègrent en un seul programme l'éditeur de liens et le chargeur.

[ii] Nous appellerons noyau du système d'exploitation le programme contenu dans le fichier "SYSTEM.PASCAL".
Il s'agit de la partie du système d'exploitation devant obligatoirement se trouver en ligne lors de l'utilisation de l'ordinateur. Le noyau ne comprend donc pas les compilateurs, ni les différents utilitaires du système.

```
  LIBRARY MAP FOR system.pascal
  *****************************

Segment °Ø:
Pre-II.1 segment
PASCALSY  completely linked segment
-------------------------------------------------------------------------
Segment ° 1:
System version = 3.Ø, code type is P-Code (least sig. lst)
USERPROG  completely linked segment
-------------------------------------------------------------------------
Segment ° 2:
System version = 3.Ø, code type is P-Code (least sig. lst)
DEBUGGER  completely linked segment
-------------------------------------------------------------------------
Segment ° 3:
System version = 3.Ø, code type is P-Code (least sig. lst)
PRINTERR  completely linked segment
-------------------------------------------------------------------------
Segment ° 4:
System version = 3.Ø, code type is P-Code (least sig. lst)
INITIALI  completely linked segment
-------------------------------------------------------------------------
Segment ° 5:
System version = 3.Ø, code type is P-Code (least sig. lst)
GETCMD    completely linked segment
-------------------------------------------------------------------------
Segment ° 6:
System version = 3.Ø, code type is P-Code (least sig. lst)
FILEPROC  completely linked segment
-------------------------------------------------------------------------
Segment °15:
Pre-II.1 segment
NONAME    completely linked segment
-------------------------------------------------------------------------
```

Signification des informations contenues dans la carte des segments :

- Numéro du segment.

- Numéro de la version du système d'exploitation utilisée lors de la compilation du segment.

- Type de code objet. (si le segment est constitué de procédures écrites en assembleur de l'ordinateur hôte, le type de processeur est indiqué)

- Informations concernant la machine virtuelle. (Il existe deux types de machines virtuelles selon la manière dont sont rangés les nombres de 16 bits dans une mémoire centrale orientée octet : octet le plus significatif à l'adresse haute ou pas) .

- Le segment est-il déjà relié ou pas (Intrinsic Unit ou Regular Unit).

(le squelette du programme source du noyau du système d'exploitation se trouve en annexe 1Ø)

- Le segment numéro Ø est constitué par le programme principal.

- Les segments 1 et 2 sont assez particuliers et sont détaillés au paragraphe suivant.

- Le segment numéro 3 est contitué d'une procédure. Il est appelé lorsqu'une erreur est détectée pendant l'exécution d'un programme (exemple : division par zéro).

- Le segment numéro 4 sert à initialiser le système (initialisation des pointeurs de la pile et du tas, et des variables utilisées par le système d'exploitation) .

- Le segment numéro 5 constitue l'interpréteur de commandes. Il lit un caractère au clavier et exécute la fonction correspondante.

- Le segment numéro 6 gère les fichiers.

- Le segment numéro 15, ne portait pas de nom; ce nom a été ajouté pour pouvoir manipuler ce segment plus aisément (certains utilitaires reconnaissent la présence ou l'absence d'un segment par la présence ou l'absence d'un nom valide pour ce segment).
Pour ces raisons, nous avons donné le nom 'NONAME' au segment 15.

Les segments 1 et 2.

Le premier segment autorise l'exécution d'un programme "utilisateur", tandis que le deuxième permet l'exécution du "DEBUGGER".

L'exécution d'un programme utilisateur.

Dans le programme principal existe un appel à la procédure numéro 1 du segment 1 (USERPROGRAM). L'instruction dans le programme "PASCALSYSTEM" effectuant cet appel est simplement

```
USERPROGRAM;
```

ce qui sera traduit par le compilateur par

```
CXP Ø1,Ø1    ;(Call eXternal Procedure segment 1
                 procedure 1)
```

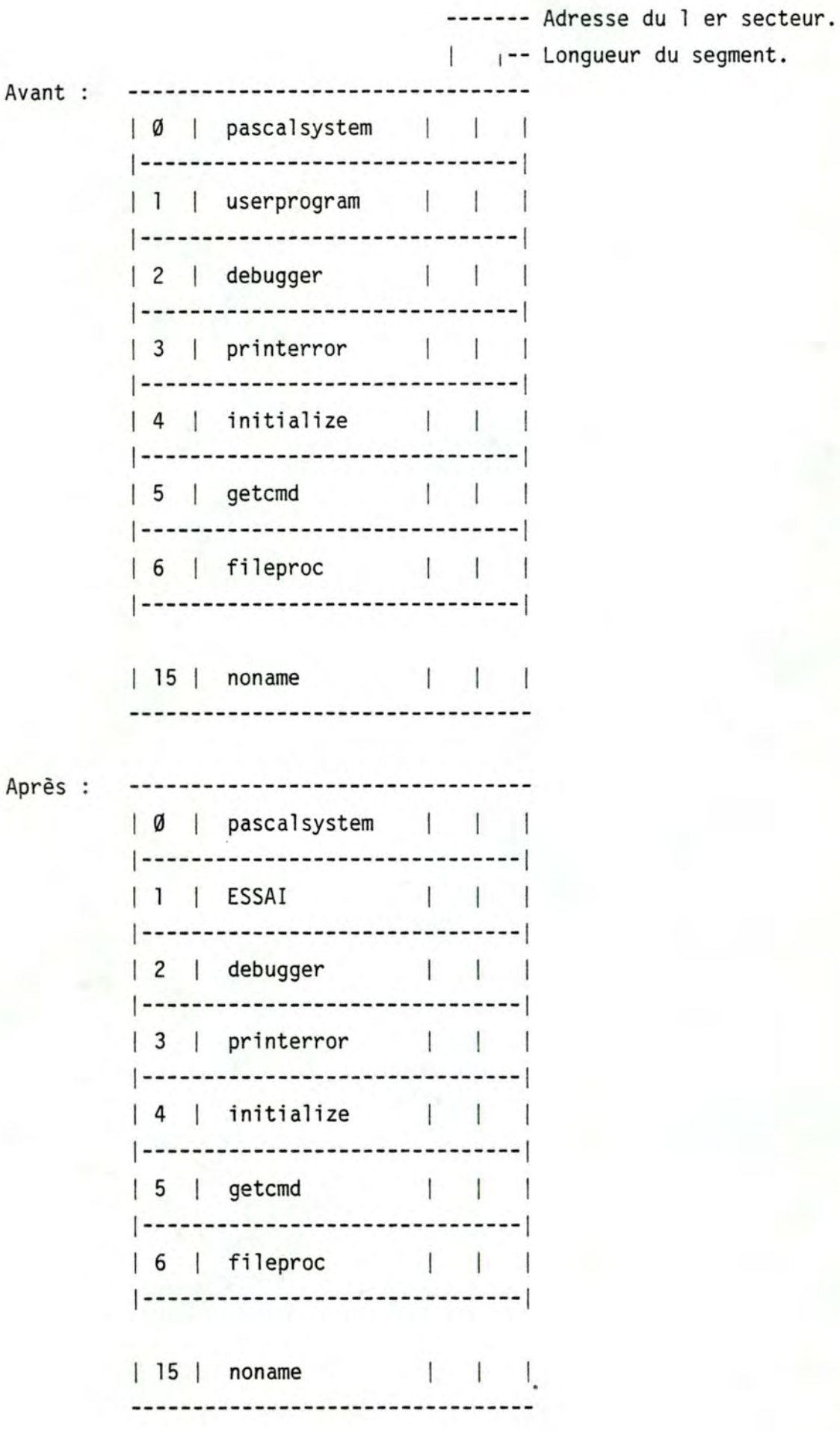
La déclaration du segment numéro 1 sera :

```
SEGMENT PROCEDURE USERPROGRAM;
BEGIN
   WRITELN('no user program');
END;
```

Lors de la demande d'exécution d'un programme (commande 'X' = 'Execute'), l'interpréteur de commandes vérifie si le fichier contenant le programme à exécuter est bien présent et vérifie s'il s'agit d'un fichier exécutable (procédure 'LOAD' de la procédure "GETCMD") .

Si les conditions sont remplies pour pouvoir exécuter le programme, l'interpréteur de commandes (procédure 'LOAD' de la procédure 'GETCMD') lit le premier bloc du fichier à exécuter (dictionnaire des segments) et le charge dans la table des segments. Le programme principal du programme utilisateur commence toujours au segment 1, procédure 1.

On trouvera sur la figure ci-dessous la structure de la table de segments pendant l'exécution de GETCMD (avant et après l'exécution de la procédure 'LOAD') : (La table de segments ne contient pas les noms des segments. Nous avons placé ceux-ci en regard des numéros correspondants pour des raisons de clarté)


------- Adresse du 1 er secteur.
| |-- Longueur du segment.
Avant :
Ø	pascalsystem		
1	userprogram		
2	debugger		
3	printerror		
4	initialize		
5	getcmd		
6	fileproc		
15	noname		
Après :
Ø	pascalsystem		
1	ESSAI		
2	debugger		
3	printerror		
4	initialize		
5	getcmd		
6	fileproc		
15	noname		

Il y a eu remplacement des références des segments du système d'exploitation par les références des segments du programme utilisateur. Comme on peut le voir sur la figure 5. , les références des segments du noyau du système d'exploitation qui n'ont pas d'équivalent dans le programme utilisateur (même numéro logique) ne sont pas détruites.

Pour lancer l'exécution de ce programme, il suffit d'exécuter une instruction CXP Ø1 Ø1 , ce qui est fait par le programme principal lors de l'exécution de la ligne contenant l'instruction "USERPROGRAM".

S'il y avait eu un appel à la procédure 1 du segment 1 sans que la table des segments ne soit modifiée (ce qui est peu probable, mais peut se présenter lorsque la première commande d'une session est 'U' (USER); Cette commande a normalement pour effet d'exécuter à nouveau le programme qui vient de se terminer. Dans le cas d'une première commande d'une session, la table des segments n'a pas encore été modifiée et contient toujours les références du segment 1 du programme "PASCALSYSTEM", c'est-à-dire la procédure "USERPROGRAM") ; le message

NO USER PROGRAM

apparaîtrait à l'écran, pusqu'il s'agit de la procédure se trouvant normalement dans le segment 1.

7. Création d'un debugger [i].

La rédaction d'un debugger poursuit deux objectifs :

a) Il s'agit d'un programme très lié au système d'exploitation. L'écriture d'un debugger permet donc de bien cerner les finesses du système d'exploitation.

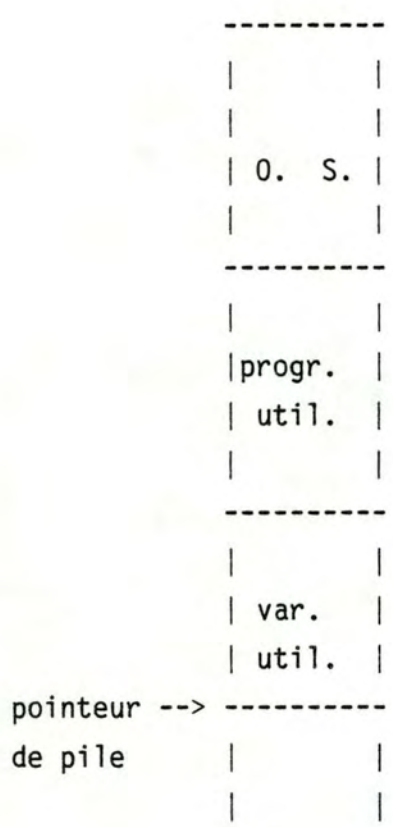
b) L'ensemble du système d'exploitation tel qu'il est vendu possède la commande permettant l'appel du debugger. Il n'est cependant pas livré de debugger.
Le programme décrit ci-dessous comble cette lacune.

Le noyau central du système d'exploitation ('PASCALSYSTEM') se trouve en mémoire en permanence.
Il est rappelé que le programme utilisateur est considéré par le système d'exploitation comme une procédure (Tout simplement parce que le système d'exploitation appelle le programme utilisateur par l'instruction Pascal "USERPROGRAM").

---

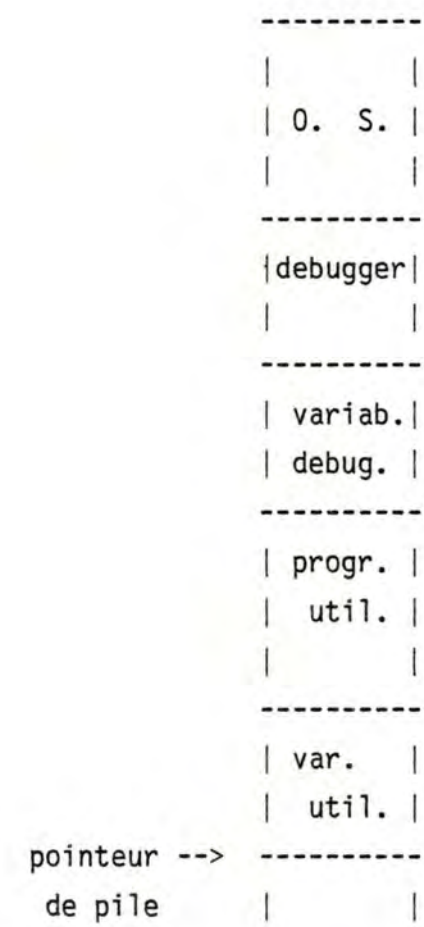
[i] Il n'est évidemment pas question de programmer un "DEBUGGER" complet mais simplement un petit programme permettant quelques fonctions de base, ceci afin de montrer ce qu'il est possible de faire.

Situation de la pile lors de l'exécution d'un programme utilisateur:



Le debugger, pour contrôler le programme utilisateur, appellera également celui-ci comme une procédure et se trouvera dans la pile entre le système d'exploitation et le programme utilisateur.

Situation de la pile pendant l'exécution du programme utilisateur, à l'aide du debugger:



Il doit donc y avoir d'abord chargement du debugger, ensuite le debugger appellera le programme utilisateur par une activation de la procédure Ø du segment 1. Le debugger sera placé dans le segment 2 qui est reservé à cet effet.

A ce moment, le programme utilisateur s'exécutera.
Si, lors de la compilation, on a spécifié l'option (*D+*) , le compilateur aura généré un point d'arrêt dans le code objet, et ce, pour chaque ligne du programme source compilée [i].

---

[i] Un point d'arrêt est une instruction de la Pseudo-machine. Lors de la rencontre de cette instruction, l'interpréteur devra rendre la main au debugger; ceci est facilement réalisable en langage machine par un appel à la procédure Ø du segment 2.

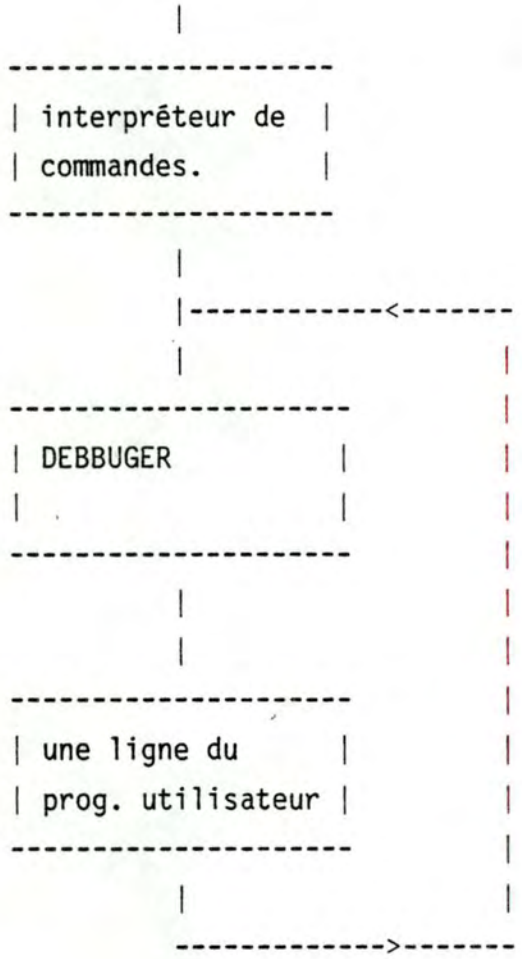
Le lancement du debugger à partir du système d'exploitation a été prévu par les concepteurs de ce système. Il s'agit de la commande 'D' de l'interpréteur de commande, cette commande activant la procédure Ø du segment 2.

Le debugger sera activé lors de la rencontre d'un point d'arrêt.
Les fonctions suivantes ont été implémentées :

-Trace.

-Pas à pas.

-Arrêt sur un point d'arrêt déterminé.



Ecriture du DEBUGGER.

Le debugger doit être considéré par le système d'exploitation comme une procédure. De plus le debugger doit pouvoir invoquer le programme utilisateur comme une procédure.

On va donc réécrire un programme complet de manière à effectuer correctement les références nécessaires. Après la compilation, on remplacera dans le programme SYSTEM.PASCAL, le segment 2 par le segment 2 du nouveau programme. Lors de l'appel du debugger, le programme activera la procédure Ø du segment 2 sans savoir qu'il y aura eu échange entre l'ancien segment 2 et le debugger nouvellement écrit.

Squelette du nouveau programme 'PASCALSYSTEM' contenant le debugger.
(Le listing complet se trouve en annexe 12) .

```
program nouveaupascalsystem;

segment procedure userprogram;
begin
end;

segment procedure debugger;
begin

  ici se trouvera le texte du programme debugger

end;

begin (* ces 2 instructions représentent le programme pascalsystem *)
end.
```

Lors de l'écriture de ce programme, la procédure debugger sera la seule à être écrite complètement, les autres procédures ne devant figurer que par leurs noms et les paramètres reçus. De cette manière, lors du remplacement du segment 2 par le debugger, toutes les références seront satisfaites (car pour le remplacement d'un segment, les numéros des segments et des procédures sont les seuls concernés) et les différents appels se feront correctement (car le nombre de paramètres sera correct).

Le remplacement d'un segment se fera à l'aide de l'utilitaire LIBRARY.CODE.

Chapitre 3.

Les entrées-sorties.

1. Introduction.

Dépourvu de périphériques d'entrées-sorties, un micro-ordinateur ne présente que peu d'intéret.
En effet, comment pourrait-il communiquer avec l'environnement extérieur ?

La gestion des entrées-sorties prend de plus en plus d'importance aux yeux des constructeurs de micro-ordinateurs.
Pour s'en convaincre, il suffit de feuilleter un magazine spécialisé pour voir que dans leur publicité, les constructeurs insistent davantage sur les possibilités d'entrées-sorties de leur matériel plutôt que sur la puissance de calcul de l'unité centrale.

De plus, l'évolution conduit actuellement à la conception de réseaux de micro-ordinateurs.
Dans ces conditions, il est important de pouvoir gérer un grand nombre de périphériques différents.

2. Organisation physique des entrées-sorties.

Généralement, les micro-ordinateurs ne possèdent qu'un seul microprocesseur.
Les interfaces des entrées-sorties sont le plus souvent constituées d'un minimum de matériel, et d'ailleurs, le plus souvent, basées sur l'utilisation d'un seul circuit intégré spécialisé.
Chaque famille de microprocesseurs comporte les circuits suivants :

- Pour les transmissions en série, un circuit effectuant la conversion parallèle-série et série-parallèle. Les décalages sont pilotés par une horloge dont la fréquence détermine le débit binaire.

- Pour les transmissions parallèles, un circuit servant de tampon entre la périphérie et le bus interne du micro-ordinateur.
Ces circuits sont capables de mémoriser un mot de données et possèdent la logique nécessaire pour un échange par poignée de main.

- Pour la gestion des mémoires de masse magnétique, un circuit effectuant automatiquement le positionnement de la tête, la lecture ou l'écriture d'un secteur et le contrôle de la validité des données transférées.

Ces circuits intégrés signalent la fin d'une opération par une interruption ou par le positionnement d'un bit dans le registre d'état de l'interface.

Voici schématisé un exemple de transmission de données en série à l'aide d'un circuit spécialisé sans utiliser d'interruptions : (Pour un schéma complet, il faudra ajouter un test pour savoir si la transmission est terminée)

initialiser l'interface (nombre de bits par
caractère, vitesse de transmission, ...).

TRANSM : tester si le caractère précédent est déjà transmis

si oui : écrire le caractère suivant dans le
registre correspondant de l'interface.

aller en TRANSM.

La gestion peut-être nettement plus complexe si une procédure "poignée de main" est utilisée.

Une procédure utilisant les interruptions ne différerait pratiquement pas de l'exemple précédent car l'ordinateur est monoprogrammé. Pour profiter de l'avantage des interruptions, une mise en oeuvre beaucoup plus complexe devrait être réalisée, et ce pour un rendement faible.

Une possibilité beaucoup plus intéressante, nécessitant moins d'interventions du microprocesseur serait l'utilisation de l'accès direct mémoire.
Dans ce cas, le micro-ordinateur utilise toujours les mêmes circuits d'interface, mais utilise également un contrôleur de DMA.
Ce contrôleur est en fait un microprocesseur spécialisé. Il ne peut jamais fonctionner seul. Il doit toujours être initialisé par le processeur principal.

A titre d'exemple, voici le schéma d'une transmission série utilisant un contrôleur de DMA :

initialisation de l'interface.

initialisation du contrôleur de DMA

adresse du premier caractère à transmettre
nombre de caractères à transmettre.

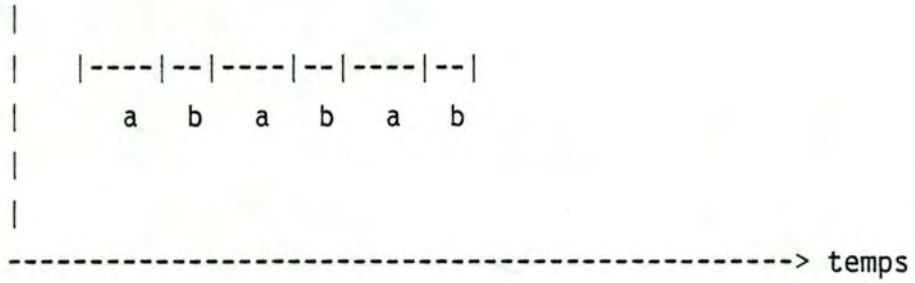
lancement du transfert.

A partir de ce moment, le microprocesseur principal pourra se consacrer à une autre tâche .Selon le mode de fonctionnement du contrôleur de DMA, le microprocesseur sera peu ou pas du tout ralenti (transparent, vol de cycle, ...) .
Il sera averti de la fin du transfert soit par une interruption, soit par le positionnement d'un bit dans le registre d'état du contrôleur de DMA.

Actuellement, les contrôleurs de DMA sont encore peu utilisés, non en raison de leur coût (environ 500 francs au détail), mais en raison des difficultés de mise en oeuvre.
Les périphériques qui profiteraient le plus d'un contrôleur de DMA sont les mémoires de masse. En effet, de très grandes quantités d'informations sont échangées, ralentissant considérablement le microprocesseur principal.

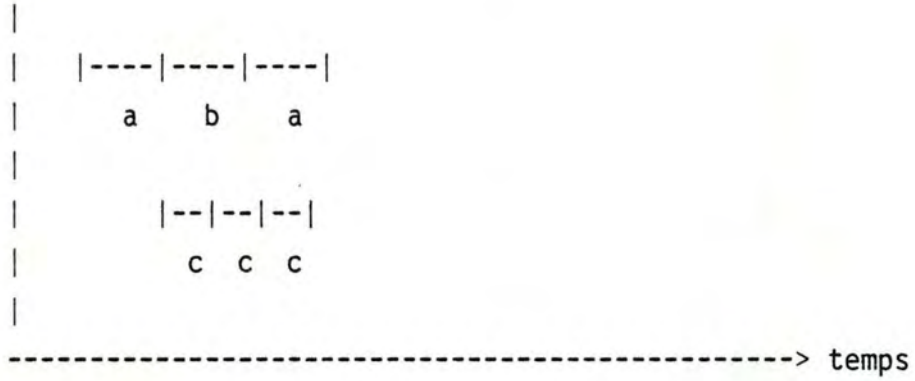
Même dans un environnement monoprogrammé, un contrôleur de DMA peut se justifier. Pour s'en convaincre, imaginons un traitement effectué sur un fichier séquentiel :

Sans DMA :



a : chargement buffer.
b : traitement buffer.

Avec DMA :



a : chargement buffer 1 en DMA.
b : chargement buffer 2 en DMA.
c : traitement buffers.

Il faut remarquer que sans contrôleur de DMA, l'utilisation de plusieurs buffers n'apporterait pas d'amélioration, alors qu'avec l'utilisation d'un contrôleur de DMA, l'utilisation d'un plus grand nombre de buffers peut être plus intéressante.

3. L'organisation des entrées-sorties sous UCSD Pascal.

Le système d'exploitation UCSD Pascal distingue deux types de périphériques :

- Les périphériques adressables.

- Les périphériques non adressables.

La gestion des périphériques adressables.

Ce point sera abordé dans le chapitre suivant.

La gestion des périphériques non adressables.

Sous UCSD Pascal, les périphériques suivants sont prédéfinis :

- CONSOLE:
  SYSTERM: Il s'agit du clavier et de l'écran cathodique, avec ou sans écho.

- PRINTER: Il s'agit d'une imprimante.

- REMIN:
  REMOUT: Il s'agit d'entrée et sortie série, normalement destinées au raccordement (direct ou via un modem) à un autre ordinateur.

Il existe différentes instructions destinées à gérer ces périphériques (Nous donnons ci-dessous, les noms des procédures tels qu'ils sont demandés par le compilateur Pascal) :

- UNITREAD,
  UNITWRITE : Il s'agit de fonctions permettant la lecture ou l'écriture d'un certain nombre de caractères vers un périphérique donné.

- UNITBUSY,
  UNITWAIT,
  UNITCLEAR : Il s'agit de fonctions permettant de savoir si un périphérique est occupé, d'attendre qu'un échange avec un périphérique soit terminé et d'arrêter un échange avec un périphérique.

Ces cinq procédures reçoivent comme paramètres le numéro du périphérique demandé, l'adresse de la mémoire centrale où se trouvent les données et la taille de la chaîne de caractères à transmettre.
Elles font partie intégrante du jeu d'instructions de la machine virtuelle, sont écrites en assembleur de l'ordinateur hôte et sont appelées directement par l'interpréteur de CODE-P.

Les récupérations d'erreurs lors des entrées-sorties.

Le compilateur Pascal génère, après chaque opération d'entrée-sortie, le code nécessaire pour vérifier la réussite de l'opération. En cas d'échec, ce code provoque une erreur d'exécution.

Une erreur lors d'une entrée-sortie peut se manifester de deux manières :

- Une erreur d'exécution entraînant l'arrêt du programme utilisateur et le retour à l'interpréteur de commandes.

- L'écriture d'un nombre dans une variable globale, connue du système d'exploitation et du programme utilisateur (IORESULT).
Le programme utilisateur peut donc, après chaque opération d'entrée-sortie, tester cette variable.
Si la valeur est égale à Ø, l'échange s'est fait correctement, dans les autres cas, la valeur contenue dans IORESULT indiquera le type d'erreur rencontré.

Pour utiliser cette deuxième solution, l'utilisateur doit le signaler au compilateur par l'ordre : '(*$I-*)' .

Le lecteur trouvera à la page suivante la liste complète des erreurs d'entrée-sortie diagnostiquées par le système. [extraits de 4]

I/O ERROR MESSAGES

Error Number	Error Message and Description
Ø	No error
1	Diskette has bad block: parity error (CRC). (Not used on the Apple.)
2	Bad device (volume) number.
3	Bad mode: illegal operation. (For example, an attempt to read from PRINTER:.)
4	Undefined hardware error. (Not used on the Apple.)
5	Lost device: device is no longer on-line, after successfully starting an operation using that device.
6	Lost file: file is no longer in the diskette directory, after successfully starting an operation using that file.
7	Bad title: illegal filename. (For example, filename is more than 15 characters long.)
8	No room: insufficient space on the specified diskette. (Files must be stored in contiguous diskette blocks.)
9	No device: the specified volume is not on-line.
1Ø	No file: the specified file is not in the directory of the specified volume.
11	Duplicate file: attempt to re-write a file when a file of that name already exists.
12	Not closed: attempt to open an already-open file.
13	Not open: attempt to access a closed file.
14	Bad format: error in reading real or integer. (For example, your program expects an integer input but you typed a character.)
15	Ring buffer overflow: characters are arriving at the Apple faster than the input buffer can accept them.

La gestion d'autres périphériques.

Il est, bien entendu, possible de gérer d'autres périphériques, mais cela impliquera généralement une programmation en assembleur de la gestion de ce nouveau périphérique.
L'utilisation, en langage évolué, d'un périphérique qui n'a pas été prévu lors de la conception du système d'exploitation posera de très gros problèmes.

4. La gestion des entrées-sorties dans le micro-ordinateur APPLE II.

Un des atouts du micro-ordinateur APPLE II est une grande possibilité d'adaptation. Le circuit imprimé principal de cet ordinateur comprend en effet 8 connecteurs pouvant recevoir des extensions aussi diverses que des interfaces séries et parallèles, des modems, des contrôleurs de minidisquettes, des interfaces pour moniteur couleur, etc... .
Encore faut-il que les programmes existent pour gérer correctement toutes ces interfaces.

Voyons ce qui existe sous UCSD :

Sous UCSD, chaque connecteur est destiné à recevoir un périphérique particulier.
Par exemple, le connecteur 1 reçoit l'interface pour l'imprimante, le connecteur 2 reçoit l'interface pour le périphérique REMIN:-REMOUT: , etc ... .

Le lecteur trouvera aux deux pages suivantes :

page 81 : la liste des différents périphériques gérés par le système UCSD Pascal.

page 82 : Pour le micro-ordinateur APPLE II, les numéros des connecteurs suivis de la liste des interfaces qu'ils peuvent recevoir.

PASCAL I/O DEVICE VOLUMES

The Apple Pascal operating system assigns volume numbers and volume names to the various input/output devices as follows:

Volume Number	Volume Name	Description of Input/Output Device
#Ø:		(not used)
#1:	CONSOLE:	Screen & keyboard (echo on input)
#2:	SYSTERM:	Screen & keyboard (no echo on input)
#3:		(not used)
#4:	<diskette name>:	Boot disk drive (slot 6, drive 1)
#5:	<diskette name>:	2nd disk drive (slot 6, drive 2)
#6:	PRINTER:	Printer (card in slot 1)
#7:	REMIN:	Remote input (card in slot 2)
#8:	REMOUT:	Remote output (card in slot 2)
#9:	<diskette name>:	5th disk drive (slot 4, drive 1)
#1Ø:	<diskette name>:	6th disk drive (slot 4, drive 2)
#11:	<diskette name>:	3rd disk drive (slot 5, drive 1)
#12:	<diskette name>:	4th disk drive (slot 5, drive 2)

APPLE I/O DEVICE SLOTS

In using an Apple computer with the Apple Pascal operating system, the Apple's peripheral equipment slots are assigned as follows:

Apple Slot	Input/Output Device and Card Assigned to That Slot	Apple Pascal Operating System Use
Ø	Apple Language Card *	Stores interpreter and I/O system
1	Printer (Communications Interface Card, Serial Interface Card, or Parallel Printer Interface Card)	PRINTER: or #6:
2	Modem, Apple-to-Apple communication, etc. (Communications Interface Card or Serial Interface Card)	REMIN: or #7: REMOUT: or #8:
3	External terminal (Communications Interface Card. Use of Serial Interface Card is tolerated.)	CONSOLE: or #1: (with echo on input) SYSTERM: or #2: (without echo on input)
4	5th disk drive (Drive 1) 6th disk drive (Drive 2) (Disk Controller Card)	diskette name: or #9: diskette name: or #1Ø:
5	3rd disk drive (Drive 1) 4th disk drive (Drive 2) (Disk Controller Card)	diskette name: or #11: diskette name: or #12:
6	Boot disk drive (Drive 1) * 2nd disk drive (Drive 2) (Disk Controller Card)	diskette name: or #4: diskette name: or #5:
7	Must not contain a disk drive (Do not install a Disk Controller Card here.)	
	Apple keyboard and screen (only if there is no Communications Interface Card in slot #3)	CONSOLE: or #1: (with echo on input) SYSTERM: or #2: (without echo on input)

Note: An asterisk (*) indicates a device which is REQUIRED to be present in that slot.

Chaque interface possède une petite PROM contenant un petit sous-programme de gestion de l'interface. Ce petit sous-programme est utilisé par l'interpréteur BASIC mais pas par le système d'exploitation UCSD Pascal.

Lors de l'initialisation du système, le programme teste le troisième et le cinquième caractères de chaque PROM et peut de la sorte savoir si le périphérique concerné est présent ou pas, et à quel type d'interface il est fait référence.
Le système conserve en mémoire centrale,et pour chaque connecteur, un byte destiné à spécifier le type d'interface associé à ce connecteur.

Sous UCSD Pascal, le BIOS contient les sous-programmes destinés à gérer toutes ces interfaces.
Lors d'une opération d'entrée-sortie avec un certain périphérique, le BIOS teste le byte caractérisant l'interface et invoque le sous-programme adéquat.

Avec ce système, il y aura en permanence, en mémoire centrale, tous les sous-programmes de gestion d'interface.
De la place en mémoire centrale sera donc perdue si des sous-programmes ne sont pas utilisés.

Une solution plus élégante aurait été de permettre une configuration du système d'exploitation en fonction des interfaces utilisées, mais plusieurs raisons ont plaidé contre ce choix :

- La base de la pile contenant les procédures et les zones de données doit être connue de l'interpréteur de P-code.
Si la taille du BIOS n'est pas fixe, un calcul d'adresses sera nécessaire pour effectuer un adressage dans la pile.
Le microprocesseur 65Ø2 de l'APPLE II ne permet que très difficilement d'effectuer un adressage avec double indexation.
Cette double indexation aurait de toute manière été très coûteuse en temps d'exécution.

Il aurait également été possible de livrer l'interpréteur sous forme symbolique et de laisser à l'utilisateur le soin de procéder à l'assemblage. Un assembleur conditionnel tel celui fourni avec le système d'exploitation aurait alors permis la personnalisation de l'interpréteur de P-code. Cela était inacceptable d'un point de vue commercial.

- Ce micro-ordinateur est destiné à être utilisé par des personnes ayant peu de connaissances en informatique.
Le choix a donc peut-être été fait pour des raisons de facilité.

- La mémoire centrale du micro-ordinateur APPLE II est scindée en deux parties :

ØØØØ à BFFF : mémoire RAM.

CØØØ à CFFF : entrées-sorties.

DØØØ à FFFF : mémoire RAM.

Ceci est dû au fait que les entrées-sorties sont projetées en mémoire et à certaines contraintes du matériel qui empêchaient de placer les entrées-sorties soit aux adresses basses, soit aux adresses hautes.

Les concepteurs du système d'exploitation ont choisi de placer l'interpréteur de P-code aux adresses comprises entre DØØØ et FFFF, la pile et le tas aux adresses comprises entre ØØØØ et BFFF.
Dès lors, il ne servait à rien de réaliser un interpréteur de P-code et un BIOS aussi petits que possible. Il suffisait qu'ils occupent un espace inférieur à 16 K octets pour ne pas empiéter sur la zone de la mémoire centrale réservée à la pile et au tas.

Nous allons montrer comment sont gérés deux périphériques importants du micro-ordinateur APPLE II :

- Le clavier.

- Les mini-disquettes.

Ensuite nous montrerons comment introduire une gestion du clavier par interruptions.

A la fin de ce paragraphe, nous introduirons une astuce qui permet de gagner 1 K. octets lors de l'utilisation d'un terminal extérieur.

La gestion du clavier de l'APPLE II.

Le clavier est un clavier ASCII standard. Le code du caractère frappé est présenté à un port d'entrée parallèle.
Tous les caractères du code ASCII comportent 7 bits.
Le port d'entrée ayant 8 bits (par la suite, nous numéroterons les bits de Ø à 7) , le huitième bit est utilisé pour signaler la présence d'un caractère (bit numéro 7) .
L'adresse du port d'entrée est CØØØ en hexadécimal. Après la lecture d'un caractère, il est nécessaire de remettre à Ø le bit 7. Cela se fera par une simple écriture à l'adresse CØ1Ø.

Voici l'exemple d'une lecture d'un caractère avec test pour voir si un caractère est présent :

```
LECTURE : LDA CØØØ    ; Lecture d'un caractère.
          BPL LECTURE ; Test du bit 7. Si pas de
                        caractère, branchement en LECTURE.
          STA CØ1Ø    ; Remise à Ø du bit 7.
          AND °7F     ; Masquage du bit 7.
          ...         ; L'accumulateur contient le code
                        du caractère frappé au clavier.
```

Le système UCSD Pascal gère un tampon de 78 caractères lus au clavier. Malheureusement, la gestion du clavier se fait par balayage.
Lors de l'exécution de certaines instructions P-code, il y a lecture du clavier et si un caractère est présent, celui-ci est placé dans le tampon. Cette procédure entraîne une perte de temps importante (cette perte de temps varie entre Ø et 5 % en fonction des instructions P-code exécutées [i]) .

Un meilleur procédé aurait été de générer une interruption lors du passage à 1 du bit 7, indiquant la présence d'un caractère.
La routine de service de l'interruption aurait placé le caractère dans le tampon avant le retour au processus principal.
De cette manière, il n'y aurait plus eu de perte de temps pour des balayages inutiles du clavier (s'il n'y a pas de caractère présent) .

Malheureusement, nous verrons lors de l'étude de la gestion des unités de mini-disquettes que la gestion des interruptions n'est pas simple.

---

[i] Pour cette estimation, quelques programmes ont été essayés, d'une part avec le système d'exploitation standard, d'autre part à l'aide du système d'exploitation où les routines de balayage ont été supprimées.

La gestion des unités de mini-disquettes.

Pour maintenir aussi bas que possible le prix de revient du matériel, la plus grande partie de la gestion des unités de mini-disquettes est faite par un programme exécuté par le microprocesseur principal lui-même.
C'est ainsi que c'est le microprocesseur qui gère l'entièreté du processus de positionnement de la tête de lecture-écriture (gestion d'un moteur pas-à-pas composé de quatre bobinages) et qui effectue une partie de la conversion série-parallèle nécessaire aux lectures et aux écritures sur le support magnétique.

La lecture d'un secteur se fait comme suit :

- Positionner si nécessaire la tête de lecture-écriture.

- Le microprocesseur lit une suite de caractères provenant de la piste concernée. Il reconnaît le début du secteur par une suite spéciale de caractères suivie de l'identification du secteur.
A partir de ce moment, il recueille régulièrement les caractères, et ce à intervalles réguliers (ces intervalles sont déterminés par une boucle programmée) .

Voyons ce qui se passe en cas d'interruption pendant la lecture ou l'écriture d'un secteur :

Lecture :

Le microprocesseur a lu quelques caractères du secteur. Il est alors interrompu au milieu d'une boucle d'attente programmée. A la fin de l'interruption, il reprendra ses lectures mais à un mauvais endroit de la disquette, car celle-ci a continué à tourner pendant toute la durée de la routine de service de l'interruption.

Il est bien évident que la lecture sera erronée (sauf par un hasard extraordinaire, si la durée de la routine de service de l'interruption correspond à un multiple de la durée de rotation de la disquette).
Cette mauvaise lecture sera détectée par le contrôle de CRC ou de Check-sum [i], et une nouvelle lecture de ce secteur sera demandée.

Ce procédé est néanmoins dangereux, car bien que très efficaces, les contrôles ne sont pas infaillibles, et à la longue, une mauvaise lecture pourrait ne pas être détectée.

Ecriture :

Ce cas est beaucoup plus dangereux, car après la routine de service de l'interruption, le sous-programme d'écriture continuera, en écrasant d'autres informations utiles.

---

[i] CRC : Contrôle par redondance cyclique.
Check-sum : Somme de contrôle.

Voir [27] page 179, [28] page 153.

Il sera dès lors obligatoire d'interdire les interruptions pendant une partie de la durée nécessaire aux accès aux disquettes. La partie minimale à masquer sera les deux sous-programmes de lecture et d'écriture d'un secteur sur les disquettes [i] .

Pendant les accès aux disquettes, le microprocesseur ne pourra rien faire d'autre. Nous avons indiqué au paragraphe précédent les avantages retirés lors d'une gestion en DMA des unités de mini-disquettes (possibilités de lire ou d'écrire le contenu d'un buffer pendant le traitement d'un autre buffer).

La disquette tourne à la vitesse de 3ØØ tours par minute. La lecture ou l'écriture d'un secteur prend donc au maximum Ø.2125 seconde (Ø.2 seconde pour une rotation de disquette + Ø.Ø125 seconde pour la lecture d'un secteur; on n'a pas tenu compte du temps de positionnement de la tête) .

---

[i] En réalité, il faudra masquer toutes les routines faisant intervenir des délais déterminés par des boucles d'attente programmées.
On peut toutefois noter que pour le micro-ordinateur APPLE II, seules les gestions des mini-disquettes et du haut-parleur font intervenir des boucles programmées.

Modification minimale permettant une gestion du clavier par interruptions.----------

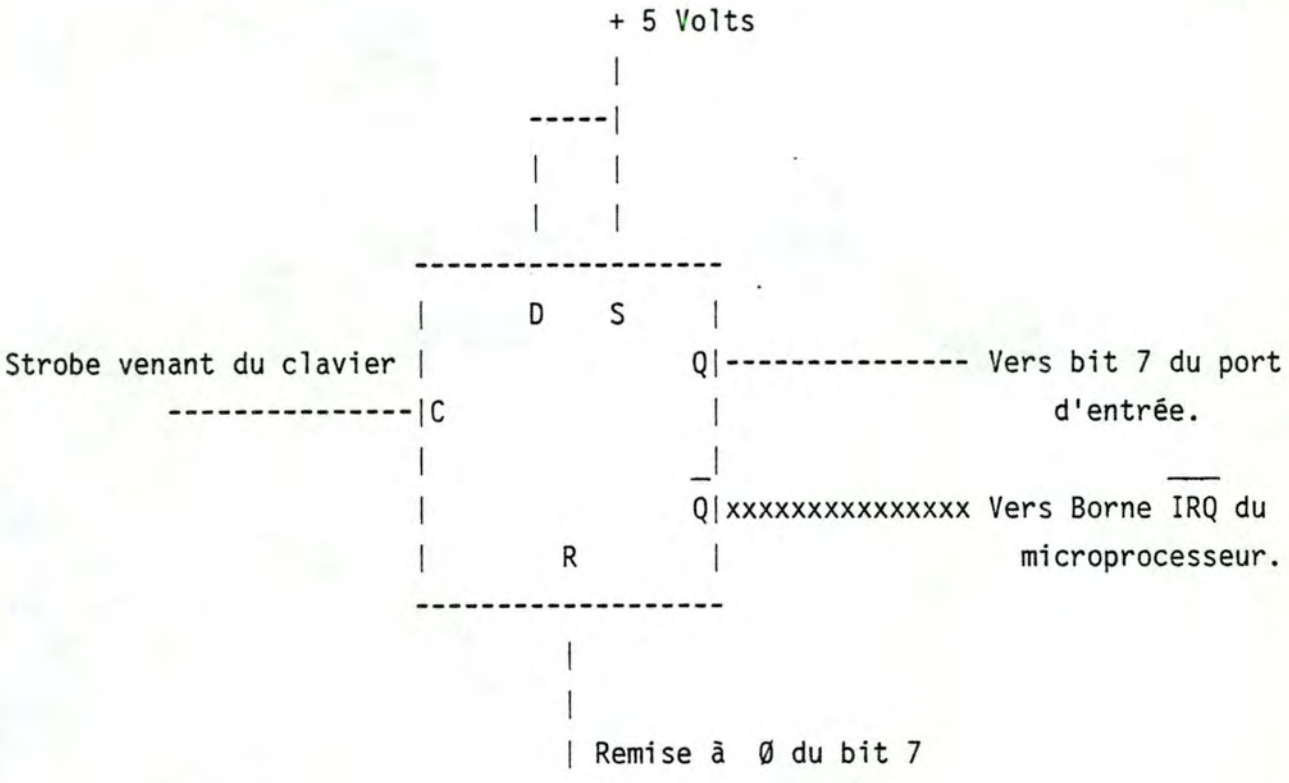
Reprenons le cas du clavier envisagé précédemment, mais cette fois, avec une gestion à l'aide d'interruptions.
Il faudra masquer les interruptions pendant les sous-programmes faisant intervenir des délais programmés.
Ce masquage peut se faire simplement par l'instruction 'SEI' du microprocesseur 6502. Voici les instructions utilisées pour masquer et démasquer les interruptions :

```
Pour le masquage :          PHP  ; sauvetage du registre d'état
                            SEI  ; masquage des interruptions

Pour le démasquage :        PLP  ; récupération du registre
                                   d'état
```

La vitesse maximale d'entrée des caractères sera d'environ 4.7 caractères par seconde (en fonction de la plus longue période de masquage) , ce qui est peu, et il risque d'y avoir des caractères perdus.

Voici la modification minimale du matériel à effectuer pour générer une interruption lorsqu'un caractère est frappé au clavier (1 seul fil ajouté) :



xxxxxxxx : Représente la connexion à ajouter.

Circuit 74LS74 de la série TTL de Texas-Instruments.
Il s'agit d'une double bascule R/S.
Ce circuit se trouve en position B1Ø sur le circuit imprimé principal du micro-ordinateur APPLE II .

Fonctionnement du circuit :

Sans interruption : lors de la frappe d'un caractère au clavier, une impulsion en provenance du clavier positionne la bascule avec un niveau haut sur la sortie 'Q'.

Par une lecture à l'adresse CØØØ (voir ci_dessus la gestion du clavier), le microprocesseur détermine la présence ou l'absence d'un caractère (test du bit 7). Si un caractère est présent, une remise à Ø de la bascule est nécessaire.
Pour ce faire, un décodeur d'adresse est simplement connecté à l'entrée 'R' (Reset) de la bascule. Une écriture à l'adresse CØ1Ø repositionnera la sortie 'Q' à Ø.

Avec interruption : lors de la frappe d'un caractère au clavier, la bascule est positionnée de la même manière que ci-dessus.

La sortie '$\overline{Q}$' est positionnée à Ø, ce qui entraîne un niveau bas sur l'entrée IRQ du microprocesseur, ce qui entraîne l'exécution d'une routine de gestion des interruptions.
La routine de gestion des interruptions devra remettre la bascule à Ø par une écriture à l'adresse CØ1Ø.
Le gérant des interruptions peut tester le bit 7 de l'adresse CØØØ pour savoir si l'interruption provient du clavier.

Bien entendu, ceci est une modification minimale. Dans le cas d'un emploi généralisé des interruptions, un encodeur de priorité d'interruptions accélérerait le traitement de celles-ci.

Le cas du connecteur numéro 3.

La gestion de l'écran du micro-ordinateur APPLE II est peu adaptée à la réalisation de gros programmes ou au traitement de textes.
En effet, cet écran ne comprend que 24 lignes de 40 caractères ce qui ne correspond qu'à la moitié d'une page d'écran normale.
De plus on ne peut afficher que les majuscules, bien que sous forme interne, le micro-ordinateur puisse manipuler les codes correspondant aux minuscules; dans ce cas, ces codes sont affichés sous forme de majuscules inversées (noir sur fond blanc) .

Il est cependant possible d'effectuer un balayage latéral de l'écran, celui-ci se comportant comme une fenêtre se déplaçant sur un écran de taille normale (24 lignes de 80 caractères) .
Cette possibilité n'est pas très pratique et la plupart des utilisateurs n'emploient que la moitié gauche de l'écran.

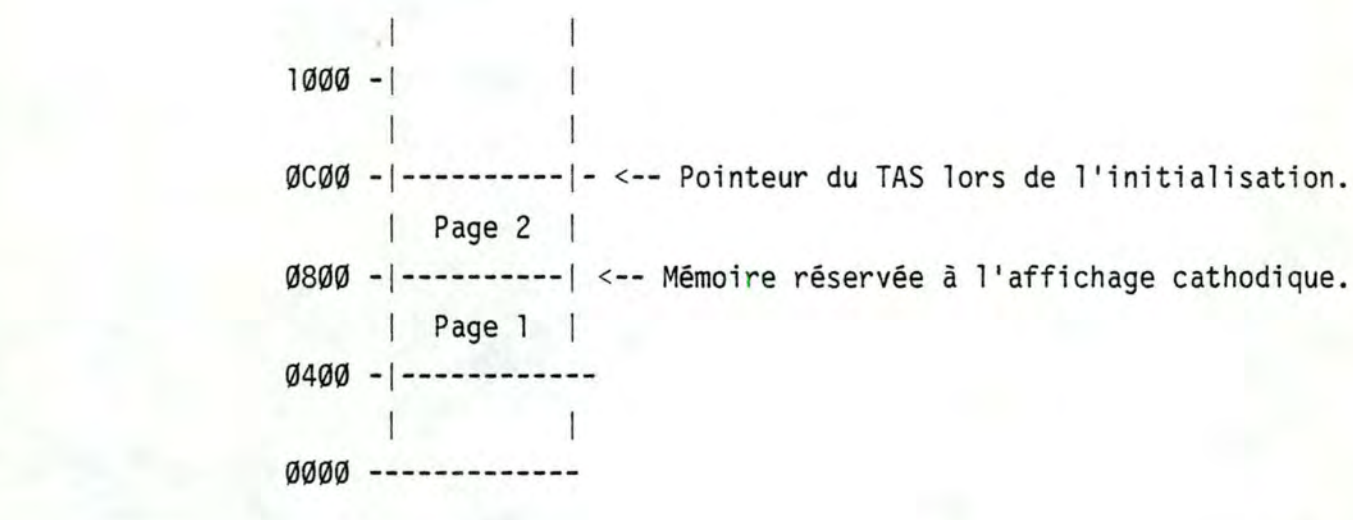
L'affichage se fait par un balayage en DMA de la mémoire centrale comprise entre 0400 et 07FF [i] . L'écriture à l'écran se fait par une simple écriture en mémoire centrale [ii] .

---

[i] Toutes les adresses physiques sont données en hexadécimal sous la forme de quatre digits.

[ii] Principes des Vidéo-Ram ou V-RAM
Voir [18] .

Voici une carte mémoire des adresses comprises entre ØØØØ et 1ØØØ :



Pour les utilisations professionnelles du micro-ordinateur, il est possible d'utiliser un terminal extérieur. Celui-ci pourra avoir n'importe quelle dimension d'écran et pourra afficher les minuscules .

L'interface utilisée pour gérer cet écran sera une interface série rapide et sera placée dans le connecteur numéro 3.

Lors de la mise sous tension, le système reconnaît la présence de l'interface série dans le connecteur 3. Le byte caractérisant ce périphérique est donc initialisé en conséquence.
Si un terminal extérieur est utilisé, la mémoire comprise entre Ø4ØØ et Ø7FF peut être utilisée pour d'autres fonctions que l'affichage cathodique. On déplace alors le pointeur du TAS vers les adresses basses (nouvelle adresse : ØSØØ) .
De cette manière, on gagne 1 K octets de mémoire vive et l'écran (correspondant aux adresses comprises entre Ø4ØØ et Ø7FF) reste noir .

Il serait possible de gagner 2 K octets de mémoire vive (nouvelle adresse du pointeur du TAS : Ø4ØØ) mais dans ce cas, l'écran montrerait des signes cabalistiques correspondant aux 1Ø24 premiers octets du TAS.

Voici à titre d'illustration les instructions qui lors de l'initialisation du système, initialisent le pointeur du TAS (Uniquement l'octet de poids fort dont l'adresse est ØØ5B) :

```
          LDA DRAPEAU  ;Byte caractérisant le connecteur 3.
          AND #Ø4
          BEQ ECRAN
          LDA #Ø8      ;Si terminal extérieur.
          STA 5B
          BPL END
ECRAN     LDA #ØC      ;Si pas de terminal extérieur.
          STA 5B
END       ...
```

5. La gestion des entrées-sorties pour les micro-ordinateurs pouvant fonctionner sous le système d'exploitation CP/M.

CP/M [i] est de loin le système d'exploitation pour micro-ordinateur le plus répandu.
Malheureusement, il n'est prévu pour fonctionner qu'avec un microprocesseur 8Ø8Ø [ii] .

La diffusion de CP/M est facilitée par le fait que Digital Research, propriétaire des droits de CP/M, demande aux constructeurs de micro-ordinateurs d'écrire un BIOS spécifique pour chaque machine.
Ce BIOS est ensuite relié à l'ensemble du système d'exploitation.

Cette manière de procéder a permis d'adapter facilement CP/M à un très grand nombre de micro-ordinateurs.

---

[i] CP/M (Control Programs for Micro-computer) est une marque déposée de Digital Research.

[ii] Ce système d'exploitation peut également fonctionner avec les microprocesseurs 8Ø85, Z8Ø et NCS 8ØØ qui admettent le jeu d'instructions du 8Ø8Ø comme un sous-ensemble de leur jeu d'instructions.

Vu la diffusion de CP/M, et plutôt que de réécrire un BIOS pour les micro-ordinateurs utilisant comme microprocesseurs des 8Ø8Ø, 8Ø85, Z8Ø, les concepteurs du système d'exploitation UCSD ont préféré utiliser pour chaque type de machine, le BIOS utilisé par CP/M.
Ce Bios est relié à l'ensemble du système d'exploitation.

Grâce à cette méthode, il n'existe qu'un seul système d'exploitation UCSD pour les micro-ordinateurs pouvant fonctionner sous CP/M. Seul le BIOS change.

Chapitre 4.

La gestion des fichiers.

1. Introduction.

La mémoire centrale est d'un accès très rapide. Sa taille est cependant limitée, particulièrement sur les micro-ordinateurs. La quantité d'informations (par exemple résultats intermédiaires) qu'il est possible de conserver en mémoire centrale pendant l'exécution d'un programme est donc limitée.

De plus la mémoire centrale étant une mémoire volatile, le contenu sera perdu lors de la coupure de l'alimentation électrique de l'appareil.

Actuellement, la manière la plus simple de conserver les informations est de les enregistrer sur des supports magnétiques tels que bandes et diques magnétiques.

Outre la bande magnétique utilisée sous forme de cassettes, semblables aux cassettes utilisées pour l'enregistrement du son ou spécialisées pour un accès plus rapide aux informations, les micro-ordinateurs utilisent principalement des disquettes magnétiques, versions simplifiées des disques magnétiques en éliminant une série de contraintes mécaniques.

Depuis quelques mois, les utilisations de disques magnétiques véritables se multiplient en raison des avantages importants de ce type de support et de l'abaissement des coûts dû à une production de masse.

Rappelons que nous appelons fichiers, une collection d'informations réunies pour des raisons fonctionnelles.
Il peut s'agir de données, de résultats ou de programmes.

Il est habituel de découper le fichier en articles. Cette découpe s'applique bien aux fichiers de données et beaucoup moins bien aux fichiers de programmes.

Sur la mémoire externe, il est nécessaire de créer un découpage en blocs. l'utilisateur doit pouvoir ignorer ce découpage. Le système doit lui venir en aide en proposant un ensemble de programmes s'occupant de toute cette gestion.

C'est le rôle des procédures de gestion des fichiers.

2 Organisation d'une disquette.

Les mémoires de masse des micro-ordinateurs monopostes sont en général des minidisquettes souples d'un diamètre de 5.25 pouces (On rencontre également des disquettes souples d'un diamètre de 8 pouces, mais leur capacité et leur prix sont beaucoup plus élevés et de ce fait, on ne les trouve que sur les appareils de haut de gamme) [i] .

Une mini-disquette souple est un support magnétique circulaire de 5.25 pouces (13.3 cm) de diamètre, disposé dans une enveloppe protectrice carrée de 5.25 pouces de côté.
Sur ce support, les informations sont enregistrées sur des pistes circulaires concentriques. (Les minidisquettes habituelles contiennent 35, 4Ø et actuellement parfois 1ØØ pistes).

La tête de lecture-écriture est translatée le long d'un rayon par un mécanisme de positionnement, généralement un moteur pas à pas [36] .

---

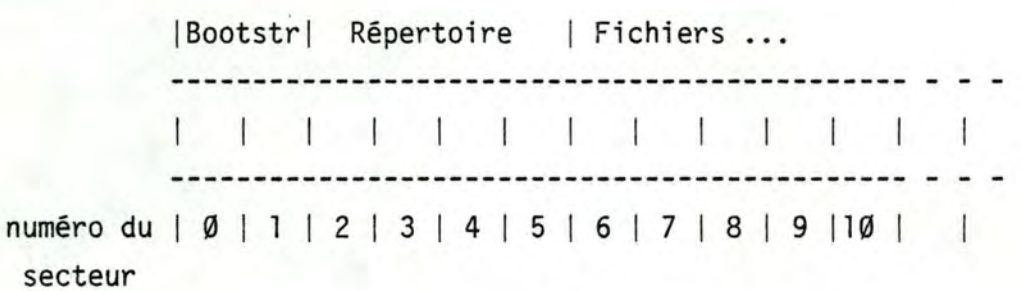
[i] L'exposé est basé sur l'utilisation de minidisquettes de 5.25 pouces. La gestion et l'utilisation des disquettes de 8 pouces sont identiques, excepté le nombre de pistes et de secteurs qui est plus grand.

Chaque piste est découpée en secteurs, généralement 10, 13, 16 secteurs. Chaque secteur contient 128 ou 256 bytes.
La capacité totale d'une mini-disquette varie de 80 K à 400 K octets selon le nombre de pistes et de secteurs.

3 Utilisation d'une mini-disquette par UCSD Pascal.

A l'origine, UCSD Pascal a été écrit pour des mini et micro-ordinateurs de DIGITAL EQUIPEMENT, PDP 11 et LSI 11. Sur ces machines, les secteurs des disques contiennent 512 octets. Pour des raisons de cohérence, UCSD Pascal simule des secteurs de 512 octets sur tous les autres ordinateurs sur lesquels il est implémenté.
La documentation du système d'exploitation emploie le mot 'bloc' pour désigner les secteurs simulés ou non de 512 octets.

Logiquement [i] , une disquette peut être représentée comme une suite de blocs de 512 octets (à partir de cet endroit, lorsque nous parlerons de bloc, nous sous-entendrons des secteurs de 512 octets) .



Lors de l'initialisation du système, un programme en ROM charge en mémoire et exécute le code objet situé sur les blocs 0 et 1. Ce programme a pour but de charger le système d'exploitation et de lancer l'exécution de l'interpréteur de commande.

[i] : Par opposition à la représentation physique.

Il existe sur toutes les disquettes un répertoire permettant l'accès à toutes les informations stockées sur la disquette. Ce répertoire contient également le nom de la disquette.

Le répertoire se trouve toujours au même endroit sur les disquettes (blocs numéros 2 à 5). Il est de longueur fixe (4 blocs).
Il peut contenir les références de 77 fichiers.

Voici la description 'Pascal' du répertoire [39] :

```
Directory=array[0..77] of record
            first_block:integer;
            last_block:integer;
            file_type:integer;
            file_name:string[15];
            num_byte:integer;
            last_date:integer;
          end;
```

Chaque enregistrement du tableau contient les références d'un fichier sauf le premier. Celui-ci contient les références de la disquette (nom de la disquette, nombre de blocs disponibles,numéro du premier bloc du premier fichier, etc ...) .

Les références des fichiers apparaissent dans le répertoire, dans l'ordre de placement des fichiers sur la disquette. Il est parfois nécessaire, en cas de création de fichier, d'insérer une nouvelle référence de fichier dans le répertoire, en décalant d'une position toutes les références qui la suivent.

L'espace d'un fichier est constitué d'un nombre de blocs contigus. Pour chaque fichier, le catalogue contient le numéro du premier bloc, le numéro du dernier bloc plus un, le type de fichier et la date de la dernière mise à jour.

Problèmes posés par la représentation contiguë.

La représentation contiguë ne nécessite pas de chaînage ou de table permettant de retrouver un bloc appartenant à un fichier, mais par contre, introduit des contraintes:

1. Une disquette peut contenir suffisamment d'espace pour recevoir un nouveau fichier, mais cet espace est scindé en plusieurs morceaux, tous trop petits pour recevoir ce fichier.

exemple [i] :

```
PASCALØ:
SYSTEM.PASCAL      43 23-Feb-82    6 Data
SYSTEM.MISCINFO     1  4-May-79   49 Data
SYSTEM.STARTUP      3 13-Nov-81   5Ø Code
SYSTEM.EDITOR      47 17-Dec-81   53 Code
SYSTEM.COMPILER    75 26-Feb-82  1ØØ Code
ACTUAL.TEXT         6  5-Mar-82  175 Text
LECTURE.TEXT       12  5-Mar-82  181 Text
ENTSOR.TEXT         8  5-Mar-82  193 Text
INITIALI.TEXT       4  5-Mar-82  2Ø1 Text
FONCTIONS.TEXT     16  5-Mar-82  2Ø5 Text
C65Ø2.TEXT          4  7-Mar-82  221 Text
ACTUAL.CODE        25  5-Mar-82  225 Code
< UNUSED >          2             25Ø
BUFFER.CODE         2 18-Mar-82  252 Code
< UNUSED >         1Ø             254
BUFFER.TEXT         4 18-Mar-82  264 Text
< UNUSED >          5             268
C65Ø2.CODE          2  7-Mar-82  273 Code
< UNUSED >          5             275
15/15 files, 22 unused, 1Ø in largest
```

---

[i] : Les listings illustrant ces exemples sont fournis par la tâche L(ist de l'utilitaire FILER.

Dans l'exemple ci-dessus, il reste 22 blocs libres mais l'espace le plus grand est de 10 blocs.

Pour cette raison, un programme manipulant un fichier séquentiel avec recopies répétées de celui-ci (exemple édition d'un texte à l'aide de l'éditeur de texte du système), se verra contraint de réserver :

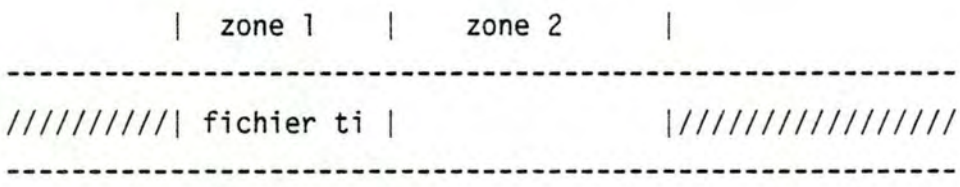
-soit deux zones libres distinctes et toutes deux suffisamment grandes pour recevoir la totalité du fichier. (il faut remarquer que les deux zones doivent absolument être distinctes)

-soit une zone unique ayant une taille au moins égale à trois fois la taille du fichier à éditer. (dans ce cas, il est nécessaire que la zone libre de la disquette soit disposée d'un seul tenant)

Exemple : Edition d'un texte

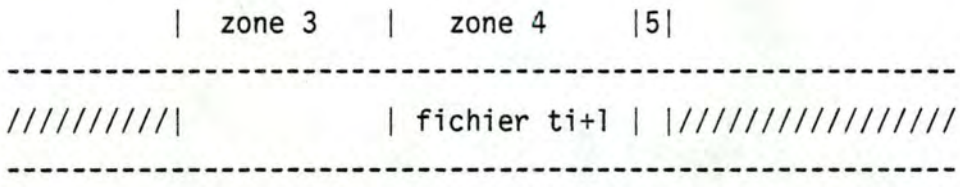
Lors de l'édition d'un texte, il est nécessaire, pour des raisons de sécurité, de recopier périodiquement le texte sur la disquette.
De plus, lors de la création d'un texte, la taille du fichier croît sans cesse.

Imaginons le travail sur une zone unique de taille inférieure à trois fois la taille du fichier. Après un certain nombre de copies, la disquette se présentera comme suit :



avec taille(zone2) supérieure taille(zone1).

Après un nouvel accroissement de la taille du fichier et une nouvelle copie, la disquette se présentera comme suit :



avec taille(zone3) inférieure taille(zone4)

taille(zone5) inférieure taille(zone4)

taille(zone3)+taille(zone5) supérieure taille(zone4).

A partir de ce moment, il sera impossible d'effectuer une nouvelle copie du fichier.

Pour remédier à ces inconvénients, la nouvelle version (version 1.1) de l'éditeur permet de recopier un fichier en écrasant l'ancien (option 'Save with purge').
Cette solution présente certains dangers :

a) En cas d'arrêt intempestif du micro-ordinateur pendant la recopie du fichier, seulement une partie de celui-ci aura été recopiée sur la disquette.
Dans ce cas, le répertoire indiquera l'adresse et la taille de l'ancien fichier, le début du fichier correspondra au début du nouveau fichier, la fin du fichier correspondra à la fin de l'ancien fichier.

b) Pour ce faire, il faut que la totalité du fichier réside en mémoire centrale.

2. Il est nécessaire de disposer d'un utilitaire de réorganisation des fichiers sur les disquettes.

exemple : tâche K(runch de l'utilitaire FILER.
compactage en avant.
La disquette de départ est la disquette de l'exemple précédent.

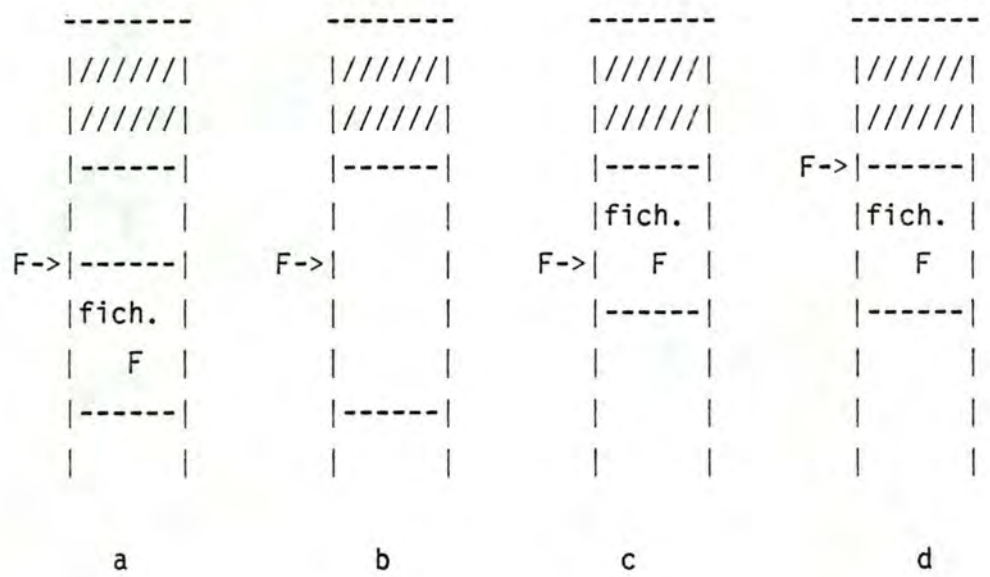
```
PASCALØ:
SYSTEM.PASCAL     43 23-Feb-82    6 Data
SYSTEM.MISCINFO    1  4-May-79   49 Data
SYSTEM.STARTUP     3 13-Nov-81   5Ø Code
SYSTEM.EDITOR     47 17-Dec-81   53 Code
SYSTEM.COMPILER   75 26-Feb-82  1ØØ Code
ACTUAL.TEXT        6  5-Mar-82  175 Text
LECTURE.TEXT      12  5-Mar-82  181 Text
ENTSOR.TEXT        8  5-Mar-82  193 Text
INITIALI.TEXT      4  5-Mar-82  2Ø1 Text
FONCTIONS.TEXT    16  5-Mar-82  2Ø5 Text
C65Ø2.TEXT         4  7-Mar-82  221 Text
ACTUAL.CODE       25  5-Mar-82  225 Code
BUFFER.CODE        2 18-Mar-82  25Ø Code
BUFFER.TEXT        4 18-Mar-82  252 Text
C65Ø2.CODE         2  7-Mar-82  256 Code
< UNUSED >        22            258
15/15 files, 22 unused, 22 in largest
```

Voici un deuxième exemple où la disquette a été compactée,

mais cette fois en libérant un espace à la suite d'un fichier préalablement choisi.

```
PASCALØ:
SYSTEM.PASCAL      43 23-Feb-82    6 Data
SYSTEM.MISCINFO     1  4-May-79   49 Data
SYSTEM.STARTUP      3 13-Nov-81   5Ø Code
SYSTEM.EDITOR      47 17-Dec-81   53 Code
SYSTEM.COMPILER    75 26-Feb-82  1ØØ Code
ACTUAL.TEXT         6  5-Mar-82  175 Text
LECTURE.TEXT       12  5-Mar-82  181 Text
ENTSOR.TEXT         8  5-Mar-82  193 Text
INITIALI.TEXT       4  5-Mar-82  2Ø1 Text
< UNUSED >         22            2Ø5
FONCTIONS.TEXT     16  5-Mar-82  227 Text
C65Ø2.TEXT          4  7-Mar-82  243 Text
ACTUAL.CODE        25  5-Mar-82  247 Code
BUFFER.CODE         2 18-Mar-82  272 Code
BUFFER.TEXT         4 18-Mar-82  274 Text
C65Ø2.CODE          2  7-Mar-82  278 Code
15/15 files, 22 unused, 22 in largest
```

L'emploi de cet utilitaire est dangereux. Le mode d'emploi en avise, d'ailleurs, l'utilisateur.
En effet, pendant la réorganisation d'une disquette, juste avant la mise à jour du répertoire de la disquette, ce répertoire ne permet plus un accès correct aux différents fichiers de la disquette. Si survient, à ce moment, un arrêt du micro-ordinateur, une partie de l'information de cette disquette risque d'être perdue.
De plus, à un certain moment pendant la réorganisation, il est possible qu'en déplaçant un fichier, ni l'ancienne, ni la nouvelle version de ce fichier ne soit complète. C'est ce qui se passe sur la figure suivante au stade b.



a : Avant la copie du fichier F.
b : Pendant la copie du fichier F.
c : La copie du fichier F est terminée.
d : Mise à jour du répertoire.

3. S'il n'y a pas d'espace libre à la fin d'un fichier, il est impossible d'étendre celui-ci.

Dans ce cas, il est nécessaire de recopier la totalité du fichier existant, dans une zone suffisamment grande pour contenir le nouveau fichier. Cette copie doit être faite par le programme utilisateur.

4. Un problème se pose lorsque l'on veut créer simultanément 2 ou plusieurs fichiers sur une disquette.

Lors de la création d'un fichier, le système choisit le plus grand espace libre de la disquette et le réserve pour ce fichier (C'est-à-dire qu'il sera impossible de placer sur cet espace de la disquette un autre fichier). S'il y a plusieurs espaces libres sur la disquette, il n'y a aucun problème, le premier espace libre est réservé pour le premier fichier, le second espace libre pour le deuxième fichier et ainsi de suite.

S'il n'y a qu'un seul espace libre sur la disquette, sans précaution spéciale, on ne pourra y placer qu'un seul nouveau fichier. Pour créer 2 ou plusieurs nouveaux fichiers dans cet espace libre, il faudra spécifier la taille maximum possible pour chacun des fichiers, en sachant que dans aucun cas, cette taille ne pourra être dépassée.

A cause de cette contrainte, il est très difficile de concevoir des fichiers dont la taille est inconnue au moment de l'ouverture du fichier ou dont la taille varie dans de grandes proportions au cours de l'exécution.

Certains programmes, (par exemple les compilateurs, l'éditeur de liens, etc ...), utilisent simultanément plusieurs fichiers, (pour les compilateurs, le fichier contenant le code objet, un fichier contenant le listing de la compilation et des fichiers temporaires ; pour l'éditeur de liens, le fichier contenant le code objet, un fichier contenant le listing de l'édition de liens). De plus la plupart de ces fichiers ont une taille inconnue et il faudra soit prévoir une taille maximum, soit n'autoriser la création que d'un seul fichier à la fois (exemple, pour l'éditeur de liens, il n'est pas possible de placer sur la même disquette le fichier contenant le code objet et le fichier contenant le listing de l'édition de liens).

5. Le meilleur emplacement possible pour le répertoire d'une disquette est, généralement, l'espace situé sur les pistes du centre de la disquette.
Dans le cas de fichiers situés obligatoirement sur des blocs contigus, il faut tâcher de ménager des espaces libres les plus grands possibles sur la disquette.
Dans ce cas, le meilleur emplacement pour le répertoire d'une disquette est l'espace situé sue les pistes les plus intérieures ou les plus extérieures de la disquette.
Cette contrainte n'est cependant pas trop gênante, dans la mesure où les performances des disquettes sont mauvaises (le temps de mise en route du moteur d'entraînement de la disquette est d'environ 1 seconde).

4. Les différents types de fichiers.

Au niveau physique:

Une suite de blocs de 512 bytes disposés de manière contiguë sur les supports magnétiques.

Au niveau logique:

Une suite d'enregistrements de longueur quelconque mais fixe.

Il existe différents types de fichiers qui sont distingués par :

- Un suffixe accolé au nom figurant dans le catalogue :

  .TEXT
  .DATA
  .CODE
  .BAD

- Un byte figurant dans le catalogue et indiquant le type du fichier.

En fonction de la valeur indiquée par le byte situé dans le catalogue et indiquant le type du fichier, on distinguera les types suivants :

-DATA : Le contenu du fichier est complètement libre. Le fichier contiendra du premier au dernier bloc les enregistrements tels qu'ils ont été définis dans le programme qui a créé le fichier.

-CODE : Il s'agit des fichiers contenant des programmes exécutables.
Le premier bloc contient le dictionnaire des segments du programme contenu dans ce fichier. Les autres blocs contiennent le code exécutable.

-TEXT : Il s'agit de fichiers prédéfinis comme :

file of char.

-BAD : Il s'agit d'une marque identifiant une zone de la mini-disquette impropre au stockage des données.
Cette marque, à la différence de tous les autres types de fichiers ne peut jamais être déplacée lors de réorganisation de disquette.

Seul le suffixe .TEXT est significatif. Dans tous les autres cas, le type indiqué par le byte situé dans le catalogue est prépondérant.
Les fichiers possédant le suffixe .TEXT contiennent au début du fichier, deux blocs destinés à contenir n'importe quelles informations désirées par l'utilisateur.

C'est ainsi que l'éditeur du système y range les différents paramètres d'édition (ceux-ci sont donc définis pour chaque fichier édité) .
Ces deux blocs sont ignorés lors d'une lecture normale du fichier(à l'aide des instructions GET ou READ) .

De plus, dans ce type de fichier, les chaînes de caractères ne chevauchent jamais une frontière de blocs entre blocs pairs et impairs.
Il s'agit d'une contrainte imposée par le compilateur Pascal du système [i] .

Il est possible de définir les fichiers suivants (soit par les instructions standard, soit en modifiant le répertoire comme expliqué au point 6 de ce chapitre) :

Bytes caractérisant le fichier	1 BAD	2 CODE	3 TEXT	4	5 DATA
Pas de bloc d'en-tête					
2 blocs d'en-tête	XXXXXXXXXX XXXXXXXXXX XXXXXXXXXX XXXXXXXXXX	XXXXXXXXXX XXXXXXXXXX XXXXXXXXXX XXXXXXXXXX			

---

[i] Pour des raisons de rapidité, le compilateur Pascal du système n'utilise pas les instructions READ et READLN pour le balayage du fichier source, mais utilise un tableau de 1Ø24 caractères (2 blocs physiques de la mini-disquette) .

Tous les types de fichiers peuvent être définis (si pas explicitement par programme, il est toujours possible de modifier certaines informations contenues dans le catalogue; nous verrons plus loin comment).
Certains types de fichier n'ont pas d'intérêt car ils ne sont pris en charge par aucun utilitaire du système. C'est notamment le cas pour les fichiers 'CODE' comportant les blocs d'en-tête, car ces fichiers contenant des programmes exécutables ne pourront être exécutés.

5. Les primitives d'accès.

Les différents accès possibles peuvent se subdiviser en trois classes :

- Les ouvertures de fichier. Le système vérifie que le fichier peut être légalement ouvert, réserve en mémoire centrale une zone destinée à la gestion du fichier (tampon) et initialise différents pointeurs.

- Les lectures et écritures d'information. l'utilisateur donne le nom symbolique d'une variable. Le système de gestion des fichiers effectue les échanges entre la disquette, le tampon en mémoire centrale et la variable du programme utilisateur.

- Les fermetures de fichier. Le système met à jour le catalogue de la disquette si nécessaire et libère la zone réservée en mémoire centrale pour le tampon.

Nous donnerons aux primitives les noms demandés par le compilateur Pascal.
La compilation de ces instructions étant traduite par un appel à des procédures standard, nous donnerons, pour des raisons de facilité, le même nom aux instructions Pascal et aux procédures standard.

Les compilateurs des autres langages doivent

-soit utiliser les procédures standard (les mêmes que le compilateur Pascal).

-soit utiliser leurs propres routines d'accès disposées dans un RUN-TIME package.

Ouverture et fermeture de fichier.

RESET, REWRITE.

Reset(nom_dans_programme,nom_dans_catalogue)

exemple :

Reset(sysout,'list.text');

Nom_dans_catalogue : doit être une chaîne de caractères représentant un nom de fichier.

Nom_dans_programme : représente l'identificateur du RECORD décrivant un enregistrement du fichier.

Cette instruction a pour effet d'établir les relations entre les tampons d'entrées-sorties et le programme utilisateur.
Sert également à initialiser les pointeurs qui pointent vers le début du fichier.

En Pascal standard, cette instruction a, également, pour effet de lire le premier article du fichier. Cela peut avoir de très gros inconvénients notamment lors de lecture de caractères au clavier.
En effet, lors de l'instruction d'initialisation du clavier par l'instruction :

reset(sysin,keyboard) par exemple ,

le système tente de lire le premier caractère.
Il y aura donc arrêt du programme juqu'au moment où l'utilisateur frappera le premier caractère.

Pour remédier à ces inconvénients, UCSD Pascal admet un nouveau type de fichier, les fichiers interactifs.

Exemple de déclaration d'un tel fichier en Pascal :

```
var sysout:interactive;
```

Ces fichiers sont, en tous points, identiques aux fichiers de caractères ("TEXT"), mis à part le fait que lors de l'instruction RESET, il n'y a plus de première lecture.

Au niveau le plus bas, la primitive "RESET" se décompose en deux parties :

-OPEN : ouverture du fichier.
-GET : lecture du premier article .

Lorsqu'un fichier est déclaré "INTERACTIF" , le compilateur Pascal ne génère plus l'instruction "GET" à la suite de l'instruction "OPEN".

Rewrite(nom_dans_programme,nom_dans_catalogue)

Cette instruction est identique à RESET, excepté le fait que les pointeurs pointent, dans ce cas, à la fin du fichier.
Le fichier ne peut exister avant l'exécution de cette instruction.
Il est possible de spécifier une taille maximale pour le fichier à créer :

Exemple : Rewrite(sysout,list.text[10]);

Le fichier occupera 10 blocs. Il pourra être étendu par la suite si les blocs qui lui sont contigus ne sont pas utilisés (voir le paragraphe 3 de ce chapitre) .

Lecture et écriture d'un article du fichier.

Il existe 4 niveaux d'accès:

-Pour tous les supports (adressables ou non) :

-UNITREAD, UNITWRITE.

Il s'agit de fonctions permettant la lecture ou l'écriture d'une suite de bytes sur un périphérique donné. Ces procédures sont écrites en assembleur de l'ordinateur hôte. Ces deux fonctions sont également utilisées pour la gestion des entrées-sorties et ont été détaillées au chapitre 3.
Le manuel d'utilisation prévient l'utilisateur du danger de ces fonctions. En effet, il n'y a aucune vérification faite par le système de gestion des fichiers. Il est possible, en se trompant simplement de numéro de bloc, d'écraser un fichier se trouvant précédemment sur la disquette.

Cependant, l'usage de ces fonctions de très bas niveau est parfois indispensable lorsque l'on désire manipuler des informations de la disquette n'appartenant à aucun fichier (Exemple : le catalogue proprement dit ou le porgramme de BOOTSTRAP) . Un exemple de travail directement sur le catalogue sera donné au paragraphe suivant.

-Uniquement pour supports adressables :

-BLOCKREAD, BLOCKWRITE.

Il s'agit de fonctions permettant de lire ou d'écrire un certain nombre de blocs de 512 bytes. Ces procédures peuvent accéder :

-à une disquette via un nom de disquette. La différence par rapport aux fonctions UNITREAD et UNITWRITE est ici une lecture et écriture d'un multiple de 512 bytes.

-à un fichier situé sur une disquette via un nom de fichier.

-GET, PUT, SEEK.

Ces procédures permettent de lire, d'écrire ou de se positionner sur un enregistrement du fichier.

-READ, WRITE (Uniquement pour fichiers TEXT, c'est-à-dire définis comme FILE OF CHAR). Ces procédures permettent de lire ou d'écrire une suite de caractères.

Les procédures READ et WRITE sont simplement une simplification des procédures GET et PUT et peuvent s'écrire comme suit:

```
READ =      x:=f^;
            get(f);

WRITE =     f^:=x;
            put(f);
```

-EOF : Cette fonction indique si la fin du fichier est atteinte.
La fin du fichier se détecte grâce à une information contenue dans le catalogue : le nombre de bytes significatifs se trouvant dans le dernier bloc du fichier.

-EOLN : Cette fonction ne peut être utilisée que pour des fichiers de caractères. Elle indique la fin d'une ligne.

6. L'accès au catalogue.

En plus des possibilités de travail sur les fichiers proprement dits, il peut être intéressant de pouvoir manipuler explicitement le catalogue.

Exemples :

a) Un programme utilise un fichier T et crée par fusion un fichier T+1 .
Lors d'une exécution ultérieure de ce programme, le fichier T+1 doit devenir le fichier T .

Il serait donc intéressant de pouvoir, par programme, changer le nom d'un fichier.

b) L'espace libre sur la disquette est devenu insuffisant (Voir le point 3 de ce chapitre) .
Normalement le programme se termine par une erreur ou alors le positionnement d'un drapeau indique l'anomalie.
Il serait intéressant de pouvoir réorganiser la disquette sans interrompre le programme.

Ces deux cas peuvent être résolus en manipulant directement le catalogue.

On déclarera une zone, en mémoire centrale, destinée à recevoir l'image du catalogue : (Voir le paragraphe 3 de ce chapitre)

```
Var directory:array[Ø.77] of record
                        first_bloc:integer;
                        last_block:integer;
                        file_type:integer;
                        file_name:string[15];
                        num_byte:integer;
                        last_date:integer;
            end;
```

Cette zone sera remplie par le contenu du catalogue par une lecture de la disquette à partir du bloc numéro 2 :

```
Unitread(4,directory,2Ø28,2);  [i]
```

---

[i] Les différents paramètres signifient :

4 : est le numéro du premier lecteur de mini-disquette.
Pour les autres lecteurs de mini-disquettes, d'autres numéros seront utilisés.

Catalogue : est le nom symbolique de la zone en mémoire centrale destinée à recevoir les informations lues.

2Ø28 : est le nombre d'octets à transmettre (5 mots de 2 octets pour les nombres entiers et 16 octets pour la chaîne de caractères, le tout à multiplier par 78)

2 : Numéro du premier secteur du catalogue;

Le programmeur a alors accès à toutes les informations du catalogue.
S'il modifie l'image du catalogue, la recopie du contenu de la variable CATALOGUE sur la disquette se fera comme suit :

```
UNITWRITE(4,catalogue,2Ø28,2);
```

Les paramètres sont identiques à ceux de la procédure UNITREAD utilisée précédemment.

L'avantage de cette méthode, est que le programmeur a accès à toutes les informations contenues dans le catalogue [i] .

L'inconvénient est qu'il devra réserver en mémoire centrale une zone de 2 K octets destinée à contenir l'image du catalogue.

---

[i] Cette méthode a été utilisée avec succès lors de la création du DEBBUGER.
Après la compilation du DEBBUGER, le fichier objet a les caractéristiques des fichiers CODE.
Cependant, après remplacement du segment numéro 2 par le DEBBUGER (à l'aide de l'utilitaire LIBRARY.CODE), le fichier objet a les caractéristiques d'un fichier DATA (tout comme les librairies de sous-programmes). Ce fichier contenant le code objet n'est dès lors plus exécutable.
Nous avons donc remplacé le code 5 (DATA) par le code 2 (CODE) .

7. La gestion des tampons.

Au niveau physique :

Une disquette est un support adressable. Cependant les lectures et écritures ne peuvent se faire directement que pour un certain nombre d'octets à la fois (1 bloc à la fois) .

Il existe, presque toujours en mémoire centrale, une zone appelée tampon et destinée à recevoir les informations contenues dans un bloc.

Au niveau logique :

La majorité des langages de haut niveau reconnaissent la notion d'article.
Un article peut être considéré comme l'ensemble des informations que l'utilisateur désire manipuler d'un seul tenant.

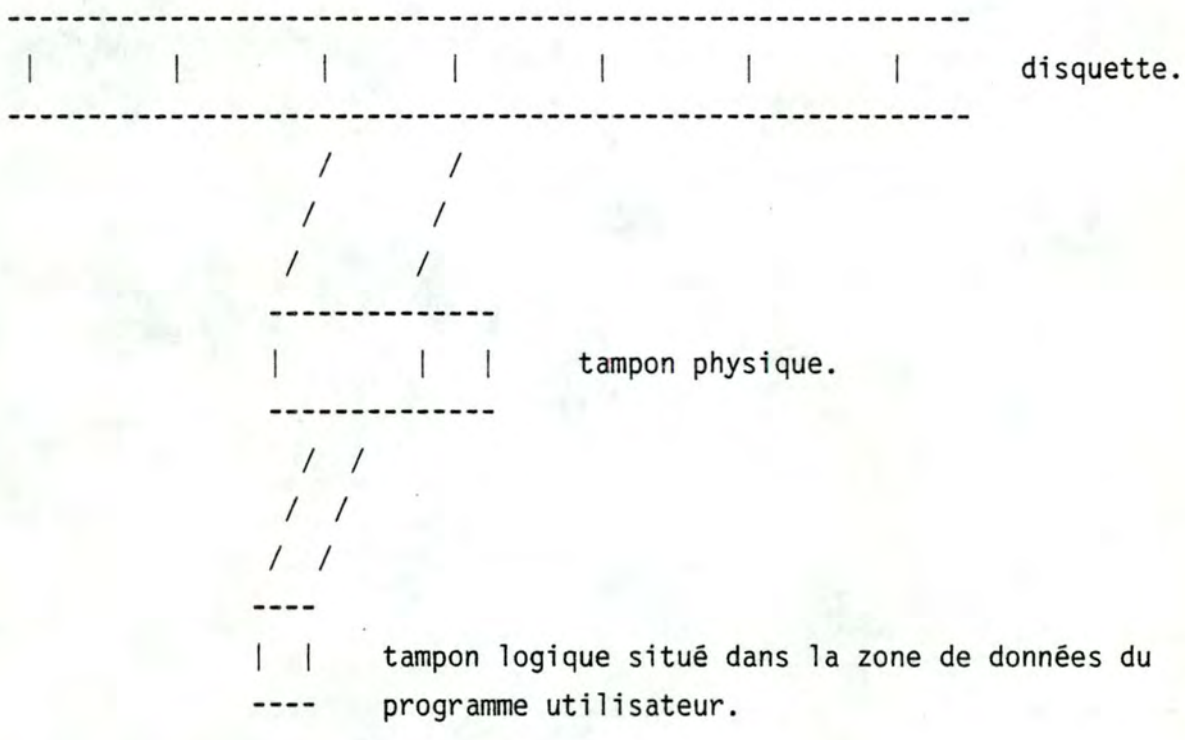
Nous distinguerons donc deux niveaux de tampons

- Tampon physique : de la taille d'un bloc de la disquette [i] .

---

[i] Il sera cependant légèrement plus grand, car il contiendra quelques autres informations, comme le nom du fichier dans le catalogue et différents pointeurs.

- Tampon logique : il s'agit de la zone de la mémoire centrale contenant un article.
Cette zone fera le plus souvent partie de la zone de données du programme utilisateur et sera connue de celui-ci par un nom symbolique.



La longueur d'un article (tampon logique) est, le plus souvent différente, de la longueur des tampons physiques.
Le système de gestion des fichiers rend la gestion transparente pour l'utilisateur.

Conclusions.

La gestion des tampons est classique.
Il est cependant dommage que l'utilisateur ne puisse pas choisir la taille des tampons physiques . C'est d'ailleurs pour cette raison que le compilateur Pascal du système utilise ses propres tampons. Ceux-ci ont une taille deux fois plus grande que la taille normale.
Toute la gestion (avec toutes les vérifications que cela entraîne) doit alors être explicitement programmée par l'utilisateur.

8. Les protections prévues.

Il semble que les protections soient efficaces.
Nous avons effectué un certain nombre d'essais, où nous avons véritablement cherché la faille. Dans presque tous les cas, le système d'exploitation s'est rendu compte de l'erreur.

Nous donnons , ci-dessous , la liste des différentes protections que nous avons rencontrées, suivie de la liste des erreurs qui se sont produites lors de nos essais.

Protections prévues.

- Les disquettes sont des supports amovibles. Le système de gestion des fichiers, avant tout accès à un fichier, vérifie si la disquette désirée se trouve bien dans le lecteur.
Si l'utilisateur possède plusieurs lecteurs de disquettes, il peut placer la disquette désirée dans le lecteur de son choix. Le système de gestion des fichiers lira le nom des disquettes se trouvant dans chacun des lecteurs et identifiera de la sorte la disquette désirée.

- Une disquette contient plusieurs fichiers. Lors d'une écriture, le système de gestion des fichiers vérifie qu'il n'y a pas écrasement d'un fichier préexistant. En cas de risque d'écrasement de fichier, il y a soit une erreur d'exécution, soit le positionnement d'un drapeau.

Erreurs rencontrées.

- Il ne s'agit pas exactement du système de gestion des fichiers, mais le cas est très semblable.
Lors du chargement d'un segment en mémoire centrale, il n'y a aucune vérification opérée sur la disquette. Il y a simplement chargement en mémoire centrale des blocs figurant sur la disquette aux emplacements indiqués par la table de segments.

Si par inadvertance, l'utilisateur a changé de disquette, ou si le programme a modifié le contenu de la disquette à l'emplacement du segment à charger, il y a beaucoup de chances pour que l'utilisateur perde le contrôle du système.
Dans ce cas une réinitialisation générale sera nécessaire.

- Nous avons indiqué précédemment le cas des fichiers possédant le suffixe .TEXT.
Les deux premiers blocs de ces fichiers ne contiennent pas d'enregistrements proprement dits. Ils sont en fait constitués, pour la plus grande partie, de Ø.
Si lors de la création d'un fichier, le nom comporte le suffixe .TEXT, et si après avoir changé de nom (en utilisant un autre suffixe, par exemple .DATA) on essaie de relire ce fichier, les premiers articles ne sembleront contenir que des Ø.
Un autre problème pourra survenir à cause du non chevauchement des frontières de pages des chaînes de caractères.
Le plus grave est que ce type d'erreur ne sera jamais détecté par le système de gestion des fichiers.

Chapitre 5.

Vérification de la portabilité des programmes.

Au niveau du langage source.

Prenons comme exemple le compilateur Pascal du système. Le programme source de ce compilateur est absolument identique d'une machine à l'autre (mis à part quelques constantes qui sont fonction de la taille de la mémoire centrale) .
Il ne devrait donc pas y avoir de problème de ce côté.

Par contre, et c'est le point le plus délicat, il existera, dans l'avenir, un grand nombre de compilateurs pour ce système, et même plusieurs compilateurs pour chaque langage de programmation.
C'est à ce niveau que des différences pourront apparaître et rendre les programmes non portables. La balle est donc dans le camp des concepteurs de compilateurs; c'est à ceux-ci de réaliser des compilateurs répondant aux normes en vigueur; encore faut-il que ces normes existent.

Nous avons eu l'occasion d'écrire un certain nombre de programmes, en langage Pascal, sur le micro-ordinateur APPLE II.
Ces programmes ont été également compilés et exécutés sur le micro-ordinateur NORT STAR Horizon sans aucun problème.

Au niveau du code objet.

Théoriquement, toutes les machines virtuelles sont identiques.
Il devrait donc être possible d'exécuter un programme objet sur n'importe quelle machine virtuelle.

Cela n'est pas tout à fait vrai pour trois raisons :

a) Il existe de petites différences d'une machine virtuelle à l'autre [i] .
Ces différences peuvent également provenir du fait que des optimisations des interpréteurs de CODE-P ont été apportées sur certaines machines .

b) Le jeu d'instructions de la machine virtuelle (CODE-P) évolue.
La deuxième version de ce code introduit quelques nouvelles instructions.
Normalement, une compatibilité vers le bas devrait être assurée, mais cela reste encore à démontrer.

---

[i] Nous avons déjà eu l'occasion de parler du type de rangement de nombres de 16 bits dans une mémoire orientée octets (Chapitre 2, paragraphe 6).
Exemple : microprocesseur 68ØØ, les nombres de 16 bits sont rangés avec les poids forts aux adresses basses.
microprocesseur 65Ø2, les nombres de 16 bits sont rangés avec les poids forts aux adresses basses.
Il est cependant possible de passer outre à cette contrainte, mais aux dépens de l'efficacité de l'interpréteur de CODE-P.

c) Certaines parties de programme peuvent être écrites en assembleur de l'ordinateur hôte.
Dans ce cas, la portabilité ne peut évidemment être assurée que pour les micro-ordinateurs utilisant le même type de microprocesseur (ou un microprocesseur ayant un jeu d'instructions compatible) .

Il semble que les différences les plus importantes proviennent des différentes versions de ce système d'exploitation et pas du micro-ordinateur utilisé.

Conclusions.

Le système d'exploitation étudié remplit bien les objectifs visés, c'est-à-dire portabilité et facilité d'utilisation.

L'utilisation d'un pseudo-code est une solution très simple, tout en offrant des avantages incontestables :

- Toutes les machines virtuelles sont identiques.

- Bonne gestion de la mémoire.

- Compacité du code généré.

Les inconvénients sont :

- Moindre vitesse d'exécution.

- Place mémoire perdue pour loger l'interpréteur de CODE-P.

Ces inconvénients diminueront avec les progrès technologiques, par l'utilisation de microprocesseurs microprogrammés en CODE-P.
De ce fait, le CODE-P utilisé pourrait bien, à brève échéance devenir un langage machine pur et simple.

La gestion des entrées-sorties correspond à un compromis entre simplicité de réalisation et performances.
Les utilisations habituelles se font sans problème. Par contre toute utilisation particulière nécessitera une programmation en assembleur [i] .

La gestion des fichiers est très bonne.
Les protections prévues sont très efficaces.

Il est cependant regrettable que l'organisation des fichiers sur les disquettes soit aussi restrictive (uniquement des secteurs contigus sans possibilités d'extension) .
Dans ces conditions les avantages apportés par les nouvelles primitives d'accès (par rapport au Pascal Standard) n'ont que peu d'intérêt.

D'un point de vue commercial, la publicité insiste surtout sur les possibilités professionnelles de ce système.
Il nous semble toutefois que les domaines où ce système remplira le mieux les objectifs visés sont l'enseignement et l'utilisation comme système de développement de programmes.
Pour des utilisations professionnelles, la disposition des fichiers sur les disquettes est trop restrictive.

---

[i] Avec tous les inconvénients de non portabilité que cela occasionne.

Références bibliographiques.

1 : APPLE COMPUTER , Communication interface card,
Installation and operating manual, CUPERTINO CALIFORNIA.

Serial interface card,
Installation and operating manual, CUPERTINO CALIFORNIA.

Parallel printer interface card,
Installation and operating manual, CUPERTINO CALIFORNIA.

2 : APPLE COMPUTER , Apple language system.
Installation and operating manual, CUPERTINO CALIFORNIA.

3 : APPLE FORTRAN , Language reference manual, CUPERTINO CALIFORNIA, 1980.

4 : APPLE PASCAL , Operating system manual, CUPERTINO CALIFORNIA, 1980.

5 : APPLE PASCAL , Language reference manual, CUPERTINO CALIFORNIA, 1980.

6 : APPLE PASCAL , Reference manual, CUPERTINO CALIFORNIA, 1979.

7 : BANINO JS., FERRIE J., KAISER C., LANCIAUX D. , Contrôle de l'accès aux objets partagés dans les systèmes informatiques, EDITIONS HOMMES ET TECHNIQUES, MONOGRAPHIE D'INFORMATIQUE DE L'AFCET, PARIS, 1978.

8 : BELL J. , "Threaded Code.", C. ACM 16, 6 (june 1973), pp. 370-372.

9 : BOWLES K. , UCSD PASCAL:A (nearly) Machine Independent Software System (for Microcomputers and Minicomputers) , THE BYTE BOOK OF PASCAL, PETERBOROUGH, NH. U.S.A, 1980, pp. 3 - 6.

10 : BOWLES Kenneth L. , Beginner's guide for the UCSD Pascal System, BYTE BOOKS (Subsidiary of McGraw-hill), PETERBOROUGH, NH U.S.A , 1980.

11 : CASSERES D. , "Bits and Bytes in Pascal; and other Binary Wonders", BYTE, octobre 1981, pp. 448-457.

12 : CHUNG K., YUEN H. , A Proposed Pascal Compiler, THE BYTE BOOK OF PASCAL, PETERBOROUGH, NH. U.S.A., 1980, pp. 23 - 24.

13 : CHUNG K., YUEN H. , A "Tiny" Pascal Compiler : Part 1: The P-Code Interpreter, THE BYTE BOOK OF PASCAL, NH. U.S.A., pp. 59 - 69.

14 : CHUNG K., YUEN H. , A "Tiny" Pascal Compiler : Part 2:The P-Compiler, THE BYTE BOOK OF PASCAL, NH. U.S.A., pp. 71 - 80.

15 : CHUNG K., YUEN H. , A "Tiny" Pascal Compiler : Part 3: P-Code to 8080 Conversion, THE BYTE BOOK OF PASCAL, NH. U.S.A., pp. 81 - 89.

16 : DAVID Daniel-Jean et DESCHAMPS Jean-Luc , Programmer en Pascal, EDITIONS DU P.S.I., PARIS, 1980.

17 : DEWAR B.K. , "Indirect Threaded Code.", C. ACM 18 (june 1975), pp. 330-331.

18 : DORIS A. , "Principes de la visualisation", Microsystèmes, Juillet-aout 1979, pp. 47-54.

19 : EHRMANN R. , Allocation dynamique de la mémoire des ordinateurs, DUNOD, MONOGRAPHIE DE L'AFCET, PARIS, 1972.

20 : FONTAINE Alain Bernard , "Le concept de logiciel relogeable pour microprocesseurs", MINIS ET MICROS, NUMERO 92, pp. 17 - 19.

21 : FORD G. , Comments on Pascal, Learning How to Program, and Small Systems, THE BYTE BOOK OF PASCAL, PETERBOROUGH, NH. U.S.A., 1980, pp. 13 - 15.

22 : FORSYTH C., HOWARD R. , Compilation and Pascal on the new Microprocessors, THE BYTE BOOK OF PASCAL, NH. U.S.A., 1980, pp. 33 - 39.

23 : JENSEN K., WIRTH N. , Pascal user manual and report, SPRINGER VERLAG, NH U.S.A, 1978(Second edition).

24 : LABAU A. , FIGUERAS J. , PINSON S. , "Pascal et les ordinateurs individuels", L'ordinateur individuel, décembre 1979, pp. 60-67.

25 : LEROY H. , Théorie des langages (1 ère partie): techniques de compilation, Notes de cours manuscrites 1 ère licence., FNDP , 1980-81.

26 : LEROY H. , Théorie des langages (2 ème partie), Notes de cours manuscrites 2 ème licence., FNDP , 1981-82.

27 : LESEA A., ZAKS R. , Techniques d'interface aux microprocesseurs, SYBEX, PARIS, 1978.

28 : LILIEN H. , Interfaces pour microprocesseurs et micro-ordinateurs, Editions Radio, Paris, 1981.

29 : LOUIS B. , "Tiny" Pascal in 8080 Assembly Language, THE BYTE BOOK OF PASCAL, NH. U.S.A., pp. 91 - 93.

30 : MACHI C. et GUILBERT J.F. , Téléinformatique, DUNOD, PARIS, 1979.

31 : MEMADIER J.P. , Structure et fonctionnement des ordinateurs, LAROUSSE, PARIS, 1971.

32 : NORI K.V. , AMMAN U. , JENSEN K., NAGELI H.H. et JACOBI Ch. , Pascal-P implementation notes, John WHILEY and SONS Ltd , NH U.S.A. , 1981.

33 : NORTHSTAR , Northstar Pascal Version 1, system reference manual, BERKELEY CALIFORNIA, 1979.

34 : OSBORNE Adam, KANE Gerry , Osborne 4 and 8 bit microprocessors handbook, OSBORNE/MC GRAW HILL, BERKELEY CALIFORNIA, 1981.

35 : OSBORNE Adam, KANE Gerry , Osborne 16 bits microprocessors handbook, OSBORNE/MC GRAW HILL, BERKELEY CALIFORNIA, 1981.

36 : PARRIEL Roger et DESCHAMPS Dominique , "Les unités mémoires à disques souples", MICRO SYSTEMES, 1980, NUMERO 11, pp. 67 - 79.

37 : RAMAEKERS J. , Systèmes d'exploitation (1 ère partie). Notes de cours 1 ère licence., FNDP , 1980-81.

38 : RAMAEKERS J. , Systèmes d'exploitation (2 ème partie). Notes de cours 2 ème licence., FNDP , 1981-82.

39 : ROSING M., McLAUREN K. , "From the very core of APPLE : PASCAL INTERNALS",CALL-APPLE (Apple Pugetsound Program Library Exchange) , VOLUME IV,NUMERO 3, 1981, pp. 9 - 22.

40 : SMITH P. , Concerning Pascal, A Hombrew Compiler Project, THE BYTE BOOK OF PASCAL, PETERBOROUGH, NH. U.S.A., 1980, pp. 21.

41 : TIREBOIS Jacques ," Pascal sur 6800 : une solution à gestion dynamique de l'espace mémoire", MINIS ET MICROS, NUMERO 141, pp. 37 - 39.

42 : WIRTH N. , Algorithms + Data Structures = Programs, Prentice-Hall Inc, Englewood N.J. U.S.A. , 1976.

43 : WIRTH N. , Pascal-S:A Subset and its Implementation, John WHILEY and SONS Ltd , NH. U.S.A. , 1981.

44 : WUYTENS G. , "Développement de logiciel avec le système P-UCSD", INFORDATA, avril 1982, pp. 49-50.

45 : ZAKS R. , Programmation du 6502, SYBEX, PARIS, 1979.

Annexes.

Les lecteurs désireux de trouver un complément d'informations, concernant la machine virtuelle, pourront analyser les listings témoins et les codes générés par le compilateur Pascal du système, lors de la compilation de petits programmes d'essai.
Une description complète du jeu d'instructions de la machine virtuelle se trouve en [4].

Les annexes 1 à 6 permettent de comparer un programme constitué d'un seul segment avec un programme constitué de deux segments.
En annexes 7 et 8, on trouvera les instructions supplémentaires générées par le compilateur lors de la rencontre de la directive spéciale (*$R Segment_name*) .
L'annexe 9 illustre la procédure à suivre pour utiliser des procédures compilées séparément.
L'annexe 10 montre ce que pourrait être le squelette du noyau du système d'exploitation (Le programme PASCALSYSTEM).
L'annexe 11 montre deux petits listings permettant de calculer la taille des tampons utilisés pour les accès aux disquettes.
Enfin l'annexe 12 donne le listing complet du DEBUGGER.

Table des annexes.

Annexe 1 : Programme ESSAI1. Page A1
Tri par la méthode SHELL en un seul segment.

-Listing témoin de la compilation.

Annexe 2 : Programme ESSAI1. A3
Tri par la méthode SHELL en un seul segment.

-Désassemblage du code généré par le compilateur.

Annexe 3 : Programme ESSAI1. A9
Tri par la méthode SHELL en un seul segment.

-Contenu du dictionnaire des segments.

Annexe 4 : Programme ESSAI2. A10
Tri par la méthode SHELL en deux segments.

-Listing témoin de la compilation.

Annexe 5 : Programme ESSAI2. A12
Tri par la méthode SHELL en deux segments.

-Désassemblage du code généré par le compilateur.

Annexe 6 : Programme ESSAI2. A18
Tri par la méthode SHELL en un seul segment.

-Contenu du dictionnaire des segments.

Annexe 7 : Programme ESSAI3. A19
Utilisation de l'option R (Résident).

-Listing témoin de la compilation.

Annexe 8 : Programme ESSAI3. A2Ø
Utilisation de l'option R (Résident).

-Désassemblage du code généré par le compilateur.

Annexe 9 : Programme ESSAI4 et 5. A23
Compilation séparée d'une procédure mise sous forme de UNIT.

-Listing témoin de la compilation.

Annexe 1Ø : Squelette du programme "PASCALSYSTEM" constituant le noyau du système d'exploitation. A25

Annexe 11 : Calcul de la taille des tampons. A27

Annexe 12 : Programme source du DEBUGGER. A28

Annexe 1.

Listing fourni par le compilateur lors de la compilation du programme ESSAI1. Les différentes colonnes représentent :

1 : le numéro de la ligne.
2 : le numéro du segment.
3 : le numéro de la procédure.
4 : le numéro d'imbrication.
5 : l'adresse relative du code généré.
6 : le programme source.

```
---------------------------------------------------------------------------
  1   2    3 4     5    6
---------------------------------------------------------------------------
      |    |  |    |    |

  1   1    1:D     1 (*$L+*)
  2   1    1:D     1 PROGRAM ESSAI1;
  3   1    1:D     3 const nb=100;
  4   1    1:D     3 var ech,ecart,i:integer;
  5   1    1:D     6     tab:array[1..nb] of integer;
  6   1    1:D   106
  7   1    2:D     1 procedure echange;
  8   1    2:D     1 var x:integer;
  9   1    2:0     0 begin
 10   1    2:1     0    x:=tab[i];
 11   1    2:1    13    tab[i]:=tab[i+ecart];
 12   1    2:1    37    tab[i+ecart]:=x
 13   1    2:0    49 end;
 14   1    2:0    64
 15   1    1:0     0 begin
 16   1    1:1     0    for i:=1 to nb do
 17   1    1:2    14       read(tab[i]);
 18   1    1:1    39    writeln('tableau non classe');
 19   1    1:1    77    writeln;
```

```
20   1    1:1    85    writeln;
21   1    1:1    93    ecart:=nb;
22   1    1:1    96    repeat
23   1    1:2    96       ecart:=ecart div 2;
24   1    1:2   101       repeat
25   1    1:3   101          ech:=0;
26   1    1:3   104          for i:=1 to nb-ecart do
27   1    1:4   118          begin
28   1    1:5   118             if tab[i]>tab[i+ecart] then
29   1    1:6   145             begin
30   1    1:7   145                ech:=1;
31   1    1:7   148                echange
32   1    1:6   148             end
33   1    1:4   150          end
34   1    1:2   150       until ech=0
35   1    1:1   158    until ecart=1;
36   1    1:1   167    writeln('tableau classe');
37   1    1:0   201 end.
```

Annexe 1.

Désassemblage commenté du code généré pour le programme ESSAI1.
---

adresses relatives des variables globales:

ech :05
ecart :04
i :03
tab :06

Procédure 'ECHANGE'.
---

adresses relatives des variables locales:

x :01

```
0000 A5 06      LAO  06      ;charger l'adresse de tab
0002 EA         SLDO 03      ;chargement de i et verification des bornes
0003 01         SLDC #01
0004 64         SLDC #64
0005 88         CHK
0006 01         SLDC #01
0007 95 00      SUBS INT
0008 A4 01      IXA  01      ;calcul de l'index
000A F8         SIND 00      ;chargement de tab[i]
000B CC 01      STL  01      ;sauvetage de x
000D A5 06      LAO  06      ;charger l'adresse de tab
000F EA         SLDO 03      ;chargement de i et vérification des bornes
```

```
0010 01          SLDC #01
0011 64          SLDC #64
0012 88          CHK
0013 01          SLDC #01
0014 95 00       SUBS INT
0015 A4 01       IXA  01        ;calcul de l'index
0017 A5 06       LAO  06        ;charger l'adresse de tab
0019 EA          SLDO 03        ;calcul de i + ecart et vérification des bornes
001A EB          SLDO 04
001B 82 00       ADD  INT
001C 01          SLDC #01
001D 64          SLDC #64
001E 88          CHK
001F 01          SLDC #01
0020 95 00       SUBS INT
0021 A4 01       IXA  01        ;calcul de l'index
0023 F8          SIND 00        ;chargement de tab[i+ecart]
0024 9A          STO            ;sauvetage dans tab[i]
0025 A5 06       LAO  06        ;charger l'adresse de tab
0027 EA          SLDO 03        ;calcul de i + ecart et vérification des bornes
0028 EB          SLDO 04
0029 82 00       ADD  INT
002A 01          SLDC #01
002B 64          SLDC #64
002C 88          CHK
002D 01          SLDC #01
002E 95 00       SUBS INT
002F A4 01       IXA  01        ;calcul de l'index
0031 D8          SLDL 01        ;chargement de x
0032 9A          STO            ;sauvetage dans tab[i+ecart]
0033 AD 00       RNP  00        ;fin de la procédure 'ECHANGE'.
```

```
0036 02     ;data size          ;taille de la zone de données.
0037 00
0038 00     ;parameter size     ;taille de la zone des paramètres.
0039 00
003A 07     ;exit IC            ;point de sortie de la procédure.
003B 00
003C 3C     ;enter IC           ;point d'entrée de la procédure.
003D 00
003E 02     ;procedure number   ;numéro de la procédure.
003F 01     ;lex level          ;numéro dans la chaîne statique.
```

```
0040 D7          NOP            ;début du programme principal
0041 D7          NOP
0042 01          SLDC #01       ;chargement de 1 dans la variable i
0043 AB 03       SRO  03
0045 64          SLDC #64       ;chargement de 100 dans la variable V6A
0046 AB 6A       SRO  6A
0048 EA          SLDO 03        ;test si fin de boucle
0049 A9 6A       LDO  6A
004B C8 00       LEQ  INT
004C A1 19       FJP  19        ;si fin de la boucle 'FOR' alors aller en 0067
004E B6 01 02    LOD  01,02
0051 A5 06       LAO  06
0053 EA          SLDO 03        ;vérifier si l'indice compris entre les bornes
0054 01          SLDC #01
0055 64          SLDC #64
0056 88          CHK
0057 01          SLDC #01
0058 95 00       SUBS INT
0059 A4 01       IXA  01        ;calcul de l'index
005B CD 00 0C    CXP  00,0C     ;read(integer)
005E 9E 00       CXP  00        ;iocheck
0060 EA          SLDO 03        ;ajouter 1 à la variable i
```

```
0061 01          SLDC #01
0062 82 00       ADD  INT
0063 AB 03       SRO  03
0065 B9 F6       UJP  F6        ;saut inconditionnel en 0048
0067 B6 01 03    LOD  01,03
006A D7          NOP
006B A6 12       LCA  'tableau non classe'          ;chargement d'une chaine
007F 00          SLDC #00                              de caractères
0080 CD 00 13    CXP  00,13     ;writestring
0083 9E 00       CXP  00        ;iocheck
0085 B6 01 03    LOD  01,03
0088 CD 00 16    CXP  00,16     ;writeln
008B 9E 00       CXP  00        ;iocheck
008D B6 01 03    LOD  01,03
0090 CD 00 16    CXP  00,16     ;writeln
0093 9E 00       CXP  00        ;iocheck
0095 B6 01 03    LOD  01,03
0098 CD 00 16    CXP  00,16     ;writeln
009B 9E 00       CXP  00        ;iocheck
009D 64          SLDC #64       ;chargement de 100 dans la variable ecart
009E AB 04       SRO  04
00A0 EB          SLDO 04        ;diviser la variable ecart par 2
00A1 02          SLDC #02
00A2 86 00       DIV  INT
00A3 AB 04       SRO  04
00A5 00          SLDC #00       ;chargement de 0 dans la variable ech
00A6 AB 05       SRO  05
00A8 01          SLDC #01       ;chargement de 1 dans la variable i
00A9 AB 03       SRO  03
00AB 64          SLDC #64       ;calacul de 100-ecart dans V6A
00AC EB          SLDO 04
00AD 95 00       SUBS INT
00AE AB 6A       SRO  6A
00B0 EA          SLDO 03        ;test si fin de boucle
00B1 A9 6A       LDO  6A
```

```
00B3 C8 00      LEQ  INT
00B4 A1 27      FJP  27       ;si fin de boucle alors saut en 00DD
00B6 A5 06      LAO  06
00B8 EA         SLDO 03       ;test si la variable i comprise entre 1 et 100
00B9 01         SLDC #01
00BA 64         SLDC #64
00BB 88         CHK
00BC 01         SLDC #01
00BD 95 00      SUBS INT
00BE A4 01      IXA  01       ;calcul de l'index
00C0 F8         SIND 00       ;chargement de tab[i]
00C1 A5 06      LAO  06
00C3 EA         SLDO 03       ;calcul de i + ecart et vérification des bornes
00C4 EB         SLDO 04
00C5 82 00      ADD  INT
00C6 01         SLDC #01
00C7 64         SLDC #64
00C8 88         CHK
00C9 01         SLDC #01
00CA 95 00      SUBS INT
00CB A4 01      IXA  01       ;calcul de l'index
00CD F8         SIND 00       ;chargement de tab[i+ecart]
00CE C5 00      GRT  INT      ;comparaison de 2 entiers
00CF A1 05      FJP  05       ;si le résultat est false alors aller en 00D6
00D1 01         SLDC #01      ;chargement de 1 dans ech
00D2 AB 05      SRO  05
00D4 CE 02      CLP  02       ;appel de la procédure 'ECHANGE'.
00D6 EA         SLDO 03       ;ajouter 1 a la variable i
00D7 01         SLDC #01
00D8 82 00      ADD  INT
00D9 AB 03      SRO  03
00DB B9 F4      UJP  F4       ;saut en 00B0
00DD EC         SLDO 05       ;test si ech=0, si oui aller en 00A5
00DE 00         SLDC #00
```

```
00DF C3 00      EQU  INT
00E0 A1 F2      FJP  F2
00E2 EB         SLDO 04       ;test si ecart=1, si oui aller en 00A0
00E3 01         SLDC #01
00E4 C3 00      EQU  INT
00E5 A1 F0      FJP  F0
00E7 B6 01 03   LOD  01,03
00EA D7         NOP
00EB A6 0E      LCA  'tableau classe'
00FB 00         SLDC #00
00FC CD 00 13   CXP  00,13    ;writestring
00FF 9E 00      CXP  00       ;iocheck
0101 B6 01 03   LOD  01,03
0104 CD 00 16   CXP  00,16    ;writeln
0107 9E 00      CXP  00       ;iocheck
0109 C1 00      RBP  00       ;retour au système d'exploitation

010C 6C     ;Jump table
010D 00
010E 69
010F 00
0110 60
0111 00
0112 CA
0113 00
0114 D0     ;data size
0115 00
0116 04     ;parameter size
0117 00
0118 0F     ;exit IC
0119 00
011A DA     ;enter IC
011B 00
011C 01     ;procedure number
011D 00     ;lex level
```

Annexe 3.

Informations contenues dans le dictionnaire des segments et dans le dictionnaire des procédures du programme ESSAI1.

SEGMENT : 0 ESSAI1

ADRESSE : (BLOC,BYTE) 1 0 LONGUEUR : (PAGE,BYTE) 1 36 SOIT : 292
NUMERO DU SEGMENT : 1 NOMBRE DE PROCEDURES : 2

DEPLACEMENT	PROCEDURE NUMBER	LEX LEVEL	ENTER IC	EXIT IC	PARAMETER SIZE	DATA SIZE
4 0	1	0	218	15	4	208
224 0	2	1	60	7	0	2

Annexe 4.

Listing fourni par le compilateur lors de la compilation du programme ESSAI2. Les différentes colonnes représentent :

1 : le numéro de la ligne.
2 : le numéro du segment.
3 : le numéro de la procédure.
4 : le numéro d'imbrication.
5 : l'adresse relative du code généré.
6 : le programme source.

```
   1   2    3 4     5    6
-------------------------------------------------------------------------------
     |    |  |    |    |

   1   1    1:D     1 (*$L+*)
   2   1    1:D     1 PROGRAM ESSAI2;
   3   1    1:D     3 const nb=100;
   4   1    1:D     3 var ech,ecart,i:integer;
   5   1    1:D     6     tab:array[1..nb] of integer;
   6   1    1:D   106
   7   7    1:D     1 segment procedure echange;
   8   7    1:D     1 var x:integer;
   9   7    1:0     0 begin
  10   7    1:1     0    x:=tab[i];
  11   7    1:1    13    tab[i]:=tab[i+ecart];
  12   7    1:1    37    tab[i+ecart]:=x
  13   7    1:0    49 end;
  14   7    1:0    64
  15   1    1:0     0 begin
  16   1    1:1     0    for i:=1 to nb do
  17   1    1:2    14       read(tab[i]);
  18   1    1:1    39    writeln('tableau non classe');
  19   1    1:1    77    writeln;
```

```
20   1    1:1    85    writeln;
21   1    1:1    93    ecart:=nb;
22   1    1:1    96    repeat
23   1    1:2    96       ecart:=ecart div 2;
24   1    1:2   101       repeat
25   1    1:3   101          ech:=0;
26   1    1:3   104          for i:=1 to nb-ecart do
27   1    1:4   118          begin
28   1    1:5   118             if tab[i]>tab[i+ecart] then
29   1    1:6   145             begin
30   1    1:7   145                ech:=1;
31   1    1:7   148                echange
32   1    1:6   148             end
33   1    1:4   151          end
34   1    1:2   151       until ech=0
35   1    1:1   159    until ecart=1;
36   1    1:1   168    writeln('tableau classe');
37   1    1:0   202 end.
```

Annexe 5.

Désassemblage du code généré pour le programme ESSAI2.

adresses relatives des variables globales:

ech :05
ecart :04
i :03
tab :06

Segment procedure 'ECHANGE'.

adresses relatives des variables locales:

x :01

```
0000 A5 06      LAO  06      ;charger l'adresse de tab
0002 EA         SLDO 03      ;chargement de i et vérification des bornes
0003 01         SLDC #01
0004 64         SLDC #64
0005 88         CHK
0006 01         SLDC #01
0007 95 00      SUBS INT
0008 A4 01      IXA  01      ;calcul de l'index
000A F8         SIND 00      ;chargement de tab[i]
000B CC 01      STL  01      ;sauvetage de x
000D A5 06      LAO  06      ;charger l'adresse de tab
000F EA         SLDO 03      ;chargement de i et vérification des bornes
```

```
0010 01          SLDC #01
0011 64          SLDC #64
0012 88          CHK
0013 01          SLDC #01
0014 95 00       SUBS INT
0015 A4 01       IXA  01       ;calcul de l'index
0017 A5 06       LAO  06       ;charger l'adresse de tab
0019 EA          SLDO 03       ;calcul de i + ecart et vérification des bornes
001A EB          SLDO 04
001B 82 00       ADD  INT
001C 01          SLDC #01
001D 64          SLDC #64
001E 88          CHK
001F 01          SLDC #01
0020 95 00       SUBS INT
0021 A4 01       IXA  01       ;calcul de l'index
0023 F8          SIND 00       ;chargement de tab[i+ecart]
0024 9A          STO           ;sauvetage dans tab[i]
0025 A5 06       LAO  06       ;charger l'adresse de tab
0027 EA          SLDO 03       ;calcul de i + ecart et vérification des bornes
0028 EB          SLDO 04
0029 82 00       ADD  INT
002A 01          SLDC #01
002B 64          SLDC #64
002C 88          CHK
002D 01          SLDC #01
002E 95 00       SUBS INT
002F A4 01       IXA  01       ;calcul de l'index
0031 D8          SLDL 01       ;chargement de x
0032 9A          STO           ;sauvetage dans tab[i+ecart]
0033 AD 00       RNP  00       ;fin de la procédure echange
```

```
0036 02     ;data size
0037 00
0038 00     ;parameter size
0039 00
003A 07     ;exit IC
003B 00
003C 3C     ;enter IC
003D 00
003E 01     ;procedure number
003F 01     ;lex level
```

```
0000 D7          NOP          ;début du programme principal
0001 D7          NOP
0002 01          SLDC #01     ;chargement de 1 dans la variable i
0003 AB 03       SRO  03
0005 64          SLDC #64     ;chargement de 100 dans la variable V6A
0006 AB 6A       SRO  6A
0008 EA          SLDO 03      ;test si fin de boucle
0009 A9 6A       LDO  6A
000B C8 00       LEQ  INT
000C A1 19       FJP  19      ;si fin de la boucle alors aller en 0027
000E B6 01 02    LOD  01,02
0011 A5 06       LAO  06
0013 EA          SLDO 03      ;vérifier si l'indice compris entre les bornes
0014 01          SLDC #01
0015 64          SLDC #64
0016 88          CHK
0017 01          SLDC #01
0018 95 00       SUBS INT
0019 A4 01       IXA  01      ;calcul de l'index
001B CD 00 0C    CXP  00,0C   ;read(integer)
001E 9E 00       CXP  00      ;iocheck
```

```
0020 EA          SLDO 03       ;ajouter 1 a la variable i
0021 01          SLDC #01
0022 82 00       ADD  INT
0023 AB 03       SRO  03
0025 B9 F6       UJP  F6       ;saut inconditionnel en 000C
0027 B6 01 03    LOD  01,03
002A D7          NOP
002B A6 12       LCA  'tableau non classe'        ;chargement d'une chaine
003F 00          SLDC #00                            de caractères
0040 CD 00 13    CXP  00,13    ;writestring
0043 9E 00       CXP  00       ;iocheck
0045 B6 01 03    LOD  01,03
0048 CD 00 16    CXP  00,16    ;writeln
004B 9E 00       CXP  00       ;iocheck
004D B6 01 03    LOD  01,03
0050 CD 00 16    CXP  00,16    ;writeln
0053 9E 00       CXP  00       ;iocheck
0055 B6 01 03    LOD  01,03
0058 CD 00 16    CXP  00,16    ;writeln
005B 9E 00       CXP  00       ;iocheck
005D 64          SLDC #64      ;chargement de 100 dans la variable ecart
005E AB 04       SRO  04
0060 EB          SLDO 04       ;diviser la variable ecart par 2
0061 02          SLDC #02
0062 86 00       DIV  INT
0063 AB 04       SRO  04
0065 00          SLDC #00      ;chargement de 0 dans la variable ech
0066 AB 05       SRO  05
0068 01          SLDC #01      ;chargement de 1 dans la variable i
0069 AB 03       SRO  03
006B 64          SLDC #64      ;calcul de 100-ecart dans V6A
006C EB          SLDO 04
006D 95 00       SUBS INT
006E AB 6A       SRO  6A
```

```
0070 EA          SLDO 03       ;test si fin de boucle
0071 A9 6A       LDO  6A
0073 C8 00       LEQ  INT
0074 A1 28       FJP  28       ;si fin de boucle alors saut en 009E
0076 A5 06       LAO  06
0078 EA          SLDO 03       ;test si la variable i comprise entre 1 et 100
0079 01          SLDC #01
007A 64          SLDC #64
007B 88          CHK
007C 01          SLDC #01
007D 95 00       SUBS INT
007E A4 01       IXA  01       ;calcul de l'index
0080 F8          SIND 00       ;chargement de tab[i]
0081 A5 06       LAO  06
0083 EA          SLDO 03       ;calcul de i + exart et vérification des bornes
0084 EB          SLDO 04
0085 82 00       ADD  INT
0086 01          SLDC #01
0087 64          SLDC #64
0088 88          CHK
0089 01          SLDC #01
008A 95 00       SUBS INT
008B A4 01       IXA  01       ;calcul de l'index
008D F8          SIND 00       ;chargement de tab[i+ecart]
008E C5 00       GRT  INT      ;comparaison de 2 entiers
008F A1 06       FJP  06       ;si le résultat est false alors aller en 0096
0091 01          SLDC #01      ;chargement de 1 dans ech
0092 AB 05       SRO  05
0094 CD 07 01    CXP  07,01    ;appel de la procédure echange
0097 EA          SLDO 03       ;ajouter 1 à la variable i
0098 01          SLDC #01
0099 82 00       ADD  INT
009A AB 03       SRO  03
009C B9 F4       UJP  F4       ;saut en 00
009E EC          SLDO 05       ;test si ech=0, si oui aller en 00
```

```
009F 00          SLDC #00
00A0 C3 00       EQU  INT
00A1 A1 F2       FJP  F2
00A3 EB          SLDO 04       ;test si ecart=1, si oui aller en 00
00A4 01          SLDC #01
00A5 C3 00       EQU  INT
00A6 A1 F0       FJP  F0
00A8 B6 01 03    LOD  01,03
00AB A6 0E       LCA  'tableau classe'
00BB D7          NOP
00BC 00          SLDC #00
00BD CD 00 13    CXP  00,13    ;writestring
00C0 9E 00       CXP  00       ;iocheck
00C2 B6 01 03    LOD  01,03
00C5 CD 00 16    CXP  00,16    ;writeln
00C8 9E 00       CXP  00       ;iochek
00CA C1 00       RBP  00       ;retour au système d'exploitation

00CC 6C     ;Jump table
00CD 00
00CE 69
00CF 00
00D0 60
00D1 00
00D2 CA
00D3 00
00D4 D0     ;data size
00D5 00
00D6 04     ;parameter size
00D7 00
00D8 0E     ;exit IC
00D9 00
00DA DA     ;enter IC
00DB 00
00DC 01     ;procedure number
00DD 00     ;lex level
```

Annexe 6.

Informations contenues dans le dictionnaire des segments et dans les dictionnaires des procédures du programme ESSAI2.

SEGMENT : 0 ESSAI2

ADRESSE : (BLOC,BYTE) 2 0 LONGUEUR : (PAGE,BYTE) 0 226 SOIT : 226
NUMERO DU SEGMENT : 1 NOMBRE DE PROCEDURES : 1

DEPLACEMENT	PROCEDURE NUMBER	LEX LEVEL	ENTER IC	EXIT IC	PARAMETER SIZE	DATA SIZE
2 0	1	0	218	14	4	208

SEGMENT : 1 ECHANGE

ADRESSE : (BLOC,BYTE) 1 0 LONGUEUR : (PAGE,BYTE) 0 68 SOIT : 68
NUMERO DU SEGMENT : 7 NOMBRE DE PROCEDURES : 1

DEPLACEMENT	PROCEDURE NUMBER	LEX LEVEL	ENTER IC	EXIT IC	PARAMETER SIZE	DATA SIZE
2 0	1	1	60	7	0	2

Annexe 7.

Listing fourni par le compilateur lors de la compilation du programme ESSAI3. Les différentes colonnes représentent :

1 : le numéro de la ligne.
2 : le numéro de segment.
3 : le numéro de la procédure.
4 : le numéro d'imbrication.
5 : l'adresse relative du code généré.
6 : le programme source.

```
   1   2    3 4     5    6
   !   !    ! !     !    !

   1   1    1:D     1 (*$L+*)
   2   1    1:D     1 program essai3;
   3   1    1:D     3
   4   7    1:D     1 segment procedure proc1;
   5   7    1:0     0 begin
   6   7    1:1     0   writeln('segment 1');
   7   7    1:0    29 end;
   8   7    1:0    42
   9   8    1:D     1 segment procedure proc2;
  10   8    1:0     0 begin
  11   8    1:0     0   (*$R proc1*)
  12   8    1:1     0   writeln('segment 2');
  13   8    1:0    31 end;
  14   8    1:0    56
  15   1    1:0     0 begin
  16   1    1:1     0   writeln('programme principal');
  17   1    1:1    41   proc1;
  18   1    1:1    44   proc2;
  19   1    1:0    47 end.
```

Annexe 8.

Désassemblage du code généré pour le programme ESSAI3.

Procédure PROC1.

```
0000 B6 02 03   LOD  02,03
0003 A6 09      LCA  'segment 1'        ;chargement d'une chaîne de caractères
000E D7         NOP
000F 00         SLDC #00
0010 CD 00 13   CXP  00,13   ;writestring
0013 9E 00      CSP  00      ;iocheck
0015 B6 02 03   LOD  02,03
0018 CD 00 16   CXP  00,16   ;writeln;
001B 9E 00      CSP  00      ;iocheck
001D AD 00      RNP  00      ;fin de la procédure PROC1

0020 00     ;data size
0021 00
0022 00     ;parameter size
0023 00
0024 07     ;exit IC
0025 00
0026 26     ;enter IC
0027 00
0028 01     ;procedure number
0029 01     ;lexical level
```

Procédure PROC2.

```
0000 B9 22      UJP  22       ;saut inconditionnel en 0024
0002 B6 02 03   LOD  02,03
0005 A6 09      LCA  'segment 2'     ;chargement d'une chaîne de caractères
0010 D7         NOP
0011 00         SLDC #00
0012 CD 00 13   CXP  00,13    ;writestring
0015 9E 00      CSP  00       ;iocheck
0017 B6 02 03   LOD  02,03
001A CD 00 16   CXP  00,16    ;writeln
001D 9E 00      CSP  00       ;iocheck
001F 07         SLDC #07
0020 9E 16      CSP  16       ;RLS
0022 B9 05      UJP  05       ;saut inconditionnel en 0029
0024 07         SLDC #07
0025 9E 15      CSP  15       ;MRK
0027 B9 F6      UJP  F6       ;saut inconditionnel en 0002
0029 AD 00      RNP  00       ;fin de la procédure PROC2

002B 00     ;Jump table
002C 2A
002D 00
002E 00
002F 00     ;data size
0030 00
0031 00     ;parameter size
0032 13
0033 00     ;exit IC
0034 34
0035 00     ;enter IC
0036 01
0037 01     ;procedure number
0038 02     ;lexical level
```

Programme principal.

```
0000 D7          NOP
0001 D7          NOP
0002 B6 01 03    LOD  01,03
0005 A6 13       LCA  'programme principal'          ;chargement d'une chaîne
                                                       de caractères
001A D7          NOP
001B 00          SLDC #00
001C CD 00 13    CXP  00,13   ;writestring
001F 9E 00       CSP  00      ;iocheck
0021 B6 01 03    LOD  01,03
0024 CD 00 16    CXP  00,16   ;writeln
0027 9E 00       CSP  00      ;iocheck
0029 CD 07 01    CXP  07,01   ;appel de la procédure PROC1
002C CD 08 01    CXP  08,01   ;appel de la procédure PROC2
002F C1 00       RBP  00      ;fin du programme

0032 00     ;data size
0033 00
0034 04     ;parameter size
0035 00
0036 07     ;exit IC
0037 00
0038 38     ;enter IC
0039 00
003A 01     ;procedure number
003B 00     ;segment number
```

Annexe 9.

Listing source du programme ESSAI3 et de la procédure 'ECHANGE', à compiler séparément sous forme de 'UNIT' ESSAI4.

```
(*$L+*)
PROGRAM ESSAI3;

uses
(*$UESSAI4.code*)
    ESSAI4;

begin
   for i:=1 to nb do
      read(tab[i]);
   writeln('tableau non classe');
   writeln;
   writeln;
   ecart:=nb;
   repeat
      ecart:=ecart div 2;
      repeat
         ech:=0;
         for i:=1 to nb-ecart do
         begin
            if tab[i]>tab[i+ecart] then
            begin
               ech:=1;
               echange
            end
         end
      until ech=0
   until ecart=1;
   writeln('tableau classe');
end.
```

```
(*$S+*)
(*$L+*)
unit ESSAI4;

interface

  const nb=100;
  var ech,ecart,i:integer;
      tab:array[1..nb] of integer;

  procedure echange;

implementation

  procedure echange;
  var x:integer;
  begin
     x:=tab[i];
     tab[i]:=tab[i+ecart];
     tab[i+ecart]:=x
  end;


begin
end.
```

Annexe 10.

Squelette du programme PASCALSYSTEM constituant le noyau du système d'exploitation.

```
Program PASCALSYSTEM;

VAR liste des variables globales;

Segment procedure USERPROGRAM;
begin
  writeln('no user program');
end;

Segment procedure DEBUGGER;
begin
  writeln('no debugger in system');
end;

Segment procedure PRINTERROR(xeqerr:integer);
begin
  writeln('error number ',xeqerr);
end;

Segment procedure INITIALIZE;
begin
end;
```

```
Segment procedure GETCMD;
   procedure load(filename);
   begin    (* charge le dictionnaire de segments du fichier
   end;          'filename' dans la table de segments du système. *)
begin
  repeat
    writeln('Command: E(dit,R(un,F(ile,C(omp,L(ink,X(ecute,A(ssem,',
            'D(ebug,?');
    read(ch);
  until ch in['E','R','F','C','L','X','A','D'];
  case ch of
    'E':load('system.editor');
    'C':load('system.compiler');
    .
    .
    'X':begin
          write('execute what file : ');
          readln(filename);
          load(filename);
        end;
  end;
end;

begin
  INITIALIZE;
  GETCMD;
  USERPROGRAM;
end.
```

Annexe 11.

Calcul de la taille des tampons.

Programme 1. Article d'une longueur de 100 mots.

```
1   1    1:D     1 (*$L+*)
2   1    1:D     1 program essai;
3   1    1:D     3 type article=array[1..100] of integer;
4   1    1:D     3 var sysout:file of article;
5   1    1:D   403
6   1    1:0     0 begin
7   1    1:1     0   reset(sysout,'essai.data');
8   1    1:1    33   close(sysout);
9   1    1:0    41 end.
```

Programme 2. Article d'une longueur de 10 mots.

```
1   1    1:D     1(*$L+*)
2   1    1:D     1 program essai;
3   1    1:D     3 type article=array[1..10] of integer;
4   1    1:D     3 var sysout:file of article;
5   1    1:D   313
6   1    1:0     0 begin
7   1    1:1     0   reset(sysout,'essai.data');
8   1    1:1    33   close(sysout);
9   1    1:0    41 end.
```

Annexe 12.

Programme source du DEBUGGER.

```
(*$S++ *)
(*$U-*)
(*$L CONSOLE:*)

PROGRAM PASCALSYSTEM;
TYPE CMDSTATE=(ETAT1,DEBUGCAL,ETAT3,ETAT4,ETAT5,ETAT6,ETAT7,ETAT8,ETAT9,ETAT10,
               DEBUGEXEC,DEBUGCONT);

     FILEPTR=↑INTEGER;

     SYSCOMREC=RECORD
                 FILLER1:ARRAY[1..3]OF INTEGER;
                 BUGSTATE:INTEGER;
                 FILLER2:ARRAY[1..8]OF INTEGER;
                 HLTLINE:INTEGER;
                 BRKPTS:ARRAY[0..3] OF INTEGER;
               END;

VAR SYSCOM:↑SYSCOMREC;
    FILLER3:ARRAY[1..67] OF INTEGER;
    STATE:CMDSTATE;

SEGMENT PROCEDURE USERPROGRAM(INPUT,OUTPUT:FILEPTR);
BEGIN
END;
```

```
SEGMENT PROCEDURE DEBUGGER;

TYPE OCTET=PACKED ARRAY[0..0] OF 0..255;

VAR BYTE:RECORD
         CASE BOOLEAN OF
           FALSE:(ADDRESS:INTEGER);
           TRUE :(MEMORY:↑OCTET);
         END;
    IPC,I:INTEGER;
    XX,YY:INTEGER;

PROCEDURE DEBON; (* MISE EN ROUTE DU DEBUGGER *)
BEGIN
  BYTE.ADDRESS:=-6096;
  BYTE.MEMORY↑[0]:=169;
  BYTE.ADDRESS:=-6095;
  BYTE.MEMORY↑[0]:=1;
  BYTE.ADDRESS:=-6094;
  BYTE.MEMORY↑[0]:=133;
END;

PROCEDURE DEBOFF; (* ARRET DU DEBUGGER *)
BEGIN
  BYTE.ADDRESS:=-6096;
  BYTE.MEMORY↑[0]:=76;
  BYTE.ADDRESS:=-6095;
  BYTE.MEMORY↑[0]:=59;
  BYTE.ADDRESS:=-6094;
  BYTE.MEMORY↑[0]:=210;
END;
```

```
FUNCTION LECTURE:INTEGER; (* LECTURE D'UN CARACTERE DANS LE BUFFER DU CLAVIER *)
VAR PTR1,PTR2:INTEGER;
BEGIN
  REPEAT
    BYTE.ADDRESS:=-16615;
    PTR2:=BYTE.MEMORY↑[0];
    BYTE.ADDRESS:=-16616;
    PTR1:=BYTE.MEMORY↑[0];
  UNTIL PTR1<>PTR2;
  PTR1:=PTR1+1;
  IF PTR1=78 THEN PTR1:=0;
  BYTE.MEMORY↑[0]:=PTR1;
  BYTE.ADDRESS:=PTR1+945;
  LECTURE:=BYTE.MEMORY↑[0];
END;

PROCEDURE SAUVETAGE; (* SAUVETAGE DES COORDONNEES DU CURSEUR *)
BEGIN
  BYTE.ADDRESS:=244;
  XX:=BYTE.MEMORY↑[0];
  BYTE.ADDRESS:=245;
  YY:=BYTE.MEMORY↑[0];
  GOTOXY(0,0);
END;

PROCEDURE BLANC; (* MISE A BLANC DE LA PREMIERE LIGNE DE L'ECRAN *)
BEGIN
  GOTOXY(0,0);
  WRITE('                                        ');
  WRITE('                                        ');
  GOTOXY(0,0);
END;
```

```
FUNCTION VALEUR:INTEGER; (* LECTURE DU NUMERO DU POINT D'ARRET *)
BEGIN
   IPC:=SYSCOM↑.HLTLINE;
   BYTE.ADDRESS:=IPC+1;
   VALEUR:=BYTE.MEMORY↑[0];
END;


PROCEDURE TRAITEMENT;FORWARD;

PROCEDURE INITIALISATION;

VAR NUMERO,NOMBRE:INTEGER;
    MOT:RECORD
        CASE BOOLEAN OF
          FALSE:(ADDRESS:INTEGER);
          TRUE :(MEMORY:↑INTEGER);
        END;

BEGIN
   SAUVETAGE;
   IF STATE=DEBUGCALL THEN WRITE(CHR(12));
   BLANC;
   WRITE('Debugger:   E(nd   S(tep   T(race   W(here');
   IF STATE=DEBUGCALL THEN WRITE('    Starting')
                      ELSE WRITE('    Line number : ',VALEUR);
   WRITE('  [1.1]');
   REPEAT
     I:=LECTURE;
   UNTIL I IN [116,84,115,83,119,87,101,69];
   GOTOXY(0,0);
   BLANC;
   CASE I OF
     116,84:BEGIN      (* TRACE *)
              SYSCOM↑.BRKPTS[0]:=0;
              SYSCOM↑.BUGSTATE:=1;
            END;
```

```
    115,83:BEGIN      (* STEP *)
             SYSCOM↑.BRKPTS[0]:=-1;
             SYSCOM↑.BUGSTATE:=2;
             IF STATE<>DEBUGCALL THEN
             BEGIN
               GOTOXY(XX,YY);
               TRAITEMENT;
             END;
           END;
    119,87:BEGIN      (* WHERE *)
             REPEAT
               BLANC;
               WRITE('Break point number : ');
               READLN(SYSCOM↑.BRKPTS[0]);
             UNTIL SYSCOM↑.BRKPTS[0]>0;
             REPEAT
               BLANC;
               WRITE('Times : ');
               READLN(SYSCOM↑.BRKPTS[1]);
             UNTIL SYSCOM↑.BRKPTS[1]>=0;
             SYSCOM↑.BUGSTATE:=3;
           END;
    101,69:BEGIN      (* END *)
             SYSCOM↑.BUGSTATE:=9;
           END;
  END;
  GOTOXY(XX,YY);
  IF STATE=DEBUGCALL THEN
  BEGIN
    STATE:=DEBUGEXEC;
    DEBON;
    USERPROGRAM(NIL,NIL);
  END;
  STATE:=DEBUGEXEC;
END;
```

```
PROCEDURE TRAITEMENT;
BEGIN
   CASE SYSCOM↑.BUGSTATE OF
    1:BEGIN
        SAUVETAGE;
        GOTOXY(0,0);
        WRITE('Trace:     execute line ',VALEUR);
        GOTOXY(XX,YY);
      END;
    2:BEGIN
        SAUVETAGE;
        BLANC;
        WRITE('Step:      execute line ',VALEUR);
        WRITE('   Press <ESC> or <ETX>');
        REPEAT
          I:=LECTURE;
        UNTIL ((I=27) OR (I=3));
        IF I=3 THEN
        BEGIN
          GOTOXY(XX,YY);
          INITIALISATION;
        END;
        GOTOXY(25,0);
        WRITE('                                        ');
        GOTOXY(XX,YY);
      END;
    3:BEGIN      (* WHERE *)
        IF SYSCOM↑.BRKPTS[0]=VALEUR THEN
        BEGIN
          IF SYSCOM↑.BRKPTS[1]=0 THEN
            INITIALISATION;
          SYSCOM↑.BRKPTS[1]:=SYSCOM↑.BRKPTS[1]-1;
        END;
      END;
```

```
    9:BEGIN      (* END *)
      END;
  END;
END;

BEGIN
  DEBOFF;
  CASE STATE OF
     DEBUGEXEC:TRAITEMENT;
     DEBUGCALL:INITIALISATION;
     DEBUGCONT:INITIALISATION;
  END;
  DEBON;
END;

BEGIN
END.
```