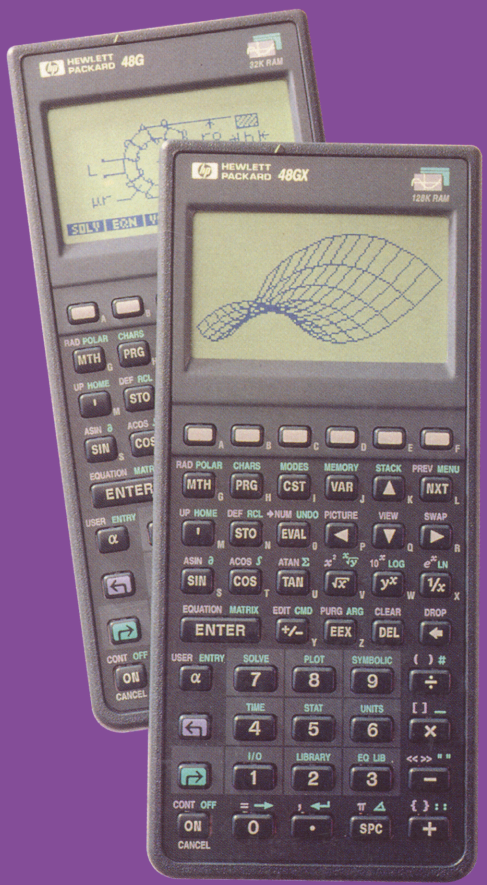
LES *SECRETS* DE LA HP48G/GX

TOME 2

Externals et Assembleur

J.-M. FERRARD



D3I DIFFUSION

Chez le même EDITEUR et du même AUTEUR, J.-Michel FERRARD :

- **La Maîtrise de la HP 28 S**
  TOME 1 : PROGRAMMATION ET EXERCICES
- **La Maîtrise de la HP 28 S**
  TOME 2 : PROGRAMMATION ET APPLICATIONS
- **La Maîtrise de la HP 48**
  TOME 1 : PROGRAMMATION ET EXERCICES (Épuisé)
- **La Maîtrise de la HP 48**
  TOME 2 : PROGRAMMATION ET APPLICATIONS
- **Mastering your HP 48**
  PROGRAMMING AND APPLICATIONS
  (Version anglaise)
- **HP 48, NOCH EFFEKTIVER NUTZEN**
  ANWENDUNGSPROGRAMME FÜR FORTGESCHRITTENE
  (Version allemande)
- **Les Secrets de la HP 48**
  TOME 1
- **Les Secrets de la HP 48**
  TOME 2 : EXTERNALS ET ASSEMBLEUR
- **Carte ROM PAC MATH pour HP 48 SX**
- **Les Secrets de la HP 48 G / GX**
  TOME 1 : 6000 BONNES ADRESSES
- **Les Secrets de la HP 48 G / GX**
  TOME 2 : EXTERNALS ET ASSEMBLEUR
- **« MATHEZ » la HP 48 G / GX en 380 programmes**

D3I DIFFUSION
20 rue Hermès - 31520 Ramonville St Agne - France
Tél. : (33) 62 24 66 12 - Fax : (33) 61 73 54 19

LES *SECRETS* DE LA HP48G/GX

TOME 2

Programmation en Externals et en Assembleur

Jean-Michel FERRARD

Agrégé de Mathématiques
Professeur de Mathématiques Spéciales

D3I
BP 49 - Tél. 62 24 66 20
31528 RAMONVILLE ST-AGNE CÉDEX (FRANCE)

Hewlett-Packard, HP 48, HP 48 S, HP 48 SX, HP 48 G, HP 48 GX,
sont des marques déposées

Première édition : Septembre 1993
© 1993 - D3I DIFFUSION
ISBN 2-908791-11-2

Tous les efforts ont été faits pour que les informations, programmes et schémas présentés dans ce livre soient aussi exacts et complets que possible. Ni les auteurs, ni l'éditeur ne pourront en aucun cas être tenus pour responsables des préjudices de quelque nature que ce soit, pouvant résulter de leur utilisation, tant dans un cadre privé, que commercial ou professionnel.

La loi du 11 mars 1957 n'autorisant, aux termes des alinéas 2 et 3 de l'article 41, d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale, ou partielle, faite sans le consentement de l'auteur ou des ayants droit, ou ayants cause, est illicite » (alinéa premier de l'article 40).

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

TABLE DES MATIERES
DU TOME 2

Introduction 7

PREMIERE PARTIE:
PROGRAMMATION EN EXTERNALs

Le programme →EXT 13
Le programme EXTERNPC 31
"Rpl-Système" et nouveaux Externals 59
Utilitaires en Externals 115

DEUXIEME PARTIE:
PROGRAMMATION EN ASSEMBLEUR

Le micro-processeur Saturn 175
Les instructions du Saturn 189
Un assembleur en Externals 237
Un assembleur sur PC 281
La RAM-Système 311
Quelques points d'entrée 345
Exemples de programmes en assembleur 365

Index 389

INTRODUCTION

Les deux tomes des "Secrets" de la HP48G/GX forment un ensemble cohérent pour apprendre et maîtriser deux langages qui vont bien au-delà du vocabulaire standard de la HP48:

- ▸ Le **Rpl-Système** (un sur-ensemble du RPL standard, utilisé par les ingénieurs de HP pour écrire une grande partie du code de la calculatrice).
- ▸ L'**Assembleur** (le nec plus ultra en matière de liberté et de rapidité, mais aussi de difficulté).

Le premier tome contient la description de tous les objets de la HP48, ainsi qu'une liste très importante d'***Externals*** (adresses de la ROM pointant sur des objets ou des sous-programmes exécutables par SYSEVAL).

Pour ceux qui veulent utiliser efficacement ces données, il est indispensable de se doter d'outils logiciels commodes. C'est le but que j'ai recherché ici avec les programmes **→EXT** et **EXTERNPC** (le premier tournant sur HP48, le second sur compatible PC).

Pour aller plus loin encore que les *Externals*, le langage machine est l'ultime recours. J'ai consacré la deuxième partie de ce livre à une initiation à la programmation en langage-machine, avec notamment deux assembleurs (**ASM48**, écrit en *Externals*, tournant sur votre HP48, et **ASMPC**, écrit en Turbo-Pascal, tournant sur compatible PC).

Vous trouverez également dans ce livre le contenu exhaustif et détaillé de la RAM-Système (utile aux programmeurs en langage-machine), et quelques points d'entrée directement accessibles depuis un programme écrit en assembleur. Enfin plusieurs exemples de programmes écrits en assembleur devraient permettre de lever les dernières inhibitions envers ce langage.

Les deux tomes des "***Secrets***" couvrent un sujet qui n'est pas facile et qui pourrait rebuter les utilisateurs débutants ou encore peu expérimentés. L'effort que je leur demande est cependant prometteur de grandes découvertes. J'ai en tout cas fait de mon mieux pour rendre cette matière aussi limpide que possible, loin de toute idée élitiste.

Toulouse, Août 1993

◆ PROGRAMMER en EXTERNALs

On sait que les programmes de la HP48 (les structures RPL) sont constitués d'objets successifs. Dans la représentation en mémoire d'un tel programme, un de ces objets peut être codé:

- par son adresse en ROM (il est alors évalué indirectement), ou
- par l'adresse-préfixe qui correspond au type de cet objet, suivie par le corps de l'objet lui-même, qui est alors évalué directement. (voir tome 1, pages 29 et suivantes).

L'objectif principal de la première partie de ce livre est de se passer de l'instruction **SYSEVAL** pour évaluer des objets non standards de la HP48, en accédant à ces objets directement et simplement par leur adresse. Voyons quels avantages on peut tirer de ce projet.

Si **adr** est l'adresse d'un objet de la ROM (qui ne soit pas un des objets intégrés à la HP48: instructions standards et entiers de -9 à 9) la méthode la plus simple pour l'évaluer est d'inclure dans le corps du programme (en ligne de commande), la séquence **#adr SYSEVAL** ("adr" étant ici exprimée sous la forme d'un entier binaire).

Si on utilise cette méthode, le programme, tel qu'il est codé en mémoire, contient quelque part:

- le codage de l'entier binaire **#adr** (l'adresse-préfixe 02A4E sur 5 quartets, suivie du corps de l'entier binaire sur 21 quartets)
- l'adresse de l'instruction **SYSEVAL** (1A52E), sur 5 quartets.

Exemple:

La structure RPL située à l'adresse **#4B1F2h** a pour effet de rappeler sur la pile les dimensions de l'objet **PICT**. La séquence **#4B1F2h SYSEVAL** apparaîtra codée, dans le programme qui la contient, sous la forme suivante (voir tome 1, page 46, pour le codage des entiers binaires):

E4A20510002F1B400000000000E25A1

◆ PLUS COURT, PLUS VITE !

Si on arrive à remplacer cette séquence de 31 quartets par la séquence **2F1B4** de 5 quartets, le résultat (au niveau de l'éxécution du programme) est exactement le même et on économise 26 quartets!

On peut également s'attendre à un gain en vitesse d'éxécution: dans la première méthode, en effet, on demande à la HP48 d'interpréter un entier binaire avant de le placer sur la pile, puis d'éxécuter, sur cet entier binaire, une instruction standard (SYSEVAL) avec ce que celà comporte de tests internes inutiles...

◆ PLUS FORT !

Il y a un autre avantage, plus subtil mais décisif, dans le fait de remplacer le codage de la séquence **#adr SYSEVAL** par les cinq quartets représentant l'adresse **adr**.

Il faut savoir que parmi les milliers de structures RPL qui apparaissent dans la ROM de la HP48, certaines (quelques centaines) apparaissent comme des outils de mise au point de toutes les autres.

Il s'agit là d'un véritable langage de programmation, insoupçonné de l'utilisateur. C'est ce langage (l'assembleur n'ayant été employé que pour les séquences en langage machine) qu'ont conçu et employé les ingénieurs de HP qui ont conçu la ROM de la HP48.

Ce langage, particulièrement riche, fait appel à des adresses qui jouent le rôle de "**commandes**". Le plus souvent, celles ci agissent sur le "**flot des instructions**". On y trouve des instructions conditionnelles ou répétitives. D'autres agissent sur la pile des retours, ou sur les objets qui les suivent dans la séquence RPL, etc...

Ces adresses seront longuement présentées dans la première partie de tome 2. Elles ne figurent pas dans le tome 1, pour la simple raison qu'il est difficile, sinon impossible, de les évaluer en utilisant l'instruction SYSEVAL. La simple présence de cette instruction dans le corps du programme peut en effet perturber (ou empêcher) le fonctionnement de ces adresses.

La seule solution, pour utiliser ces adresses, est dès lors d'être à même de pouvoir les inclure directement dans le codage des programmes.

Enfin, pour permettre de bien comprendre le rôle de ces adresses mystérieuses, il faut se doter d'outils efficaces. Les programmes **→EXT** et **EXTERNPC**, présentés dans cette partie, ont été conçus pour celà. Ils font l'objet des deux chapitres qui suivent.

◆ LE LANGAGE-MACHINE

La deuxième partie de ce livre est une introduction au langage-machine:

- ► Une présentation des principes de ce langage
- ► Le dictionnaire des instructions
- ► Un assembleur écrit entièrement en Externals
- ► Un assembleur écrit en Turbo-Pascal, pour compatibles IBM PC.
- ► Une sélection de points d'entrée, à utiliser directement dans vos routines en assembleur.
- ► Quelques exemples de programmes en langage-machine.

Les deux assembleurs que j'ai écrits vous permettront, je l'espère, de faire vos premiers pas dans cet art difficile, puis de progresser rapidement.

PREMIERE PARTIE

La Programmation en Externals

LE PROGRAMME →EXT

Le tome 1 des "*Secrets* de la HP48" contient plusieurs milliers de points d'entrées de la HP48. Ce tome 2 en contient quelques centaines.

Nombre de ces adresses (notamment celles du tome 2) ne peuvent pas utilisées avec l'instruction SYSEVAL. Il faut donc se doter d'outils permettant de contourner cette difficulté.

Le programme **→EXT** présenté ici-même vous permettra d'écrire des structures Rpl quelconques, directement sur votre HP48.

Dans ce chapitre, nous allons voir un programme très simple, qui, sur votre HP48, vous permettra de créer des structures RPL faisant appel à des adresses non standards de la HP48 (adresses que nous appelons **Externals**, pour simplifier), à l'image des structures RPL qui peuplent la mémoire morte de la HP48.

Le programme dont il est question s'appelle **→EXT**.

Il prend comme argument une liste non vide (les délimiteurs { et } jouant provisoirement le rôle des délimiteurs « et » des programmes standards).

Il applique une transformation particulière aux nombres réels ou aux entiers binaires de cette liste, de la manière suivante:

- Un entier binaire **b** est remplacé par l'adresse correspondante dans la mémoire de la HP48. Ainsi #3188h sera remplacée par l'adresse 03188 (qui est celle de la forme interne **dup** de l'instruction **DUP**).

  Rappelons à ce propos que la ROM "utile" de la HP48 s'étend de l'adresse #0 à l'adresse #7FFFFh, les adresses suivantes permettant d'accéder indifféremment (selon la configuration) à la "ROM couverte" (256 K de #80000h à #FFFFFh), ou à la RAM, ou au contenu d'une carte, etc... Il semble donc raisonnable de limiter l'entier binaire **b** à l'intervalle [#0h,#7FFFFh].

  Si b est supérieur à #FFFFFh, on ne considère que son reste dans la division par #100000h (les cinq chiffres héxas de poids faible).
- Un réel **r** est converti en l'entier-système correspondant (les entiers-système sont certainement les objets les plus utilisés dans la ROM de la HP48). Par exemple 12 sera transformé en <Ch> (Le réel r est arrondi à l'entier le plus proche).
- Le même procédé est appliqué, récursivement, aux sous-listes qui peuvent entrer dans la composition de la liste initiale.
- Le programme '**→EXT**' "*détaggue*" (sans les modifier davantage) tous les objets taggués de la liste initiale. Le meilleur moyen de conserver un entier binaire à sa valeur, ou un réel, ou une sous-liste, est donc de le tagguer au départ.

Enfin la liste obtenue, après ces transformations, est convertie en **structure RPL** (les délimiteurs { et } de listes sont transformés en les délimiteurs de structure RPL). Dans la pratique, il ne s'agit que de remplacer l'adresse-préfixe de la liste par une adresse-préfixe de structure RPL (voir **tome 1**, pages 197 et 198).

Le fait d'utiliser une liste au départ offre plusieurs avantages:

- Dans la saisie de la liste en ligne de commande, on utilise le mode de saisie **PRG**.
- Les délimiteurs { et }, appelés à disparaître, sont un bon moyen de ne pas utiliser les délimiteurs « et » (qui n'ont pas d'intérêt dans l'exécution d'un programme)
- Dans les cas un peu compliqués, la liste peut être constituée sur la fin (instruction **→LIST**) à partir d'objets qui peuvent être des entiers-système, des réels longs, des structures RPL contenant des Externals...,etc...

Nous allons voir une première version de →**EXT**. Il est intéressant de noter que cette première version permettra de construire la deuxième version, plus performante !
'→**EXT**': (première version) CheckSum #C31Ah, taille 266.5 octets

```
« 1  OVER  SIZE  FOR  k  k  DUP2  GET
     CASE
        DUP  TYPE  10  SAME  THEN
          #4003h  SYSEVAL  #62B9Ch  SYSEVAL
        END
        DUP TYPE 0 SAME THEN #18CD7h SYSEVAL END
        DUP  TYPE  8  SAME  THEN
          #3F9Fh  SYSEVAL  #5AB3h  SYSEVAL  →EXT
        END
        DUP  TYPE  5  SAME  THEN  →EXT  END
        DTAG
     END
     PUT
  NEXT
  #3FB3h  SYSEVAL  #5AB3h  SYSEVAL
»
```

Voici en quelques lignes la logique du programme précédent:

- On parcourt la liste initiale du début à la fin (boucle **FOR**).
- Chaque objet de la liste est ainsi examiné tour à tour (**GET**).
- Quatre types d'éléments sont particulièrement intéressants: les entiers binaires (**DUP TYPE 10 SAME**), les réels (**DUP TYPE 0 SAME**), les programmes (**DUP TYPE 8 SAME**) et les listes (**DUP TYPE 5 SAME**).
- Les objets taggués éventuels sont alors *détaggués*.
- Dans le cas d'un entier binaire **b**:
  On place l'entier-système **<2h>** sur la pile (**#4003h SYSEVAL**), puis on appelle l'instruction de mnémonique "**get(cp,s)**" (voir tome 1, page 182), qui considère l'entier binaire b=#xyztu (qui est codé en mémoire E4A20 51000 utzyx000...) comme un objet-composé (tome 1 pages 181 et suivantes) et en extrait le deuxième élément (ici celui dont l'adresse en mémoire est xyztu).
  Cette méthode est "gonflée" (il y a avait plus simple sur la HP48S/SX, et je n'ai pas encore trouvé l'équivalent sur la G/GX), mais ça marche...
- Dans le cas d'un objet de type programme: On place l'entier-système **<2A74h>** sur la pile (représentant l'adresse-préfixe des listes) sur la pile (**#3F9Fh SYSEVAL**). On exécute **#5AB3h SYSEVAL** pour convertir le programme en une liste, et on appelle →**EXT** pour la transformer.
- Dans le cas d'une sous-liste: elle est immédiatement transformée par →**EXT**.
- Chaque objet de la liste, éventuellement transformé, est replacé dans la liste, à sa position initiale (**PUT**).
- On sort alors de la boucle. On place l'entier-système **<2D9Dh>** (adresse-préfixe des structures Rpl) sur la pile (**#3FB3h SYSEVAL**). On transforme enfin la liste en structure Rpl par **#5AB3h SYSEVAL**.

N.B: on constate la présence à deux reprises de la séquence **#5AB3h SYSEVAL** dans le programme **→EXT**. L'action de ce SYSEVAL (modificateur d'adresse-préfixe) a été longuement décrite dans le tome 1 (pages 197 et suivantes).

Remarques sur →EXT:

- Vous pouvez vous demander l'intérêt d'avoir prévu le cas où l'objet considéré dans la liste initiale est du type programme.
  Il s'agit en fait de prendre en compte la possibilité, pour la liste donnée au départ, de contenir des instructions composées du genre **IF..THEN..ELSE**, ou **WHILE..REPEAT..END**, etc...
  En effet si on considère une liste telle que **{ ROT + * }**, cette liste est visiblement de taille 3.
  Au contraire, la liste **{ IF ROT THEN + ELSE * END }** est de taille 1, et le seul objet qu'elle contient est une structure RPL!
  Dans ces conditions, il fallait prévoir la possibilité de "rentrer" dans de tels objets, pour y procéder éventuellement à des conversions de réels en adresses...

- Le problème soulevé ci-dessus n'est en fait pas d'une importance capitale dans la mesure où, précisément, certains ***Externals*** permettront bien vite de remplacer les instructions composées que nous venons d'évoquer.

- Le fait que le programme **→EXT** détaggue les objets taggués de la liste initiale est utile pour empêcher que **→EXT** ne modifie certains éléments de cette liste (éléments de type 10, 8, ou 5).
  Ainsi, si une sous-liste est "tagguée", elle ne sera pas transformée par **→EXT**. Elle sera simplement détagguée. Il en va de même pour les entiers binaires, les réels, et les structures Rpl.

- Le programme **→EXT** présente quelques avantages et quelques inconvénients assez évidents, quand on le compare au programme **EXTERNPC**, qui fera l'objet du chapitre suivant:
  →EXT ne demande aucun matériel extérieur à la HP48 (ni PC, ni cable, ni logiciel KERMIT). C'est un atout de taille par rapport à **EXTERNPC**.
  Néanmoins, le principal problème rencontré dans l'utilisation de **→EXT** est qu'il nous oblige à travailler avec des listes presque exclusivement composées d'entiers binaires (destinés à être traduits en adresses).
  Il en résulte un manque de lisibilité des programmes (ce qui implique des risques d'erreurs, et une plus grande difficulté de "mise au point"). Il vous faut constamment compulser le tome 1 à la recherche de l'adresse correspondant à une action donnée (ou alors créer vous-même votre table de SYSEVALs).
  Au contraire, le programme **EXTERNPC** permet de travailler sur des fichiers sources lisibles (utilisation de mnémoniques) et donc aisément modifiables.

EXEMPLES D'UTILISATION DE '→EXT'

EXEMPLE 1:

On veut écrire un programme inversant l'ordre des n premiers éléments de la pile. Le programme s'appelle **'REVST'** et sa syntaxe doit être: **n REVST** (l'argument n ne devant pas être considéré comme l'un des n premiers objets de la pile).

Si l'entier n est supérieur au nombre d'objets de la pile, il doit se produire une erreur "*Two Few Arguments*".

Le programme **'REVST'**, peut s'écrire, dans le langage usuel:

```
« 2 FOR i i ROLLD -1 STEP »
```

La taille de ce programme est, sans compter le nom, de 31.5 octets.

Pour réécrire **'REVST'** en Externals, on va utiliser les adresses suivantes:

- **#18C4Ah** (Mnémonique **need(r)**): transforme un réel n en entier-système, avec une erreur "*Two Few Arguments*" si la taille de la pile est inférieure à n.
- **#5DE7Dh** (Mnémonique **revB**): inverse l'ordre des éléments du premier bloc de la pile (voir tome 1, pages 208 et suivantes).
- **#3244h** (Mnémonique **drop**): pour détruire l'entier-système.

On place donc la liste **{ #18C4Ah #5DE7Dh #3244h }** au niveau 1, et on lui applique le programme **'→EXT'**. On obtient alors sur la pile l'objet suivant, que vous pouvez placer dans la variable **'REVST'**.

```
External  External  External  External
External  External  External  External  External
External  External  <0h>  External  External
External  External  External  External  External
External  External  External  External  External
```

Evidemment celà n'est pas particulièrement clair!

L'instruction **BYTES** nous apprend que le CheckSum de cet objet est **#B080h** et que sa taille est de **12.5 octets** (25 quartets). Effectivement, le codage en mémoire de ce programme est constitué de 5 adresses de 5 quartets chacune:

```
D9D20      #02D9Dh = Adresse-Préfixe des Rpl
   A4C81   need(r)
   D7ED5   RevB
   44230   drop
B2130      #0312Bh = Adresse-Suffixe des Rpl
```

L'exemple de REVST

Avant d'utiliser ce qui précède pour placer une remarque très générale sur les programmes utilisant des Externals, on peut comparer les performances des deux versions de **'REVST'**:

Au niveau de la taille, la deuxième version constitue une certaine économie (12.5 octets, au lieu de 31.5). Ce résultat doit cependant être relativisé.

En effet, il faut inclure au début de la deuxième version un test vérifiant que l'argument n est présent et qu'il est bien de type réel (prévoir deux adresses, au moins, soit 5 octets).

Les **SYSEVALs** permettant de traiter les problèmes de reconnaissance d'arguments seront décrits plus loin dans cette partie du livre.

Pour ce qui est de la rapidité d'exécution, les deux versions de **'REVST'** ont été testées sur une pile de 500 objets (inversion de la pile sur toute sa longueur).

Les temps d'exécution moyens des deux versions sont les suivants:

- Première version: 5.97 secondes.
- Deuxième version: 3.36 secondes.

Le gain de temps est donc assez net. L'utilisation d'Externals permet souvent d'accélérer beaucoup plus encore la vitesse d'exécution des programmes. On le verra sur les exemples qui suivent.

Remarques sur les programmes avec Externals:

Quand on travaille avec des Externals, il ne faut pas confondre l'image en mémoire d'un programme (constituée d'adresses) avec celle qui est obtenue sur la pile (par décompilation répétée de ces adresses, jusqu'à ce que la HP48 reconnaisse des objets classiques, comme ici <0h>, ou des objets du type "*Primitive Code*", qu'elle affiche sous le nom **External**).

En reprenant l'exemple précédent, voici comment la HP48 en arrive à afficher tant d'Externals, alors que le programme ne fait appel qu'à trois adresses non standards:

- Les adresses **#18C34h** et **#3244h**, qui désignent déjà des "*Primitive Code Objets*", sont uniquement responsables du premier et du dernier des Externals ci-dessus.
- L'adresse **#5DE7Dh** désigne une structure Rpl constituée de 7 adresses dont 4 désignent des "*Primitive Code Objects*".
- Les 3 adresses restantes désignent des structures Rpl. C'est dans la première d'entre elles que figure l'objet **<0h>** que l'on a remarqué ci-dessus. Pour les autres, il faut continuer la décompilation pour finalement arriver partout à des "*Primitive Code Objects*".

(Pour plus de précision sur les "*Primitive Code Objects*", cf tome 1, page 32).

Il n'est pas rare qu'une même adresse non standard soit décompilée par la HP48 en un très grand nombre d'Externals. Pour autant l'objet ainsi décompilé ne compte que pour 5 quartets dans la taille de la structure Rpl qui le contient....

Pour terminer, signalons que les programmes contenant des ***Externals*** ne sont pas éditables en ligne de commande. Si vous passez outre et que vous validez par ***ENTER***, le processus de compilation ne reconnaît aucun des Externals. Il les remplace au contraire par le nom global **'External'**.

D'autres problèmes peuvent empêcher l'édition de structures Rpl telles que nous allons en créer tout au long de ce livre.

Ainsi, et en reprenant l'exemple précédent, une tentative de validation, en ligne de commande, de la deuxième version de ***'REVST'*** se solde par une erreur de syntaxe tant que vous conservez l'objet <0h>.

EXEMPLE 2:

On veut écrire un programme ne conservant sur la pile que les n premiers éléments (tous les objets des niveaux n+1 et plus étant détruits). Le programme s'appelle **'KEEP'** et sa syntaxe doit être: **n KEEP** (l'argument n ne devant pas être considéré comme l'un des n premiers objets de la pile).

Si l'entier n est supérieur au nombre d'objets de la pile, il doit se produire une erreur "*Two Few Arguments*".

Une façon très simple d'écrire ce programme est:

```
« →LIST  DROP  CLEAR  LASTARG  OBJ→  DROP »
```

La taille de cet objet est de 25 octets. Pour fonctionner, il faut que la fonction **LASTARG** ne soit pas désactivée. On remarquera l'astuce consistant à détruire par **DROP** les seuls éléments que l'on souhaite conserver, avant de les récupérer par la fonction **LASTARG**.

Pour obtenir le même résultat avec des Externals, on va utiliser les adresses suivantes (il ne s'agit que d'une variante, utile pour montrer qu'on peut très bien conserver certaines adresses standards, et en remplacer d'autres):

- On conserve la commande **→LIST**
- On utilise l'adresse **#634CFh** (Mnémonique **Ldef''**): elle crée une variable locale de nom vide en y plaçant l'objet du niveau 1, qui est ici la liste précédemment formée (voir tome 1, pages 310 et suivantes).
- On conserve la commande **CLEAR**.
- On utilise l'adresse **#634B6h** (Mnémonique **1Lget;Lpop**) qui rappelle sur la pile le contenu de la variable locale qui vient d'être créée, et qui en même temps libère la mémoire de cette variable devenue inutile.
- On utilise enfin l'adresse **#62B88h** (Mnémonique **break;drop**), qui "libère" la liste sur la pile, tout en détruisant l'entier-système représentant la longueur de cette liste.

On applique donc →**EXT** à la liste:

```
{ →LIST  #634CFh  CLEAR  #634B6h  #62B88h  }
```

Le résultat est le suivant, au niveau 1 (il doit être placé dans la variable **'KEEP'**). La taille est 17.5 octets (au lieu de 25), et le CheckSum est #D3DBh.

```
→LIST  { }  External  External  CLEAR
External  External  External  External
```

On y distingue parfaitement, les deux commandes "survivantes" →**LIST** et **CLEAR**. La liste en 2-ième position n'est pas vide: elle contient le nom local '' (cf tome 1, p.60).

Remarque: La deuxième version de **'KEEP'** conduit à un gain de 7.5 octets. Celui-ci peut être pratiquement annulé en appliquant le programme →**EXT** à la liste:

```
{ →LIST  DROP  CLEAR  LASTARG  OBJ→  DROP  }
```

Celà conduit en effet à la structure Rpl suivante:

```
→LIST  DROP  CLEAR  LASTARG  OBJ→  DROP
```

Ce qui correspond au programme **'KEEP'** initial, privé des deux délimiteurs « et » (qui ne servent à rien dans l'exécution du programme lui-même). La taille de cet objet est de 20 octets (on gagne 5 octets en supprimant deux adresses).

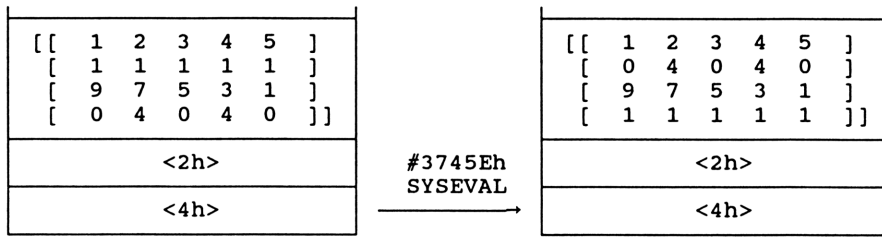
EXEMPLE 3:

On veut écrire un programme en Externals réalisant le travail de l'instruction **RSWP**, c'est-à-dire permettant d'échanger deux lignes quelconques d'une matrice (réelle ou complexe) ou deux coefficients d'un vecteur.

On nomme ce programme **'ECHL'**, et sa syntaxe doit être: **m n ECHL**, où m et n sont les numéros de lignes (ou de coefficients) à échanger.

Nous allons résoudre ce problème en utilisant quelques Externals bien choisis... L'outil principal sera l'adresse **#3745Eh** (Mnémonique **SwapRows(ar,2s)**), dont l'action est d'échanger deux lignes dans une matrice, ou deux coefficients dans un vecteur (cf tome 1, page 176), ces deux lignes ou ces deux coefficients étant désignés par leur numéro d'ordre (dans le tableau initial) sous la forme de deux entiers-système.

Voici un schéma montrant le fonctionnement de ce **SYSEVAL**:



On constate que les deux entiers-système désignant les numéros de ligne (ou de coefficient) conservent leur position initiale sur la pile.

Ces deux entiers-système peuvent être égaux (rien ne se passe alors), mais, **surtout**, ils doivent être compris entre <1h> et le nombre de lignes de la matrice (ici <4h>) ou le nombre de coefficients du vecteur. Le non-respect de cette condition peut entraîner un "*Memory Clear*".

Au vu de ce risque, nous allons soigner particulièrement l'écriture en Externals d'une nouvelle version de **'ECHL'**. Nous allons en particulier tester sérieusement la présence, sur la pile initiale, des arguments requis par le problème (ainsi que la conformité de ces arguments).

Celà va nous obliger à utiliser deux des nombreuses adresses qui seront décrites dans la suite de cette première partie du livre.

La première adresse (**#18A68h**, mnémonique **need3**) vérifie la présence d'au moins 3 arguments sur la pile (sinon erreur "*Too Few Arguments*").

La seconde adresse vérifie ensuite que la pile contient bien, au départ, un tableau au niveau 3, et deux réels ou niveaux 2 et 1.

Cette vérification utilise le couple formé par l'adresse **#18FB2h** (Mnémonique **ChkArg**) et l'entier-système **<411h>** (4 pour la présence d'un tableau au niveau 3, suivi de deux 1 pour la présence des deux réels aux deux premiers niveaux).

Le codage du programme en mémoire peut alors être représenté, symboliquement, de la manière suivante:

```
D9D20          #02D9Dh est l'adresse-préfixe des Rpl
   86A81         need3
   2BF81         ChkArg
   <411h>        identificateur types arguments
   ......              objet à exécuter
B2130          #0312Bh est l'adresse-suffixe des Rpl
```

Dans le cas particulier ci-dessus, le test représenté par l'adresse **#18FB2h** et l'entier-système **<411h>** fonctionne de la manière suivante: si les objets des trois premiers niveaux ne sont pas du type attendu, il y a une erreur "***Bad Argument Type***". Dans le cas contraire, l'objet désigné par ..... dans le listing ci-dessus est évalué (ici il s'agira d'une structure Rpl regroupant les différentes instructions).

Notons que ce test n'est pas destructif: les arguments vérifiés restent sur la pile, que le test ait réussi ou non. Dans la suite de ce livre, de tels tests seront longuement présentés (dans le cas qui nous occupe, il n'est pas besoin d'en savoir plus).

Il reste maintenant à écrire la suite d'instructions à exécuter dans le cas où les arguments ont passé les deux premiers tests.

Il faut d'abord vérifier que les deux réels placés aux niveaux 2 et 1 sont bien (après arrondi à l'entier le plus proche) compris entre 1 et le nombre de lignes de la matrice.

Nous utiliserons les adresses suivantes:

- **#194F7h** (Mnémonique **2r=>2s**): transforme deux réels r2 et r1 en les deux entiers-système correspondants, avec arrondi à l'entier le plus proche. Les réels négatifs sont transformés en <0h> (tome 1, page 196).

- **#31ACh** (Mnémonique **dup2**): forme interne de **DUP2**.

- **#6362Dh** (Mnémonique **min(2s)=0?**): teste si l'un au moins de deux-entiers système est nul. Le résultat est un booléen (**True** ou **False**, voir tome 1, pages 151 et suivantes).

- **#63B05h** (Mnémonique **?:BAV!**): examine le booléen du niveau 1. S'il est égal à True, alors exécute l'erreur "*Bad Argument Value*" (tome 1, page 361).

- **#624C6h** (Mnémonique **max(2s)**): maximum de deux entiers-système.

- **#6121Ch** (Mnémonique **4pick**): forme rapide de **4 PICK**.

- **#2CC97h** (Mnémonique **nrows(ar)**): donne le nombre de lignes d'une matrice, ou le nombre de coefficients d'un vecteur. Le résultat est un nombre réel (voir tome 1, page 176).

- **#18CD7h** (Mnémonique **r=>s**): transforme un réel en entier-système.

- **#3D83h** (Mnémonique **>(2s)?**): applique l'opérateur logique **>** à deux entiers-système. Le résultat est un booléen.

- **#3745Eh** (Mnémonique **SwapRows(ar,2s)**): adresse décrite ci-dessus.

- **#3258h** (Mnémonique **drop2**): forme interne de **DROP2**, pour éliminer les deux entiers-système qui subsistent après utilisation de l'adresse précédente.

La liste que nous allons donner en pâture au programme **→EXT** doit être formée sur le modèle exposé précédemment, à savoir:

```
{  #18A68h  #18FB2h  #411h  {....}  }
```

Ici la sous-liste {....} sera transformée par **→EXT** en la structure Rpl représentant l'objet à évaluer après le premier test vérifiant la présence et la nature des arguments.

Voici donc la liste sur laquelle il faut appliquer **→EXT**. Les commentaires (séparés du reste par un trait vertical) n'en font évidemment pas partie. Le CheckSum de cette liste est: #2645h et sa taille est de 218 octets.

```
{ #18A68h                    | need3
  #18FB2h                    | ChkArg
  #64F2Ch                    | <411h> (on peut utiliser 1041)
  {                          | début structure Rpl à évaluer
    #194F7h    #31ACh        |    2r=>2s        dup2
    #6362Dh    #63B05h       |    min(2s)=0?      ?:BAV!
    #31ACh     #624C6h       |    dup2       max(2s)
    #6121Ch    #2CC97h       |    4pick       nrows(ar)
    #18CD7h    #3D83h        |    r=>s       >(2s)?
    #63B05h                  |    ?:BAV!
    #3745Eh    #3258h        |    SwapRows(ar,2s)    drop2
  }                          | fin de la structure Rpl à évaluer
}
```

Il ne servirait à rien d'afficher l'image sur la pile de la structure Rpl obtenue en appliquant **→EXT** à cette liste. Disons simplement que cette image fait apparaître 23 Externals, et l'entier-système <411h> en troisième position.

Plus utilement, vous devez vérifier que le CheckSum de cette structure Rpl est **#FEFBh**, mais surtout que sa taille n'est que de **52.5 octets** (à comparer aux 120 octets de la première version...).
On peut alors placer la structure Rpl ainsi obtenue dans la variable **'ECHL'**.

Il est utile de vérifier qu'un message d'erreur adéquat accompagne toute tentative d'exécuter le programme **'ECHL'** avec des données incomplètes ou incorrectes (pas de risque de *Memory Clear*).

On peut remarquer que ces messages d'erreur ne sont pas attribués à une commande particulière, et que quelque soit l'endroit où survient l'incident, vous retrouvez votre pile initiale (et non pas la pile au moment où s'est produite l'erreur). De tout celà, nous reparlerons beaucoup plus en détail dans quelque temps....

L'exemple de ECHL

Comparons maintenant les rapidités d'exécution du programme **'ECHL'** et de la commande intégrée **RSWP**.

Le test est le suivant: Effectuer 10 échanges de lignes successifs sur une matrice d'ordre 10 (peu importe quelle matrice).

- Avec la version en Externals: 0.14 secondes.
- Avec l'instruction intégrée **RSWP**: 0.38 secondes !

Remarque:

Un point mérite cependant d'être signalé (qu'on pourrait qualifier de défaut) concernant le fonctionnement du programme **'ECHL'**. L'échange de deux lignes de la matrice (ou de deux coefficients du vecteur) s'effectue en effet **sans aucun newob**.

Pour bien comprendre cette notion un peu subtile, on se reportera au tome 1, pages 173 et suivantes, ainsi que pages 346 et suivantes.

En quelques mots, voilà ce que celà implique: le tableau présent au niveau 1 de la pile est stocké, en mémoire, à une certaine adresse. Aucune nouvelle copie du tableau n'est effectuée avant que de le modifier. Celà signifie que si le tableau provient du contenu d'une variable (rappelé par **RCL**), la modification se fait directement dans la variable.

De même, si le tableau est préalablement dupliqué par **DUP**, la modification se fait aux deux niveaux de la pile.

Pour éviter ce problème, la seule solution est d'exécuter une instruction **NEWOB** sur le tableau avant de songer à le modifier.

Dans la liste ayant servi à fabriquer le programme **'ECHL'**, il suffit d'intercaler les deux adresses **#366CEh** et **#60FACh**, juste au début de la sous-liste. Voilà la signification de ces deux adresses:

- **#366CEh** (Mnémonique: **rot;NewobIfNotFree**): déplace le tableau du niveau 3 vers le niveau 1, et lui applique **newob** si nécessaire (voir tome 1, page 351).

- **#60FACh** (Mnémonique: **3rolld**): replace le tableau au niveau 3.

Une telle modification ralentit légèrement l'exécution du programme (le test évoqué ci-dessus s'exécute maintenant en environ 0.23 secondes, ce qui est encore un peu plus rapide que l'instruction intégrée **RSWP**).

Pour en terminer avec cet exemple, signalons qu'avec le programme **EXTERNPC** (présenté dans le chapitre suivant), il faut partir (au lieu de la liste utilisée ci-dessus) d'un programme source s'écrivant:

```
begin
   need3    ChkArg    <411h>
   begin
      rot;NewobIfNotFree      3rolld
      2r=>2s          dup2
      min(2s)=0?        ?:BAV!
      dup2         max(2s)
      4pick        nrows(ar)        r=>s
      >(2s)?       ?:BAV!
      SwapRows(ar,2s)        drop2
   end
end
```

ce qui est tout de même plus clair que la liste de la page précédente...

EXEMPLE 4: (Nouvelle version de **→EXT**)

Nous allons utiliser la première version de **→EXT** pour en former une deuxième version pleine d'Externals....

Comme dans l'exemple précédent, nous devons faire attention à ce qu'aucune erreur d'exécution ne puisse avoir de conséquence fâcheuse...

Deux adresses permettent de vérifier la présence d'un objet sur la pile, puis de tester que cet objet est bien une liste.

- **#18AB2h**: (Mnémonique **need1**): vérifie que la pile contient au moins un objet (sinon erreur "*Too Few Arguments*").

- **#19443h**: (Mnémonique **dup;?#li=>BAT!**): vérifie que l'objet du niveau 1 est bien une liste, sans quoi une erreur "*Bad Argument Type*" est générée (ce test ne détruit pas l'objet du niveau 1).

Par prudence, on doit tester si la liste transmise au programme est vide. Si tel est le cas on la transforme en une structure Rpl vide (la précédente version de **→EXT** aurait conduit à un erreur).

Pour celà on utilise les adresses:

- **#54AFh** (Mnémonique **breaks(cp)**): Développe un objet composé (donc une liste) sur la pile. Au niveau 1 apparaît la longueur de cette liste, sous forme d'un entier-système.
- **#61891h** (Mnémonique **dup;=0(s)?:dq/sk)**: Teste, en le dupliquant, si l'entier-système est nul. S'il l'est, exécute l'objet suivant dans le rpl en cours, puis quitte ce rpl. Sinon saute par dessus cet objet et continue le Rpl en cours.

La liste ayant été développée sur la pile, on va effectuer des rotations successives (à l'aide de l'instruction **ROLL**), pour amener chaque objet au niveau 1 (pour mieux l'étudier). Il faut bien sûr écrire ces traitements individuels dans une boucle. Les boucles par Externals seront amplement commentées dans la suite de ce livre.

Pour l'heure, on se contentera d'utiliser deux adresses pour générer cette boucle (**start(1,s)** et **next**, voir plus loin).

Quand le traitement de ces objets est terminé, il suffit de reconstituer la liste initiale. Les adresses utiles ici sont:

- **#634CAh** (Mnémonique **dup;Ldef''**): duplique l'objet du niveau 1, puis le stocke dans une variable locale de nom vide (cf tome 1, page 320). L'utilité de ce stockage apparaît dans la suite du programme.
- **#73DBh** (Mnémonique **start(1,s)**): démarre une boucle de type **START** dont l'indice est un entier-système prenant toutes les valeurs de <1h> jusqu'à l'entier-système placé au niveau 1 (et qui représente ici la taille de la liste). En l'occurence, l'indice de la boucle ne sera jamais explicitement utilisé ici.
- **#613B6h** (Mnémonique **1Lget**): rappelle le contenu de la dernière variable locale définie (qui ici renferme la taille de la liste). Voir le tome 1 page 314.
- **#3325h** (Mnémonique **roll(s)**): effectue une instruction **ROLL** sur l'objet situé au niveau s (où s est un entier-système placé au niveau 1, mais qui ne doit pas être compté dans la numérotation des niveaux).

- Ici doit se situer la procédure de traitement d'un quelconque des objets de la liste initiale.

- **#7334h** (Mnémonique **next**): indique la fin d'un passage dans la boucle, incrémente l'indice de cette boucle (et sort si cet indice dépasse le maximum autorisé).
- **#634B6h** (Mnémonique **1Lget;Lpop**): Rappelle le contenu de la dernière variable locale créée, puis libère la mémoire de cette variable, devenue inutile (cf tome 1, page 309).
- **#5445h** (Mnémonique **−>rpl(s)**): crée une struture Rpl de **s** objets, en utilisant le nombre d'éléments spécifié par l'entier-système **s** placé au niveau 1.

Il reste maintenant à écrire la suite d'instructions réalisant la transformation éventuelle d'un objet particulier de la liste initiale, selon que cet objet est

- un entier binaire, ou
- un réel, ou
- une structure Rpl, ou
- une sous-liste, ou
- un objet-taggué.

Si on reprend la logique du programme tel qu'il est déjà connu, l'objet à transformer est au niveau 1, et le résultat de cette transformation doit être également placé au niveau 1 (les niveaux supérieurs de la pile ne devant pas subir de modification).

On va utiliser, pour "aiguiller" le flot des instructions en fonction du type de l'objet considéré, une adresse qui a été employée à plusieurs reprises.

- **#18FB2h** (Mnémonique **ChkArg**): Cette adresse permet de tester successivement plusieurs combinaisons possibles de 1 à 5 arguments sur la pile, et de diriger sur un objet à évaluer spécifiquement dès qu'une de ces combinaisons est reconnue.
  Cette adresse doit figurer dans une structure Rpl (en principe vers le début). Si aucune combinaison d'arguments n'est reconnue avant la fin de cette structure, il y a une erreur "*Bad Argument Type*".

- Chaque combinaison d'arguments est représentée par un entier système. Pour ne pas trop entrer dans les détails, et nous limiter à ce qui est utile ici (nous en verrons beaucoup plus dans la suite de ce livre), signalons que:

L'entier **<8h>** permet de reconnaître un **entier-binaire** au niveau 1.
L'entier **<1h>** permet de reconnaître un **nombre réel** au niveau 1.
L'entier **<8h>** permet de reconnaître une **structure Rpl** au niveau 1.
L'entier **<5h>** permet de reconnaître une **liste** au niveau 1.
L'entier **<0h>** permet de reconnaître une **pile quelconque**.

D'une façon symbolique, la structure Rpl traitant le problème de la transformation éventuelle de l'objet du niveau 1 est:

```
begin                                  Dans cette représentation, les
   ChkArg                              objets Objet_#, Objet_Rpl, ...
   <Bh>    Objet_#                     Objet_{} sont respectivement
   <1h>    Objet_Real                  évalués quand, au niveau 1, on
   <8h>    Objet_Rpl                   trouve: un binaire, un réel, un
   <5h>    Objet_{}                    Rpl ou une liste.
   <0h>    dtag                        Sinon    dtag    est évalué.
end
```

Quand un des quatre types d'argument est reconnu, l'objet qui lui correspond est évalué, puis on sort de la séquence Rpl décrite ci-dessus.

Si aucun argument de type entier binaire, ou liste, ou Rpl, n'est reconnu, **dtag** est évalué (c'est la forme interne de **DTAG**: il n'y a pas d'erreur si l'objet n'est pas taggué).

Nous allons maintenant écrire chacune des séquences de traitement évoquées ci-dessus sous le nom **Objet_#, Objet_Real, Objet_Rpl, Objet_{}**.

<u>Cas des objets intégrés:</u>

La commande **TYPE** (dans la première version de **→EXT**) permettait de faire la différence entre les commandes intégrées (type=19), les fonctions intégrées (type=18) et les autres structures Rpl (type=8).

Le schéma ci-dessus ne permet plus de distinguer ces objets (tous sont des Rpl). Comme nous ne souhaitons pas toucher aux objets intégrés, **Objet_Rpl** devra en tenir compte.

Traitement d'un entier binaire:

On procède comme on l'a fait dans la première version de **→EXT**: on place l'entier-système <2h> (représenté ici par le réel 2) sur la pile, et on exécute l'instruction get(cp,s) (adresse **#62B9Ch**)

Traitement d'un nombre réel:

On exécute l'instruction **r=>s** (conversion d'un réel en entier binaire, avec d'abord *round*, puis *abs*), qui se trouve à l'adresse **#18CD7h**.

Traitement d'une séquence Rpl:

Comme celà a été dit plus haut, il faut vérifier si cette structure est un objet intégré à la HP48. Si tel est le cas, on ne la modifie pas. On utilise pour celà les adresses suivantes:

- **#61FA9h** (Mnémonique **dup;StdObj?**): Duplique l'objet et ajoute un booléen égal à True si c'est un objet intégré.

- **#61A3Bh** (Mnémonique **?:q**): cette adresse teste si le booléen du niveau 1 vaut True. Si tel est le cas, on quitte le Rpl en cours.

Sinon, on transforme la structure Rpl en une liste, sur laquelle on applique le programme **→EXT**. On utilise pour celà:

- L'adresse **#3F9Fh**: c'est celle de l'entier-système **<2A74h>**, représentant l'adresse-préfixe des listes.

- L'adresse **#5AB3h** (Mnémonique **ModType(_,s)**): son rôle est de convertir la structure Rpl en une liste (ct tome 1, pages 197 et suivantes).

- Le nom **→EXT**, pour l'appel récursif.

Traitement d'une sous-liste:

Très simple: l'objet à évaluer est tout simplement le nom **→EXT**.

Traitement d'un objet quelconque:

Cette éventualité n'apparaît qu'après les trois autres.
On se contente d'un appel à l'adresse **#64775h** (Mnémonique **dtag**), qui est la forme interne de **DTAG**.

Nous sommes maintenant en mesure d'écrire la liste que va devoir transformer la première version de **→EXT**. Voici cette liste:

```
{  #18AB2h   #19443h   #54AFh   #61891h
   #5445h   #634CAh   #73DBh   #613B6h
   #3325h
      {  #18FB2h
         11  {  2   #62B9Ch  }
         1   #18CD7h
         8   {  #61FA9h   #61A3Bh
                #3F9Fh   #5AB3h   →EXT
             }
         5   →EXT
         0   #64775h
      }
   #7334h
   #634B6h   #5445h
}
```

Cette liste n'est évidemment pas très présentable. Elle est cependant la conséquence exacte des indications précédentes.

Pour améliorer tout de même sa lisibilité, elle est plus ou moins "indentée" à droite, pour qu'y apparaisse la logique du programme.

N.B: pour éviter toute erreur, vérifiez bien que la liste ci-dessus a une taille de **318 octets**, et un CheckSum de **#7125h**...

Il ne vous reste plus qu'à appliquer la première version de **→EXT** (la seule pour l'instant) à cette liste. On obtient une structure Rpl de taille **100 octets** (à comparer avec les 258 octets de la version initiale !) et de **CheckSum #4D9Eh**.

On stocke ensuite la structure Rpl obtenue, dans la variable **→EXT**, pour obtenir la deuxième version de ce programme (la vie est injuste: la première version de **→EXT** vient à l'instant de créer une 2-ième version, qui, à peine née, la remplace).

IMPORTANT: à partir de maintenant et jusqu'à désormais, quand on fera allusion au programme →EXT, il s'agira toujours de cette deuxième version.

On peut comparer la rapidité des deux versions du programme **→EXT** (bien que celà ne soit pas fondamental ici: on passe beaucoup plus de temps à concevoir la liste qu'à la transformer par le programme...). Les deux versions ont été appliquées à la liste de la page précédente. Les résultats sont éloquents:

- Première version: 1.98 seconde.
- Deuxième version: 0.17seconde.

La deuxième version (avec Externals) est donc plus de 10 fois plus rapide que la première (tout en étant 2.5 fois plus courte) !

→*EXT (deuxième version)*

Pour en terminer avec ce dernier exemple (et en même temps avec ce chapitre), et pour assurer une transition habile, voici quel texte il aurait fallu utiliser avec le programme **EXTERNPC** pour créer la deuxième version de **→EXT**.

```
begin
   need1      dup;?#li=>BAT!
   breaks(cp)      dup;=0(s)?;dq/sk      ->rpl(s)
   dup;Ldef''
   start(1,s)
      1Lget      roll(s)
      begin
         ChkArg
         <Bh>     begin    <2h>    get(cp,s)    end
         <1h>     r=>s
         <8h>     begin
            dup;StdObj?      ?:q
            <2A74h>     ModType(_,s)      →EXT
         end
         <5h>     →EXT
         <0h>     dtag
      end
   next
   1Lget;Lpop      ->rpl(s)
end
```

LE PROGRAMME EXTERNPC

Ce chapitre est consacré à la présentation d'un programme qui vous permettra d'écrire, beaucoup plus rapidement qu'avec →EXT, des programmes en Externals.

Le programme EXTERNPC, comme son nom l'indique, tourne sur les compatibles PC, sous DOS. Il constitue, si on l'interface avec le logiciel KERMIT et un éditeur de textes ASCII, un environnement intégré performant.

Les capacités de stockage de l'ordinateur vous permettent alors d'écrire vos programmes en utilisant directement les identificateurs issus du fichier de mnémoniques que vous aurez vous-même constitué.

Le chapitre précédent a été consacré au programme **→EXT**, dont la principale qualité est qu'il fonctionne directement sur la HP48, sans aucune aide matérielle extérieure.

Son principal défaut est qu'il nous oblige à travailler sur des données peu lisibles (des adresses exprimées sous la forme d'entiers binaires). Il faut alors compulser sans cesse le tome 1 (mais je ne vous dirai pas que c'est une mauvaise lecture...) à la recherche des adresses correctes.

Le programme **EXTERNPC** va nous permettre de passer à la vitesse supérieure. Il s'agit en fait d'un véritable interpréteur permettant de traduire un texte représentant symboliquement une séquence Rpl (les ***adresses*** qui forment cette séquence étant remplacés par des ***mnémoniques***).

Les quelques exemples du chapitre précédent vous ont donné une idée du "***code source***" que vous serez amené à utiliser. Ces **fichiers sources** seront traduits par le programme **EXTERNPC** sous forme de **fichiers objets**, directement chargeables sur la HP48, par la liaison série. Pour celà, il faut tout de même un peu de matériel.

◆ LE MATÉRIEL REQUIS

Il vous faut, pour pouvoir utiliser le programme **EXTERNPC**:

- Une **HP48** (jusque là, ça doit aller).
- Un ordinateur de type **compatible PC** (un seul lecteur suffit, mais un disque dur est recommandé).
- Un éditeur permettant de produire des **textes ASCII** sur l'ordinateur (n'importe quel traitement de texte à une possibilité de sauvegarde en ASCII). On peut également utiliser l'éditeur d'un langage de programmation, ou celui du système d'exploitation de votre ordinateur.
- Le "*Serial Interface Kit*" de **Hewlett-Packard** (logiciel **KERMIT** et cable de liaison avec l'ordinateur compatible PC).

Pour ce qui est de l'ordinateur, il est tout à fait possible de travailler avec un **MacIntosh** (l'interface série de **Hewlett-Packard** existe là aussi) ou un autre type de machine (**Commodore Amiga**, **Atari**, etc...) à condition de se procurer le logiciel **KERMIT** (il est dans le domaine public) et le cable de liaison, et à condition également d'adapter à chaque ordinateur, dans un langage approprié, le programme **EXTERNPC.**

◆ LES DONNÉES REQUISES

Le programme **EXTERNPC**, listé ci après, a été écrit en Turbo-Pascal (pour version 4 ou ultérieures).

Il vous faut bien entendu disposer de ce langage de programmation (ou alors vous faites l'effort de la traduction dans un autre langage).

Vous devez saisir le listing du programme **EXTERNPC**, puis le compiler sur disque. Peu importe le nom que vous donnez au fichier source et au fichier objet: j'ai personnellement choisi les noms **EXTERNPC.PAS** pour le source et **EXTERNPC.EXE** pour l'exécutable.

Vous devez ensuite constituer un **fichier de mnémoniques**: vous y mettrez les adresses que vous êtes susceptible d'utiliser, accompagnées des mnémoniques que vous aurez choisis pour les désigner (vous pouvez bien sûr conserver les mnémoniques que je vous propose dans les deux tomes de ce livre). La façon de constituer ce fichier est décrite plus loin.

Enfin vous devez enregistrer les fichiers-sources des programmes que vous voulez faire traduire à **EXTERNPC** lors de la prochaine session de travail. Dans toute la suite de cette présentation, on appellera "*fichiers sources*" les textes écrits en ASCII à l'aide de mnémoniques (et que l'on veut faire traduire par **EXTERNPC**) et "*fichiers objets*" les résultats de cette traduction (directement chargeables sur la HP48).

◆ UNE SESSION DE TRAVAIL AVEC EXTERNPC

Voilà comment peut se passer une séance de travail avec **EXTERNPC**:

- On lance le programme **EXTERNPC.EXE** sur l'ordinateur.
- Vous indiquez le *fichier de mnémoniques* que vous désirez utiliser (c'est un texte ASCII). Le programme procède alors à la lecture de ce fichier, puis au tri des enregistrements qu'il contient (par ordre croissant des mnémoniques).
- Vous pouvez alors indiquer le nom d'un *fichier source* à traduire (c'est un texte ASCII) et celui du *fichier objet* qui recevra cette traduction (ce sera un fichier binaire).
- Le programme procède alors à cette traduction, en remplaçant chacun des *mnémoniques* reconnus par l'*adresse* qui lui correspond. La traduction est écrite dans le *fichier objet*.
- Au cours de la traduction, le programme **EXTERNPC** vous indique les erreurs éventuelles dans le fichier source et vous permet de les corriger.
- Une fois que vous avez terminé les opérations de traduction d'un fichier source, vous pouvez recommencer, ou quitter le programme, ou pourquoi pas utiliser un nouveau fichier de mnémoniques.
- Tout en restant dans le programme **EXTERNPC**, si vous avez connecté la HP48 à votre ordinateur, vous pouvez lancer le programme **KERMIT**. Les fichiers objets que vous avez générés avec **EXTERNPC** sont alors chargeables sur la HP48, en mode binaire.

◆ LE FICHIER DES MNÉMONIQUES

La création de ce texte est un préalable indispensable à l'utilisation du programme **EXTERNPC**. Le nom que vous lui donnez n'a aucune importance.

Un certain nombre de conditions doivent impérativement être réunies pour que le programme **EXTERNPC** puisse l'utiliser correctement:

Format du fichier des mnémoniques:

Le fichier des mnémoniques doit être un texte **ASCII**. Si vous l'éditez sur un traitement texte standard, vous ne devriez pas rencontrer de problème pour produire un tel document. Evitez en tout cas d'utiliser des "*enrichissements*" (souligné, italique, gras, changement de police de caractères, etc...).

Dans ce fichier ASCII, chaque ligne ne commençant pas par le caractère **# ne sera pas prise en compte** par le programme **EXTERNPC**. C'est un bon moyen d'inclure des commentaires ou des titres dans votre texte.

Couples "adresse-mnémonique":

Pour indiquer une adresse et un mnémonique, vous devez procéder de la manière suivante:

Indiquez d'abord **l'adresse**, sous la forme d'un entier binaire **a cinq chiffres hexadécimaux**, en utilisant le format: **#xxxxx**. Il est indispensable que le caractère **#** débute la ligne. Pensez bien aux 5 chiffres, même si le premier est nul (ainsi l'adresse de la forme interne de **DUP** doit être écrite **#03188**).

Indiquez ensuite **le mnémonique** correspondant, **sur la même ligne**, séparé de l'adresse par (au moins) **un espace ou une tabulation**.

Caractères autorisés:

Un mnémonique peut être constitué d'une séquence quelconque de caractères **à l'exception des espaces et des tabulations**, et du caractère | (code ASCII 124).

Seuls les 20 premiers caractères du mnémonique seront cependant pris en considération par le programme **EXTERNPC** (mais ce paramètre peut être modifié).

Commentaires:

Un mnémonique peut être suivi d'un commentaire. Ceux-ci doivent être séparés par (au moins) un espace ou une tabulation. Le caractère | (code 124) peut également servir de séparateur.

Les commentaires éventuels suivant un mnémonique ne sont pas pris en considération par le programme **EXTERNPC**.

Sur une même ligne (commençant donc par le caractère **#**) ne peut donc prendre place qu'un seul couple adresse+mnémonique.

Tri des mnémoniques:

Vous n'avez pas à vous soucier de l'ordre dans lequel apparaissent ces couples les uns par rapport aux autres. Le programme **EXTERNPC** procèdera en effet à un tri alphabétique croissant des mnémoniques (rassurez vous, la procédure de tri est très rapide: même avec 2500 couples adresse+mnémonique, le tri s'effectue en moins de trois secondes).

Problèmes d'homonymie:

Evitez bien sûr d'attribuer un même mnémonique à deux adresses différentes (et souvenez vous que seuls les vingt premiers caractères sont pris en compte).

Par contre vous pouvez très bien associer plusieurs mnémoniques distincts à une même adresse. L'adresse **#0312Bh** peut ainsi être considérée comme l'adresse-suffixe des structures Rpl (et mériter le mnémonique **end**) ou l'adresse-suffixe des listes (et être alors représentée par le mnémonique **}**).

Nombre maximum de mnémoniques:

Le nombre maximum de couples adresse+mnémonique que peut accepter le programme **EXTERNPC** est pour l'instant fixé à 5000, mais vous pouvez modifier ce paramètre. Il s'agit en fait d'un équilibre à trouver entre le nombre de mnémoniques autorisés et leur taille maximum (pour l'instant 20 caractères).

◆ LES MNÉMONIQUES A CHOISIR

Même si le nombre de 5000 mnémoniques peut paraître très important, il vous faut tout de même faire un choix: en effet le tome 1 de ce livre contient plus de 6000 adresses (et le tome 2 quelques centaines). Peut-être voudrez vous également y faire figurer les adresses des instructions standards de la HP48 (voir tome 1, pages ??? et suivantes).

Toutes ces adresses ne sont bien sûr pas indispensables, et on verra plus loin qu'il est possible, dans le fichier source, de spécifier directement des adresses à inclure dans le fichier objet, sans utiliser de mnémonique.

Néanmoins, voilà quelques conseils pour choisir au mieux ces mnémoniques.

Distinguons d'abord les adresses correspondant à des données de celles qui pointent sur des objets exécutables (*structures Rpl*, *primitive code objects*, *code objects*).

Le programme **EXTERNPC** permet d'inclure dans le fichier source des données quelconques appartenant à quelques types courants (nombres réels ou complexes, chaînes de caractères, entiers-système, entier binaire, noms globaux ou locaux, characters).

Pour tout objet d'un de ces types qui apparaîtrait en ROM, il n'est donc pas indispensable d'utiliser un mnémonique (vous pouvez reconstituer l'objet directement dans votre fichier source).

Il est intéressant tout de même de prévoir quelques adresses pointant sur des objets simples et classiques (entiers-système, réels).

Ainsi l'adresse **#03FEFh** pourra être incluse dans le fichier des mnémoniques, accompagnée du mnémonique **<0h>** (cette adresse pointe en effet sur cet entier système). Mais un entier-système moins usuel, tel que **<800h>**, pourra fort bien être entré directement dans le fichier source (le programme se chargeant de le traduire).

En règle générale donc, on traitera sous la forme adresse+mnémonique les objets de type "*donnée simple et usuelle*" (par exemple les entiers-système de <0h> à <FFh>, ou encore les réels de -9 à 9 qui sont d'ailleurs considérés par la HP48 comme des objets intégrés).

L'intérêt qu'il y a à remplacer (dans le programme objet) un réel tel que 6, par exemple, par son adresse en ROM (#2A332h) plutôt que par le contenu de cet objet, est que l'adresse n'occupe 2.5 octets alors que le contenu en occupe 10.5 ...

Pour ce qui est des adresses pointant sur des objets exécutables (les innombrables **SYSEVALs** de la deuxième partie du tome 1, ainsi que ceux qui seront décrits dans la suite de ce tome 2), il n'est pas trop difficile de choisir les adresses les plus utiles: dans chacun des chapitres du tome 1, par exemple, les SYSEVALs sont organisés en tableaux selon leur action et leur utilité.

De toutes façons, le programme **EXTERNPC** vous donne la possibilité de charger simultanément plusieurs fichiers de mnémoniques (et de renoncer au fichier en cours pour en charger d'autres).

Une bonne stratégie consiste donc à prévoir plusieurs fichiers séparés:

- Un fichier principal, contenant tous les mnémoniques vraiment indispensables.
- Des fichiers plus spécialisés, que vous ne fusionneriez avec le précédent que pour assurer la traduction de fichiers sources particuliers (en restant toujours dans la limite des 5000 mnémoniques simultanés).

Objets intégrés de la HP48:

Il vous faut sans doute prévoir d'inclure dans un fichier de mnémoniques un certain nombre d'**adresses standards** de la HP48 (voir tome 1, pages ??? et suivantes). Pour une telle adresse, on utilisera évidemment comme mnémonique le nom de l'instruction intégrée qui lui correspond.

Majuscules et minuscules:

Le programme **EXTERNPC**, quand il scrute votre fichier source, fait la différence entre caractères majuscules et minuscules. C'est pourquoi j'ai en général utilisé des minuscules pour désigner les mnémoniques correspondant à des adresses non standards (par exemple, le mnémonique **dup** pour l'adresse **#03188h**, qui désigne la forme interne de l'instruction **DUP**, qui se trouve, elle, à l'adresse **#1FB87h**).

◆ EXEMPLE DE FICHIER DE MNÉMONIQUES

L'exemple suivant montre un fichier de mnémoniques réduit à sa plus simple expression. Il sert uniquement à décrire à nouveau le format de présentation évoqué dans les lignes précédentes.

```
NOMBRES RÉELS

#2A2B4h:        0
#2A2C9h:        1
#2A2DEh:        2
#2A2F3h:        3

ENTIERS-SYSTEME

#03FEFh:        <0h>
#03FF9h:        <1h>
#04003h:        <2h>
#0400Dh:        <3h>

NOMS LOCAUX

#34D30h:        local''         C'est le nom local vide

LA PILE

#03188h:        dup             Forme interne de DUP
#031ACh:        dup2            Forme interne de DUP2
#03223h:        swap            Forme interne de SWAP
#03244h:        drop            Forme interne de DROP

BOOLÉENS

#03A81h:        true            Le booléen "Vrai"
#03AC0h:        false           Le booléen "Faux"

ADRESSES STANDARDS

#1B4ACh:        SIN
#1B505h:        COS
#1B55Eh:        TAN

ENTIERS-SYSTEME

#03DEFh:        1+(s)           Incrémente un entier-système
#03E0Eh:        1-(s)           Décrémente un entier-système
#03E2Dh:        2+(s)           Ajoute <2h> à un entier-système
#03E4Eh:        2-(s)           Retranche <2h> à un entier-système

ADRESSES PRÉFIXES ET SUFFIXES

#02A74h:        {               Début d'une liste
#02D9Dh:        begin           Début d'une structure Rpl
#0312Bh:        end             Fin d'une structure Rpl
#0312Bh:        }               Fin d'une liste
```

Rappelons que le programme **EXTERNPC**, quand il lit le fichier des mnémoniques, ne s'intéresse qu'aux lignes commençant par le caractère #, et qu'il lit l'adresse débutant cette ligne en utilisant les caractères des colonnes 2,3,4,5,6.

Deux adresses sont capitales dans l'exemple précédent:

- **#02D9Dh** (Mnémonique **begin**): adresse préfixe d'une structure Rpl
- **#0312Bh** (Mnémonique **end**): adresse suffixe d'une structure Rpl

Ces deux mnémoniques seront sans doute les plus employés dans tous les exemples de fichier source donnés tout au long de ce livre. Ils servent en effet de délimiteur aux structures Rpl (la structure principale et toutes les structures imbriquées).

◆ LES INCLUSIONS DE FICHIERS SECONDAIRES:

Le programme **EXTERNPC** vous permet d'insérer dans votre fichier-source principal, des **fichiers-source secondaires**, ainsi que le contenu de **fichiers-code**. Il devient alors plus facile de créer des applications importantes.

Un **fichier-source secondaire** est un texte constitué de mnémoniques, tout comme le fichier-source principal.

Pour inclure un tel fichier secondaire, vous devez faire précéder son nom par le caractère [(crochet ouvrant). Dans ce nom, le suffixe par défaut est '**SRC**'. Attention, il ne s'agit pas ici d'un fichier binaire (contenant une structure Rpl) tel que le crée le programme **EXTERNPC**. Les mnémoniques du fichier-source inclus sont converties par **EXTERNPC** et leur traduction est ajoutée au programme-objet en cours de création, avant retour au programme-source principal.

Aucune vérification n'est effectuée sur la nature du fichier à inclure. Dans chaque fichier-source inclus, vous pouvez placez des directives d'inclusion d'autres fichiers secondaires.

Un **fichier-code** est un fichier dont le contenu est un objet de type code, sauvegardé depuis la HP48 en mode binary, ou créé par le programme **ASMPC** (voir le chapitre 8).

Pour inclure un tel fichier-code, faites précéder son nom de deux crochets ouvrants: [[. Dans ce nom, le suffixe par défaut est '**COD**' (attention il s'agit d'un fichier binaire, et non pas d'un texte qui contiendrait les instructions en assembleur). Le contenu de ce fichier-code est ajouté au fichier-objet en cours de création, avant retour au programme-source principal.

Le programme **EXTERNPC** vérifie seulement que le fichier que vous voulez inclure contient bien un objet de type-code. C'est tout le contenu de cet objet-code qui est incorporé au fichier-objet final, y compris l'adresse-préfixe et la taille sur 5 quartets qui suit cette adresse-préfixe.

◆ LE LISTING DU PROGRAMME EXTERNPC:

Les pages suivantes contiennent le code-source du programme **EXTERNPC**. Les spécialistes de Turbo-Pascal trouveront sans doute, ça et là, bien des améliorations possibles. Je ne prétends pas avoir produit là un programme sans défauts; je me contente de le trouver, à travers mon expérience personnelle, particulièrement utile.

Les esthètes de Turbo-Pascal ne me pardonneront peut-être pas de ne pas avoir ordonné les différentes procédures selon les canons intangibles de la programmation structurée: même si certaines procédures sont logiquement imbriquées, j'ai préféré, pour des raisons de présentation, les placer les unes à la suite des autres.

D'autre part, toujours pour des questions typographiques, j'ai été amené à grouper (assez souvent) plusieurs instructions consécutives sur une même ligne du code-source (je sais, ça ne se fait pas).

Je n'ai pas indiqué de commentaires pour toutes les lignes du programme. Seuls les en-têtes de procédures et de fonctions sont succintement documentés.

Il semble cependant utile, pour faciliter votre compréhension du programme, et bien sûr son adaptation à vos besoins, de présenter rapidement le plan logique (le "*script*") d' **EXTERNPC**.

Programme principal:

Il Appelle la procédure "**Init**", qui contient le choix du chemin où trouver **KERMIT**, et de celui où trouver votre traitement de texte. Cette procédure effectue également quelques initialisations (d'où son nom).

Il affiche un menu permettant de choisir une options parmi 8.

- Option 0: quitter le programme
- Option 1: lancer l'édition d'un fichier-source spécifié (procédure "**Edition**")
- Option 2: convertir un fichier-source en fichier-objet (procédure "**Conversion**")
- Option 3: transmettre un fichier-objet à la HP48 (procédure "**Transmission**")
- Option 4: lancer le logiciel **KERMIT**, sans indiquer de fichier particulier
- Option 5: lancer le traitement de texte, sans indiquer de fichier-source
- Option 6: charger un fichier de mnémoniques (procédure "**Charger_mnemos**")
- Option 7: "*oublier*" les mnémoniques déjà chargés.

On revient à ce menu principal dès que l'option choisie a été exécutée.

Procédure "Charger_mnemos":

Elle appelle la procédure "**Lecture_mnemos**", qui demande le nom du fichier des mnémoniques, le lit et mémorise chacun de ses mnémoniques, et chacune des adresses en ROM qui lui correspondent.
Elle trie la liste des mnémoniques chargés (procédure "**Tri_mnemos**").

Procédure "Edition":

Elle vous demande le fichier à éditer (procédure "**Choix_Fichier_Source**").
Elle lance le traitement de texte en lecture de ce fichier.

Procédure "Conversion":

Elle ouvre le fichier-source en lecture (procédure "**Ouvrir_fichier_source**").
Elle ouvre le fichier-objet en écriture (procédure "**Ouvrir_fichier_objet**").
Elle écrit, dans le fichier-objet le "*header*" (HPHP48-M, par exemple) pour permettre son identification par la HP48.
Elle appelle le procédure **ConvFile** qui traite chacune des lignes du fichier-source (procédure "**Convertir_ligne**").
Elle ferme les deux fichiers (procédure "**CloreConversion**").

Procédure "Transmission":

Elle choisit le fichier-objet à tansmettre (procédure "**Choix_Fichier_Objet**").
Elle lance **KERMIT** avec la commande S (*send*) appliquée à ce fichier.

Procédure "Convertir_ligne"

(dans la procédure "**ConvFile**")

Elle lit une ligne du fichier-source.
Elle y efface les commentaires éventuels (procédure "**CleanComment**").
Tant que cette ligne n'est pas vide (càd lue entièrement):
1) Elle isole un identificateur (Fonction "**Item**").
2) Elle traite cet identificateur (Procédure "**Traiter_item**").

Fonction "Item"

Elle nettoie les espaces et tabulations de début de ligne.
Si le premier caractère rencontré est un " (début de chaîne), alors elle lit cette chaîne (jusqu'au prochain ").
Sinon elle lit les caractères successifs jusqu'à rencontrer un espace, une tabulation, ou la fin de la ligne.
L'identificateur est maintenant isolé. La ligne est purgée de celui-ci, de manière à être prête pour l'identificateur suivant.

Procédure "Traiter_item"

Un identificateur vient donc d'être isolé dans le fichier-source.
On trouve sa position d'insertion dans le fichier des mnémoniques, au moyen de la fonction "**position**" (recherche dichotomique)
Si à cette position on trouve bien un mnémonique identique à l'identificateur en cours de traitement, alors le résultat de l'interprétation de ce mnémonique est l'adresse qui lui correspond (fonction "**adresse**" et variable "**result**").
Sinon on essaie d'interpréter cet identificateur (procédure "**Traiter_unknown**")

Procedure "Traiter_unknown"

Selon le premier caractère de cet identificateur non reconnu dans le tableau des mnémoniques, on dirige l'interprétation dans telle ou telle direction:

- Identificateur débutant par [: Procédure **Inclure**.
- Identificateur débutant par < : Procédure "**Traiter_system_integer**"
- Identificateur débutant par 0,..,9 ou le signe + ou −: Procédure "**Traiter_real**"
- Identificateur débutant par \ : Procédure "**Traiter_sequence**"
- Identificateur débutant par # : Procédure "**Traiter_binary**"
- Identificateur débutant par (: Procédure "**Traiter_complex**"
- Identificateur débutant par " : Procédure "**Traiter_string**"
- Identificateur débutant par $: Procédure "**Traiter_character**"
- Identificateur débutant par £ : Procédure "**Traiter_local**"

Dans les autres cas, on lance la procédure "**Proposer_correction**" qui permet, au choix:

- De proposer un autre identificateur.
- De traduire l'identificateur comme un nom global (procédure "**Traiter_global**")
- De spécifier une adresse en ROM pour cet identificateur.
- D'interrompre la conversion du fichier-source.

Chacune des procédures évoquées ci-dessus peut conduire à une erreur (par exemple un identificateur nommé **<ABC**, interprété sans succès comme un entier-système). Dans ce cas, la variable **ErrNum** est chargée avec le code d'erreur qui correspond à la situation. Au retour dans la procédure **Convertir_Ligne**, il y aura un appel à la procédure "**Proposer_correction**".

Quand l'identificateur lu dans le fichier source a été correctement interprété, son image est ajoutée dans le tableau **tab** (procédure "**ajouter**"). L'écriture dans le fichier-objet s'effectue à chaque fois que ce tableau est plein (procédure "**EcrireBuffer**").

Lorsque l'un des mnémoniques est une chaîne de caractères, ou un nom de variable locale ou globale, chaque caractère (par exemple **C**, de code ascii **67=#43h**) doit être codé sur deux quartets dans le fichier-objet final, mais d'abord être codé sur deux caractères dans le tableau **tab**. La procédure **ConvertAscii** se charge de la conversion.

Un appel à la procédure "**ConvertAscii**" est précédé par un appel à la procédure "**Traduction**" qui remplace la séquence de deux caractères – > (dans le fichier-source) par le caractère → de code 141 (dans le fichier-objet). Vous pouvez bien sûr envisager d'autres conversions, pour vous adapter à la table des caractères de code > 128 qu'utilise la HP48.

Remarque: toujours pour obtenir, dans le fichier-objet, le caractère → de la HP48, qui n'existe pas dans la table ASCII utilisée par les compatibles PC, vous pouvez utiliser le caractère ì (code ascii 141) dans votre fichier-source.

Procedure Inclure

C'est elle qui est chargée d'assurer la lecture des fichiers secondaires appelés depuis le fichier-source principal. Elle est appelée pour interpréter un mnémonique (non localisé dans le fichier des mnémoniques) commençant par le caractère [.

Ce caractère [est d'abord éliminé.

Si le premier caractère est encore [(c'est-à-dire si le mnémonique commençait par [[), alors on appelle la procédure **InclureCode**, pour ajouter au code-objet le contenu du fichier binaire dont le nom suit [[. Comme il a déjà été dit, ce contenu doit être un objet de type code (adresse préfixe #02DCCh).

Sinon la procédure **Inclure** interprète le nom suivant le caractère [comme celui d'un fichier contenant du texte (sous forme de mnémoniques). Ce texte sera traduit par un appel (récursif puisque l'on est actuellement sous le contrôle de cette procédure) à la procédure **ConvFile**.

Les suffixes de fichier:

Quand vous indiquez un nom de fichier au programme **EXTERNPC** (soit à partir du menu principal, lors d'un appel à une des procédures **Edition**, **Transmission**, ou **Conversion**, soit dans le fichier-source principal, dans le cas d'une directive d'inclusion) et si votre nom de fichier ne comporte pas d'extension, alors le programme **EXTERNPC** lui attribue d'office l'une des extensions suivantes:

- **SRC** pour un fichier-source (constitué de mnémoniques).
- **OBJ** pour un fichier-objet (résultant de la traduction d'un fichier-source).
- **COD** pour des fichiers-code (par exemple créés par le programme **ASMPC**) dont le contenu doit être un objet-code.

Le programme **EXTERNPC** mémorise le derniers nom de fichier-source donné par l'utilisateur au clavier. Dans la procédure **Conversion**, par exemple, le programme **EXTERNPC** propose systématiquement de donner au fichier-objet à écrire le même nom que le fichier-source à traduire (à l'extension près, qui devient **OBJ**).

Après toutes ces explications, venons-en au listing de **EXTERNPC**...

```
(************  DEBUT DU LISTING DU PROGRAMME EXTERNPC ************)

Program EXTERNPC;

{$N-} (* Pour désactiver le coprocesseur arithmétique *)
{$M $4000,0,$19000} (* Réserve la place pour exécuter KERMIT, etc..*)

Uses Crt, Dos;

Const
    MaxAdr = 5000; (* nombre maximum de mnémoniques *)
    comment = '|';  (* caractère de code 108 *)
    version :char = 'M';  (* M = version de la HP48 *)
    max=20; (* longueur maximum utilisée d'un mnémonique *)

Type mnemonic = string[max]; string3= string[3];

Var adresse: array[1..MaxAdr] of String[5];  (* adresses retournées *)
    mnemonique: array[1..MaxAdr] of ^mnemonic;
    source: text; objet: file;
    nombre, numline, count, ErrNum: word;
    filename, line, result: string; choix: char;
    ok, stop:boolean; pdebut: pointer;
    pathK: string[79]; (* Chemin pour trouver KERMIT *)
    pathTT: string[79]; (* Chemin pour trouver un éditeur *)
    oldsrc, oldobj: string[79]; (* noms de fichiers utilisés *)
    tab: array[0..2047] of char;
    buffer: array[0..1023] of byte;

Function Valeur(var k:char):byte;
  (* Transforme un caractère de '0' à 'F', en un entier de 0 à 15 *)
  begin
    case k of
      '0'..'9': valeur:=ord(k)-48;
      'A'..'F': valeur:=ord(k)-55
    end;
  end;

Function Car(k:byte): char;
  (* Transforme un entier de 0 à 15, en un caractère de '0' à 'F' *)
  begin
    case k of
      0..9: car:=chr(48+k);
      10..15: car:=chr(55+k);
    end;
  end;
```

```
Function Adr(n:longint):string;
  (* transforme un entier en une adresse, chaîne de 5 caractères *)
  var st:string; i:integer;
  begin
    st:='00000'; if n<0 then n:=n+1+MaxLongInt;
    for i:=1 to 5 do
    begin st[i]:=car(n mod 16); n:=n div 16; end;
    adr:=st;
  end;

Procedure ConvertAscii(var st:string);
  var st2:string; i,n:integer;
  begin
    ErrNum:=11; n:=length(st); if n>127 then exit;
    st2[0]:=chr(2*n);
    for i:=1 to n do begin
      n:=ord(st[i]); st2[2*i-1]:=car(n mod 16);
      st2[2*i]:=car(n div 16);
    end;
    st:=st2; ErrNum:=0;
  end;

Function OkHexa(var st:string): boolean;
  var i:integer; test:boolean;
  begin
    test:=true;
    for i:=1 to length(st) do
      test := test and (st[i] in ['0'..'9','A'..'F']);
    OkHexa:=test;
  end;

Procedure Pause(mess:string);
  (* Emet un bip, puis attend un appui sur ENTRÉE, pour continuer *)
  begin
     write(^G+mess+' (ENTRÉE pour continuer)'); readln; clrscr
  end;

Procedure Choix_Fichier_Source(suff: string3);
  (* Choix d'un fichier-source, pour lancer son édition dans le
     traitement de texte, ou pour le compiler en un fichier-objet *)
  begin
    ok:=true; if oldsrc<>'' then write('('+oldsrc+') ');
    readln(filename);
    if filename<>'' then oldsrc:=filename else ok:=(oldsrc<>'');
    if pos('.',oldsrc)=0 then oldsrc:=oldsrc+'.'+suff;
  end;
```

```
Procedure Ouvrir_Fichier_Source(suff: string3);
  (* Ouverture du fichier source, contenant les instructions sous
       forme de mnémoniques en assembleur *)
  begin
     repeat
       write('Nom du fichier source: '); Choix_fichier_source(suff);
       if ok then begin
         if pos('.',oldsrc)=0 then oldsrc:=oldsrc+'.'+suff;
         assign(source,oldsrc);
         {$I-};reset(source);{$I+};
         ok:= (IoResult=0);
       end;
       if not ok then  begin  write(^G); oldsrc:='' end
     until ok;
  end;

Procedure Choix_Fichier_Objet(suff: string3);
  (* Choix d'un fichier-objet, pour recevoir la traduction
       d'un fichier-source, ou pour transmission à la HP48 *)
  begin
    ok:=true;
    if oldsrc<>'' then  oldobj:=copy(oldsrc,1,pos('.',oldsrc))+suff;
    if oldobj<>'' then write('('+oldobj+') ');
    readln(filename);
    if filename<>'' then  oldobj:=filename
                    else  ok:=(oldobj<>'');
    if pos('.',oldobj)=0 then oldobj:=oldobj+'.'+suff;
  end;

Procedure Ouvrir_Fichier_Objet(suff:string3);
  (* Ouverture du fichier-objet pour recevoir la
        traduction d'un fichier-source *)
  begin
     repeat
       write('Nom du fichier objet: '); Choix_fichier_objet(suff);
       if ok then
       begin
         assign(objet,oldobj);
         {$I-};rewrite(objet,1);{$I+};
         ok:=(IoResult=0);
       end;
       if not ok then begin write(^G); oldobj:='' end
     until ok;
  end;
```

```
Procedure Transmission(suff: string3);
  (* Transmet un fichier-objet à la HP48, via Kermit *)
  begin
    repeat
      Write('Nom du fichier à transmettre: ');
      Choix_Fichier_Objet(suff);
      if not ok then begin write(^G); oldobj:='' end
    until ok;
    Exec(PathK,'S '+oldobj); ClrScr;
    if (DosError+DosExitCode)=0
       then pause('Transmission terminée')
       else pause('Transmission non effectuée');
  end;

Procedure Edition(suff: string3);
  (* Lance la saisie d'un fichier-source, sur l'éditeur *)
  begin
    repeat
      Write('Nom du fichier source: '); Choix_Fichier_Source(suff);
      if not ok then begin write(^G); oldsrc:='' end
    until ok;
    Exec(PathTT,oldsrc); ClrScr;
    if (DosError+DosExitCode)=0
      then pause('Edition terminée')
      else pause('Edition non effectuée');
  end;

Procedure CleanComment(var st:string);
  (* Efface les caractères de la ligne, à partir du
       premier caractère | (code 108) rencontré *)
  var k:integer;
  begin
    k:=pos(comment,st); if k>0 then st:=copy(st,1,k-1);
  end;

Procedure Init;
  (* Initialise le nom, avec le chemin d'accès, du fichier KERMIT
       et de votre éditeur ASCII préféré. Vous remplacerez bien
       sûr les déclarations de PathK et pathTT ci-dessous,
       qui correspondent à mon installation, par les vôtres... *)
  begin
    ClrScr; nombre:=0; oldsrc:=''; oldobj:=''; mark(pdebut);
    PathK:= '\HP\KERMIT\KERMIT.EXE'; (* où trouver Kermit *)
    PathTT:= '\TP6\TURBO.EXE'; (* où trouver un éditeur ascii *)
  end;
```

```
Procedure Affiche_Erreur;
  (* Un erreur a été rencontrée, soit dans la lecture du fichier
       des mnémoniques, soit dans celle d'un fichier source *)
  var mess:string;
  begin
    write(^G);
    case ErrNum of
      1: mess:='Adresse erronée';
      2: mess:='Mnémonique non trouvé';
      3: mess:='Entier-système incorrect';
      4: mess:='Entier binaire incorrect';
      5: mess:='Nombre réel incorrect';
      6: mess:='Nombre complexe incorrect';
      7: mess:='Séquence \... incorrecte';
      8: mess:='Character incorrect';
      9: mess:='Inclusion de fichier incorrecte';
     10: mess:='Inclusion de fichier-code incorrecte';
     11: mess:='Chaîne trop longue';
    end;
    writeln(mess,' en ligne ',numline);
  end;

Function Item(var line:string):string;
  (* Cette fonction isole un nouvel identificateur du fichier source.
     Elle modifie le contenu de la variable "line", qui représente
     la ligne en cours de lecture dans ce fichier source *)
  var st:string;
  begin
    st:='';
    while (line<>'') and (line[1] in [#9,#32])
      do delete(line,1,1);
    if line<>'' then
      if line[1]=#34 then
        begin
          repeat
            st:=st+line[1]; delete(line,1,1);
          until (line='') or (line[1]=#34);
          delete(line,1,1); st:=st+#34;
        end
        else repeat
          st:=st+line[1]; delete(line,1,1);
        until (line='') or (line[1] in [#9,#32]);
    item:=st;
  end;
```

```
Function Position(var ch:string):integer;
  (* Donne la position d'un identificateur du fichier-source,
        dans le tableau des mnémoniques *)
  var deb,fin,mil:longint; st:mnemonic;
  begin
    deb:=1; fin:=nombre;
    while deb<fin do
    begin
      mil:=(deb+fin) div 2;
      st:=mnemonique[mil]^;
      if ch<=st then fin:=mil
                else deb:=mil+1;
    end;
    position:=deb;
  end;

Procedure Tri_Mnemos;
  (* tri rapide de l'ensemble des mnémoniques chargés *)
  var i,j:word; aide1,aide2 :mnemonic;
  Procedure tri(gauche,droite:word);
    begin
      i:=gauche; j:=droite;
      aide1:= mnemonique[(i+j) div 2]^;
      while i<j do
      begin
        while mnemonique[i]^ < aide1 do inc(i);
        while mnemonique[j]^ > aide1 do dec(j);
        if i<=j then
        begin
          aide2:=mnemonique[i]^;
          mnemonique[i]^:=mnemonique[j]^;
          mnemonique[j]^:=aide2;
          aide2:=adresse[i];
          adresse[i]:=adresse[j];
          adresse[j]:=aide2;
          inc(i); dec(j);
        end;
      end;
      if gauche < j then tri(gauche,j);
      if i < droite then tri(i,droite);
    end;
  begin (*de la procedure tri_mnemos*)
    tri(1,nombre);
  end;
```

```
Procedure Lecture_Mnemos;
  (* Lecture du fichier des mnémoniques, et mémorisation de ceux-ci
      ainsi que leurs adresses en ROM *)
  var  MnemoFile: text; name:string;
       st,adr:string[5]; mnemo: mnemonic;
       hexa:longint; i,err:integer;
  begin
    repeat
       write('Nom du Fichier des mnémoniques: '); readln(name);
       assign(MnemoFile,name);
       {$I-};reset(MnemoFile);{$I+};
       ok:=(IoResult=0); if not ok then write(^G);
    until ok;
    writeln('Lecture du fichier des mnémoniques....');
    numline:=0;
    while not eof(MnemoFile) do
    begin
      readln(MnemoFile,line); CleanComment(line); inc(numline);
      if (line<>'') and (line[1]='#') then
      begin
        ErrNum:=0;
        st:=copy(item(line),2,5); val('$'+st,hexa,err);
        if err>0 then ErrNum:=1 else
        begin
          mnemo:=item(line);
          if mnemo='' then ErrNum:=2 else
          begin
            inc(nombre); new(mnemonique[nombre]);
            mnemonique[nombre]^:=mnemo; adr:='';
            for i:=1 to 5 do adr:=st[i]+adr;
            adresse[nombre]:=adr;
          end
        end;
        if ErrNum>0 then Affiche_Erreur;
      end;
    end;
    close(MnemoFile);
    writeln('Lecture terminée: (',nombre,' mnémoniques)');
  end;

Procedure Charger_Mnemos;
  (* Charge puis trie les mnémoniques *)
  begin
    lecture_mnemos; writeln('Tri du tableau en cours....');
    tri_mnemos; pause('Tri terminé');
  end;
```

```
Procedure Traiter_Item(id:string); Forward;
Procedure Traiter_Global(var id:string); Forward;

Procedure Proposer_Correction(var id:string);
  (* L'identificateur "id" n'a pas été reconnu. Le programme vous
       propose de l'interpréter ou de le corriger *)
  var rom:longint; c:char;
  begin
    If ErrNum>0 then Affiche_Erreur;
    writeln; ErrNum:=0;
    writeln('Traitement de l''identificateur: ',id);
    writeln('  1: Proposer un autre identificateur');
    writeln('  2: Traiter cet identificateur comme un nom global');
    writeln('  3: Donner l''adresse en ROM');
    writeln('  4: Interrompre la conversion du fichier');
    repeat
      write('Choisissez: '); readln(c);
    until c in ['1'..'4'];
    case c of
      '1': begin
             writeln; write('Nouvel identificateur: ');
             readln(id); traiter_item(id);
             ClrScr; writeln('Conversion en cours...');
           end;
      '2': traiter_global(id);
      '3': begin
             writeln; write('Adresse en ROM: '); readln(rom);
             id:='\'+adr(rom); traiter_item(id);
           end;
      '4': Stop:=true;
    end;
    ClrScr;
  end;

Procedure EcrireBuffer;
  var i,n: integer;
  begin
    if odd(count) then  begin tab[count]:=chr(0); inc(count); end;
    n:=(count div 2);
    for i:=0 to n-1 do
      Buffer[i]:=valeur(tab[2*i])+16*valeur(tab[2*i+1]);
    BlockWrite(objet,Buffer,n);
    count:=0;
  end;
```

```
Procedure Ajouter(var st:string);
  var i:integer;
  begin
    for i:=1 to length(st) do begin
       tab[count]:=st[i]; inc(count);
       if count=2048 then EcrireBuffer;
    end;
  end;

Procedure InclureCode(var str1: string);
  var CodFile:file; k:integer; n: longint; b:byte;
      Buff: Array[0..12] of byte;
  begin
    ErrNum:=10; delete(str1,1,1);
    if pos('.',str1)=0 then str1:=str1+'.cod';
    assign(CodFile,str1);
    {$I-};reset(CodFile,1);{$I+}; If IoResult<>0 then exit;
    BlockRead(CodFile,Buff,13,k); If k<13 then exit;
    n:=Buff[8]+256*Buff[9]+65536*(Buff[10] mod 16);
    if n<>$02DCC then exit;
    n:=(Buff[10] div 16)+16*Buff[11]+4096*Buff[12];
    result:='CCD20'+adr(n); ajouter(result);
    k:=0; n:=n-5;
    While k<n do begin
      BlockRead(CodFile,b,1); result:=car(b mod 16);
      inc(k,2); if k<=n then result:=result+car(b div 16);
      ajouter(result);
    end;
    ErrNum:=0; result:=''; close(CodFile);
  end;

Procedure ConvFile(name: string; var source: text); forward;

Procedure Inclure(var str1: string);
  var source2: text;
  begin
    delete(str1,1,1);
    if (str1<>'') and (str1[1]='[') then InclureCode(str1) else
    begin  if pos('.',str1)=0 then str1:=str1+'.src';
      assign(source2,str1);
      {$I-};reset(source2);{$I+}; ok:=(IoResult=0);
      if ok then begin ConvFile(str1,source2); close(source2); end
        else ErrNum:=9;
    end;
    result:='';
  end;
```

```
Procedure Traiter_System_Integer(var id:string);
  (* On traite ici les identificateurs inconnus, qui débutent par le
       caractère <, en les interprétant comme des entiers-système *)
  var L,base,i,k: integer; num: longint;
  begin
    L:=length(id); ErrNum:=3;
    if (L<4) or (id[L] <> '>') then exit;
    case id[L-1] of
      'b': base:=2;
      'o': base:=8;
      'd': base:=10;
      'h': base:=16
      else exit;
    end;
    num:=0;
    for i:=2 to L-2 do
    begin
      k:=valeur(id[i]);
      if k>=base then exit;
      num:=num*base+k;
      if num >$FFFFF then exit;
    end;
    Result:='11920'+adr(num); ErrNum:=0;
  end;

Procedure Traiter_Binary(var id:string);
  (* On traite ici les identificateurs inconnus, qui débutent par le
       caractère #, en les interprétant comme des entiers binaires *)
  var L,i:integer; st:string;
  begin
    L:=length(id); ErrNum:=4;
    if L<3 then exit;
    if id[L] <> 'h' then exit;
    for i:=L to 17 do insert('0',id,2);
    L:=length(id); st:='';
    for i:=L-1 downto 2 do
    begin
      if valeur(id[i])>=16 then exit;
      st:=st+id[i];
    end;
    result:='E4A20'+adr(L+3)+st;
    ErrNum:=0;
  end;
```

```
Function Real_String(var chn:string): string;
  (* donne la chaîne image d'un nombre réel, hors adresse préfixe *)
  var err,i,xpon:integer; num:real;
      st,st1,st2:string;
  begin
    val(chn,num,err);
    if err>0 then begin ErrNum:=5; exit; end;
    st:='0000000000000000'; (* chaîne de longueur 16 *)
    if num<>0 then
    begin
      str(num,st1);
      if num>0 then st1[1]:='0'
               else st1[1]:='9'; (* place le signe *)
      delete(st1,3,1);           (* enlève le point décimal *)
      st2:=copy(st1,14,3);       (* chaîne de l'exposant *)
      st1:=copy(st1,1,12)+'0';   (* signe et mantisse *)
      val(st2,xpon,err);       (* xpon= valeur exposant *)
      if xpon<0 then xpon:=1000+xpon;
      str(xpon,st2);
      while length(st2)<3 do st2:='0'+st2; (* chaîne exposant *)
      for i:=1 to 3 do st[i]:=st2[4-i];
      for i:=4 to 16 do st[i]:=st1[17-i];
    end;
    real_string:=st;
  end;

Procedure Traiter_Real(var id:string);
  (* Traitement d'un identificateur inconnu, débutant par + ou - ou
       par un chiffre, et donc interprété comme un réel *)
  begin
    result:='33920'+real_string(id);
  end;

Procedure Traiter_Complex(var id:string);
  (* Traitement d'un identificateur inconnu, débutant par
       une parenthèse (, et interprété comme un complexe *)
  var k:integer; st1,st2:string;
  begin
    k:=length(id); ErrNum:=6;
    if (id[k]<>')') or (k=1) then exit;
    id:=copy(id,2,k-2); k:=pos(',',id); if k=0 then exit;
    st1:=copy(id,1,k-1); st2:=copy(id,k+1,maxint);
    result:='77920'+real_string(st1)+real_string(st2);
    ErrNum:=0;
  end;
```

```
Procedure Traiter_Sequence(var id:string);
  (* On traite ici les identificateurs inconnus, qui débutent par le
     car. \, et les interprétant comme des séquences de quartets *)
  begin
    ErrNum:=7; if id='\' then exit;
    delete(id,1,1); if not OkHexa(id) then exit;
    result:=id; ErrNum:=0;
  end;

Procedure Traduction(var id:string);
  var k: byte;
  begin
    while pos('->',id)>0 do begin
      k:=pos('->',id); delete(id,k,2); insert(chr(141),id,k);
    end;
  end;

Procedure Traiter_String(var id:string);
  (* On traite ici les identificateurs inconnus, qui débutent par le
    caractère ", en les interprétant comme des chaînes *)
  var L:integer;
  begin
    Traduction(id); L:=length(id); result:='C2A20'+adr(2*L+1);
    id:=copy(id,2,L-2); ConvertAscii(id); result:=result+id;
  end;

Procedure Traiter_Local(var id:string);
  (* On traite ici les identificateurs inconnus, qui débutent par le
    caractère £, en les interprétant comme des noms locaux sans £ *)
  var L: integer;
  begin
    delete(id,1,1); Traduction(id); L:=length(id);
    result:= 'D6E20' + car(L mod 16) + car(L div 16);
    ConvertAscii(id); result:=result+id;
  end;

Procedure Traiter_Global(var id:string);
  (* On traite ici les identificateurs inconnus, en les interprétant
      comme des noms globaux *)
  var L: integer;
  begin
    Traduction(id); L:=length(id);
    Result:='84E20' + car(L mod 16) + car(L div 16);
    ConvertAscii(id); result:=result+id;
  end;
```

```
Procedure Traiter_Character(var id:string);
  (* On traite ici les identificateurs inconnus, qui débutent par le
    car. $, en les interprétant comme des Characters sans le $ *)
  var L,err:integer; n:byte;
  begin
    L:=length(id); if L=1 then proposer_correction(id) else
    begin
      delete(id,1,1); val(id,n,err);
      if err>0 then
      begin  id:='$'+id; proposer_correction(id);  end
      else result:='FB920'+car(n mod 16)+car(n div 16);
    end;
  end;

Procedure Traiter_Unknown(var id:string);
  (* Traitement des identificateurs non localisés
       dans le tableau des mnémoniques *)
  begin
     case id[1] of
       '['     : Inclure(id);
       '<'     : traiter_system_integer(id);
       '-','+','0'..'9'
               : begin
                    if id[2]<>'>' then traiter_real(id)
                                  else proposer_correction(id);
                 end;
       '\'     : traiter_sequence(id);
       '#'     : traiter_binary(id);
       '('     : traiter_complex(id);
       #34     : traiter_string(id);
       '$'     : traiter_character(id);
       '£'     : if id<>'£' then traiter_local(id)
                            else proposer_correction(id);
       else proposer_correction(id);
    end;
  end;

Procedure Traiter_Item(id:string);
  (* Traitement d'un identificateur lu dans le fichier-source *)
  var p:integer;
  begin
    if nombre>0 then p:=position(id);
    if (nombre=0) or (copy(id,1,max) <> mnemonique[p]^)
       then traiter_unknown(id) (* item non reconnu *)
       else result:=adresse[p]; (* item reconnu, on écrit l'adresse *)
  end;
```

EXTERNPC, Listing

```
Procedure Convertir_Ligne;
  (* Traitement d'une des lignes du fichier source *)
  var ident: string;
  begin
    CleanComment(line);
    ErrNum:=0;
    while (not stop) and (line<>'') do
    begin
      ident:=item(line);
      if ident <> '' then  traiter_item(ident);
      while ErrNum>0 do Proposer_Correction(ident);
      if (not stop) and (ident<>'') then ajouter(result);
    end;
  end;

Procedure ConvFile(name: string; var source: text);
  (* Conversion d'un fichier-source, principal ou inclus *)
  begin
    while (not stop) and (not eof(source)) do
    begin
      readln(source,line); inc(numline);
      Convertir_Ligne;
    end;
  end;

Procedure CloreConversion;
  (* Fin de conversion du fichier-source en le fichier-objet *)
  begin
    close(source); close(objet);
    if stop then pause('Conversion interrompue')
          else pause('Conversion terminée');
  end;

Procedure Conversion(suff1, suff2: string3);
  (* compile le fichier-source en un fichier-objet *)
  var i:byte;
  begin
    numline:=0; count:=0; result:='HPHP48-'+version; stop:=false;
    ConvertAscii(result); ajouter(result);
    ouvrir_fichier_source(suff1); ouvrir_fichier_objet(suff2);
    ConvFile(oldsrc,source);
    if (count>0) and (not stop) then EcrireBuffer;
    CloreConversion;
  end;
```

```
Function Menu:char;  (*** Menu Initial ***)
  var c:char;
  begin
     clrscr;
     writeln('0: Sortir du programme');
     writeln('1: Editer un fichier-source');
     writeln('2: Convertir un fichier-source en fichier-objet');
     writeln('3: Transmettre un fichier-objet à la HP48');
     writeln('4: Lancer le programme KERMIT');
     writeln('5: Lancer votre éditeur de textes ASCII');
     writeln('6: Charger un fichier de mnémoniques');
     writeln('7: Annuler les ',nombre,' mnémoniques chargés');
     writeln;
     repeat
        write('Faites votre choix: '); readln(c);
     until c in ['0'..'7'];
     menu:=c; ClrScr;
  end;

begin
  Init;
  repeat
    choix:=menu;
    case choix of
      '1':Edition('src');
      '2':conversion('src','obj');
      '3':transmission('obj');
      '4':Exec(PathK,'');
      '5':Exec(PathTT,'');
      '6':charger_mnemos;
      '7':begin nombre:=0; release(pdebut); end;
    end;
  until choix='0';
end.

(************  FIN DU LISTING DU PROGRAMME EXTERNPC ************)
```

"RPL SYSTEME" ET NOUVEAUX EXTERNALs

Les très nombreux points d'entrée décrits dans le tome 1 des "*Secrets* de la HP48" peuvent tous être exploités avec l'instruction SYSEVAL.

Les adresses présentées dans ce chapitre n'ont, pour la plupart, pas cette propriété. Les programmes →EXT et EXTERNPC, vus dans les pages précédentes, permettent de les utiliser.

Il est donc temps de prendre connaissance de ces nouveaux Externals, qui constituent les éléments de base de la programmation en "Rpl Système".

Comme dans le tome 1, on utilise les notations suivantes, dans les tableaux présentant ces nouveaux Externals:

Si une adresse #*****h est précédée du signe ▪, elle est garantie par HP (elle ne changera pas sur les futures versions).

Si elle est précédée de ≈, elle a changé depuis la HP48S/SX. En l'absence de signe distinctif, l'adresse n'est pas "supportée" par HP, mais elle n'a jamais encore changé depuis la version A de la HP48S/SX.

Le langage "RPL Système"

Le langage de programmation de la HP48, tel qu'il est décrit dans le manuel fourni avec la calculatrice, est (c'est le moins qu'on puisse dire) très complet. Nous l'appelerons "**RPL utilisateur**".

Chacune des instructions qui constitue ce langage est disponible en toute sécurité (sauf bien sûr l'instruction SYSEVAL). De nombreux tests internes permettent d'être certain du bon déroulement de toute séquence d'instructions: les erreurs éventuelles sont dues à l'utilisateur, et n'ont en principe pas de conséquences fâcheuses (on pourra s'en prendre à son étourderie, mais pas au matériel).

Cette sécurité, ce confort, se payent doublement. D'une part, toutes ces vérifications internes se font au détriment de la vitesse d'exécution. D'autre part, le constructeur ne nous autorise pas, dans sa grande sagesse, les opérations (souvent risquées) que notre imagination ne manquerait pas de nous proposer.

Le langage "**Rpl utilisateur**" n'est que la partie visible d'un langage plus passionnant encore, et que nous appelerons "**Rpl système**". C'est dans ce dernier langage qu'est écrite la plus grande partie de la ROM de la HP48. Le "**langage machine**" est bien sûr présent, pour les routines les plus internes, mais de façon minoritaire.

Le langage "Rpl utilisateur" est, en pratique, un sous-ensemble du langage "Rpl système". Ce dernier autorise en effet la manipulation d'un plus grand nombre de types de données (entiers-système, réels et complexes longs, etc...) et il est constitué d'instructions extrêmement nombreuses, parmi lesquelles celles qui forment le langage usuel. L'instruction **TAN** (tangente) par exemple, qui apparaît à l'utilisateur comme un "*atome*" du langage est en fait une routine écrite en "Rpl Système".

En quoi consiste ce "Rpl Système" ? La réponse a cette question est en partie donnée dans le tome 1, pages 26 et suivantes. Essayons cependant de formaliser davantage...

Le fonctionnement du "Rpl système" s'articule autour de quelques mots simples:

- ▸ Les objets.
- ▸ L'évaluation directe.
- ▸ L'évaluation indirecte.
- ▸ La notion de pile (pile "utilisateur" et pile "des retours").
- ▸ Les pointeurs.

Toute cette architecture est contrôlée par une boucle principale ("***Main Loop***"). Quand la HP48 est en mode "*exécution directe*", elle attend que vous décidiez quelle action elle doit réaliser. Vous précisez vos intentions au clavier et vous validez par **ENTER** (ou par une touche qui exécute implicitement **ENTER**).

Dès lors, la calculatrice évalue un "objet" (une séquence d'instructions), pour mener à bien l'ordre que vous lui avez donné.

Quand cet objet est "évalué", la calculatrice revient à sa position d'attente, et on recommence...

L'objet qu'évalue la calculatrice est en principe une structure Rpl. Celle-ci est composée d'objets. Ces objets sont évalués en séquence. Cette séquence est le *"flot d'instructions"*.

On sait que dans une séquence Rpl, les objets sont représentés ou bien par leur contenu (ce dernier débutant par l'adresse-préfixe correspondant au type de l'objet), ou bien par une adresse (un *pointeur*) désignant l'emplacement de cet objet (nécessairement intégré à la HP48) dans la ROM de la calculatrice. On parle, pour cet objet, d'exécution directe (dans le premier cas) ou indirecte (dans le deuxième cas).

Du point de vue de l'exécution d'une séquence Rpl, le fait qu'un objet soit évalué de manière directe ou indirecte ne change rien à l'affaire.

Parmi les objets qui figurent dans la structure Rpl évaluée, figurent souvent des sous-structures Rpl (effectivement présentes, ou bien représentées par leur adresse). Celles-ci sont évaluées à leur tour, et il n'est pas rare qu'elles fassent appel elles-mêmes à d'autres structures Rpl, etc...

Cette souplesse de fonctionnement nécessite un certain nombre d'outils et de propriétés. Il faut en effet que la calculatrice puisse passer facilement d'un objet à l'autre. Pour celà il est nécessaire de ne ne pas perdre de vue l'adresse où se trouve l'objet qui suit celui qui est en cours d'évaluation. Le fait que de nombreuses structures Rpl puissent être emboîtées les unes dans les autres impose la gestion d'une pile des retours de sous-programmes...

◆ Le pointeur sur "l'objet suivant"

A l'intérieur d'une structure Rpl, pendant qu'un des objets qui la constituent est évalué, la calculatrice doit impérativement retenir l'adresse où se trouve le prochain objet de la structure.

Le registre **D0** de la HP48 est affecté à cette tâche. Il contient donc en permanence l'adresse du prochain objet à évaluer.

Pour que celà soit possible, il est absolument nécessaire que la longueur de l'objet en cours d'évaluation soit calculable *"à priori"* (avant même que ne commence cette évaluation), pour mettre à jour le contenu de **D0**.

- De nombreux objets ont une longueur fixe (réels, complexes, entiers-système, etc...). Pour ceux-là il n'y a donc pas de problème.

- Pour d'autres (tableaux, chaînes de caractères, noms locaux ou globaux, etc...) la longueur est inscrite dans le corps même de l'objet (précisément juste après l'adresse-préfixe).

- Enfin certains objets sont dits *"composés"* (structures Rpl, listes, expressions, objets-unités,...) et donc *"scannés"* composante par composante, jusqu'à l'adresse postfixe (**#0312Bh**) qui marque leur extrémité.

◆ La pile des "retours"

La conservation dans le registre **D0** de l'adresse du prochain objet à évaluer n'est effective qu'au sein d'une même structure Rpl. Dans le cas de structures Rpl emboîtées, il est nécessaire que soit sauvegardée l'adresse de retour correspondant à la sous-structure Rpl qui est en passe d'être évaluée.

Ces structures Rpl pouvant être emboîtées ***ad libitum*** (ou presque), il est nécessaire de placer les adresses de retour sur une pile.

Cette pile est du type **LIFO** (*Last In, First Out*), car la dernière instruction de retour à être placée sur cette pile est la première à être "*récupérée*" (au sortir de la structure Rpl la plus interne).

Cette pile (qui n'a rien à voir avec la pile "utilisateur") est créée dynamiquement en mémoire vive. Plus précisément, le registre **B** de la HP48 pointe en permanence sur l'adresse où sera empilée la prochaîne adresse de retour.

Comme on le verra dans les pages qui suivent, certaines instructions du langage "Rpl système" interviennent sur cette pile de la même manière que nous intervenons sur la pile utilisateur. Il est en particulier possible d'effectuer les opérations suivantes:

- ▸ Dupliquer une adresse de retour.
- ▸ Détruire une adresse de retour.
- ▸ Intervertir deux adresses de retour.

Les registres **B** et **D0**, dont nous venons de parler, sont des ***pointeurs*** essentiels dans l'exécution des programmes par la HP48. Ils sont automatiquement gérés et modifiés par la calculatrice, sans que nous ayons à nous en occuper.

Si par contre nous écrivons une séquence d'instructions en langage machine, nous devons impérativement veiller à ce que le contenu de ces registres soit le même en entrée et en sortie de la routine, sinon c'est le *crash* assuré (il en est de même pour le registre **D1** qui pointe sur l'adresse du niveau 1 de la pile-utilisateur, et le registre **D**, qui contient la taille de mémoire disponible, exprimée en blocs de 5 quartets consécutifs).

On vient de voir que la longueur des objets qui constituent une structure Rpl donnée est toujours calculable à priori. Celà a deux conséquences très différentes:

- ▸ Il est possible de "sauter" par dessus un objet de la structure (le plus souvent après un test). On verra que le langage "***Rpl Système***" fait un très large usage de cette possibilité.
  Pour "sauter" toute une série d'objets consécutifs, il suffira de les incorporer dans une même structure Rpl, et de "sauter" cette structure...

- ▸ Il est impossible d'exécuter un "**Primitive Code Object**" (voir tome 1, page 32) de façon *directe* dans une structure Rpl (notamment parce que rien ne permet de calculer la longueur d'un tel objet *à priori*). Seule est alors possible l'exécution *indirecte*, le "**Primitive Code Object**" étant référencé par son adresse (nécessairement en mémoire morte).

◆ Et le langage machine ?

Pendant qu'une structure Rpl est évaluée, le microprocesseur de la HP48 n'est évidemment pas au repos. On sait que celui-ci ne connaît qu'un seul langage: le "langage machine". Celui-ci est pourtant bizarrement plutôt absent de la description précédente, où il est surtout question d'adresses, de pointeurs, d'objets...

La contradiction n'est qu'apparente. L'évaluation des différents objets qui constituent la structure Rpl en cours d'exécution n'est en effet rien d'autre que l'exécution des programmes (tous écrits en langage machine) correspondant aux adresses-préfixes de ces objets.

Imaginons qu'une structure Rpl soit en cours d'exécution, et que l'évaluation d'un des objets qui la composent vienne à se terminer. On sait que le registre **D0** contient l'adresse du prochain objet de la structure.

Que trouve-t-on réellement à l'adresse contenue dans **D0** ?

- Ou bien l'objet qui doit maintenant être évalué est réellement présent dans la structure Rpl: à l'adresse contenue dans **D0** on trouve les 5 quartets constituant l'adresse-préfixe de l'objet, précédant le corps même de celui-ci.
- Ou bien l'objet à évaluer est représenté par une adresse **adr** en mémoire morte, et ce sont les 5 quartets de cette adresse que l'on peut lire à l'adresse contenue dans **D0**.

  Dans ce deuxième cas, à partir de l'adresse **adr**, et sur 5 quartets, on trouve l'adresse-préfixe de l'objet à évaluer, suivie du corps de celui-ci, du moins dans la mesure où cet objet n'est pas du type "*Primitive Code*". Dans le cas d'un "objet" de ce type, on trouve (le plus souvent) en **adr** 5 quartets formant l'adresse **adr + 5** qui est chargée dans le registre **PC** du microprocesseur.

Dans les deux cas évoqués ci-dessus, la routine correspondant à l'adresse-préfixe de l'objet à évaluer est exécutée. C'est elle en particulier qui permet de modifier le contenu du registre **D0**, de manière à pointer sur le prochain objet:

- Dans le cas d'une évaluation indirecte (l'objet à évaluer est représenté par son adresse en ROM) le contenu de **D0** est augmenté de 5 (la longueur de cette adresse). Rappelons que c'est toujours le cas quand il s'agit d'évaluer un "*Primitive Code Object*".
- Dans le cas d'une évaluation directe, le contenu de **D0** est augmenté de 5 (pour pointer directement après l'adresse-préfixe) puis d'une valeur égale à la longueur (hors-préfixe) de l'objet (cette dernière incrémentation étant réalisée par la routine correspondant à l'adresse-préfixe).

Ce sont enfin les routines correspondant aux différentes adresses-préfixes qui, par comparaison des registres **D0** et **PC** du microprocesseur, permettent de distinguer entre l'évaluation directe d'un objet et son évaluation indirecte (la différence essentielle tenant dans la façon dont le contenu de **D0** est alors modifié).

Nouveaux Externals

Les nombreuses adresses qui nous restent à examiner n'ont pu trouver place dans le tome 1 dans la mesure où, pour la plupart d'entre elles, il n'est pas possible de les utiliser avec l'instruction SYSEVAL. En effet ces points d'entrée ont en commun d'agir sur le "*flot des instructions*", et la présence même de l'instruction SYSEVAL peut conduire à des ambiguités ou à des erreurs.

Il n'est réellement possible d'utiliser efficacement ces adresses que si l'on écrit directement en "*Rpl Système*". Pour celà le développement d'outils comme les programmes →**EXT** et **EXTERNPC** s'est avéré indispensable.

Nous voici donc en mesure de découvrir ces nouvelles adresses. Le programme **EXTERNPC** (ou du moins sa syntaxe) nous sera ici particulièrement utile. En effet les principales adresses de ce chapitre seront illustrées d'exemples, sous la forme de textes écrits pour le programme **EXTERNPC**. Un ou plusieurs résultats d'exécution de ces programmes seront indiqués, facilitant la compréhension et l'utilisation future de ces points d'entrée de la ROM de la HP48.

Ces adresses peuvent être groupées en quelques familles distinctes. Les frontières entre les différents cas ci-dessous ne sont pas hermétiques (certaines adresses combinent en effet plusieurs des caractéristiques évoquées). D'autre part cette liste ne prétend pas être exhaustive, certains cas étant assez inclassables...

Nous rencontrerons donc:

- Des instructions de test du nombre et du type des objets présents sur la pile. Elles sont particulièrement utiles pour écrire des programmes en "Rpl système" en minimisant le risque de "plantage"...

- Des adresses modifiant d'autorité le "*flot d'instructions*" (sauter un objet, quitter prématurément la structure Rpl en cours, placer l'objet suivant sur la pile sans évaluation, etc...). On peut placer dans cette catégorie les adresses agissant directement sur la "*pile des retours*".

- Des boucles "*start*" et des instructions apparentées (notamment en ce qui concerne la gestion de l'indice de boucle, ou des valeurs extrêmes de celui-ci).

- Des instructions répétitives (dans le genre "*tant que... répéter...*").

- Des tests et les conséquences de ces tests (dans le genre *if....then...else...*). Ces instructions sont particulièrement nombreuses, notamment par les variantes qu'elles autorisent.

- Des instructions de contrôle des erreurs pouvant éventuellement se produire à l'exécution, dans le genre de "*iferr...then...end*".

Vérifications d'arguments

Les adresses que nous allons voir ici sont très importantes pour qui veut programmer en "*Rpl Système*". Autant les instructions standards sont protégées contre une utilisation maladroite ou non appropriée, autant leurs formes internes sont le plus souvent dénuées de telles sécurités.

Pour prendre un exemple simple, l'instruction **drop** (adresse **#3244h**, forme interne de **DROP**) exécutée alors que la pile est vide, conduit à un "*Memory Clear*".

Il est nécessaire de se prémunir contre de tels risques. On peut se demander alors où est l'intérêt d'utiliser les formes internes des instructions standards si on les accompagne de tests sur la présence et la qualité des arguments présents sur la pile. Autant utiliser les instructions intégrées et revenir au "*Rpl utilisateur*". A cette question (légitime) il y a deux réponses:

- On envisage la création d'un programme. En général celui-ci nécessite un ou plusieurs arguments d'un ou de plusieurs types bien particuliers. Il suffit de contrôler la présence et la qualité de ces arguments au début du programme, une fois pour toutes. Ensuite le programme peut se dérouler sans qu'on ait sans cesse à effectuer tous ces contrôles.
- Enfin rappelons que l'utilisation d'adresses en ROM ne se limite pas, loin de là, à l'emploi des formes internes des instructions intégrées. Dans de tels cas, aucun test n'est effectué par la calculatrice, et il faut bien prendre soi-même toutes les précautions utiles.

Les adresses que nous allons maintenant découvrir offrent plusieurs visages différents. Voici les vérifications qu'elles permettent de réaliser:

- S'assurer seulement de la présence d'un certain nombre d'objets sur la pile. Dans le cas où un ou plusieurs objets seraient manquants, la conséquence est en principe une erreur "*Two Few Arguments*".
- Tester (en général après la vérification précédente) la nature des objets sur la pile. Orienter éventuellement le déroulement du programme suivant le résultat obtenu. Si aucune combinaison satisfaisante d'objets n'est détectée, le signaler par une erreur "*Bad Argument Type*".

Quand une erreur survient durant l'exécution d'un programme écrit en "*Rpl utilisateur*", un message est affiché qui indique la nature de l'erreur et (c'est très important ici) l'instruction intégrée qui a provoqué cette erreur.

On en vient à se demander comment la HP48 peut mémoriser cette instruction fautive (dans la mesure où l'incident se produit quelque part dans le corps, parfois fort volumineux, de celle-ci).

On se demande aussi comment les arguments initiaux reviennent sur la pile (si **LASTARG** est active), alors qu'au moment où se produit l'erreur, la calculatrice les a peut-être transformés et/ou a placé d'autres données sur la pile (comme des entiers-système, disparaissant comme par enchantement)...

◆ Le marquage de la pile

Nous allons voir que les Externals qui testent le nombre et la qualité des arguments "*marquent*" la pile dans l'état où ils la trouvent, permettant ainsi un retour à la situation initiale en cas d'erreur.

Ce "*marquage*" consiste en les opérations suivantes:

- Copier les arguments du programme dans la mémoire réservée à la fonction **LASTARG** (dans la mesure où cette fonction n'est pas désactivée).
- Mémoriser le niveau de la pile situé immédiatement après le ou les arguments. Cette partie de la pile ne doit en principe pas être affectée par le déroulement du programme (celà n'est pas une obligation, mais c'est un principe élémentaire de "saine" programmation).
- On peut utiliser et transformer les arguments du programme et ajouter des objets sur la pile, pour tous les calculs nécessaires à l'exécution de celui-ci.
- Si une erreur survient, la pile est "*nettoyée*" jusqu'au niveau "*marqué*" au départ. Si la fonction **LASTARG** est active, les arguments initiaux sont restaurés sur la pile.

Bien évidemment, si un sous-programme du programme en cours d'exécution contient une instruction de marquage de la pile, cette dernière se substitue à la précédente.

◆ Mémoriser la dernière instruction:

Quant à la mémorisation de la dernière instruction intégrée, elle n'est le fait que de la moitié des adresses présentées ici.

On ne peut mémoriser le nom de la dernière instruction que s'il s'agit d'une entrée visible d'une librairie donnée (le plus souvent une librairie intégrée).

Toutes les instructions intégrées de la HP48 débutent ainsi par une adresse mémorisant le nom de la commande à exécuter.

Cet "*External*" doit figurer en tête du programme. Il examine les 6 quartets qui précèdent le corps de celui-ci dans la librairie (voir tome 1, page 113). Ces 6 quartets contiennent le numéro de librairie et le numéro d'objet dans la librairie. Il suffit alors à cet "*External*" de lire le nom de l'objet dans la **Hash-Table** (voir tome 1, page 114). Si ce nom est absent, on se contente du nom **XLIB**.

Ce nom est mémorisé en RAM, et il est un des éléments du message d'erreur qui peut alors survenir à tout moment.

Dans le cas d'un programme ne figurant pas comme entrée d'une librairie, la calculatrice ne retient pas le nom. On aura donc recours à des Externals marquant certes la pile, mais n'affichant pas le nom de l'instruction fautive en cas d'erreur.

Venons-en maintenant à quelques Externals qui se contentent de vérifier que la pile contient un nombre donné d'objets (sans tester le type de ces objets, qui restent à leur place). Toutes ces adresses "*marquent*" la pile, au sens des explications précédentes.

Mnémonique	Adresse	Type	Commentaires
need0 need1 need2 need3 need4 need5	#18A15h #18AB2h #18A8Dh #18A68h #18B9Fh #18B7Ah	LM LM LM LM LM LM	Ces adresses "marquent" la pile, puis testent la présence de 0 à 5 objets Si la pile ne contient pas le nombre suffisant d'objets, alors il y a une erreur "Two Few Arguments". Sinon le programme continue en séquence.
LibNeed0 LibNeed1 LibNeed2 LibNeed3 LibNeed4 LibNeed5	#18A1Eh #18AA5h #18A80h #18A5Bh #18B92h #18B6Dh	LM LM LM LM LM LM	Comme ci-dessus, mais ces adresses retiennent le nom de la commande. Ces adresses doivent être réservées aux instructions d'une librairie, et apparaître en première position dans cette instruction. Le nom de la commande figure alors dans le message d'erreur éventuel.
need(r)	#18C4Ah	Rpl	Marque la pile, vérifie que l'objet au niveau 1 est un réel, et que le nombre d'objets au-dessus de ce réel est au moins égal à r. Transforme alors r en entier-système. A utiliser avec des programmes qui obéissent à une syntaxe analogue à celle de PICK, ROLL, ROLLD, etc...
LibNeed(r)	#18C34h	LM	Comme ci-dessus mais retient le nom de la commande. Est à need(r) ce que LibNeed1 est à need1.

Si vous utilisez une des instructions du type **LibNeed** avec un programme qui n'est pas un objet d'une librairie, et s'il se produit une erreur dans ce programme, le message correspondant commencera par une ligne du type, par exemple, **XLIB 2369 84 Error:**, ce nom **XLIB** étant ici sans signification.

N.B: la pile restaurée lors d'une erreur dépend du moment où est intervenue la dernière instruction de marquage de la pile. Considérons les deux exemples suivants:

begin	need0	0	INV	end
begin	need0	0	inv(r)	end

Le premier programme utilise l'instruction intégrée **INV**, alors que l'autre en utilise la forme interne pour un argument réel (adresse **#2AAAFh**). Les 2 provoquent une erreur "*Infinite Result*". Le premier laisse le 0 sur la pile (car celle-ci est marquée par l'instruction **INV**). Le second restaure la pile dans l'état initial.

◆ Les identificateurs de types

On sait que l'instruction **TYPE** permet de distinguer les différents types d'objets en leur assignant une valeur réelle comprise entre 0 et 27 (voir tome 1, page 27).

Les adresses que nous allons maintenant considérer utilisent des codes (sous la forme d'entiers-système) pour identifier un ou plusieurs types d'objets simultanément. De tels entiers-système seront appelés ici "***identificateurs de types***".

Dans un premier temps voici les entiers-système permettant d'identifier un type particulier. La dernière colonne du tableau contient une adresse où on peut trouver chacun des entiers-système utiles.

Nom du type	Identificateur	adresse
Objet quelconque	<0h>	▪#03FEFh
Nombre réel	<1h>	▪#03FF9h
Nombre complexe	<2h>	▪#04003h
Chaîne de caractères	<3h>	▪#0400Dh
Tableau réel ou cplxe	<4h>	▪#04017h
Liste	<5h>	▪#04021h
Nom global	<6h>	▪#0402Bh
Nom local	<7h>	▪#04035h
Structure Rpl	<8h>	▪#0403Fh
Expressions	<9h>	▪#04049h
Objet "algébrique"	<Ah>	▪#04053h
Entier binaire	<Bh>	▪#0405Dh
Objet-graphique	<Ch>	▪#04067h
Objet-taggué	<Dh>	▪#04071h
Objet-unité	<Eh>	▪#0407Bh
Nom XLIB	<Fh>	▪#04085h
Entier-système	<1Fh>	▪#04125h
Répertoire	<2Fh>	▪#64B30h
Réel long	<3Fh>	▪#64BD0h
Complexe long	<4Fh>	▪#64C2Ah
Linked Array	<5Fh>	#1CCD8h
Character	<6Fh>	▪#64CC0h
Code Object	<7Fh>	#1CD16h
Librairie	<8Fh>	#64D2Eh
Objet-Backup	<9Fh>	#64D60h
Library data	<AFh>	#1CD69h

Ce tableau appelle quelques commentaires:

- Les objets "*algébriques*" sont les expressions et les noms (globaux ou locaux).
- Aux tableaux correspond un seul identificateur: l'entier <4h>. On ne peut donc distinguer ici les tableaux réels des tableaux complexes (ni même des autres sortes de tableaux, comme on sait maintenant qu'il est possible d'en créer).

Pour distinguer entre des tableaux de formats différents, on pourra utiliser certains SYSEVALs du tome 1 (pages 180 et 197 en particulier).

- L'entier-système **<0h>** permet d'"identifier" n'importe quel objet. Il sera utile dans deux cas différents. D'une part comme dernier test de type (pour éviter une erreur "*Bad Argument Type*" si tous les tests précédents ont échoués), et d'autre part dans le cas d'un test multiple (voir ci-après) si l'un des arguments peut être quelconque.
- Enfin, comme on va le voir, il est très facile d'assurer plusieurs vérifications successives sur un ou plusieurs objets simultanément, ces tests n'étant pas *destructifs* (contrairement à l'instruction intégrée **TYPE**, qui "*consomme*" l'objet auquel elle s'applique).

Il se peut qu'un même objet (ou une même séquence d'objets) satisfasse à plusieurs tests de type différents. C'est le cas par exemple pour une expression algébrique, qui peut être reconnue par les identificateurs **<9h>** et **<Ah>**. Le cas le plus flagrant est l'identificateur **<0h>** qui se satisfait d'un objet quelconque (ou même de l'absence de tout objet).

Dans un tel cas, seul le premier test à être satisfait compte. C'est important quand on sait que ces tests successifs orientent (à la manière de l'instruction **CASE...THEN...END**) le déroulement du programme.

◆ Identifications multiples

Il est possible de contrôler simultanément le type de plusieurs objets (au maximum 5), à l'aide d'un seul identificateur. Ces objets sont naturellement ceux qui occupent les premiers niveaux de la pile.

Le principe est très simple et quelques exemples permettront de le comprendre. On sait que l'identificateur de type d'un réel est **<1h>**, que celui d'une liste est **<5h>**, et que celui d'une chaîne est **<3h>**.

- L'entier-système **<51h>** permet d'identifier une liste au niveau 2 et un réel au niveau 1.
- **<15h>** permet d'identifier un réel au niveau 2 et une liste au niveau 1.
- **<55h>** fait de même pour deux listes aux niveaux 2 et 1.
- **<513h>** identifie la pile suivante 3:liste 2:réel 1:chaîne
- **<330h>** identifie la pile 3:chaîne 2:chaîne 1:(objet quelconque)
- **<5000h>** identifie une chaîne au niveau 4 et trois objets quelconques aux niveaux 1,2 et 3.
- **<51331h>** identifie la pile: 5:Chaîne 4:réel 3:liste 2:liste 1:réel

Il suffit donc de lire l'entier-système de droite à gauche pour y voir un ou plusieurs identificateurs individuels, s'appliquant aux objets des niveaux 1,2,... On comprend qu'il soit dès lors possible d'identifier au plus cinq types simultanément.

Un ambiguité peut surgir des identificateurs de à deux chiffres (de **<1Fh>** pour un entier-système à **<AFh>** pour un "*Library Data*").

L'entier-système **<1Fh>** identifie un entier-système (au niveau 1) mais pas une pile du type **2:réel 1:Nom XLIB**. C'est au programmeur d'être prudent.

VÉRIFICATIONS D'ARGUMENTS

Nous considérons maintenant deux adresses qui ne marquent pas la pile, et dont le rôle est de tester le type d'un ou de plusieurs objets, jusqu'à ce qu'une configuration de types soit reconnue. Le programme est alors orienté en conséquence. Si aucune configuration n'est reconnue, il y a une erreur du type "*Bad Argument Type*".

En principe une telle vérification doit figurer au début du programme principal ou de l'un de ses sous-programmes. Dans le premier cas, une solution très recommandée est de faire précéder cette vérification de type par un test sur le nombre d'arguments (mnémoniques **need0**,...,**need5**, ou **need(r)**, vus précédemment).

Mnémonique	Adresse	Type	Commmentaires
ChkArg	▪#18FB2h	LM	Entame un ou plusieurs tests successifs sur le type d'un ou de plusieurs objets (jusqu'à 5) des niveaux inférieurs de la pile. Si tous ces tests sont négatifs, recommence en "détaggant" les objets qui seraient éventuellement taggués. Oriente le programme dans le cas où une configuration est reconnue. Dans le cas contraire, erreur "Bad Argument Type".
ChkArg!	▪#18F9Dh	LM	Comme ci-dessus. Mais en cas d'échec des différents tests, ne recommence pas une nouvelle passe "détagguée".

Intéressons nous plus particulièrement à "**ChkArg**", la deuxième instruction étant très voisine. Le schéma fonctionnel est le suivant:

ChkArg <Id1> Obj1 <Id2> Obj2 ..... <Idn> Objn end

<Id1>, **<Id2>**, ..., **<Idn>** sont ici des identificateurs de type(s) (voir les pages précédentes) pouvant tester jusqu'à 5 éléments simultanément. **Obj1**, **Obj2**, ... **Objn** sont des objets dont l'évaluation est conditionnée par la reconnaissance d'une configuration de type(s).

Les configurations de type(s) représentées par ces entiers-système sont testées les unes après les autres. Si la première à être reconnue est celle qui correspond à l'identificateur **<Idk>**, alors **Objk** est évalué et le programme continue après le "**end**" ci-dessus (c'est-à-dire qu'on sort de la structure Rpl).

Si aucune configuration de type n'est reconnue, il y a une erreur "*Bad Argument Type*". Le meilleur moyen pour éviter une telle erreur finale est d'inclure <Idn> = **<0h>** comme dernier identificateur de type, l'action associée étant l'instruction vide (représentée par le mnémonique **nop**, à l'adresse **#6E8Eh**).

Bien entendu, si on désire exécuter plusieurs instructions pour une configuration de type(s) donnée, il faut les inclure dans une même structure Rpl (débutant par le mnémonique **begin** et se terminant par **end**).

Les identificateurs **<Idk>** et les objets **Objk** qui leur sont associés peuvent être présents "physiquement" ou être représentés par une adresse (en mémoire morte).

Le fait que **ChkArg** (adresse **#18FB2h**) exécute une deuxième passe en détaggant les objets taggués (si la première passe n'a rien donné) est justifié par l'idée que le **tag** d'un objet taggué est une chose secondaire par rapport à son contenu. C'est d'ailleurs ce que font la plupart des instructions intégrées de la HP48 (si, en mode degré, vous calculez le sinus de l'objet **:ABC:30**, vous obtiendrez **0.5**, comme si on avait calculé directement **SIN(30°)**. On n'utilisera **ChkArg!** (adresse **#18F9Dh**) que lorsqu'il est indispensable de reconnaître un ou plusieurs objets taggués en tant que tels.

◆ Tests de type dans les librairies

Il reste maintenant à considérer cinq Externals qui combinent les deux actions décrites précédemment:

Mnémonique	Adresse	Type	Commentaires
LibNeed&Chk1	#18ECEh	LM	L'adresse LibNeed&Chkn (où n est
LibNeed&Chk2	#18EDFh	LM	compris entre 1 et 5) vérifie la
LibNeed&Chk3	#18EF0h	LM	présence de n objets sur la pile,
LibNeed&Chk4	#18F01h	LM	puis teste les types des objets
LibNeed&Chk5	#18F12h	LM	à la manière de ChkArg.

Comme les mnémoniques choisis ici le laissent présager, ces adresses doivent (de préférence) être utilisées avec des entrées de librairies. Elles doivent, plus précisément, être la première instruction de la structure Rpl correspondant à cette entrée.

En effet ces adresses "marquent" la pile mais surtout vont chercher dans la librairie, et dans les 6 quartets précédant la structure Rpl, le numéro XLIB de l'objet en cours d'exécution. Ce numéro XLIB est converti en un nom (si l'entrée de la librairie y est une entrée "visible") et ce nom apparaît dans le message d'erreur accompagnant toute erreur survenant dans la structure Rpl.

Toutes les entrées de la librairie intégrée N°2 de la HP48 (la plus volumineuse) qui nécessitent au moins un argument débutent par un des Externals ci-dessus.

Exemples:

En utilisant les mnémoniques choisis dans le tome 1, l'instruction **SIN** (sinus) est codée, dans la ROM de la HP48, de la manière suivante:

```
begin   LibNeed&Chk1
  <1h>   sin(r)   <2h>  sin(c)   <Ah>   sin(ex)   <Eh>   sin(uo)
end
```

De même, l'instruction **COMB**, qui prend deux arguments, est codée:

```
begin   LibNeed&Chk2
  <11h>   comb(2r)   <1Ah>   comb(r,ex)
  <A1h>   comb(ex,r)   <AAh>   comb(2ex)
end
```

VÉRIFICATIONS D'ARGUMENTS (exemples)

Une instruction du type **LibNeed** ou **LibNeed&Chk** mémorise le nom de la commande de librairie à laquelle elle correspond (cette instruction étant le premier objet exécutable de cette commande).

Dans le cas où cette commande s'évalue sans erreur, il n'est pas nécessaire que son nom puisse être attaché à une erreur qui pourrait survenir par la suite (tant qu'une nouvelle instruction, **LibNeed**, ou **LibNeed&Chk** ne sera pas exécutée, le nom de la commande initiale restera mémorisé et apparaîtra dans le message d'erreur éventuel).

Considérons l'exemple suivant:

```
begin     *     need2     end
```

Stockons ce programme dans la variable 'PROG':

- La séquence: **"A" 2 7 PROG** conduit à la pile **2:"A" 1:14**
- La séquence: **CLEAR 2 7 PROG** place 14 au niveau 1 et provoque une erreur **"* Error: Two Few Arguments"**.

Le fait que l'erreur soit ici imputée à la commande * est injuste car elle a bien fait son travail. L'erreur vient du fait que l'instruction **need2** (marquage de la pile et vérification de la présence de deux objets) ne trouve qu'un seul argument sur la pile.

On comprend dès lors l'utilité de l'instruction **ClrLastName** (adresse **#1884Dh**) qui efface le nom de la dernière commande de librairie mémorisée.

L'instruction **need0;ClrLastName** (adresse **#40BC9h**) commence par "*marquer*" la pile, puis efface le nom de la dernière commande mémorisée. Cette instruction peut être considérée comme faisant de la pile, dans son état actuel, la "*propriété*" de l'utilisateur (si une erreur survient, avant que l'on ait de nouveau marqué la pile ou utilisé une commande de librairie, la pile ne sera pas rétablie dans une configuration précédente, et aucune commande de librairie ne sera tenue responsable de cette erreur).

De même l'instruction **ClrLastName;need1** (adresse **#1592Dh**) vérifie que la pile n'est pas vide, en générant une erreur "*Two Few Arguments*" dans le cas contraire (cette erreur n'étant imputée à aucune commande de librairie).

Mnémonique	Adresse	Type	Commentaires
ClrLastName	▪#1884Dh	LM	Efface le nom de la dernière commande de librairie mémorisée par "LibNeed.."
need0;ClrLastName	▪#40BC9h	Rpl	Marque la pile, puis ClrLastName
ClrLastName;need1	▪#1592Dh	Rpl	ClrLastName, puis marque la pile en vérifiant la présence d'un argument.

◆ Commandes "Listables":

On dit d'une commande (s'appliquant à un ou à plusieurs arguments, et renvoyant un résultat) qu'elle est "listable" si elle peut agir, terme à terme, sur des listes d'arguments.

Les HP48G/GX ont marqué l'apparition de cette propriété pour de nombreuses commandes intégrées. Prenons deux exemples:

- La commande **SQ** (élévation au carré de son unique argument) peut s'appliquer à une liste:
  { 2 4 'A' } SQ donne la liste **{ 4 16 'SQ(A)' }**
- La commande ***** (produit de deux arguments) peut s'appliquer à deux listes, et renvoyer alors la liste des résultats.

Pourtant, quand on observe la façon dont est codée, par exemple, l'instruction **SQ**, on ne trouve pas trace (après l'instruction initiale **LibNeed&Chk1**) de l'action éventuelle de cette commande sur une liste:

```
begin   LibNeed&Chk1   <1h>   sq(r)   <2h>   sq(c)
   <4h>   sq(ar)   <Ah>   sq(ex)   <Eh>   sq(uo)
end
```

La réponse à ce mystère apparent est dans l'instruction **LibNeed&Chk1**, qui procède de la manière suivante: si tous les types possibles d'argument ont été testés sans succès par l'instruction **LibNeed&Chk1**, alors elle vérifie si l'argument est une liste: si c'est le cas, la commande est appliquée à tous les éléments de la liste (dans la mesure où le type correspond aux différents cas de figure prévus initialement).

Il faut bien sûr qu'au départ, le type liste ne fasse pas partie des types d'argument accepté par la commande (car alors ce type d'argument est reconnu dès le premier "passage").

Considérons l'exemple suivant (on veut ajouter 1 à un réel, ou à un entier-système):

```
begin   LibNeed&Chk1   <1h>   1+(r)   <1Fh>   1+(s)   end
```

Si on applique ce programme à la liste {3,<5h>}, on obtient la liste {4,<6h>}.

L'instruction **LibNeed&Chk2** fonctionne de manière analogue, en testant "*in fine*" si les deux arguments sont des listes (on obtient alors la liste des résultats terme à terme) ou si l'un des arguments est une liste (l'autre argument étant alors appliqué à tous les éléments de cette liste).

On peut de même envisager d'appliquer terme à terme 3, 4, ou 5 listes d'arguments (avec les instructions **LibNeed&Chk3**, **LibNeed&Chk4**, ou **LibNeed&Chk5**).

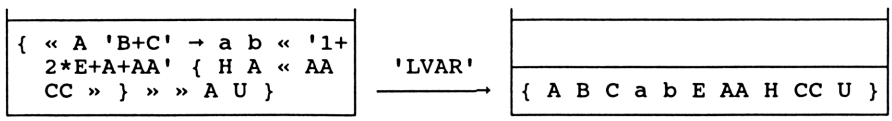
◆ Exemples

Deux exemples nous permettront de mettre en oeuvre les instructions du type "***need***" (tester la présence d'un nombre donné d'arguments) et "***ChkArg***" (orienter le flot des instructions suivant le type de ces arguments).

On veut écrire un programme donnant la liste de tous les noms globaux ou locaux figurant à l'intérieur d'un objet donné (l'objet initial peut être lui même un nom global ou local, ou une liste, ou une structure Rpl ou une expression algébrique).

```
begin
   need1    {}    swap
   ≈
   begin
      breaks(cp)     dup;=0(s)?:drq
      dup;roll(s+2)    swap     start(1,s)
         BeforeStopIndex    roll(s+1)    1Lget    eval
      next
   end
   ≈
   begin
      ChkArg
      <5h>  2Lget;eval
      <8h>  begin
               dup;StdObj?    ?:drq    2Lget;eval
            end
      <9h>  2Lget;eval
      <6h>  +IfNew(cp,_)
      <7h>  +IfNew(cp,_)
      <0h>  drop
   end
   {2['']}    Ldef(_,1i)
   1Lget    eval
end
```

Cet exemple n'est pas le plus simple qu'on puisse imaginer pour illustrer l'action de **ChkArg**, mais il a le mérite de montrer comment cette instruction permet de gérer des choix mutliples selon le ou les types d'objets reconnus. Si on baptise **LVAR** (pour '*liste des variables*') ce programme, voici un exemple de fonctionnement:



Le temps d'exécution est ici de 0.31 seconde. On remarque que le programme **LVAR** donne la liste de tous les noms présents dans l'objet initial, sans répétitions, et dans l'ordre où ils y apparaissent.

Voici tout de même quelques explications permettant de comprendre le fonctionnement du programme précédent:

- On commence par vérifier la présence d'un objet sur la pile (mnémonique **need1**), puis on place une liste vide au-dessus de cet objet.
- On place dans deux variables locales deux structures Rpl (la première du listing ci-dessus est la variable n°2, rappelable par **2Lget**; la seconde porte le numéro 1 et on rappelle son contenu par **1Lget**).
- Ces deux variables locales vont s'appeler mutuellement (récursivité croisée).
- La variable n°1 est celle qui débute par **ChkArg**. C'est elle qui est appelée dans le programme principal pour examiner l'objet initial (au-dessus duquel on a placé une liste vide)

  Ce sous-programme teste si l'objet présent au niveau 1 est un objet-composé (identificateurs **<5h>**, **<8h>** ou **<9h>**), un nom (identificateurs **<6h>** ou **<7h>**), ou sinon un objet d'un autre type (identificateur **<0h>**).

  Si l'objet reconnu n'est ni un nom, ni un objet composé, il est éliminé de la pile (mnémonique **drop**), la liste passant donc du niveau 2 au niveau 1. Si c'est un nom, celui-ci est ajouté à la liste du niveau 2, dans la mesure où il n'y figure pas déjà (mnémonique **+IfNew(cp,_)**).

  Si l'objet reconnu est composé (liste, structure Rpl, ou expression): dans le cas particulier où c'est une structure Rpl représentant un objet intégré de la HP48 (mnémonique **dup;StdObj?**), on l'élimine de la pile. Dans le cas général, on transmet cet objet composé à la structure Rpl contenue dans la variable n°2 (mnémonique **2Lget;eval**).
- La variable n°2 étale sur la pile le contenu de l'objet composé qu'on lui a transmis (mnémonique **breaks(cp)**). Dans le cas d'un objet de longueur nulle, on quitte le sous-programme.

  Dans le cas d'un objet-composé non vide, on replace la liste au niveau 1 de la pile (mnémonique **dup;roll(s+2)**). On entame ensuite une boucle consistant à placer successivement chacun des éléments de l'objet composé au niveau 1 (sous la liste, mnémoniques **BeforeStopIndex roll(s+1)**) et à lui appliquer le sous-programme n°1 (mnémoniques **1Lget eval**).
- On comprend ainsi qu'il soit possible au programme **LVAR** de "scanner" tout les sous-objets de l'objet initial (l'exemple de la page précédente l'illustre bien).

Un certain nombre de mnémoniques utilisés dans l'exemple ci-dessus seront décrits dans la suite de ce chapitre. Il s'agit en particulier de ≈ (adresse **#6E97h**) qui permet de placer sur la pile l'objet suivant de la structure Rpl, sans l'évaluer (ce qui est très important quand cet objet est lui-même une structure Rpl).

Il en est de même pour les mnémoniques associés aux boucles (**start**, **BeforeStopIndex**, et **next**), qui sont décrits dans les pages suivantes.

Enfin les mnémoniques **dup;=0(s)?:drq** et **?:drq** parlent d'eux mêmes (**drq** étant l'abréviation de **drop quit**). Le premier considère l'entier-système placé au niveau 1, teste s'il est nul (après l'avoir dupliqué). S'il est nul, cet entier-système est détruit et on sort prématurément de la structure Rpl en cours.

VÉRIFICATIONS D'ARGUMENTS (exemples)

Les utilitaires développés dans le chapitre suivant vous familiariseront avec ce type de programmation. Le listing ci-dessus a été donné pour être utilisé avec le programme **EXTERNPC**. Si vous voulez utiliser **→EXT**, voici la liste qu'il faut lui appliquer:

Pour cette liste la commande **BYTES** donne les résultats suivants:
Taille: 462 octets. CheckSum: #6DE7h
Pour la structure Rpl qui en résulte (et qu'on peut stocker sous le nom **LVAR**):
Taille: 105 octets. CheckSum: #9E0Ah

```
{ #18AB2h   #55E9h   #3223h   #6E97h
  {  #54AFh   #63CBDh   #61305h   #3223h   #73DBh
     #5182Fh   #612F3h   #613B6h   #6F8Eh   #7334h
  }
  #6E97h
  {  #18FB2h   #4021h   #632E5h
     #403Fh   { #61FA9h   #6194Bh   #632E5h }
     #4049h   #632E5h
     #402Bh   #35491h
     #4035h   #35491h
     #3FEFh   #3244h
  }
  #37DB9h   #74D0h
  #613B6h   #6F8Eh
}
```

Terminons sur un exemple beaucoup plus simple, qui nous permettra d'utiliser l'instruction de mnémonique **need(r)**. Le programme suivant, baptisé **CHKT** vérifie que la pile contient **n** objets de type **t**, aux **n** premiers étages de la pile (une fois les arguments **n** et **t** retirés de la pile).

La syntaxe est: **t n CHKT**.

Le programme **CHKT** doit provoquer une erreur "*Two Few Arguments*" si la pile ne contient pas au moins **n+2** objets (les deux arguments **t** et **n**, et **n** objets situés au-delà). Il doit y avoir une erreur "*Bad Argument Type*" si les objets des niveaux 1 et 2 ne sont pas des réels et/ou si les **n** objets des niveaux au-delà ne sont pas tous de type **t**. Le réel **t** désigne ici un type d'objet au sens de l'instruction **TYPE** (ce n'est pas un identificateur de type au sens où nous l'avons vu précédemment).

```
begin
   need2   ChkArg   <11h>
   begin
      1+(r)   need(r)   start(1,s-1)
         StopIndex   roll(s)   type!
         rot;over   =(2r)?   ?#:BAT!
      next
      drop
   end
end
```

Boucles en Externals

La programmation en "*Rpl système*" autorise les boucles du type "**Start...Next**" et "**Start...Step**". Ces structures ont un certain nombre de points communs avec les boucles **START** et **FOR** du langage usuel, et en même temps, il y a des différences importantes. Voici les particularités essentielles des boucles "*start*" en "*Rpl système*":

- Les boucles peuvent être emboîtées les unes dans les autres, sans limitation apparente autre que la mémoire disponible.
- L'indice d'une boucle est implicite (il n'est pas référencé par une variable nommée). Il est pourtant toujours accessible (au moyen de l'instruction de mnémonique "**index**", comme on le verra plus loin).
- L'indice d'une boucle est toujours un entier-système.
- Si plusieurs boucles sont emboîtées, il n'est possible de désigner au moyen des instructions "**index**" et "**UpIndex**" que les indices de la boucle "*la plus interne*" et de celle qui la contient immédiatement. Pour les boucles plus *externes* encore, il faudra (par exemple) mémoriser les indices dans des variables locales.

◆ Démarrer une boucle

Une boucle commence par une instruction du type "**start**" (il y a plusieurs variantes) et se termine par une instruction du type "**next**" ou "**step**". Voyons tout d'abord les différentes façons de démarrer une boucle en "*Rpl Système*".

Dans le tableau ci-dessous, **k** désigne l'indice de boucle, et **s** désigne l'entier-système présent au niveau 1 de la boucle quand intervient l'instruction "**start**".

Mnémonique	Adresse	Type	Commentaires
start(0,s-1) start(1,s-1) start(1,s) start(s1,s2-1)	▪#073C3h ▪#073CEh ▪#073DBh ▪#073F7h	LM LM LM LM	De k=<0h> à k=s-1 De k=<1h> à k=s-1 De k=<1h> à k=s De k=s1 à k=s2-1 , où s1 et s2 désignent des entiers-système placés aux niveaux 1 et 2.
dup;start(0,s-1)	▪#6347Fh	Rpl	Duplique d'abord l'entier-système
dup;start(1,szcp)	▪#63498h	Rpl	Un objet-composé doit se trouver au niveau 1: duplique cet objet puis entame une boucle où l'indice va de <1h> à la taille de l'objet (traduisez "szcp" = "size of cp")
dup;start(1,szar)	#37BCBh	Rpl	Comme ci-dessus, sauf que le niveau 1 doit contenir un tableau et que l'indice de boucle va varier de <1h> au nombre d'éléments du tab.

Remarques:

Comme dans le boucles **START** ou **FOR** du langage usuel, les boucles "**start**" écrites en "*Rpl Système*" sont toujours au moins exécutées une fois (et ce même si l'indice de départ est supérieur à l'indice de fin).

Les entiers-systèmes qui interviennent dans le déroulement d'une boucle sont comptés positivement de <0h> à <FFFFFh>.

♦ Terminer une boucle

Il y a essentiellement deux instructions possibles pour marquer la fin d'une boucle. Ou bien on incrémente l'indice de boucle d'une unité, ou bien on lui ajoute un entier-système quelconque.

Dans les deux cas, la boucle est définitivement terminée si la nouvelle valeur de l'indice de boucle est supérieure strictement à la valeur maximum qui était spécifiée dans l'instruction **start** initiale.

Sinon, on exécute de nouveau le corps de la boucle, avec la nouvelle valeur de l'indice de boucle.

Mnémonique	Adresse	Type	Commentaires
next step(s)	▪#07334h ▪#073A5h	LM LM	Fin de boucle et incrément de <1h> Fin de boucle avec incrément de s, où "s" est l'entier-système placé au niveau 1 de la pile
swap;next drop;next	▪#6344Dh ▪#63466h	Rpl Rpl	swap puis next drop puis next
NextInArray	≈#357FDh	LM	Utilisable dans une boucle débutant par "dup;start(1,szar)" (adresse #37BCBh, voir page précédente) pour fermer la boucle.

Remarques:

Il n'y a aucune raison que l'incrément utilisé dans une instruction "**step**" soit le même pour chacun des passages dans la boucle.

Celà permet de provoquer une sortie prématurée de la boucle, ou au contraire de provoquer un passage supplémentaire (par **<0h> step(s)**).

On peut même simuler une boucle infinie en la terminant toujours par l'instruction **<0h> step(s)**. Pour en sortir, il faudra utiliser une instruction du type **stop=0** décrite dans le tableau de la page suivante.

◆ Agir sur les indices de boucle

Nous allons voir qu'il est possible de rappeler ou de modifier la valeur actuelle de l'indice d'une boucle; il est de même possible de rappeler ou de modifier la valeur "stop" de la boucle (c'est-à-dire la valeur immédiatement supérieure à la plus grande valeur autorisée pour l'indice de boucle).

Toutes ces opérations s'entendent pour la boucle "*la plus interne*" (la dernière dans laquelle on soit entré), ou, dans certains cas, pour la boucle qui la contiendrait immédiatement.

Mnémonique	Adresse	Type	Commentaires
index	▪#07221h	LM	Rappelle la valeur du compteur de la boucle "la plus interne".
UpIndex	▪#07258h	LM	Même chose avec la boucle qui la contient immédiatement.
StopIndex	▪#07249h	LM	Rappelle la valeur immédiatement supérieure à la dernière valeur autorisée pour le compteur de la boucle "la plus interne".
StopUpIndex	#07264h	LM	Même chose pour la boucle qui la contient immédiatement.
BeforeStop	▪#5182Fh	Rpl	Indique ce qui sépare la valeur de l'indice actuel de la valeur de stop ("StopIndex-Index") pour la boucle la plus interne.
NewIndex(s)	▪#07270h	LM	L'entier-système devient la valeur de l'indice de la boucle la plus interne.
NewUpIndex(s)	#072ADh	LM	Même chose pour la boucle qui la contient immédiatement.
NewStop(s)	▪#07295h	LM	Définit une nouvelle valeur de stop pour la boucle la plus interne.
NewUpStop(s)	#072C2h	LM	Même chose pour la boucle qui la contient immédiatement.
dup;stop=0	#3F78Ch	Rpl	L'instruction "stop=0" donne la va-
stop=0;dup	#3F7EBh	Rpl	-leur <0h> à l'indice de stop de
stop=0	▪#6400Fh	Rpl	la boucle la plus interne, ce qui provoque une sortie de la boucle lors du prochain "next" ou "step"
drop;stop=0	#4334Fh	Rpl	
dup;index	▪#63411h	Rpl	Exécution d'une opération simple
swap;index	▪#63425h	Rpl	sur la pile, avant le rappel de
over;index	▪#63439h	Rpl	l'indice de la boucle.
index;-(2s)	▪#63790h	Rpl	Rappelle "index" puis exécute -(2s)

◆ Exemple de boucles en Externals

L'exemple qui suit montre comment procéder au tri d'un vecteur de **n** nombres réels par la méthode de la "*bulle*". Rappelons brièvement le principe de cette méthode (qui est certainement la plus rustique des méthodes de tri, mais qui nous permet ici d'écrire deux boucles imbriquées):

- On parcourt le vecteur: à chaque étape, on compare deux éléments consécutifs et on les échange s'ils ne sont pas dans l'ordre croissant.
- A la fin du premier parcours, le plus grand élément a été placé en fin de liste. On reprend le parcours précédent, mais sur les n-1 premiers éléments, etc...
- A chaque parcours, un nouvel élément est correctement placé. La longueur du parcours diminue d'une unité à chaque fois. Le tri est terminé quand le dernier parcours (uniquement sur les 2 premiers éléments du vecteur) a été effectué.

On doit donc prévoir deux boucles imbriquées:

- La boucle "externe" indique la longueur du parcours à effectuer.
- La boucle "interne" effectue ce parcours: si celui-ci est de longueur m, l'indice k de boucle varie de 1 à (m-1). On compare les termes de rang k et k+1 (et on les échange s'ils ne sont pas dans l'ordre croissant).

```
begin
   need1     ChkArg     <4h>
   begin
      dup;ra?#:BAT!  dup  length(ar)  dup;=1(s)?  ?:drq
      start(1,s-1)
         BeforeStop     start(1,s)
            index     DupGet(ra,s)
            over;index    1+(s)    get(ar,s)
            dup2    ≤(2r)?    pifte    drop2
            begin
               swap(2,3)   index    @put(ra,nb,s)!
               swap;index   1+(s)   @put(ra,nb,s)!
            end
         next
      next
   end
end
```

La plupart des mnémoniques ci-dessus sont bien connus. Pour ceux qui s'appliquent spécifiquement aux tableaux, on consultera le tome 1 aux pages 361 et suivantes (et à la page 352 pour le mnémonique **dup;ra?#:BAT!**).

Rappelons que la méthode de la "bulle" est lente. Il est cependant intéressant de noter que pour un vecteur de 25 éléments, le programme précédent met 1.6 secondes, alors que le même programme, écrit en "Rpl utilisateur" met 15.3 secondes.

Boucles "indéfinies"

Hormis les boucles "définies" du type **START** ou **FOR** (une séquence d'instructions répétée un nombre de fois fixé au départ) le langage usuel de la HP48 autorise deux types de structures permettant de répéter une ou plusieurs instructions successives:

- Tant qu'une condition est satisfaite (**WHILE...REPEAT...END**)
- Jusqu'à ce qu'une condition soit satisfaite (**DO...UNTIL...END**)

Le "*Rpl Système*" permet des constructions analogues. Nous parlerons ici de "*boucles indéfinies*" dans la mesure où le nombre de fois où la même séquence d'instructions est effectivement répétée n'est à priori pas fixé.

◆ La structure "do....until"

Il s'agit ici d'exécuter une ou plusieurs instructions, jusqu'à ce qu'une condition (que l'on peut qualifier de condition de sortie) soit réalisée. C'est l'équivalent du **DO...UNTIL...END** usuel. On sait que dans cette structure, la position du mot **UNTIL**, entre **DO** et **END**, est sans importance (le test de sortie est évalué quand l'instruction **END** est rencontré, et ce test de sortie fait partie intégrante des instructions à exécuter dans le corps de la boucle).

La forme "Rpl Système" de cette structure reflète l'inutilité d'une instruction intermédiaire entre celle qui ouvre la structure et celle qui la termine en évaluant le test de sortie.

Commençons par donner les deux adresses utiles:

Mnémonique	Adresse	Type	Commentaires
do	#071A2h	LM	Début de la structure "do...until"
until	#071C8h	LM	Fin de la structure "do...until"

Le principe de fonctionnement est le suivant. L'instruction "**do**" ouvre la structure: elle est suivie d'une ou de plusieurs instructions. Cette séquence doit, au moment où l'instruction "**until**" est rencontrée, avoir déposé un booléen sur la pile. Si ce booléen est égal à "**true**", alors on sort de la structure répétitive. Sinon on recommence la séquence d'instructions juste après l'instruction initiale "**do**".

Il n'est pas nécessaire que les instructions entre "**do**" et "**until**" soient incluses dans un objet unique (une structure Rpl).

Notons enfin qu'il est tout à fait autorisé d'imbriquer plusieurs structures "**do...until**". Bien sûr, chacune de ces structures doit être entièrement incluse dans la précédente, et/ou contenir entièrement la suivante: il est impossible que deux telles structures ne fassent que se chevaucher partiellement.

Exemple: le programme suivant permute circulairement les objets de la pile, jusqu'à ce qu'une touche soit pressée:

```
begin
   do     sdepth     roll(s)     KeyPressed?     until
end
```

Les deux adresses ci-dessous sont des versions plus élaborées de l'instruction "**until**" finale:

- La première permet d'exécuter une séquence d'opérations jusqu'à ce qu'un entier système (placé au niveau 1 de la pile) soit nul (cet entier-système étant dupliqué avant d'être testé, de manière à pouvoir être réutilisé s'il n'est pas nul).
- La seconde permet d'exécuter une séquence d'opérations jusqu'à ce que le booléen placé au niveau 1 soit égal à "**false**" (et non pas à "**true**" comme dans le cas initial).

Mnémonique	Adresse	Type	Commentaires
dup;=0(s)?;until	▪#62B6Fh	Rpl	Répétition jusqu'à ce que l'entier-système soit nul
not(?);until	▪#633C6h	Rpl	Répétition jusqu'à ce que le booléen soit égal à "false"

◆ La structure "while...repeat...endwhile"

Dans la structure usuelle **WHILE...REPEAT...END**, la position qu'occupe l'instruction **REPEAT** est importante. D'autre part, l'examen de la manière dont cette structure est codée en mémoire montre que la séquence d'instructions entre **REPEAT** et **END** est placée dans une structure Rpl unique (voir tome 1, page 372).

Nous allons retrouver ces particularités dans la structure correspondante en langage "Rpl Système". Il s'agit donc d'exécuter une séquence d'instructions tant qu'une condition (testée initialement) est vraie.

Voici tout d'abord les trois adresses utiles:

Mnémonique	Adresse	Type	Commentaires
while	▪#071A2h	LM	Les trois composantes obligatoires
repeat	▪#071EEh	LM	de la structure répétitive
endwhile	▪#071E5h	LM	"while...repeat...endwhile"

Symboliquement, la structure étudiée ici peut être représentée par:

while *instructions_vers_booléen* repeat *objet* endwhile

Ici "***instructions_vers_booléen***" est une séquence quelconque d'objets qui doit, entre autres, produire au niveau 1 de la pile un booléen qui est testé par "**repeat**":

- Si ce booléen vaut "**true**", l'objet entre "**repeat**" et "**endwhile**" est évalué (un seul objet, éventuellement une structure Rpl, doit donc séparer "**repeat**" de "**endwhile**"), et on repart après l'"instruction "**while**" initiale.
- Si le booléen vaut "**false**", le flot des instructions continue après "**endwhile**".

Remarques:
On constate qu'une seule et même adresse (**#71A2h**) sert à la fois d'instruction "**while**" (dans la séquence "**while...repeat...endwhile**" et "**do**" (dans la séquence "**do...until**"). Il est cependant préférable de lui continuer à lui attribuer l'un des deux mnémoniques, selon le rôle que l'on fait jouer à cette adresse.
Comme dans le cas des structures "**do..until**" il est tout à fait possible d'imbriquer plusieurs structures "**while..repeat..endwhile**". L'important est que ces différentes structures ne se chevauchent pas.

Voici 2 variantes permettant de répéter l'objet situé entre "**repeat**" et "**endwhile**":
- Tant que le booléen produit par la séquence d'instructions située entre "**while**" et "**repeat**" est égal à "**false**".
- Tant qu'un entier-système, placé au niveau 1 de la pile, est non nul (cet entier-système étant dupliqué pour pouvoir être réutilisé s'il différent de <0h>).

Mnémonique	Adresse	Type	Commentaires
`not(?);repeat`	`▪#633DFh`	Rpl	Continuer tant que le booléen est égal à "false"
`dup;<>0(s)?;repeat`	`▪#633F8h`	Rpl	Continuer tant que l'entier-syst. est non nul

Exemple: voici un programme calculant le PGCD de deux réels:

```
begin
   need2     ChkArg     <11h>
   begin
      while     dup     <>0(r)?     repeat
      begin     swap;over     mod(2r)     end
      endwhile
      drop    abs(r)
   end
end
```

BOUCLES INDÉFINIES

Exemple: trier les n premiers objets de la pile.

Le programme **'TRSTK'** présenté ci-dessous effectue le tri des **n** premiers objets situés sur la pile. La syntaxe est **n TRSTK** (l'entier **n** ne devant pas être considéré comme figurant sur la pile au moment où le programme commence).

A la fin du programme les **n** premiers objets sont triés:

- **Si n>0**: Dans l'ordre inverse du niveau qu'ils occupent (les plus petits éléments aux niveaux supérieurs, les plus grands aux niveaux inférieurs.
- **Si n<0**: dans l'ordre inverse de celui qui vient d'être décrit (bien sûr, c'est alors la valeur absolue de **n** qui désigne le nombre d'éléments à trier).

On utilise ici la méthode d'"**insertion**", dont voici le principe:

- On place correctement, l'un par rapport à l'autre, les objets des niveaux 1 et 2 (ce qui signifie un échange éventuel).
- On place alors correctement l'objet du niveau 3 par rapport aux objets des niveaux 1 et 2 (qui eux sont déjà bien ordonnés).
- On continue ainsi: à la k-ème étape, l'élément du niveau (k+1) est inséré dans la position qu'il doit occuper parmi les objets des niveaux 1 à k (que les étapes précédentes ont correctement ordonnés).
- A l'issue de la (n-1)ème étape, le tri est terminé.

N.B: les objets situés aux niveaux 1 à n doivent pouvoir être comparés par l'opérateur intégré **<**. Il peut s'agir de réels, de chaînes, etc...

```
begin
   need1     ChkArg    <1h>
   begin
      dup    >0(r)?    Ldef''    abs(r)    need(r)
      dup    >1(s)?    ?#:drq
      start(1,s-1)
         index     roll(s+1)    index     1+(s)
         while
            dup;pick(s+1)     3pick
            1Lget    pifte     <     >     <>0(r)?
         repeat    1-(s)
         endwhile
         rolld(s)
      next
      Lpop
   end
end
```

A titre indicatif, un tel programme, testé sur une pile de 100 entiers aléatoires, effectue le tri en moins de 10 secondes (alors que le même programme, écrit en "Rpl utilisateur", met environ 1 minute).

◆ La structure "do...again"

La structure "**do Obj0 Obj1 ... Objn again**" permet de répéter indéfiniment l'évaluation des objets **Obj0**, ..., **Objn**. La seule façon de sortir de cette boucle infernale est de provoquer (ou de subir) une erreur, ou d'utiliser une instruction du type "**quit**".

L'instruction "**do**" est ici celle qui nous a servi pour la séquence **do...until**: elle est toujours à l'adresse **#71A2h**.

Voici où trouver l'instruction "**again**":

Mnémonique	Adresse	Type	Commentaires
again	▪#071ABh	LM	Fin de la structure do...again

Exemple:

Le programme ci-dessous attend un réel au niveau 1. Ce réel doit désigner une touche du clavier, avec le format "**rc.p**" (par exemple 34.4 pour la touche α-P). Il attend alors que la touche correspondant à ce réel soit actionnée au clavier.

Ce n'est qu'à ce moment que l'on sort effectivement du programme, et le message "**BONJOUR**" est affiché au niveau 1. Un appui sur **ON** n'interromp pas ce programme. Le double appui sur **ON** et **C** reste bien sûr possible.

Si on place ce programme comme l'objet de configuration d'une librairie, voilà un bon moyen de protéger l'accès à votre HP48.

```
begin
   need1    ChkArg    <1h>
   begin
      rc.p=><n><p>
      begin
         do    dup2    GetTouch    rot    =(2s)?
            3rolld     =(2s)?    and?:dq/sk
            begin    drop2    DropRet    end
         again
      end
      "Bonjour"
   end
end
```

Les instructions **and?:dq/sk** et **DropRet** seront présentées dans les pages qui suivent. Pour les instructions **rc.p=><n><p>** et **GetTouch**, on se reportera au tome 1, page 263.

Structures "If...then...else"

On sait que le langage usuel de la HP48 nous permet d'effectuer des tests du type **IF..THEN..END** ou **IF..THEN..ELSE..END**. Dans les deux cas la position du mot **IF** n'a pas grande importance. Il est également possible d'utiliser les instructions **IFT** et **IFTE**.

Les mêmes facilités existent en "Rpl Système", plus beaucoup d'autres. On rencontre en effet un très grand nombre de variantes, la plupart d'entre elles utilisant un test prédéfini.

Pour décrire les opérations ci-après, on utilisera des abréviations telles que **Obj1**, **Obj2**, **ObjT**, **ObjF** pour désigner des objets dans le flot des instructions (les sous-structures Rpl, les listes, sont comptées comme un seul objet), ou des objets présents sur la pile utilisateur au moment où une instruction est rencontrée.

Voici les 4 adresses les plus importantes ici:

- Les instructions **ifte** et **ift** rappellent **IFTE** et **IFT** (à ceci près qu'elles utilisent un booléen, qui doit être placé au niveau 1 de la pile).
- Les instructions **pifte** et **pift** rappellent **IF..THEN..ELSE** et **IF..THEN**. Là encore on utilise un booléen. Ces deux instructions sont particulièrement adaptées au cas où les objets à évaluer à l'issue du test sont des objets "*exécutables*" (notamment des structures Rpl).

Mnémonique	Adresse	Type	Commentaires
ifte	▪#070C3h	LM	La pile étant: 3:ObjT 2: ObjF 1: booléen, Si le booléen vaut "true", alors ObjF est détruit et ObjT est évalué. Si le booléen vaut "false", alors ObjT est détruit et ObjF est évalué.
ift	▪#070FDh	LM	La pile étant: 2: ObjT 1: booléen, Si le booléen vaut "true", alors ObjT est évalué. Si le booléen vaut "false", alors ObjT est détruit.
pifte	▪#61AD8h	LM	Abréviation de "previous ifte" Soit la séquence: Booléen pifte Obj1 Obj2. Si le booléen vaut "true", alors Obj1 est évalué, puis on continue après Obj2. Si le booléen vaut "false", alors Obj2 est évalué, puis on continue en séquence.
pift	▪#619BCh	LM	Abréviation de "previous ift" Soit la séquence: Booléen pift Obj1. Si le booléen vaut "true", alors Obj1 est évalué, puis on continue en séquence. Si le booléen vaut "false", on "saute" Obj1 et on continue en séquence.

Voici maintenant un certain nombre d'adresses dont le point commun est d'effectuer un test produisant un booléen, puis d'appeler l'instruction **pift** ou **pifte**.

Mnémonique	Adresse	Type	Commentaires
=?;pift =?;pifte	▪#63E2Fh ▪#63E75h	Rpl Rpl	Ici =? teste l'égalité des adresses des objets des niveaux 1 et 2
dup;=0(s)?;pift =0(s)?:pifte dup;=0(s)?;pifte	▪#63E48h ▪#63E89h ▪#63EC5h	Rpl Rpl Rpl	Pour ces trois exemples, on teste d'abord si l'entier-système du niveau 1 est nul
=(2s)?;pifte <(2s)?;pifte >(2s)?;pifte	▪#62C2Dh ▪#63E9Dh ▪#63EB1h	Rpl Rpl Rpl	Instruction "pifte" précédée d'un test comparant les entiers-syst. des niveaux 1 et 2
and(2?);pifte	▪#63E61h	Rpl	D'abord un "and" entre 2 booléens
uflag(s)?;pifte sflag(s)?;pifte	▪#63ED9h ▪#63EEDh	Rpl Rpl	uflag(s)? et sflag(s)?: le flag n° s est-il armé (flag utilisateur si "uflag", système si "sflag") ?

Exemple:

Le programme ci-après attend un entier-système au niveau 1. Tant que l'entier-système **s** du niveau 1 n'est pas égal à **<1h>**, on lui applique l'opération suivante:

- On duplique l'entier **s**.
- Si l'entier **s** est impair, alors on remplace **s** par **3*s+1**. Sinon on divise **s** par 2.

```
begin     need1     ChkArg     <1Fh>
   begin
      dup;=0(s)?;pift      doerr203
      while      dup;=1(s)?      not(?);repeat
      begin
         dup;dup    <2h>    mdv(2s)    swap    =0(s)?;pifte
         drop(2)
         begin     drop     <3h>     *(2s)     1+(s)     end
      end
      endwhile
   end
end
```

Remarque:

On constate (mais on ne sait pas le démontrer) que toutes les suites ainsi définies finissent par atteindre la valeur 1 (à ce moment le programme s'arrête, laissant sur la pile tous les termes obtenus). A partir de 1, la suite devient périodique sur 1,4,2.

Certaines valeurs initiales conduisent à des suites plutôt longues, et passant par des "sommets" étonnants. Essayez par exemple avec s=<27d> (on obtient, en 0.8 secondes, les 112 termes de la suite, le maximum étant <9232d>).

Tests divers

Le paragraphe précédent portait sur les structures du type "**If..then**" ou "**If..then..else..**" en Rpl Système.

Voici maintenant quelques adresses de la même famille: elles ont pour point commun d'attendre un booléen au niveau 1 de la pile (ou d'effectuer une opération simple créant ce booléen) et d'exécuter ou non une instruction selon que ce booléen vaut "**true**" ou "**false**".

Mnémonique	Adresse	Type	Commentaires
?#:drop	▪#61B72h	LM	Si "false" alors drop.
?#:swap	▪#62D9Fh	Rpl	Si "false" alors swap.
?:swap	▪#62F1Bh	Rpl	Si "True" alors swap
?:drop/drop(2)	▪#62F43h	Rpl	Si "true", détruit objet niveau 1 sinon détruit objet niveau 2.
?:drop(2)/drop	▪#62F5Ch	Rpl	Si "true", détruit objet niveau 2 sinon détruit objet niveau 1.
over;=1(s)?:3+(s)	#3F944h	Rpl	Ces adresses nécessitent deux
over;=2(s)?:1+(s)	#3F962h	Rpl	entiers-système s2 et s1.
over;=3(s)?:2+(s)	#3F980h	Rpl	On teste s2, et si le résultat
over;=2(s)?:1-(s)	#3F99Eh	Rpl	est "true" on exécute une opé-
over;=3(s)?:1+(s)	#3F9BCh	Rpl	-ration simple sur s1.
?:1get(cp)	▪#6317Dh	Rpl	Si "true", extrait 1er élément d'un objet composé.
over;=?:drq	#517C6h	Rpl	"=?" est l'égalité des adresses. Si "true" alors drop puis quitte la structure Rpl en cours.
<>2(s)?:fq	#5B32Fh	Rpl	Si s2<>s1 alors place false au niveau 1, et quitte le Rpl.
?:drop;skip	▪#61A58h	LM	Si "true" alors drop puis "skip".
dup2;<(2r)?:swap	▪#62D81h	Rpl	Si r2<r1 alors inverse r2 et r1.
?:q/drop	▪#638E4h	Rpl	Si "true" alors quitte le Rpl, sinon exécute drop.

Instructions "skip"

Nous entamons à présent l'étude de quelques instructions très intéressantes du langage "Rpl Système". Ces instructions sont beaucoup plus souvent utilisées en conjonction avec d'autres (par exemple à l'issue d'un test) que de manière isolée. Elles n'ont pas d'équivalent dans le langage usuel de la HP48.

Nous commençons par les instructions du type "**skip**", qui effectuent un saut par dessus un ou plusieurs objets dans la structure rpl en cours.

Mnémonique	Adresse	Type	Commentaires
skip	▪#0714Dh	LM	"saute" l'objet suivant dans la structure Rpl en cours.
2skip	#0715Ch	LM	"saute" les deux objets suivants dans la structure Rpl en cours.
?:skip	▪#0712Ah	LM	Attend un booléen au niveau 1. S'il est égal à "true" alors on exécute l'instruction "skip"
drop;skip	#60DD3h	Rpl	drop, puis skip.
?#:skip	▪#619BCh	LM	Autre nom pour l'instruction "pift" Si le booléen du niveau 1 est false alors on exécute "skip"
=0(s)?:skip	▪#6333Ah	Rpl	Ces quatre instructions effectuent
=1(s)?:skip	▪#63353h	Rpl	d'abord un test. Si le booléen
=(2s)?:skip	▪#6336Ch	Rpl	obtenu est égal à "true" alors
>(2s)?:skip	▪#63399h	Rpl	on exécute l'instruction "skip"

N.B: l'instruction **skip**, devant une instruction **end** de fin de Rpl, est sans effet.

Exemples: Pour illustrer ces adresses, nous utiliserons les programmes **Prog1** à **Prog4**. Les deux derniers supposent (sans le vérifier) qu'un entier système est présent au niveau 1 et réagissent selon que cet entier-système est nul ou non.

```
(Prog1) begin   <0h>    skip   <1h>   <2h>   <3h>   end
(Prog2) begin   <0h>    2skip  <1h>   <2h>   <3h>   end
(Prog3) begin   =0(s)?   ?:skip   <1h>   <2h>   end
(Prog4) begin   =0(s)?   ?#:skip   <1h>   <2h>   end
```

- Le résultat de **Prog1** est: 3:<0h> 2:<2h> 1:<3h>
- Le résultat de **Prog2** est: 2:<0h> 1:<3h>
- Pour **Prog3**, on peut obtenir:
  1:<0h> ---[Prog3]---▸ 1:<2h>
  1:<4h> ---[Prog3]---▸ 2:<1h> 1:<2h>
- Pour **Prog4**, on peut obtenir:
  1:<0h> ---[Prog4]---▸ 2:<1h> 1:<2h>
  1:<4h> ---[Prog4]---▸ 1:<2h>

Instructions "Quit"

Le langage "Rpl Système" utilise de manière très fréquente des instructions permettant de quitter de manière anticipée la structure Rpl en cours d'exécution. Le plus souvent une telle sortie prématurée se produit à l'issue d'un test.

Tout se passe alors comme si l'on rejoignait immédiatement la prochaine instruction **end** de retour de sous-programme.

Il n'est pas possible d'inclure directement une instruction **end** supplémentaire dans le corps d'un programme sans que la HP48 ne considère ce "**end**" comme la fin de la structure Rpl, ce qui peut avoir des conséquences fort désagréables. C'est pourquoi on ne trouvera pas ici de mnémonique correspondant à une instruction "**quit**" isolée.

Par conséquent, cette instruction "**quit**" s'accompagne souvent d'autres opérations (soit avant la sortie du sous-programme, soit immédiatement après). Il est même possible d'effectuer plusieurs retours de sous-programme en même temps.

Dans la mesure ou l'instruction "**quit**" n'est jamais la seule opération réalisée, elle sera le plus souvent représentée, dans les mnémoniques, par l'abréviation **q**.

Commençons par quelques variantes usuelles dans lesquelles l'instruction "quit" est précédée par une opération simple.

Mnémonique	Adresse	Type	Commentaires
Lpop;q	#2F0C7h	Rpl	Détruit les dernières variables locales créées, puis "quit".
drop;q drop;q true;q false;q dropn(s);q	#517DAh #58A0Ch #5DE41h #5DE55h #603E2h	Rpl Rpl Rpl Rpl Rpl	L'instruction "quit" est ici précédée d'une instruction simple (drop,true,false...)

Exemples:

```
(Prog1)begin   <1h>
          begin    <2h>    <3h>    drop;q    <4h>    end
          <5h>    <6h>
       end
```

```
(Prog2)begin    <1h>
          begin     <2h>    <3h>
             begin    <4h>     <5h>     drop;q     <6h>    end
             <7h>     <8h>     drop;q     <9h>
          end     <Ah>     <Bh>
       end
```

Prog1 laisse le résultat suivant sur la pile:

4:<1h> 3:<2h> 2:<5h> 1:<6h>

Prog2 donne, quant à lui, le résultat suivant:

7:<1h> 6:<2h> 5:<3h> 4:<4h> 3:<7h> 2:<Ah> 1:<Bh>

Les adresses du tableau ci-dessous utilisent un booléen placé au niveau 1 de la pile. Souvent ce booléen est créé par l'instruction elle-même (à l'issue d'un test simple). De la valeur de ce booléen dépend la sortie ou non de la structure Rpl en cours.

Éventuellement, la sortie de la structure Rpl peut s'accompagner d'opérations auxiliaires (effectuer **drop** ou **drop2**, et/ou placer sur la pile un booléen qui pourra être ensuite être utilisé comme "*témoin*" de cette sortie prématurée).

Les mnémoniques ci-dessous utilisent un schéma que l'on retrouvera dans toutes les adresses où la valeur d'un booléen est utilisée pour réagir de telle ou telle manière.

- Un mnémonique du type **?:action** signifie que si le booléen du niveau 1 est "**True**", alors "**action**" est réalisée. Sinon, on continue en séquence.
- Un mnémonique du type **?#:action** signifie qu'au contraire "**action**" n'est exécutée que si le booléen du niveau 1 est égal à "**false**".

Dans les deux cas "**action**" peut consister en une succession d'opérations simples. Par exemple, en reprenant des exemples ci-dessous: "**drq**" signifie "***drop, puis quit***". De même "**?:drop;bip;quit**" signifie que si le booléen du niveau 1 vaut "**true**" alors on exécute "**drop**" puis la HP48 émet un "**bip**", avant que l'on ne quitte la structure Rpl.

N.B: Dans le mnémonique **over;=?;drq** ci-dessous, "**=?**" teste l'égalité des adresses (cf tome 1, page 153).

Mnémonique	Adresse	Type	Commentaires
?#:q	#61A2Ch	LM	"Quit" si le booléen est faux
?:q	#61A3Bh	LM	"Quit" si le booléen est vrai
=0(s)?:q	#61A18h	LM	"Quit" si l'entier-système est nul
over;=?:drq	#517C6h	Rpl	drq est l'abréviation de drop;quit.
?:drq	#6194Bh	LM	dr2q est celle de drop2;quit
?#:drq	#61960h	LM	Dans toutes ces adresses, un test
?:dr2q	#61970h	LM	est d'abord effectué. Selon que
?#:dr2q	#61984h	LM	le résultat est "True" ou "False"
dup;=0(s)?:drq	#63CBDh	Rpl	on quitte le Rpl en cours.
?#:fq	#5FB49h	Rpl	Ici "fq" et "tq" sont les abrévia-
<>2(s)?:fq	#5B32Fh	Rpl	-tions de "false;quit" et de
?:tq	#634E3h	Rpl	"true;quit".
?:fq	#6359Ch	Rpl	Le test préalable permet de décider
?#:tq	#638CBh	Rpl	si on sort du Rpl en cours.
?:drfq	#6356Ah	Rpl	Ici "drfq" et "drtq" sont les abré-
?:dr2fq	#63583h	Rpl	-viations de "drop;false;quit" et
?:drtq	#638B2h	Rpl	de "drop;true;quit".
?#:drfq	#63AECh	Rpl	"dr2fq"="drop2;false;quit"
over;=(2s)?:drfq	#63547h	Rpl	
?:drop;bip;q	#63BD2h	Rpl	Si le booléen vaut "true" on quitte
?:bip;q	#63BEBh	Rpl	le Rpl, avec un bip...

INSTRUCTIONS "QUIT"

Voici maintenant un certain nombre d'adresses qui utilisent toutes la séquence baptisée "**do;quit**" (et notée abréviativement **dq**). Celle-ci sera très souvent utilisée dans des adresses que nous verrons plus loin (en tant que l'une des deux alternatives à la valeur d'un booléen, l'autre étant l'instruction "**skip**").

Le tableau ci-dessous donne déjà un aperçu des possiblités qu'offre cette adresse, utilisée souvent en conjonction avec d'autres instructions simples.

Mnémonique	Adresse	Type	Commentaires
dq 5skip;dq 3skip;dq 2skip;dq skip;dq	▪#06FD1h ▪#626AEh ▪#626DCh ▪#626E5h ▪#626EEh	LM LM LM LM LM	Le couple "do;quit" (noté dq) fait que l'objet "Obj" qui suit dans le Rpl est placé sur la pile. On quitte ce Rpl, puis on évalue l'objet "Obj". Les instructions skip à 5skip permettent de "sauter" d'abord 1 à 5 objets dans le Rpl

D'une façon simplifiée (qui convient la plupart du temps) on peut considérer que la séquence "**do;quit**" consiste à évaluer l'objet "**Obj**" qui suit dans la structure Rpl, puis à quitter cette structure. Les choses ne sont pas aussi simples quand l'objet "**Obj**" est une instruction agissant sur le flot des instructions (comme par exemple une instruction du type "**skip**"). Le mieux est donc de considérer que la séquence "**do;quit**" fait de l'objet suivant ("**Obj**") le premier objet à évaluer après la sortie immédiate de la structure Rpl en cours d'exécution.

```
(Prog1) begin    <1h>
           begin    <2h>    dq    <3h>    <4h>    end
           <5h>   <6h>    <7h>
        end
```

```
(Prog2) begin    <1h>
           begin    <2h>    dq    skip    <4h>    end
           <5h>   <6h>    <7h>
        end
```

```
(Prog3) begin   <0h>
           begin    <1h>
              begin    <2h>    dq    dq    <4h>    end
              <5h>   <6h>
           end
           <7h>    <8h>
        end
```

Voici la pile que laissent les trois programmes précédents:

Prog1: 6:<1h> 5:<2h> 4:<3h> 3:<5h> 2:<6h> 1:<7h>
Prog2: 4:<1h> 3:<2h> 2:<6h> 1:<7h>
Prog3: 6:<0h> 5:<1h> 4:<2h> 3:<5h> 2:<7h> 1:<8h>

Le tableau ci-dessous contient quelques variantes supplémentaires de la séquence "**do;quit**" (qui est ici précédée d'une instruction simple sur la pile).

Mnémonique	Adresse	Type	Commentaires
drop;dq	#28585h	Rpl	drop puis do;quit
drop;dq	#51807h	Rpl	comme ci-dessus
swap;dq	▪#63312h	Rpl	swap puis do;quit
drop(2,3);dq	▪#63326h	Rpl	échange levels 2 et 3, puis do;quit
<1h>;dq	▪#63A15h	Rpl	<1h> puis do;quit

Voici maintenant des adresses plus sophistiquées, dont certaines seront illustrées par des exemples à la page suivante.

Comme on le voit il est possible de quitter plusieurs structures Rpl en une même instruction...

Mnémonique	Adresse	Type	Commentaires
q;skip;dq	#283C4h	Rpl	quit;skip puis "do;quit"
dq2	▪#62958h	LM	do;quit;quit (voir ci-après)
q;dq	▪#6296Dh	LM	quit puis do;quit
q;skip	▪#633B2h	Rpl	quit puis skip

Remarques:

L'adresse **#62958h** (mnémonique **dq2**) se comporte comme "**do;quit**", sauf que l'on quitte simultanément deux structures Rpl. L'objet qui suit immédiatement cette instruction est toujours placé sur la pile, et n'est évalué qu'après la sortie de ces deux structures Rpl.

L'adresse **#6296Dh** (mnémonique **q;dq**) provoque déjà une sortie sans condition de la structure Rpl en cours. On se retrouve alors dans la structure Rpl qui la contenait. Dans celle-ci, on exécute la séquence "**do;quit**".

L'adresse **#283C4h** (mnémonique **q;skip;dq**) provoque elle aussi une sortie sans conditions de la structure Rpl en cours. Cette sortie est suivie d'une instruction "**skip**" puis de la séquence "**do;quit**".

Voici enfin quatre adresses assez inclassables, mais qui contiennent elles aussi une instruction de sortie de la structure Rpl en cours d'exécution.

Mnémonique	Adresse	Type	Commentaires
eval;tq	#5FB08h	Rpl	eval puis true;quit
q;eval	▪#61A6Dh	LM	quitte le Rpl, puis "eval"
q;ifte	▪#61A8Eh	LM	quitte le Rpl, puis "ifte"
pifte;q	▪#6381Ch	Rpl	Exécute pifte, puis quitte le Rpl.

Quelques exemples, pour illustrer les adresses précédentes:

```
(Prog1) begin    <1h>
           begin    <2h>    q;skip    <3h>    end
           <4h>    <5h>    <6h>
        end
```

```
(Prog2) begin    <1h>
           begin    <2h>
              begin    <3h>    dq2    <4h>    end
              <5h>    <6h>
           end
           <7h>    <8h>
        end
```

```
(Prog3) begin    <1h>
           begin    <2h>
              begin    <3h>    dq2    skip    end
              <5h>    <6h>
           end
           <7h>    <8h>    <9h>
        end
```

```
(Prog4) begin    <1h>
           begin    <2h>
              begin    <3h>    q;dq    <4h>    end
              <5h>    <6h>    <7h>
           end
           <9h>    <Ah>    <Bh>
        end
```

```
(Prog5) begin    <1h>
           begin    <2h>
              begin    <3h>    q;dq    <4h>    end
              skip    <6h>    <7h>
           end
           <9h>    <Ah>    <Bh>
        end
```

```
(Prog6) begin    <1h>
           begin    <2h>
              begin    <3h>    q;skip;dq    <4h>    end
              <5h>    <6h>    <7h>    <8h>
           end
           <9h>    <Ah>
        end
```

Voici la pile que laissent les programmes **Prog1** à **Prog6**:

Prog1: 4:<1h> 3:<2h> 2:<5h> 1:<6h>
Prog2: 6:<1h> 5:<2h> 4:<3h> 3:<4h> 2:<7h> 1:<8h>
Prog3: 5:<1h> 4:<2h> 3:<3h> 2:<8h> 1:<9h>
Prog4: 7:<1h> 6:<2h> 5:<3h> 4:<5h> 3:<9h> 2:<Ah> 1:<Bh>
Prog5: 5:<1h> 4:<2h> 3:<3h> 2:<Ah> 1:<Bh>
Prog6: 6:<1h> 5:<2h> 4:<3h> 3:<6h> 2:<9h> 1:<Ah>

L'alternative "dq/sk"

Sous ce nom quelque peu barbare se cache l'une des caractéristiques les plus employées du langage "Rpl Système" (du moins dans la ROM de la HP48).

Sa forme générale est **Test?:dq/sk**, où "*Test*" est une instruction produisant un booléen.

Considérons par exemple la structure Rpl suivante:

..... **Obj1 Test?:dq/sk Obj2 Obj3** .....

- Si le booléen obtenu par "**Test**" vaut **True**, alors on exécute la séquence "**do;quit**" (rappelons que celà consiste à placer l'objet **Obj2** sur la pile, à sortir immédiatement de la structure Rpl en cours, puis à évaluer l'objet **Obj2**).
- Si le booléen obtenu par "**Test**" vaut **False**, alors on exécute l'instruction "**skip**" (autrement dit on "saute" par dessus **Obj2**, et le programme continue normalement, en séquence).

On rencontrera également des instructions pour lesquelles le mnémonique choisi est du type **Test?#:dq/sk** (conformément à des conventions décrites précédemment). La différence est que la séquence "**do;quit**" est exécutée si le booléen produit par "**Test**" vaut **False**, et que "**skip**" est exécuté si ce booléen vaut **True**.

Les deux formes les plus simples de cette instruction sont: **?:dq/sk** (adresse **#61993h**) et **?#:dq/sk** (adresse **#619ADh**). Dans ces deux cas, la séquence **Test** est absente (et le booléen nécessaire est tout simplement lu sur la pile).

D'une façon générale, les instructions utilisant l'alternative "**dq/sk**" sont idéales pour simuler l'instruction **CASE** du langage usuel de la HP48, afin de gérer des situations où plusieurs tests successifs doivent être effectués, jusqu'à ce que l'un d'eux soit satisfait.

Dans un premier temps, voici les formes les plus simples de l'alternative "**dq/sk**", celles où le booléen nécessaire est lu sur la pile, ou résulte d'une opération logique sur deux booléens présents aux niveaux 1 et 2.

Mnémonique	Adresse	Type	Commentaires
?:dq/sk	▪#61993h	LM	Si true, alors "do;quit" sinon "skip"
?#:dq/sk	▪#619ADh	LM	Si false alors "do;quit" sinon "skip"
or?:dq/sk	▪#629BCh	Rpl	Ces trois adresses forment le booléen
and?:dq/sk	▪#63CEAh	Rpl	nécessaire à l'alternative "dq/sk"
and?#:dq/sk	▪#63DDFh	Rpl	à partir des booléens des niv 1 & 2

L'ALTERNATIVE "dq/sk"

Voici maintenant d'autres adresses, où l'alternative "**dq/sk**" est précédée par un test permettant de former le booléen nécessaire.

Mnémonique	Adresse	Type	Commentaires
over;=?:dq/sk =?:dq/sk same?:dq/sk same?#:dq/sk	▪#61933h ▪#63CD6h ▪#63CFEh ▪#63DF3h	LM Rpl Rpl Rpl	"=?" signifie ici l'égalité des adresses, alors que "same?" signifie celles des contenus.
over;=(2st)?:dq/sk	▪#0BD54h	LM	Teste égalité de chaîne2 et de chaîne1 (chaîne2 conservée)
=(2s)?:dq/sk over;=(2s)?:dq/sk dup;=0(s)?;dq/sk =0(s)?;dq/sk <(2s)?:dq/sk =1(s)?:dq/sk <>(2s)?:dq/sk >2(s)?:dq/sk >(2s)?:dq/sk	▪#6186Ch ▪#6187Ch ▪#61891h ▪#61896h ▪#63D12h ▪#63D26h ▪#63D3Ah ▪#63D4Eh ▪#63D67h	LM LM LM LM Rpl Rpl Rpl Rpl Rpl	Ces différentes adresses forment le booléen nécessaire pour l'alternative "dq/sk" à partir de la comparaison de deux entiers-système
dup;|r|≤1?:dq/sk =0(nb)?:dq/sk dup;=0(r)?:dq/sk dup;=0(c)?:dq/sk =1(nb)?:dq/sk dup;=1(r)?:dq/sk dup;=1(c)?:dq/sk =2(nb)?:dq/sk dup;=2(r)?:dq/sk dup;=2(c)?:dq/sk -1(nb)?:dq/sk dup;-1(r)?:dq/sk dup;=-1(c)?:dq/sk =0(r)?:dq/sk	#1B6EFh ▪#5F0FAh ▪#5F127h #5F13Bh ▪#5F154h ▪#5F181h #5F19Fh ▪#5F1BDh ▪#5F1EAh #5F208h ▪#5F23Ah ▪#5F267h #5F285h ▪#63D7Bh	Rpl Rpl Rpl Rpl Rpl Rpl Rpl Rpl Rpl Rpl Rpl Rpl Rpl Rpl	Ici "nb" désigné un nombre, qui peut aussi bien être un réel qu'un complexe. Toutes ces adresses forment le booléen de l'alternative "dq/sk" en testant le nombre présent au niveau 1.
dup;r?:dq/sk dup;bk?#:dq/sk dup;lb?#:dq/sk dup;li?#:dq/sk dup;rpl?#:dq/sk rpl?#:dq/sk nb-ar?:dq/sk gn?:dq/sk nm?:dq/sk r?:dq/sk dup;ar?:dq/sk dup;li?:dq/sk dup;gn?#:dq/sk dup;r?#:dq/sk	#21638h #27224h #27234h ▪#27244h ▪#27254h ▪#5E984h #5EDFCh ▪#5F0AAh ▪#5F0CDh ▪#63D8Fh ▪#63DA3h ▪#63DB7h ▪#63E07h ▪#63E1Bh	Rpl LM LM LM LM Rpl LM Rpl Rpl Rpl Rpl Rpl Rpl Rpl	Ici, on teste le type de l'objet présent au niveau 1 de la pile. "nb-ar" signifie un nombre (réel ou complexe) ou un tableau. Rappel: la notation test?#:dq/sk signifie que "skip" est exécuté si "test" donne "true"

Exemple:

A titre d'illustration de deux des adresses de la page précédente, voici de quelle manière est codée, dans la ROM de la HP48, la forme interne "**merge(r)**" de l'instruction **MERGE** (à ce stade, celle-ci a déjà vérifié la présence d'un nombre réel sur la pile).

```
begin
   r+=>s       dup;=0(s)?;dq/sk    doerr00A
   dup     >2(s)?:dq/sk    doerr00A
   dq     merge(s)
end
```

Comme on le voit, "**merge(r)**" exécute les opérations suivantes:

- Transformer le réel **r** en un entier-système **s** (si **r**<0, **s**=<0h>).
- Si **s**=<0h>, il y a une erreur "*Port Not Avalaible*" (**doerr00A**).
- Si **s** est supérieur à <2h>, c'est la même erreur.
- Sinon (**s**=<1h> ou **s**=<2h>), on place sur la pile le contenu de l'instruction "**merge(s)**", et on exécute cette instruction à la sortie de la structure Rpl (seul le port 1 pouvant être "mergé" sur la GX, voilà une survivance de la SX).

◆ Variante de l'alternative "dq/sk"

A la page précédente sont apparues des adresses du type "**test:dq/sk**" où l'objet sur lequel portait "**test**" était préalablement dupliqué (afin qu'on puisse l'utiliser ultérieurement, notamment pour d'autres vérifications). Dans le cas où "**test**" donne la valeur "**true**", l'objet ainsi sauvegardé peut devenir inutile: l'instruction "**dq**" (**do;quit**) peut alors s'accompagner de la destruction par **drop** ou **drop2** d'un ou de deux objets.

On en arrive aux adresses suivantes:

Mnémonique	Adresse	Type	Commentaires
over;=(2st)?:drdq/sk	▪#2856Ch	Rpl	Dans toutes ces adresses, la
over;same?:drdq/sk	▪#517F3h	Rpl	séquence "drdq" signifie
dup;=0(s)?:drdq/sk	▪#618A8h	LM	"drop, puis do;quit".
over;=?:drdq/sk	▪#618BAh	LM	De même "dr2dq" signifie
over;=(2s)?:drdq/sk	▪#618D3h	LM	"drop2, puis do;quit".
?#:drdq/sk	▪#618E8h	LM	Les adresses qui commencent
?:drdq/sk	▪#618F7h	LM	par "?" ou "?#" prennent le
?#:dr2dq/sk	▪#61910h	LM	booléen au niveau 1.
?:dr2dq/sk	▪#6191Fh	LM	"same?": égalité des contenus.
same?:drdq/sk	▪#63CA4h	LM	"=?": égalité des adresses.

Exemple:

Le programme ci-dessous attend un entier-système au niveau 1. Il teste successivement si cet entier est égal à **<0h>**, **<3h>**, **<7h>**, ou **<8h>**. Dans le cas où une comparaison a réussi, la chaîne "**Prog**" correspondante est placée sur la pile. Sinon c'est la chaîne "**Prog?**" qui est obtenue.

Plutôt que des chaînes de caractères, ce sont des structures Rpl qui auraient pu être évaluées lors de la reconnaissance de l'entier initial. Cet exemple montre donc bien comment on peut gérer des choix multiples dans le monde du "*Rpl Système*".

```
begin     need1     ChkArg     <1Fh>
   begin
      <0h>     over;=(2s)?;drdq/sk     "Prog0"
      <3h>     over;=(2s)?;drdq/sk     "Prog3"
      <7h>     over;=(2s)?;drdq/sk     "Prog7"
      <8h>     over;=(2s)?;drdq/sk     "Prog8"
      drop     "Prog?"
   end
end
```

Voici maintenant deux adresses qui, dans un premier temps, quittent (sans conditions) la structure Rpl en cours d'exécution. On teste ensuite la valeur du booléen situé au niveau 1 de la pile, puis on exécute ensuite la séquence "**dq/sk**".

Mnémonique	Adresse	Type	Commentaires
q;?:dq/sk	▪#62986h	LM	Quitte le Rpl en cours, puis exécute la séquence ?:dq/sk
q;?#:dq/sk	▪#629A1h	LM	Quitte le Rpl en cours, puis exécute la séquence "?#:dq/sk"

Ces deux adresses sont employées dans la ROM de la HP48 pour créer certaines des entrées qui ont été décrites dans les deux pages précédentes. Par exemple, voici comment est programmée l'instruction "**<(2s)?:dq/sk**":

- D'abord l'instruction de mnémonique "**<(2s)?**"
- Puis l'instruction de mnémonique "**q;?:dq/sk**"

Terminons-en avec deux adresses un peu inclassables, se terminant comme les autres par "**dq/sk**", mais qui ne semblent pas devoir être d'une utilité quotidienne.

Mnémonique	Adresse	Type	Commentaires
ConstNumMode?:dq/sk	#5EA72h	Rpl	Est-on en mode d'évaluation numérique des constantes?
dup;ListOfNames?:dq/sk	#21DF1h	Rpl	L'objet au niveau 1 est-il une liste de noms?

L'instruction ≈ d'évaluation différée

L'instruction dont le mnémonique est ≈ (caractère de code ascii 247) permet de placer sur la pile, sans l'évaluer, l'objet qui suit dans la structure Rpl en cours d'exécution.

Cette instruction est utilisée pour retarder l'évaluation d'objets tels que des structures Rpl ou des noms. Ces objets, placés sur la pile, peuvent alors être évalués (ou détruits) selon (éventuellement) le résultat d'un test.

Par exemple les deux structures suivantes (dont on ne prétend pas qu'elles soient utiles...) sont équivalentes (**ifte** et **pifte** ont été décrits précédemment, dans le paragraphe traitant des structures **If..then..else** en "*Rpl Sytème*").

```
begin
   need1  ChkArg  <1h>
   dup  ≥0(r)?
   pifte
   begin  cos(r)  1+(r)  end
   begin  sin(r)  1-(r)  end
end
```

```
begin
   need1  ChkArg  <1h>
   dup  ≥0(r)?
   ≈  begin  cos(r)  1+(r)  end
   ≈  begin  sin(r)  1-(r)  end
   ifte
end
```

De la même manière, si on veut s'assurer que la variable de nom **'ESSAI'** existe dans le répertoire en cours (et poser alors sur la pile le contenu de cette variable), on utilisera la séquence (voir tome 1, page 305): **≈ ESSAI dup;RclInVars(nm).**

Sans l'utilisation de l'instruction représentée par ≈, le nom **ESSAI** (s'il existe dans le répertoire en cours, ou dans le chemin conduisant vers **HOME**) serait évalué avant que l'instruction **dup;RclInVars(nm)** ne puisse se poser la moindre question.

Ce genre de facilité est utilisé dans les programmes écrits en langage "*Rpl utilisateur*", par exemple dans les cas suivants:

- Si on place un nom entre deux cotes ''.
- Si on insère un sous-programme entre « et » à l'intérieur du programme principal (N.B: il ne s'agit pas ici du sous-programme suivant immédiatement la création d'une ou ou plusieurs variables locales).
- Si on place une expression algébrique dans le programme principal.

Dans ces différents cas , il ne s'agit pas vraiment de l'instruction ≈, mais de l'une de ses variantes. Nous en reparlerons dans les pages suivantes.

Voici donc l'adresse où se trouve l'instruction ≈:

Mnémonique	Adresse	Type	Commentaires
≈	▪#06E97h	LM	Place l'objet suivant sur la pile, sans l'évaluer.

L'INSTRUCTION ≈ D'ÉVALUATION DIFFÉRÉE

Comme d'autres instructions importantes du "*Rpl Système*", l'opération ≈ possède de nombreuses variantes. Voici tout d'abord un certain nombre d'adresses qui spécifient l'objet que l'instruction ≈ doit laisser sur la pile.

Mnémonique	Adresse	Type	Objet placé sur la pile
≈NullRpl	▪#055FDh	Rpl	Une structure Rpl vide "begin end"
≈bip	▪#3FDFEh	Rpl	Une instruction "bip;freeze"
≈bip;true	▪#3FE12h	Rpl	Comme ci-dessus, et ajoute "True"
≈nop	▪#632BDh	Rpl	Une instruction "no opération"
≈X	#50FC4h	Rpl	Le nom global 'X'
≈Y	#50FDCh	Rpl	Le nom global 'Y'
≈*	▪#63B5Ah	Rpl	L'opérateur intégré *
≈deriv	▪#63B6Eh	Rpl	L'opérateur intégré "dérivation"

Dans le tableau ci-dessous, l'instruction ≈ est exécutée normalement (l'objet suivant dans le Rpl est placé sur la pile) puis une autre opération est exécutée.

Mnémonique	Adresse	Type	Commentaires
≈;=?	▪#635F1h	Rpl	"=?" teste l'égalité des adresses
dup;≈;=?	▪#635ECh	Rpl	Idem, mais "dup" au préalable.
≈;EvalNoChk	#18F6Ah	Rpl	"EvalNoChk" = évaluation sans les tests habituels (voir ci après).
≈;SymbEval	▪#5A036h	Rpl	"SymbEval" = évaluation dans le mode symbolique, avec retour au mode initial ensuite.
≈;eval;q	#5FB5Dh	Rpl	Evalue l'objet posé sur la pile par ≈, puis quitte le Rpl en cours d'exécution.

L'adresse **#18F6Ah** (mnémonique **≈;EvalNoChk**) est un bon moyen d'évaluer une instruction standard de la HP48 (pendant l'exécution d'un programme écrit en "*Rpl système*") sans que le marquage de la pile qui accompagne habituellement ces instructions ne soit effectué (et sans que cette instruction ne soit mémorisée comme responsable des erreurs qui pourraient dès lors survenir).

```
begin    need1   ChkArg    <1h>
   begin    0    ≈;EvalNoChk    /    end
end
```

Le programme précédent provoque une erreur de division par zéro. Il attend un réel au niveau 1, puis tente de le diviser par 0. On constate que le réel 0 ne revient pas sur la pile avec l'erreur, et que le message d'erreur ne mentionne pas l'instruction intégrée /, ce qui prouve bien que le dernier marquage de pile est réalisé par l'instruction **need1**.

Pour cette nouvelle salve de points d'entrée, suivez les commentaires (pour la signification de "**HardSto(_,nm)**", on se reportera au tome 1, page 306).

Mnémonique	Adresse	Type	Commentaires
dup;≈ swap;≈ drop;≈ over;≈ HardSto(_,nm);≈ true;≈ <1h>;false;≈ false;≈ 1+(s);≈	▪#63925h ▪#63939h ▪#6394Dh ▪#63961h ▪#63975h ▪#63989h ▪#6399Dh ▪#639B6h ▪#639CAh	Rpl Rpl Rpl Rpl Rpl Rpl Rpl Rpl Rpl	Dans toutes ces adresses, une ou deux instructions simples sont exécutées avant l'opération ≈ elle-même.
Drop3Next q;≈;≈;ifte ≈;over;=?:dq/sk ≈;over;=?:drdq/sk	#283D8h ▪#61AE9h ▪#629D0h ▪#629E9h	Rpl LM Rpl Rpl	Détruit les 3 objets suivants. Quitte le Rpl en cours, place les 2 objets suivants sur la pile puis exécute "ifte" Ici "=?" teste l'égalité des adresses, et pas des contenus.

◆ Les instructions du type ≈PostEnd

Nous allons maintenant voir plusieurs adresses qui montrent bien l'étendue des possibilités offertes par le "*Rpl Système*", quand on souhaite bousculer un peu la monotonie du "*flot des instructions*".

Considérons la situation suivante, qui montre deux structures Rpl emboîtées:

```
begin                          Début de la structure "Rpl externe"
  .....                            Instructions "Rpl externe"
  begin                            Début "structure Rpl interne"
    ...Obj0  Obj1...                   Contenu "Rpl interne"
  end                              Fin "Structure Rpl interne"
  Obj2  Obj3 ....                  Retour dans "Rpl externe"
end                            Fin de la structure "Rpl externe"
```

L'une des deux structures Rpl est à l'intérieur de l'autre. Nous l'appelerons ici "*Structure Rpl interne*". L'objet **Obj0** est l'un de ses éléments, et il est suivi par l'objet **Obj1**. **Obj2** et **Obj3** sont les deux premiers éléments de la structure "*Rpl externe*" au retour de la structure interne. Il sont éventuellement suivis de quelques autres...

Supposons que **Obj0** soit en fait l'adresse **#6EEBh** (mnémonique **≈1PostEnd**): alors l'évaluation de **Obj0** place le contenu de **Obj2** sur la pile, puis poursuit l'exécution de la structure Rpl *interne* au niveau de **Obj1**.

Le retour dans la structure *externe* s'effectue alors au niveau de **Obj3**. Tout se passe donc comme si le contenu de **Obj2** avait été retiré de la structure Rpl externe, pour être placé sur la pile, juste avant l'évaluation de **Obj1**.

D'une façon plus précise, voici comment les choses se passent:

- Pendant l'exécution de la structure Rpl interne, la "*pile des retours*" contient, au niveau 1, l'adresse de l'objet **Obj2** (qui désigne la direction que doit prendre le "*flot des instructions*" après l'instruction "**end**" de la structure Rpl interne).
- Au moment où l'instruction **≈ 1PostEnd** est rencontrée, La HP48 place sur la pile-utilisateur l'adresse de l'objet **Obj2** (tout se passe, on le sait bien, comme si le contenu de cet objet était réellement déposé sur la pile).
- L'adresse du niveau 1 de la pile des retours de sous-programmes est alors incrémentée pour pointer sur l'objet **Obj3**. Ainsi, après le "**end**" de la structure interne, le retour à la structure externe se fera au niveau de **Obj3**.

Voici le tableau regroupant **≈ 1PostEnd** et trois autres instructions apparentées.

Mnémonique	Adresse	Type	Commentaires
≈1PostEnd	#06EEBh	LM	Voir explications ci-dessus.
≈1PostEnd;eval	#06F66h	LM	Evalue l'objet placé sur la pile par l'instruction ≈1PostEnd.
≈1PostEnd;drop	#63880h	Rpl	≈1PostEnd, puis drop.
≈2PostEnd	#639DEh	Rpl	Effectue ≈1PostEnd;≈1PostEnd.

Exemples:

Le langage "*Rpl Système*" est essentiellement inspiré par la notation polonaise inverse: les opérations ne peuvent être envisagées que lorsque leurs opérandes ont déjà été spécifiés. Autrement dit, toutes les commandes sont essentiellement ***postfixées***. Il peut cependant être utile de prévoir des opérations ***préfixées*** (la commande étant introduite avant ses arguments).

C'est essentiellement à celà que sert l'instruction **≈ 1PostEnd**.

Le programme ci-dessous vise à rappeler sur la pile le contenu d'une variable (globale ou locale), au moyen de l'instruction **ChkRcl(nm)** (voir tome 1 page 304).

```
begin    ≈1PostEnd    ChkRcl(nm)    end
```

Plaçons ce programme dans une variable nommée par exemple, **PrefRcl**, alors on pourra, dans un programme écrit en "*Rpl système*", pour rappeler le contenu de la variable nommée **ExVar**, utilisée les deux formes équivalentes:

```
begin  ...  ≈  ExVar  ChkRcl(nm)  ...  end
```

```
begin  ...  PrefRcl  ExVar  ...  end
```

On voit comment l'utilisation de l'instruction **≈ 1Postend** a permis de définir une instruction préfixée de rappel du contenu d'une variable (évitant ainsi l'utilisation de l'instruction **≈**, nécessaire dans le premier cas pour placer le nom **ExVar** sur la pile sans l'évaluer).

De la même manière, on peut définir une instruction **PrefSto** permettant de placer un objet donné dans une variable ayant un nom donné (Pour la signification de **HardSto(_,nm)**, voir le tome 1, page 306).

Il suffit par exemple de stocker sous le nom **'PrefSto'** le programme suivant:

```
begin    ≈2PostEnd    HardSto(_,nm)    end
```

Si on souhaite stocker la structure Rpl: "**begin *(2s) +(2s) end**" sous le nom **ExVar**, on peut utiliser les deux méthodes suivantes:

```
begin
   ≈    begin   *(2s)    +(2s)    end
   ≈    ExVar
   HardSto(_,nm)
end
```

```
begin
   PrefSto
   begin   *(2s)    +(2s)   end
   ExVar
end
```

Là encore, on voit comment l'utilisation de l'instruction **≈2PostEnd** a permis de créer une instruction préfixe de stockage d'un objet quelconque sous un nom donné.

Remarque:

Quand le "**end**" de la structure Rpl *interne* (celle qui contient l'instruction **≈1PostEnd**) est suivi immédiatement du "**end**" de la structure *externe*, alors l'instruction **≈1PostEnd** place sur la pile une structure Rpl vide, et l'adresse située au niveau 1 de la pile des retours de sous-programmes n'est pas incrémentée.

```
begin
   "A"
   begin
      "B"
      begin   "C"   ≈1PostEnd   "D"   end
   end
   "E"
end
```

Le résultat du programme ci-dessus est, sur la pile:

6:"A" 5:"B" 4:"C" 3: 2:"D" 1:"E"

(Le niveau 3 contient la structure Rpl vide "**begin end**")

L'INSTRUCTION ≈ D'ÉVALUATION DIFFÉRÉE

Il existe une adresse (**#61B89h**, mnémonique **≈1PostEnd?**) qui fonctionne à la manière de **≈1PostEnd**, avec la différence suivante:

- Si le "**end**" de la structure *interne* est immédiatement suivi du "**end**" de la structure *externe*, l'instruction **≈1PostEnd?** renvoie le booléen *false* au niveau 1 (là où **≈1PostEnd** aurait placé une structure Rpl vide).

- Dans le cas contraire, l'instruction **≈1PostEnd?** renvoie sur la pile (au niveau 2) le premier objet qui devait être évalué lors du retour dans la structure *externe*, et place le booléen *true* au niveau 1. L'adresse de retour de sous-programme est alors incrémentée comme l'aurait fait **≈1PostEnd**.

Mnémonique	Adresse	Type	Commentaires
≈1PostEnd?	▪#61B89h	LM	Voir explications ci-dessus.

Les deux adresses **≈1PostEnd** et **≈2PostEnd** permettent (en reprenant la terminologie déjà employée) de placer sur la pile-utilisateur le premier ou les deux premiers objet(s) qui apparaissent dans la structure Rpl *externe*, lors du retour dans celle-ci (après achèvement de la structure *interne*). En même temps, le retour dans la structure *externe* s'effectue juste après ce ou ces deux objet(s).

L'instruction **≈Postend** (adresse **#701Fh**), que nous allons maitenant décrire, place sur la pile tout ce qui reste de la structure Rpl *externe*, à partir du point où on la rejoint lors de l'achèvement de la structure *interne*, et jusqu'à son instruction "**end**". Tous les objets concernés sont placés dans une unique structure Rpl, elle-même placée sur la pile.

Un exemple est ici préférable à un long discours:

```
begin    "A"
   begin    "B"
      begin    "C"    ≈PostEnd    "D"    end
      "E"    "F"    "G"
   end
   "H"    "I"
end
```

Le résultat de ce programme, sur la pile, est:
7:"A" 6:"B" 5:"C" 4:"E" "F" "G 3:"D" 2:"H" 1:"I"
(Le niveau 4 contient la structure Rpl: **begin "E" "F" "G" end**)

A la manière de **≈1PostEnd**, l'instruction **≈PostEnd** dépose sur la pile une structure Rpl vide si le "**end**" de la structure *interne* est immédiatement suivi du "**end**" de la structure *externe*.

L'instruction **≈PostEnd** possède une variante (mnémonique **≈PostEnd!**, adresse **#7012h**) qui place également sur la pile, dans une structure Rpl unique, ce qui reste à courir de la structure Rpl *externe*, mais ne détruit pas l'adresse au niveau 1 de la pile des retours. Le retour dans le Rpl *externe* s'effectue donc normalement.

En reprenant l'exemple précédent:

```
begin    "A"
   begin    "B"
      begin    "C"    ≈PostEnd!    "D"    end
      "E"    "F"    "G"
   end
   "H"    "I"
end
```

Le résultat est maintenant, sur la pile:
10:"A" 9:"B" 8:"C" 7:"E" "F" "G"
6:"D" 5:"E" 4:"F" 3:"G" 2:"H" 1:"I"

Voici enfin deux proches cousines des opérations précédentes. Les instructions "**EvalPost2End**" (adresse **#61B45h**) et "**EvalPost3End**" (adresse **#61B55h**) évaluent le premier objet qui vient après les deux (respectivement les trois) prochains "**end**". Cet objet n'est, pour autant, pas éliminé de la structure à laquelle il appartient (et peut donc être évalué de nouveau à l'issue de ces 2 ou 3 retours de sous-programmes).

Voici un exemple d'utilisation de "**EvalPost2End**":

```
begin    "A"
   begin    "B"
      begin    "C"    EvalPost2End    "D"    end
      "E"
   end
   begin    "F"    "G"    end
   "H"
end
```

Le résultat de ce programme est, sur la pile:
10:"A" 9:"B" 8:"C" 7:"F" 6:"G" 5:"D" 4:"E" 3:"F" 2:"G" 1:"H"

Voici, regroupées, les adresses dont nous venons de parler:

Mnémonique	Adresse	Type	Commentaires
≈PostEnd	▪#0701Fh	LM	Pour ces quatre adresses, voir les explications qui précèdent.
≈PostEnd!	▪#07012h	LM	
EvalPost2End	▪#61B45h	LM	
EvalPost3End	▪#61B55h	LM	

Gérer la pile des retours

Pour ne pas se perdre dans l'évaluation de plusieurs structures Rpl imbriquées, la HP48 gère une pile des retours de sous-programmes (qu'on ne confondra pas avec la pile des retours du microprocesseur).

Considérons par exemple la situation suivante:

```
begin
   Obj0
   begin    Obj1
      begin    Obj2   Obj3   Obj4   end
      Obj5   Obj6
   end
   Obj7   Obj8
end
```

Obj0, ..., **Obj8** représentent ici des "*objets*" au sens le plus large, c'est-à-dire des objets réellement présents dans la structure (et donc précédés de leur adresse préfixe), où des pointeurs (adresses) désignant de tels objets.

Lors de l'évaluation de la structure ci-dessus, la pile des retours subit les mouvements suivants:

- Quand le *flot des instructions* entre dans la deuxième structure Rpl pour y évaluer **Obj1**, un pointeur est placé au niveau 1 de la pile, pour désigner l'adresse où se trouve **Obj7** (qui correspond au retour dans la première structure Rpl).
- Quand on entre dans la troisième structure Rpl (la plus interne), pour y évaluer **Obj2**, la HP48 place sur la pile des retours l'adresse où se trouve **Obj5**. L'adresse de **Obj7** passe alors au niveau 2.
- Quand la troisième structure Rpl est terminée (après l'évaluation de **Obj4**), la HP48 charge le registre **D0** du microprocesseur avec l'adresse du niveau 1 de la pile des retours. La séquence d'instructions peut alors continuer avec l'évaluation de **Obj5**.
- Quand la deuxième structure Rpl est terminée, le registre **D0** est chargé avec l'adresse figurant au niveau 1 de la pile des retours, c'est-à-dire maintenant avec l'adresse où on trouve **Obj7**, ce qui permet de terminer l'exécution de la structure Rpl la plus externe.

Comme on peut le prévoir, il est possible d'effectuer quelques manipulations sur cette pile. Les trois opérations directement accessibles par une simple adresse sont:

Mnémonique	Adresse	Type	Commentaires
DropRet	▪#06FB7h	LM	Détruit le niveau 1 de la pile des retours
DupRet	▪#14EA5h	LM	Duplique le niveau 1 de la pile des retours
SwapRet	▪#60EBDh	LM	Echange niveaux 1,2 de la pile des retours

Il suffit d'un exemple pour se rendre compte de la façon dont fonctionne chacune des trois expressions précédentes, et ce qu'elles vous permettent d'imaginer.

◆ L'adresse "DropRet":

Quand elle est rencontrée, cette instruction détruit la prochaine adresse de retour (celle qui permet de revenir dans la structure Rpl contenant la structure en cours d'exécution).

```
begin    "A"
   begin   "B"
      begin   "C"   DropRet   "D"   "E"   end
      "F"   "G"
   end
   "H"   "I"
end
```

Le résultat de ce programme est:
7:"A" 6:"B" 5:"C" 4:"D" 3:"E" 2:"H" 1:"I"

On voit bien comment l'instruction DropRet a permis de "*shunter*" les deux objets "**F**" et "**G**", en supprimant le retour dans la deuxième structure Rpl, au profit d'un retour direct dans la première structure Rpl (la plus externe).

◆ L'adresse "DupRet":

Quand elle est rencontrée, cette instruction duplique la prochaine adresse de retour. Ce qui reste à courir dans la structure Rpl externe (celle dans laquelle on reviendra après le "**end**" de la structure Rpl en cours) sera donc évalué une deuxième fois.

```
begin    "A"
   begin   "B"
      begin   "C"   DupRet   "D"   end
      "E"   "F"
   end
   "G"
end
```

Le résultat de ce programme est:
9:"A" 8:"B" 7:"C" 6:"D" 5:"E" 4:"F" 3:"E" 2:"F" 1:"G"

On voit bien que la séquence "**E**" "**F**" (à l'occasion du retour dans la seconde structure Rpl) a été évaluée deux fois.

◆ L'adresse "SwapRet":

Voilà certainement une instruction déconcertante. Quel intérêt peut-on avoir à permuter les deux prochaines adresses de retours? L'exemple suivant n'apporte sans doute pas une réponse crédible mais il montre le résultat de cette opération.

```
begin    "A"
   begin   "B"
      begin   "C"
         begin   "D"   SwapRet   "E"   end
         "F"   "G"
      end
      "H"   "I"
   end
   "J"
end
```

Le résultat de ce programme est, sur la pile:

10:"A" 9:"B" 8:"C" 7:"D" 6:"E"
5:"H" 4:"I" 3:"F" 2:"G" 1:"J"

On voit bien qu'au sortir de la 4ème structure Rpl (celle qui contient l'instruction "**SwapRet**") le retour s'est fait dans la structure n°2, puis la sortie de celle-ci s'est faite dans la structure n°3, avant que l'on ne revienne dans la structure n°1 (la plus externe) comme si de rien n'était.

Pour se convaincre de l'utilité de l'instruction **SwapRet**, il faut noter que certaines des opérations qui viennent d'être décrites sont programmées en utilisant **SwapRet**.

Par exemple, l'instruction **≈ 2PostEnd** est programmée comme suit:

```
begin   SwapRet   ≈1PostEnd   ≈1PostEnd   SwapRet   end
```

De même, l'instruction **≈ 1PostEnd;drop** s'écrit:

```
begin   SwapRet   ≈1PostEnd   drop   SwapRet   end
```

Enfin l'instruction **drop;DropRet**, évoquée ci-après, est codée:

```
begin   drop   SwapRet   DropRet   end
```

Voici enfin quelques adresses utilisant l'instruction **DropRet**, ou qui permettent de détruire d'un coup les deux ou les trois prochaines adresses de retour de sous-programmes.

Mnémonique	Adresse	Type	Commentaires
drop;DropRet	▪#632F9h	Rpl	"drop", puis "DropRet"
≈1PostEnd;DropRet	▪#639FCh	Rpl	"≈1PostEnd" puis "DropRet"
Drop2Ret	▪#6114Eh	LM	Détruit les 2 prochaines adresses de retour de sous-programme.
Drop3Ret	▪#61160h	LM	Détruit les 3 prochaines adresses de retour de sous-programme.

Terminons par un même exemple, utilisant successivement les instructions **Drop2Ret**, et **Drop3Ret**:

```
begin   "A"
  begin   "B"
     begin   "C"
        begin   "D"
           begin    "E"
              Drop2Ret   "F"
           end
           "G"
        end
        "H"
     end
     "I"
  end
  "J"
end
```

```
Le résultat de ce programme
  est, sur la pile:

  8:   "A"
  7:   "B"
  6:   "C"
  5:   "D"
  4:   "E"
  3:   "F"
  2:   "I"
  1:   "J"
```

```
begin   "A"
  begin   "B"
     begin   "C"
        begin   "D"
           begin    "E"
              Drop3Ret   "F"
           end
           "G"
        end
        "H"
     end
     "I"
  end
  "J"
end
```

```
Le résultat de ce programme
  est, sur la pile:

  7:   "A"
  6:   "B"
  5:   "C"
  4:   "D"
  3:   "E"
  2:   "F"
  1:   "J"
```

Remarque:
Même si celà ne semble pas très "catholique", il est possible d'utiliser individuellement les instructions **do** (adresse **#71A2h**), **repeat** (adresse **#71EEh**) , **until** (adresse **#71C8h**) **endwhile** (adresse **#71E5h**) et **again** (adresse **#71ABh**).
En effet chacune d'elle effectue une opération très simple sur la pile des retours de sous-programmes.

◆ L'instruction "do" seule

Elle copie le contenu du registre **D0** (c'est-à-dire l'adresse du prochain objet dans la structure Rpl en cours) au niveau 1 de la pile des retours.
La conséquence est que ce qui suit l'instruction **do**, dans la structure Rpl en cours, sera évalué deux fois (à moins qu'une erreur se produise, ou qu'une instruction du type "**quit**" en décide autrement).
Exemple: Le résultat du programme ci-dessous est:
5:"A" 4:"B" 3:"C" 2:"B" 1:"C"

```
begin    "A"   do    "B"   "C"   end
```

◆ L'instruction "repeat" seule:

Elle attend un booléen au niveau 1 de la pile. Si ce booléen vaut "**true**", rien ne se passe. S'il vaut "**false**", on saute les deux objets suivants de la structure Rpl en cours et on détruit l'adresse au niveau 1 de la pile des retours.
L'instruction **repeat** pourrait donc s'écrire: **?#:DropRet;2skip.**
Exemple: Le programme suivant (nommé par exemple '**ESSAI**') attend un réel au niveau 1, et teste s'il est nul.

```
begin   "A"
   begin   "B"
      begin
         "C"   4roll   =0(r)?   repeat
         "D"   "E"   "F"
      end
      "G"
   end
   "H"
end
```

Voici deux exemples d'utilisation du programme '**ESSAI**' ci-dessus:
▸ **0 ESSAI** donne la pile:
8:"A" 7:"B" 6:"C" 5:"D" 4:"E" 3:"F" 2:"G" 1:"H".
▸ **1 ESSAI** donne la pile:
5:"A" 4:"B" 3:"C" 2:"F" 1:"H"

◆ L'instruction "until" seule

Elle attend un booléen au niveau 1. Si ce booléen est "**true**", alors on détruit l'adresse du niveau 1 de la pile des retours. Si ce booléen est "**false**" on copie dans le registre **D0** l'adresse du prochain retour de sous-programme (cette adresse reste cependant sur la pile des retours), ce qui revient à quitter le Rpl en cours et a évaluer deux fois la séquence nous séparant du prochain "**end**".
L'instruction **until** pourrait donc s'écrire: **?:DropRet/qd** ("**qd**" étant l'abréviation de "**quit;do**"). Elle pourrait tout aussi bien s'écrire **?:DropRet/(DupRet;q)**.
Exemple: On reprend le programme '**ESSAI**' précédent.

```
begin   "A"
   begin   "B"
      begin
         "C"   4roll   =0(r)?   until
         "D"   "E"   "F"
      end
      "G"
   end
   "H"
end
```

Voici deux exemples d'utilisation du programme '**ESSAI**' ci-dessus:
▸ **0 ESSAI** donne la pile:
7:"A" 6:"B" 5:"C" 4:"D" 3:"E" 2:"F" 1:"H".
▸ **1 ESSAI** donne la pile:
6:"A" 5:"B" 4:"C" 3:"G" 2:"G" 1:"H"

◆ L'instruction "endwhile" seule

Elle copie l'adresse de retour dans le registre **D0**, ce qui revient à quitter le Rpl en cours, et à évaluer deux fois ce qui reste à courir avant le "**end**" suivant.

```
begin    "A"
   begin   "B"   endwhile   "C"    "D"   end
   "E"   "F"
end
```

Le résultat du programme ci-dessus est:
6:"A" 5:"B" 4:"E" 3:"F" 2:"E" 1:"F"

◆ L'instruction "again" seule

Son action est la même que celle de "**endwhile**". On pourrait donc la nommer aussi **DupRet;q**.

Branchements vers adresses fixes

Le langage "*Rpl Système*" permet d'effectuer, à l'intérieur d'une struture Rpl, des branchements inconditionnels vers une adresse donnée.

La syntaxe est: ..... **branch adr** ....., où "*adr*" est une adresse (sur 5 quartets) où doit continuer le *flot des instructions* (tout revient à charger le registre **D0** du microprocesseur avec cette adresse).

Ces branchements ne doivent pas introduire de confusion dans la pile des retours. Autrement dit, une instruction **branch** ne doit pas diriger vers une adresse dont le niveau logique (en termes de retours de sous-programmes) est différente de celle de la structure Rpl depuis laquelle s'est effectué le branchement.

Certains programmes de la ROM de la HP48 font un usage important de cette instruction (notamment le programme de résolution d'équation, qui est le noyau de l'instruction **ROOT**). Il ne semble guère possible d'en faire autant pour des programmes écrits en mémoire vive (d'ailleurs ce ne sera pas une très bonne idée...).

Voici les adresses qui permettent de tels branchements. Même si elles sont inutiles, elles vous permettront de décrypter certains programmes de la ROM de la HP48.

Mnémonique	Adresse	Type	Commentaires
branch	#619CBh	LM	Branchement à l'adresse qui suit.
?:branch	#619E0h	LM	Si "true", alors branchement...
?#:branch	#619F3h	LM	Si "false", alors branchement...

Une possibilité d'utilisation consisterait à dresser une liste d'adresses de la ROM de la HP48 où on puisse se brancher depuis notre propre programme. Considérons par exemple l'adresse **#18C4Ah** (mnémonique **need(r)**) où on trouve une structure Rpl permettant de vérifier que la pile contient bien déjà au moins **r** éléments (**r** étant un réel présent au niveau 1). Ce programme est codé de la manière suivante:

```
begin   need1   dup;r?#:BAT!   r+=>s
   sdepth   dup2;<(2s)?   ?:drq   doerr201
end
```

L'adresse #18C4Ah+20=#18C5Eh est celle où se trouve l'instruction **sdepth** (qui place au niveau 1 le nombre d'éléments de la pile, sous la forme d'un entier-système). Il suffit de se brancher à cette adresse pour tester si la pile contient au moins **s** éléments (**s** entier-système présent au niveau 1) et de provoquer, si tel n'est pas le cas une erreur de code #201h ("Two Few Arguments").

Par exemple, la séquence (en reprenant la syntaxe du programme **EXTERNPC**, pour inclure l'adresse **#18C5Eh** dans le code source): **<5h> branch \E5C81**, vérifie que la pile contient au moins 5 éléments (sans pour autant "marquer" la pile au sens des instructions **need1**, **need2**, ..., **need5**.

Contrôle des erreurs

Le langage usuel de la HP48 permet d'intercepter les erreurs éventuelles d'une procédure, grâce aux instructions: **IFERR...THEN...END** et **IFERR...THEN...ELSE...END**. Il en va de même dans le langage "Rpl Système" où nous disposons des adresses suivantes:

Mnémonique	Adresse	Type	Commentaires
IfError	▪#04E5Eh	LM	Contrôle les erreurs éventuelles pour l'objet suivant dans la structure Rpl
ThenError	▪#04EB8h	LM	Précède l'objet à évaluer si une erreur s'est produite dans l'objet précédant (si "IfError" était actif).

La syntaxe est: **IfError Objet_Risque ThenError Objet_Traitement**, où:

- "**Objet_Risque**" est l'objet où peut éventuellement se produire une erreur.
- "**Objet_Traitement**" est l'objet à évaluer si "**Objet_Risque**" conduit à une erreur.

Dans les deux cas possibles (une erreur ou pas d'erreur dans "**Objet_Risque**"), la séquence d'instructions continue après "**Objet_Traitement**" (à moins que ce dernier ne contienne une instruction interrompant ou détournant le flot des instructions).

Dans le cas où une erreur s'est effectivement produite dans l'évaluation de "**Objet_Risque**", la procédure "**Objet_Traitement**" peut prendre connaissance de ce qui s'est passé, en utilisant les instructions suivantes (décrites dans le tome 1, page 356):

- **rclerr** (adresse **#4CE6h**): Rappelle le numéro de l'erreur survenue, sous la forme d'un entier-système. s= <70000h> est celui des erreurs personnalisées.
- **message(s)** (adresse **#4D64h**): Rappelle le message de numéro **s** (où **s** est un entier-système). Si s= <70000h>, il s'agit d'un message personnalisé.

Il est enfin possible de produire volontairement une erreur dans "**Objet_Risque**", au moyen des adresses suivantes (tome1, pages 356 et 357):

- **errsto(s)** (adresse **#4D0Eh**): sélectionne l'adresse n°**s**.
- **uerrm(st)** (adresse **#4E37h**): sélectionne le message d'erreur.
- **error!** (adresse **#4ED1h**): évalue l'erreur sélectionnée (sort de "**Objet_Risque**" et passe la main à "**Objet_Traitement**").

Quand "**Objet_Traitement**" est évalué, il peut être amené à réagir différemment suivant l'erreur détectée. Dans certains cas, il pourra effectivement générer cette erreur (instruction **error!**). Dans d'autres cas, il pourra la neutraliser avec l'instruction **clerr**.

- **clerr** (adresse **#4D33h**): annule la sélection d'une erreur.

LE CONTROLE DES ERREURS

Si l'évaluation de "**Objet_Risque**" devait conduire à plusieurs erreurs, la première d'entre elles interromp "**Objet_Risque**" et est accessible facilement par "**Objet_Traitement**" (au moyen surtout de l'instruction **rclerror**).

Le tome 1 contient (pages 356 et suivantes) un grand nombre d'adresses provoquant une erreur déterminée. D'autre part, la plupart des instructions intégrées de la HP48 peuvent conduire à une erreur d'exécution (ne serait-ce qu'une erreur du type "***Two Few Arguments***" ou "***Bad Argument Type***").

Toutes ces erreurs sont suspendues si elles surviennent dans l'évaluation de "***Objet_Risque***".

Considérons un exemple simple (on suppose que le flag-utilisateur n°-21 est armé, pour qu'un "*overflow*" se traduise par une erreur).

```
begin    need2    ChkArg    <11h>
   begin
      IfError    COMB
      ThenError    begin    rclerr    DropRet    end
      "OK"
   end
end
```

Le programme précédent commence par vérifier la présence de deux arguments réels, et il "*marque*" la pile initiale.

Il tente de calculer **COMB(r2,r1)**. Si tout se passe bien, on évalue la chaîne "**OK**". Sinon, on passe la main à la structure suivant "**ThenError**", qui place sur la pile le numéro de l'erreur, puis exécute l'instruction **DropRet** (détruit la prochaine adresse de retour).

Si on place ce programme dans la variable '**ESSAI**', on obtient:

- 2:15 1:7 ---[ESSAI]--▸ 2:6435 1:"OK"
- 2:15 1:-7 ---[ESSAI]--▸ 1:<203h> (n° de "Bad Argument Value")
- 2:2000 1:1000 ---[ESSAI]--▸ 1:<303h> (n° de "Overflow")

Si on remplace, dans l'exemple précédent, "**DropRet**" par **error!**, on obtient:

- Dans le premier cas:
  2:15 1:7 ---[ESSAI]--▸ 2:6435 1:"OK"
- Dans le deuxième cas:
  2:15 1:-7 ---[ESSAI]--▸ 2:15 1:-7 et erreur "*Bad Argument Value*".
- Dans le troisième cas:
  2:2000 1:1000 ---[ESSAI]--▸ 2:2000 1:1000 et erreur "*OverFlow*".

On voit sur ce dernier exemple que l'instruction **error!** rétablit la dernière pile "*marquée*" (en l'occurence la pile est marquée deux fois dans l'exemple précédent: une première fois par **need2**, et une seconde fois par l'instruction intégrée **COMB**).

UTILITAIRES EN EXTERNALs

Nous allons faire un bilan de ce qui précède, à l'aide d'exemples utilisant certains des Externals qui ont été décrits dans les tomes 1 et 2, pour illustrer les méthodes de programmation qui ont été traitées jusqu'à présent.

Les programmes présentés ici sont de difficulté variée. Ils montrent l'étendue des possibilités qui nous sont maintenant offertes. Les listings des programmes sont parfois présentés dans le langage usuel, mais toujours donnés sous la forme attendue par les utilitaires **→EXT** et **EXTERNPC**.

L'ADRESSE DU NIVEAU 1

On sait que la pile de la HP48 ne contient réellement que des adresses. C'est l'opération de décompilation des objets situés à ces adresses, réalisée pour l'affichage, qui donne l'illusion que les objets sont présents aux différents niveaux de la pile.

Pour autant, il n'est pas très facile de connaître ces adresses. Nous allons confier à quelques Externals le soin de résoudre ce problème.

L'outil essentiel est ici l'adresse **#6595Ah** (Mnémoniqe **Rcl16Nib(2b)**) dont le rôle est le suivant:

2: #b2 1: #b1 ---[#6595Ah SYSEVAL]---▸ 1:#b3

#b3, #b2, #b1 sont des entiers binaires. #b2 est quelconque, et #b1 désigne une adresse (entre #0 et #FFFFFh mais de toute façon seuls les 5 chiffres les plus faibles sont pris en compte).

L'opération place dans l'entier binaire #b2 les 16 quartets (*nibbles*) à partir de l'adresse #b1, ce qui conduit à un résultat #b3.

L'autre point important est que le registre **D1** du microprocesseur pointe constamment sur la base de cette pile (c'est à dire sur l'objet du niveau 1). Le contenu du registre **D1** est en permanence sauvegardé à l'adresse **#806F8h**.

En appliquant **"Rcl16Nib(2b)"** à cette adresse, on obtiendra le contenu de **D1**, c'est-à-dire l'adresse où se trouve le niveau 1 de la pile (mais pas l'adresse qui est placée *sur* ce niveau).

Il suffit alors d'appliquer à nouveau cet External. Le reste est une question de détail.

Le programme **'→ADR'** ci-dessous remplace l'objet situé au niveau 1 de la pile par son adresse. Son schéma fonctionnel est donc:

1:Obj ---[→ADR]---▸ 1:AdrObj

Commençons par un listing dans le langage usuel:

```
« #0h  NEWOB  DUP
  #806F8h
  #6595Ah  SYSEVAL
  10   +
  #6595Ah   SYSEVAL
  #FFFFFh   AND
  SWAP   DROP
»
```

```
3:Obj   2:#0h   1:#0h
4:Obj  3:#0h  2:#0h  1:#806F8h
3:Obj   2:#0h   1:#...AdrNiv1
3:Obj   2:#0h   1:#...AdrNiv3
2:Obj   1:#...AdrObj
2:Obj   1:#AdrObj
1:#AdrObj
```

Commentaires:

- Vous pouvez être sceptique sur la présence de **NEWOB** en première ligne. Sachez que si on ne l'utilise pas, le programme continue à fonctionner parfaitement, mais l'entier binaire **#0h** est directement modifié **dans le corps du programme** (celui-ci se transforme donc lui-même: un programme mutant?).
- La notation **1#...AdrNiv1h** signifie ici qu'on a obtenu un entier binaire dont les 5 derniers chiffres hexadécimaux désignent l'adresse du niveau 1 de la pile.
- L'instruction **10 +** est nécessaire pour accéder au niveau 3 de la pile (où se situe alors l'adresse de **Obj**).
- La dernière instruction (**#FFFFFh AND**) sert à isoler les 5 chiffres constituant l'adresse de l'objet **Obj** des 11 chiffres suivants, également obtenus par le dernier **#6595Ah SYSEVAL**.

Exemple:

Le programme '**→ADR**' permet d'obtenir l'adresse de tous les objets intégrés de la HP48 (ce qui peut permettre de faire l'économie de ces adresses dans le fichier des mnémoniques nécessaire pour utiliser le programme **EXTERNPC**).

Ainsi: **1:COS** ---[**→ADR**]---▸ **2:COS 1:#1B505h**
1:−9 ---[**→ADR**]---▸ **2:−9 1:#2A42Eh**
Mais **1:10** ---[**→ADR**]---▸ **2:10 1:#8DF0Dh**
(cette dernière adresse pouvant varier, bien sûr).

Au format →EXT:

Vous appliquez le programme **→EXT** à la liste suivante, pour obtenir une séquence Rpl à stocker dans '**→ADR**' (N.B: la commande BYTES, appliquée à la structure Rpl obtenue, donne une taille de **66.5 octets**, et un CheckSum de **#10B8h**). On remarquera, dans la liste ci-dessous, les trois entiers binaires "taggués" (le programme **→EXT** ne devant pas traduire ces entiers en adresses).

```
{ #18AB2h     ::#0h      #6657h     need1    #0h      newob
  #3188h      ::#806F8h  #6595Ah    dup  #806F8h  Rcl16Nib(2b)
  #650E7h     #54349h    #6595Ah    10   +(b,r)   Rcl16Nib(2b)
  ::#FFFFFh   #53D04h    #60F9Bh    #FFFFFh  and(2b)   drop(2)
}
```

Au format EXTERNPC:

Voici le texte à soumettre au programme **EXTERNPC**:

```
begin
   need1   #0h   newob   dup   #806F8h   Rcl16Nib(2b)
   10  +(b,r)  Rcl16Nib(2b)  #FFFFFh   and(2b)   drop(2)
end
```

CONVERSION OBJET ⇒ CHAINE

Il ne s'agit pas ici d'utiliser la commande **→STR**, mais plutôt de former la chaîne représentant un objet tel qu'il est codé en mémoire.

Considérons la liste **{ 1 }**: sa représentation en mémoire est: **47A209C2A2B2130** (**47A20** pour l'adresse-préfixe #02A74h des listes, **9C2A2** pour l'adresse #2A2C9h du réel 1, **B2130** pour l'adresse-suffixe #0312Bh des listes).

Le but du programme '**→STRI**', présenté ci-après, est de convertir, en reprenant cet exemple, la liste **{ 1 }** en la chaîne "**47A209C2A2B2130**". Le schéma fonctionnel est donc:

1:Obj ---[→STRI]---▸ 1:StrObj

Remarque:

Il est très important que l'objet que l'on veut transformer en chaîne ne reste pas basé en ROM (s'il l'est au départ), car sinon il est représenté par leur adresse.

On aura recours à l'adresse **#6657h** (Mnémonique **newob**) qui est la forme interne de **NEWOB**, et qui contrairement à cette dernière génère une nouvelle copie des objets basés dans la ROM.

Considérons l'objet **{ 1 }**, évoqué ci-dessus et baptisons le **Obj**. Sa taille, exprimée en quartets sous la forme d'un entier système, est **<Fh>** (quinze quartets). Si on lui ajoute <5h> (on verra pourquoi dans un instant), on obtient l'entier-système **<14h>**, dont le codage en mémoire est **1192041000**: **11920** pour l'adresse-préfixe **#02911h** des entiers-système et **41000** pour le contenu égal à 14h).

Si on applique l'instruction **get1(cp)** (adresse **#5089h**, voir tome 1 page 182) à cet entier-système, on place sur la pile l'External (invalide) correspondant à l'adresse #00014h.

On crée une liste avec ces deux objets (d'abord l'External, pour être sûr que l'objet **Obj** le suive immédiatement en mémoire).

L'image de cette liste en mémoire est (les points sont non significatifs: ils sont là uniquement pour aider la lecture):

47A20.41000.47A209C2A2B2130.B2130

On applique alors à cette liste l'instruction **MapStr(b)** (adresse **#15E83h**) dont le rôle est de convertir en chaîne de caractères ascii les caractères héxadécimaux d'un entier binaire (tome 1, page 162).

En l'occurrence, la HP48 ne vérifie pas que l'objet auquel on applique cette instruction est bien un entier binaire: elle se contente d'interpréter les quartets qui suivent l'adresse préfixe (les 5 premiers indiquant la longueur).

Voici comment le programme **'→STRI'** peut s'écrire, dans le langage usuel de la HP48 (Taille 113 octets; CheckSum: #A726h):

```
«  #6657h      SYSEVAL                    newob
   DUP      #5902h      SYSEVAL           dup  nibbles(_)
   #6258Ah      SYSEVAL                   5+(s)
   #6657h      SYSEVAL                    newob
   #5089h      SYSEVAL                    get1(cp)
   SWAP      2      →LIST                 SWAP  2  →LIST
   #15E83h      SYSEVAL                   MapStr(b)
»
```

Au format →EXT:

Voici la liste sur laquelle vous appliquerez le programme **→EXT**. Elle se déduit très simplement du listing ci-dessus, mais elle contient un appel à l'adresse **#18AB2h** (Mnémonique **Need1**) pour vérifier que la pile n'est pas vide.

On y utilise d'autre part les adresses suivantes:

#3188h et **#3223h**: formes internes de DUP et SWAP.

#631B9h (Mnémonique **2->list**)

#5089h (Mnémonique **get1(cp)**: voir tome 1 page 182).

```
{  #18AB2h      #6657h      #3188h      #5902h
   #6258Ah      #6657h      #5089h      #3223h
   #631B9h      #15E83h
}
```

La commande **BYTES**, appliqué à la structure Rpl obtenue par **→EXT** à partir de cette liste, donne une **taille de 30 octets**, et un **CheckSum de #F8EAh**.

Cette structure Rpl devient le nouveau contenu du programme **'→STRI'** (notez le passage de 113 à 30 octets!).

Au format EXTERNPC:

Voici le texte à utiliser avec **EXTERNPC**:

```
begin
   need1    newob    dup    nibbles(_)    5+(s)
   newob    get1(cp)   swap    2->list    MapStr(b)
end
```

Remarque:

Le programme **→STRI** est rapide: il transforme par exemple la matrice identité d'ordre 15 en 0.23 seconde (la chaîne obtenue ayant 3630 caractères).

CONVERSION CHAINE ⇒ OBJET

Il s'agit là de l'opération inverse de celle réalisée par le programme **'→STRI'** précédent. Autrement dit, et pour reprendre l'exemple qui avait alors été utilisé, il s'agit de remplacer la chaîne **"47A209C2A2B2130"** par l'objet **{ 1 }** qui est codé **47A209C2A2B2130** en mémoire.

Remarque:

Il convient ici de placer une observation valable autant pour ce programme que pour le programme précédent. Supposons par exemple que la chaîne **"47A209C2A2B2130"** soit présente au niveau 1 de la pile. A ce niveau figure en fait une adresse qui pointe sur le début de la séquence suivante (représentant le codage de la chaîne en mémoire):

C2A20320004373142303933423142324231333 03.

Notre but est ici de transformer la chaîne **"47A209C2A2B2130"** en l'objet **{ 1 }**, c'est-à-dire de faire en sorte que le niveau 1 de la pile contienne maintenant une adresse pointant sur le début de la séquence (représentant le codage de la liste **{ 1 }** en mémoire): **47A209C2A2B2130**

On voit bien que les deux séquences sont totalement différentes. Dans le premier cas par exemple, le caractère **"4"** qui débute la chaîne est en fait représenté par son code ASCII **34h** (codé sur un octet à l'envers) et doit être remplacé par le quartet **4**.

On peut songer à effectuer une transformation caractère par caractère, mais celà conduit à des algorithmes et relativement longs et lents. On peut également imaginer que la ROM de la HP48 contient quelque part des routines permettant de telles conversions. Tel est bien le cas, la difficulté étant de les trouver...

L'adresse **#15E83h** (Mnémonique **MapStr(b)**) nous a aidé dans le programme précédent. Un autre adresse va nous tirer d'affaire ici.

Nous allons appeler **'→OBJ'** le programme inverse de **'→STRI'**.

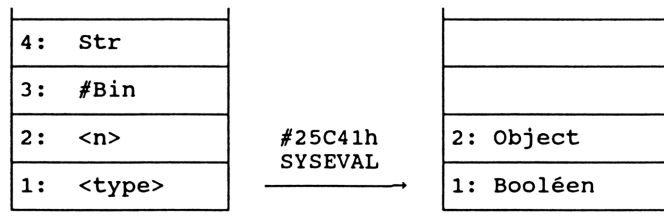
Son schéma fonctionnel doit donc être:

1:StrObj ---[→OBJ]---▸ 1:Obj

Autrement dit **→OBJ** doit reconstituer un objet **Obj**, connaissant la chaîne de caractères **StrObj** qui représente son image en mémoire.

Nous allons utiliser l'adresse **#25C41h** de mnémonique **MapBin(st,b,2s)**.

Voilà comment fonctionne ce SYSEVAL:



- Ici, **Str** est une chaîne, **#Bin** un entier binaire, **<n>** et **<type>** deux entiers-système (**<type>** représentant une adresse-préfixe).

- **Object** est un objet de type défini par l'entier-système **<type>**. Il résulte de la copie du corps de **Str** sur l'entier binaire **#Bin**, à partir de la position **<n>** (la position <5h> par exemple, signifie celle qui suit immédiatement l'adresse-préfixe).

- La copie de **Str** sur **#Bin** s'effectue (c'est bien sûr capital ici) en transformant les octets 03,13,..,93,14,24,...,64 (qui représentent les codes Ascii respectifs de 0,1,...,9,A,B,...,F) en les quartets 0,1,...,9,A,B,...,F.

- L'entier binaire **#Bin** doit être "assez long" pour recevoir le contenu de **Str**. N'oublions pas que nous pouvons avec l'adresse **#EDE1h** (Mnémonique **makeb(s)**) construire des entiers binaires aussi longs que nécessaire.

- Le booléen obtenu vaut **True** si tout s'est bien passé. Il vaut **False**, par exemple, si un caractère autre que "0","1",...",F" est apparu dans la chaîne **Str**.

Reprenons l'exemple de la chaîne **"47A209C2A2B2130"**. Elle est longue de 15 caractères. Nous créons un entier binaire de 15 chiffres, qui est donc codé **E4A20·41000·000000000000000**.

Nous appliquons la chaîne sur l'entier binaire, à partir de la position **<Ah>** (juste après le champ 41000). D'autre part, nous ne modifions pas l'adresse-préfixe.

Le résultat est un entier binaire dont l'image en mémoire est **E4A20·41000·47A209C2A2B2130**.

Ce n'est pas terminé. Il va falloir faire preuve encore d'un peu de culot. Nous considérons l'entier binaire obtenu comme un objet composé (au même titre qu'une liste, une structure Rpl, une expresssion,...). Le premier élément de cet "objet composé" est **41000** qui correspond à l'adresse **#00014h**: c'est un *External*. Le deuxième élément est précisément l'objet **47A209C2A2B2130** recherché, c'est-à-dire la liste **{ 1 }**.

Il suffit donc d'extraire ce deuxième élément par une instruction du type 2 GET (plus précisément, en utilisant les mnémoniques, **<2h> get(cp,s)**: cf tome 1, p.182).

Remarque:

Cette dernière astuce recèle un (tout petit) problème. Il ne faut pas que le champ représentant la longueur de l'entier-binaire contienne une adresse-préfixe, sans quoi l'instruction GET évoquée ci-dessus serait mise en défaut. Le résultat obtenu ne serait pas l'objet souhaité, mais un *External*. Sachant qu'il y a une trentaine d'adresses-préfixe, de #2911h=10513 à #2E92h=11922, celà concerne une trentaine de longueurs de chaînes, toutes supérieures à 10000. Autant dire que le risque est nul....

La dernière opération à réaliser est d'effectuer un **NEWOB** sur l'objet obtenu, pour le libérer complètement de l'entier binaire qui le "*contient*" (ce dernier peut alors être *nettoyé* lors du prochain "*Garbage Collection*").

Le programme →OBJ

Au format →EXT:

Voici la liste qu'il faut soumettre au programme **→EXT**. Elle contient des tests permettant de vérifier qu'une chaîne est bien présente au niveau 1 de la pile, et qu'elle est non vide. Si cette chaîne est incorrecte (présence de caractères autres que "0"..."F"), alors il y a une erreur "*Bad Argument Value*".

Pour cette liste, la taille est 202.5 octets, et le CkeckSum est #4452h.

```
{  #18AB2h   #18FB2h
   #400Dh
   {  #63209h   #63B05h
      #627BBh   #EDE1h
      #4053h    10830
      #25C41h
      #63B19h
      #60F9Bh
      #4003h    #62B9Ch
      #6657h
   }
}
```

```
need1  ChkArg
<3h>, pour une chaîne
dup;""=?   ?:BAV!
dup;ssize(st)   makeb(s)
<Ah>  <2A4Eh>
MapBin(st,b,2s)
?#:BAV!
drop(2)
<2h>  get(cp,s)
newob
```

Le programme **→EXT** transforme cette liste en une structure Rpl que vous pouvez stocker sous le nom '**→OBJ**'. Pour cette structure Rpl, la taille est **50 octets**, et le CheckSum est **#3D12h**.

Remarque: les programmes '**→STRI**' et '**→OBJ**' sont extrêmement rapides. Si on considère par exemple la matrice identité d'ordre 50 (obtenue par 50 IDN, et dont la taille est supérieure à 20000 octets), elle est transformée par '**→STRI**' en 2.3 secondes, et '**→OBJ**' recrée la matrice initiale en 2.4 secondes (on ne compte pas ici les temps d'affichage, indépendants des programmes).

Au format EXTERNPC:

Voici le texte à utiliser avec le programme **EXTERNPC**:

```
begin
   need1    ChkArg    <3h>
   begin
      dup;""=?    ?:BAV!
      dup;ssize(st)    makeb(s)    <Ah>    <2A4Eh>
      MapBin(st,b,2s)    ?#:BAV!
      drop(2)    <2h>    get(cp,s)    newob
   end
end
```

LIRE AVEC PEEK

On veut lire une séquence de quartets (de longueur quelconque) à partir d'une adresse donnée. De tout temps, les programmes réalisant ce genre d'opération se sont appelés **PEEK**. Nous n'allons pas déroger à cette habitude.

Le schéma fonctionnel doit être le suivant:

2:#Adr 1:n ---[PEEK]---▸ 1:Str

Ici **#Adr** est un entier binaire précisant l'adresse à laquelle doit s'effectuer la lecture, et **n** est un réel (positif, qui sera arrondi à l'entier le plus proche) spécifiant le nombre de quartets à lire à partir de cette adresse.

Le résultat **Str** est une chaîne formée par l'image des **n** quartets lus (par exemple, si on lit 5 quartets égaux respectivement à 8,A,0,0,1 alors la chaîne sera "8A001").

Voici quelques éléments permettant de comprendre l'algorithme utilisé:

- Il convient que le programme **'PEEK'** teste la présence de deux objets sur la pile (adresse **#18A8Dh**, mnémonique **need2**) sous peine d'une erreur "*Too Few Arguments*", et teste la nature de ces arguments (erreur "*Bad Argument Type*"). Cette dernière vérification est réalisée par l'adresse **#18FB2h** (Mnémonique **ChkArg**) déjà rencontrée plusieurs fois, associée à l'identificateur **<B1h>** ("B" pour l'entier binaire au niveau 2 et "1" pour le réel au niveau 1).

- On s'assurera que le réel **n** s'arrondit bien à un entier strictement positif.

- L'outil principal de **'PEEK'** est l'adresse **#6595Ah** (Mnémonique **Rcl16Nib(2b)**) qui a été décrite dans le programme **'→ADR'**, au début de ce chapitre.
  Le problème, avec cette adresse, est qu'elle ne permet de lire que 16 quartets à la fois. Il faut donc imaginer une boucle effectuant de telles lectures de 16 quartets, en partant de l'adresse **#Adr** (incrémentée de 16 à chaque passage dans la boucle), jusqu'à avoir effectivement lu les **n** quartets requis.

- Le résultat de chacune de ces lectures est placé dans un entier binaire. Il faudra donc *concaténer* ces entiers (adresse **#518Ah**, mnémonique **conc(2b)**).

- Le résultat final est alors un entier binaire de **n** chiffres hexadécimaux, qu'il faut convertir en une chaîne de **n** caractères "0" à "F". On sait depuis le programme **'→STRI'** que cette opération est réalisée par l'adresse **#15E83h** (Mnémonique **MapStr(b)**).

Le problème étant maintenant bien délimité, on peut proposer une solution.

Le programme PEEK

Au format →EXT:

Voici la liste sur laquelle vous devez appliquer le programme **→EXT**. La taille de cette liste est de **325.5 octets**, et sa somme de contrôle est **#CF7Dh**.

```
{  #18A8Dh   #18FB2h
   #64DC4h
   {
      #18CEAh
      #62266h   #63B05h
      #55D5h
      #3295h   #5A03h
      #3188h   #62DE5h
      #73F7h
         ::#0h    #6657h
         #7221h   #59CCh
         #6595Ah
         #518Ah
      #408Fh   #73A5h
      #3FF9h   #3295h
      #5815h
      #15E83h
   }
}
```

```
need2   ChkArg
<B1h>

r+=>s
dup;=0(s)?    ?:BAV!
entier binaire vide
rot    b=>s
dup   4pick;+(2s);swap
start(s1,s2-1)
#0h   newob
index   s=>b
Rcl16Nib(2b)
conc(2b)
<10h>   step(s)
<1h>   rot
sub(b,2s)
MapStr(b)
```

La structure Rpl obtenue a les caractéristiques suivantes (obtenues par **BYTES**): la taille est de **80.5 octets**, et le CheckSum est **#B4C1h**.

N.B: le programme **'PEEK'** ainsi obtenu est très rapide. Il effectue par exemple une lecture de 1000 quartets en 0.49 secondes....

Au format EXTERNPC:

Si vous utilisez le programme **EXTERNPC**, vous lui soumettrez le texte suivant:

```
begin
  need2     ChkArg    <B1h>
  begin
     r+=>s     dup;=0(s)?      ?:BAV!
     #0h      rot      b=>s
     dup     4pick;+(2s);swap
     start(s1,s2-1)
        #0h     newob     index     s=>b
        Rcl16Nib(2b)      conc(2b)
     <10h>     step(s)
     <1h>     rot     sub(b,2s)     MapStr(b)
  end
end
```

ECRIRE AVEC POKE

Comme d'habitude, pas de **PEEK** sans **POKE**... Le rôle de **POKE** est d'écrire une séquence de quartets à partir d'une adresse donnée. C'est une opération risquée si elle est effectuée *au petit bonheur*, car elle peut se traduire par un ***Memory Clear***.

Le schéma fonctionnel est ici:

2:#Adr 1:Str ---[POKE]---▸.....

#Adr est un entier binaire indiquant l'adresse où doit commencer l'écriture (en mémoire vive...) et **Str** est une chaîne dont chaque caractère représente un quartet à écrire. Si par exemple la chaîne **Str** est égale à "A024E", alors cinq quartets, respectivement égaux à A,0,2,4,E doivent être écrits à partir de l'adresse **#Adr**.

L'adresse essentielle ici est **#6594Eh** (Mnémonique **Put16Nib(2b)**) dont le fonctionnement peut être représenté de la manière suivante:

2: #b2 1: #b1 ---[#6594Eh SYSEVAL]---▸.....

Ce SYSEVAL écrit, à partir de l'adresse désignée par l'entier binaire **#b1**, le contenu de l'entier binaire **#b2**. Celui-ci doit posséder au plus 16 chiffres hexadécimaux (sinon rien ne se passe), et il y a autant de quartets écrits que de chiffres hexadécimaux dans #b2. Cette opération ne permet donc pas d'écrire plus de 16 quartets à la fois, mais elle permet d'en écrire de 1 à 16.

Le programme **POKE** va effectuer, en gros, les opérations suivantes:

- Vérifier la présence de deux arguments sur la pile et leur nature (un entier binaire au niveau 2, une chaîne au niveau 1).

- S'assurer que la chaîne **Str** n'est pas vide, puis la convertir en un entier binaire **#b**. La chaîne "A18247" sera par exemple transformée en l'entier binaire #74281Ah dont le codage en mémoire est E4A20B0000A18247.
  Cette opération (ou chaque caractère de la chaîne initiale est converti en le quartet qui lui correspond, en conservant l'ordre en mémoire) est assurée par l'adresse **#25C41h** (Mnémonique **MapBin(st,2b,s)**) que nous avons déjà utilisée dans le programme **'→OBJ'** ci-avant.

- Ecrire le contenu de l'entier binaire **#b** ainsi obtenu, à partir de l'adresse **#Adr**, au moyen de l'adresse **#6594Eh** (Mnémonique **Put16Nib(2b)**). Il faut nécessairement prévoir une boucle permettant d'écrire par tranches de 16 quartets, si l'entier **#b** a plus de 16 chiffres hexadécimaux (la dernière tranche pouvant être de longueur inférieure à 16).

- Il ne reste plus qu'à écrire la liste des adresses à utiliser. On remarquera une grande analogie entre le début de **'POKE'** et celui de **'→OBJ'**.

Le programme POKE

Au format →EXT:

Vous appliquez le programme **→EXT** à la liste suivante, pour obtenir une structure Rpl que vous stockerez sous le nom **'POKE'**. La commande **BYTES** donne:

- Pour la liste: taille **369 octets**, CheckSum **#A45Eh**.
- Pour la structure Rpl: taille **85 octets**, CheckSum **#C042h**.

```
{  #18A8Dh    #18FB2h
   179
   {  #63209h    #63B05h    #627BBh
      #EDE1h     #4053h     10830
      #25C41h    #63B19h    #60F9Bh
      #3188h     #5616h
      #73CEh
         #63411h    #4085h
         #6372Ch    #5815h
         #611FEh    #6594Eh
         #1CD3Ah    #3295h
         #54330h    #3223h
      #408Fh     #73A5h
      #3258h
   }
}
```

```
need2     ChkArg
deviendra <B3h>
( voir le
programme
'→OBJ' )
dup     quart(b)
start(1,s-1)
dup;index     <Fh>
over;+(2s)  sub(b,2s)
3pick   Put16Nib(2b)
16     rot
+(r,b)     swap
<10h>     step(s)
drop2
```

Exemple: Placez le nom **'→EXT'** sur la pile. Effectuez **DUP →ADR 7 +** (où **→ADR** est le programme vu précédemment). On obtient alors l'adresse où débute le codage des caractères du nom **→EXT** (après avoir sauté le préfixe et la longueur). Placez **"45"** sur la pile (le code ascii de **T** est #54h) et exécutez **POKE**. Vous obtenez **'TEXT'** au niveau 1...

Au format EXTERNPC:

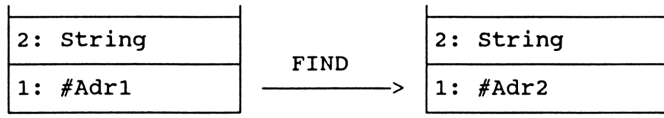
Voici le texte que vous pouvez utiliser avec le programme **EXTERNPC**.

```
begin
   need2   ChkArg    <B3h>
   begin
      dup;""=?   ?:BAV!
      dup;ssize(st)   makeb(s)   <Ah>   <2A4Eh>
      MapBin(st,b,2s)   ?#:BAV!   drop(2)
      dup   quart(b)    start(1,s-1)
         dup;index   <Fh>   over;+(2s)
         sub(b,2s)   3pick   Put16Nib(2b)
         16   rot   +(r,b)   swap
      <10h>   step(s)
      drop2
   end
end
```

RECHERCHE EN MÉMOIRE

Le programme **PEEK** (lecture d'une séquence de quartets en mémoire) permet de concevoir un algorithme de recherche d'une séquence donnée, à partir d'une adresse donnée.

Le schéma fonctionnel de **FIND** est:



Ici **String** est la séquence de quartets à rechercher (sous forme d'une chaîne de caractères de "0" à "F", comme c'était le cas pour le programme **POKE**), et **#Adr1** est un entier binaire représentant l'adresse où doit débuter la recherche.

#Adr2 est l'adresse où la première occurrence de **String** a été trouvée (la recherche s'effectue dans le sens des adresses croisssantes).

Le schéma fonctionnel est tel qu'il suffit d'ajouter 1 à **#Adr2** pour relancer la recherche de l'occurrence suivante.

Bien sûr, si la séquence de quartets représentée par **String** n'est pas trouvée, ou si la recherche vous semble trop longue, vous pouvez interrompre le programme **FIND** par un appui sur la touche **ON** (où qu'en soit le déroulement de ce programme, l'entier binaire se trouvant sur la pile est une adresse indiquant la progression dans la recherche). Voici donc le programme **FIND**, qui ne nécessite pas d'*External*:

```
« OVER SIZE #200h DUP ROT + B→R
  → s n
  « s -
    DO s + DUP 1 DISP DUP n PEEK
    UNTIL 3 PICK POS END
    LASTARG + 1 -
  »
»
```

Remarques:

- La séquence **DUP 1 DISP** affiche, en haut et à gauche de l'écran, l'adresse où s'effectue la recherche, donnant ainsi une idée de la progression.
- Le nombre #200h ci-dessus (#200h=512) indique la quantité **s** dont augmente l'adresse de base de la recherche à chaque étape. Ce nombre doit être supérieur à la longueur de la séquence **Str** mais il n'est pas avantageux de le choisir trop élevé.
- Au-delà de l'adresse #80000h, et sur 32K ou 128K, c'est de la mémoire vive.

Exemple: en environ 6 secondes (et #220Ah=8714),
2:"B2130" 1:#0h --[FIND]--▸ 2:"B2130" 1:#220Ah

UNE PILE A 7 NIVEAUX

L'affichage de la HP48 ne permet de visualiser qu'au plus quatre niveaux de la pile opérationnelle. Dans cet exemple, nous allons voir comment afficher en permanence les 7 premiers niveaux.

On sait que la plupart des actions au clavier, ou presque, redessinent la pile. Il serait donc frustrant que l'affichage sur 7 niveaux soit "fugitif" (et que l'on soit contraint d'évaluer constamment le programme chargé de produire cet affichage).
La solution existe avec le mode **USER** (tout le monde sait ce que c'est) et le mode "***ENTER redirigé***" (c'est déjà moins connu). Rappelons en quelques lignes ce principe.

On sait que de nombreuses touches du clavier exécutent la commande **ENTER**. Les choses se passent alors ainsi:

1) La ligne de commande est examinée (au cas où elle recèlerait des erreurs de syntaxe, par exemple), puis évaluée si cette vérification n'a rien décelé d'anormal.
2) L'action associée à la touche pressée est alors exécutée.

Quand on est en mode **USER** (indicateur -62 armé), et que l'indicateur -63 est armé (pour activer le mode "***ENTER redirigé***") les choses se passent un peu différemment:

1) La HP48 cherche si une variable nommée '**αENTER**' (*Alpha-ENTER*) existe dans le répertoire en cours, ou sur le chemin menant jusqu'à **HOME**.
   Si '**αENTER**' n'est pas trouvée, la ligne de commande est évaluée normalement.
   Si cette variable est trouvée, La ligne de commande est placée au niveau 1 de la pile, sous la forme d'une chaîne de caractères (sans être vérifiée). Le contenu de '**αENTER**' est alors évalué.
   '**αENTER**' se subsitue donc à l'instruction **OBJ→** qu'applique implicitement la HP48 quand elle évalue la ligne de commande.
2) L'action associée à la touche pressée est exécutée normalement.
3) La HP48 cherche si une variable nommée '**ßENTER**' (*Bêta-ENTER*) existe dans le répertoire en cours, ou sur le chemin menant jusqu'à **HOME**.
   Si '**ßENTER**' n'est pas trouvée, rien ne se passe.
   Si cette variable est trouvée, la touche ayant été à l'origine de tout celà est placée sur la pile, sous forme d'une chaîne de caractères, puis le contenu de '**ßENTER**' est évalué.
   '**ßENTER**' permet donc d'intervenir immédiatement après chaque évaluation d'une touche exécutant **ENTER**.

Une application classique du mode "***ENTER redirigé***" consiste à "***Tracer***" (sur imprimante par exemple), chacune des opérations où la touche **ENTER** est (implicitement, le plus souvent) évaluée.

Pour ce qui nous concerne, nous nous limiterons à une utilisation de la variable **'ßENTER'**, en lui demandant:

- D'éliminer du niveau 1, par **DROP**, la chaîne représentant la touche qui a exécuté **ENTER** (chaîne qui ne nous est d'aucune utilité).
- D'exécuter la procédure produisant un affichage de 7 niveaux.

Il faudra songer à ce que cette procédure d'affichage se termine par un ***gel*** de l'écran (de manière à ce que l'aspect habituel de la pile ne réapparaisse pas au moindre changement de menu par exemple).

Notons symboliquement **PROC7** le corps de la procédure générant, l'affichage sur 7 niveaux. Le contenu de **'ßENTER'** pourrait donc être: **« DROP PROC7 »**

Nous allons créer deux programmes, baptisés **'→ST7'** et **'→ST4'**, permettant de sélectionner les affichages "*7 niveaux*" ou "*4 niveaux*".

Le rôle de **'→ST7'**:

- Armer l'indicateur **−61** ("*User Mode Lock*"). Un seul appui sur la touche **USR** passe alors en mode **USER *bloqué*** (le fait de rester en mode **USER** pour une seule touche n'aurait en effet pas grand intérêt).
- Armer l'indicateur **−62** ("*User Mode*"): on est alors en mode **USER *bloqué***.
- Armer l'indicateur **−63** ("*Vectored ENTER*"): enclenche le mode *ENTER redirigé*.
- Placer le programme évoqué ci-dessus dans la variable **'ßENTER'** (dans **HOME**, pour que l'affichage sur 7 niveaux soit effectif où que l'on se trouve).

Le rôle de **'→ST4'**:

- Désarmer l'indicateur **−62** (pour sortir du mode **USER**).
- Désarmer l'indicateur **−63** (pour sortir du mode *Enter redirigé*).
- Purger la variable **'ßENTER'** du répertoire **HOME** (ça n'a rien d'obligatoire).

Les deux programmes **'→ST4'** et **'→ST7'** permettent donc de passer du mode "*4 niveaux*" au mode "*7 niveaux*", et inversement. L'affichage "*7 niveaux*" subsiste tant que l'on exécute pas **'→ST4'**, ou tant que l'on ne sort pas du mode **USER** par un appui sur la touche **USR**.

Nous allons commencer par **'→ST4'**, qui est bien sûr le plus simple. Il n'est pas indispensable d'avoir recours ici à des Externals.

Voici donc une proposition pour **'→ST4'**:

« −62 CF −63 CF PATH HOME 'ßENTER' PURGE EVAL »

Les programmes →ST7 et →ST4

Il faut maitenant en venir aux choses sérieuses!

L'affichage sur 7 niveaux:

L'idée n'est pas très compliquée. On crée une boucle dont l'indice **k** varie de 1 à 7. Pour une valeur donnée de cet entier, on considère le niveau **k** de la pile.

- Si ce niveau est vide, c'est-à-dire si DEPTH < k, alors on crée une chaîne **Str** vide. Si un objet **Obj** se trouve à ce niveau, on le transforme en une chaîne **Str**.
- On ajoute la chaine **"k:"** au début de la chaîne **Str** (**k** étant remplacé par sa valeur). Il faut veiller à ce que la chaîne obtenue montre bien l'objet *cadré à droite*, ce qui peut nécessiter d'introduire des espaces entre **"k:"** et **Str** (du moins si on utilise le langage de programmation standard, mais on verra qu'avec nos *Externals*, c'est beaucoup plus simple).
- On affiche cette chaîne au moyen de l'instruction **DISP**. La chaîne représentant le niveau **k** doit subir un **(8 – k) DISP**.

L'intérêt de programmer ce problème en *Externals* est double (au moins): le programme obtenu est plus rapide (on s'en doutait) mais il est beaucoup plus compact (car on verra que certaines adresses font une partie du travail toutes seules).

Voici les adresses les plus marquantes qui seront utilisées dans la solution:

- **#44197h** (Mnémonique **pick(s)?**): Essaye d'appliquer la séquence **s PICK**, où **s** est l'entier-système du niveau 1. Le résultat est **2:Obj 1:true** si la pile est assez haute, et **1:False** sinon (voir tome 1, page 150).
- **#15978h** (Mnémonique **->StackStr!**): Transforme un objet en la chaîne qui le représenterait sur la pile, en mode *uniligne* (voir tome 1, page 200).
- **#1685Ch** (Mnémonique **s:Str(st,s)**): Concatène la chaîne **st** avec l'entier-système **s** de manière à produire une chaîne de 22 caractères, la sous-chaîne **Str** étant cadrée à droite. L'entier-sytème **<3h>**, par exemple, est remplacé par **"3:"** (cf tome 1, page 201).
- **#5182Fh** (Mnémonique **BeforeStop**): Dans le courant d'une boucle, si **s** est l'index en cours, et si **n** est la valeur de sortie de boucle, alors calcule **n-s**.
- **#3919Eh** (Mnémonique **3freeze!**): Il s'agit d'un gel *sévère* de la ligne d'état et de la zone de la pile réunies (plus sévère que ne le serait un **3 FREEZE** normal. En particulier le fait d'entamer une ligne de commande ou de changer de menu ne redessine pas l'écran).
- **#6E97h** (Mnémonique **≈**): Place sur la pile l'objet qui suit dans le Rpl en cours, sans l'évaluer.
- **#61AD8h** (Mnémonique **pifte**): La pile doit contenir un booléen au niveau 1. Soient **Obj1** **Obj2** les deux objets qui suivent ***"pifte"*** dans le Rpl en cours. Si le booléen vaut **True**, alors c'est **Obj1** qui est évalué, sinon c'est **Obj2** (l'autre étant ignoré). Traduire ***"pifte"*** par ***"previous IFTE"***.
- **#2E9E6h** (Mnémonique **HomeSto(_,gn)**): Stocke l'objet du niveau 2 dans le nom global précisé au niveau 1, et dans le répertoire **HOME**.

Au format →EXT:

Les commentaires précédents dessinent la solution que voici. Vous créez cette liste, puis vous lui appliquez le programme **→EXT**.

Le résultat est une structure Rpl, que vous stockez sous le nom **'→ST7'**.

Signalons que la liste a une taille de **311.5 octets** et un CheckSum de **#494Ah**.

Le Rpl obtenu a une taille de **70 octets**, et un CheckSum de **#1F54h**.

```
{  #64BBCh     #53731h
   #64BC6h     #53731h
   #64BD0h     #53731h
   #6E97h
   {  DROP
      #403Fh    #73CEh
         #7221h    #44197h
         #61AD8h
            #15978h    #55DFh
         #7221h    #1685Ch
         #5182Fh
         #12429h
      #7334h
      #3919Eh
   }
   #6E97h    #409DFh
   #2E9E6h
}
```

```
   <3Dh>    SysSf(s)
   <3Eh>    SysSf(s)
   <3Fh>    SysSf(s)
   ≈

   <8h>     start(1,s−1)
   index    pick(s)?
   pifte
   ->StackStr!     ""
   index    s:Str(st,s)
   BeforeStop
   disp(st,s)
   next
   3freeze!

   ≈    ßENTER
   HomeSto(_,gn)
```

Au format EXTERNPC:

Voici le texte à soumettre au programme **EXTERNPC**.

```
begin
   <3Dh>  SysSf(s)  <3Eh>  SysSf(s)  <3Fh>  SysSf(s)
   ≈  begin
      DROP
      <8h>   start(1,s−1)
         index   pick(s)?
         pifte   ->StackStr!   ""
         index   s:Str(st,s)
         BeforeStop   disp(st,s)
      next
      3freeze!
   end
   ≈  ßENTER
   HomeSto(_,gn)
end
```

L'INVERSE DE →EXT

Le programme **→EXT** permet de créer une structure Rpl contenant des appels à des routines non standards de la HP48.

Il suffit pour celà de transmettre une liste à **→EXT**, où ces adresses sont représentées par un entier binaire. L'utilisation de sous-listes dans la liste initiale permet de créer des "sous-structures Rpl". Les exemples précédents vous ont familiarisés avec ce principe.

Nous allons voir maintenant un programme, baptisé **EXT→**, qui permet de défaire ce qu'a fait **→EXT**. Au delà du simple exercice de style, ce projet revêt une utilité certaine:

- Si vous ne disposez plus de la liste qui vous a permis de créer une structure Rpl au moyen de **→EXT**, vous pouvez maintenant la reconstituer. C'est le moyen de plus simple (par une modification de cette liste et une nouvelle utilisation de **→EXT**) de modifier cette structure Rpl.

- Si vous avez chargé sur votre HP48, en mode binaire, une variable dont le contenu est une structure Rpl contenant des *Externals*, vous avez maintenant la possibilité d'en examiner facilement le contenu.

- Le programme **EXT→** vous permet également de comprendre de quoi sont faites les innombrables structures Rpl qui peuplent la ROM de la HP48.

On veut, autant que possible, que les programmes **→EXT** et **EXT→** soient inverses l'un de l'autre. En particulier, il faut que, partant d'une structure Rpl, un appel de **EXT→** (pour la traduire en liste et sous-listes), puis de **→EXT**, permette de revenir à la structure Rpl initiale.

Le programme **EXT→** procède de la manière suivante:

- Il s'assure de la présence d'une structure Rpl au niveau 1 (utilisation des adresses **#18AB2h** (**need1**) et **#18FB2h** (**ChkArg**))

- Il développe cette struture sur la pile (**breaks(cp)** à l'adresse **#54AFh**).

- Si la taille du bloc obtenu est nulle (**<0h>**), on en fait une liste vide.

- Sinon on traite un à un les différents éléments de la structure Rpl étudiée (à suivre).

- Une fois ce traitement individuel terminé, on regroupe ces différents éléments en une liste (**->list(s)**, à l'adresse **#5459h**).

Venons en maintenant au traitement que le programme **EXT→** va appliquer à chacun des éléments de la structure Rpl initiale. Voici les tests effectués successivement et les actions qui en résultent:

- Les objets intégrés (commandes et fonctions standards) ne sont pas modifiés.
- Les réels, entiers binaires, et listes seront reproduits taggués (par un tag vide ::), de manière à assurer la réversibilité des deux programmes **→EXT** et **EXT→**. De la même manière, les entiers-système seront transformés en nombres réels.
- Les objets situés en ROM (adresse < #80000h) sont remplacés par leur adresse sous la forme d'un entier binaire (utilisation du programme **→ADR**).
- Les structures Rpl sont transformées en listes par un appel récursif à **EXT→**.

Au format →EXT:

Vous placez la liste suivante au niveau 1, et vous lui appliquez le programme **→EXT**. Vous obtenez une structure Rpl que vous stockez sous le nom **'EXT→'**.

```
{  #18AB2h
   #18FB2h   #403Fh
   {  #54AFh
      #61891h
      #5459h
      #634CAh
      #73DBh
         #613B6h
         #3325h
         {  #61FA9h    #61A3Bh
            #1A2DAh    #61993h
            →ADR
            #621E7h
            #61993h     EXT→
            #6212Ah
            #61993h     #18DBFh
            #6213Fh
            #32C2h     #62216h
            #3B75h     #32C2h
            #6216Eh    #3B75h
            #61A2Ch
            #55DFh     #5E81h
         }
      #7334h
      #634B6h     #5459h
   }
}
```

```
need1
ChkArg   <8h>
breaks(cp)
dup;=0(s)?:dq/sk
->list(s)
dup;Ldef''
start(1,s)
1Lget
roll(s)
dup;StdObj?     ?:q
dup;inROM?  ?:dq/sk

dup;rpl?
?:dq/sk     EXT->
dup;s?
?:dq/sk      s=>r
dup;b?
over      li?
or(2?)     over
r?     or(2?)
?#:q
""      →tag!(_,st)

next
1Lget;Lpop    ->list(s)
```

N.B: la commande **BYTES** donne les résultats suivants:
Pour la liste précédente: taille 446 octets, CheckSum **#22EBh**.
Pour la structure Rpl qui en résulte: taille 110 octets, **#72D1h**.

Au format EXTERNPC:

Voici le texte à utiliser avec **EXTERNPC**.

```
begin
   need1    ChkArg    <8h>
   begin
      breaks(cp)   dup;=0(s)?:dq/sk    ->list(s)
      dup;Ldef''
      start(1,s)
         1Lget   roll(s)
         begin
            dup;StdObj?   ?:q
            dup;InRom?   ?:dq/sk     ->ADR
            dup;rpl?   ?:dq/sk    EXT->
            dup;s?   ?:dq/sk    s=>r
            dup;b?   over   li?   or(2?)
            over   r?   or(2?)   ?#:q
            ""    ->tag!(_,st)
         end
      next
      1Lget;Lpop   ->list(s)
   end
end
```

Commentaires:
La plupart des adresses utilisées ici ont déjà servi dans les exemples précédents, et il est inutile de les commenter à nouveau. Seuls quelques points méritent d'être signalés:

- L'instruction **ChkArg** (adresse **#18FB2h**) utilise des identificateurs (ce sont des entiers-système) pour désigner des combinaisons de types d'objets sur la pile. La présence d'une structure Rpl au niveau 1 est ainsi identifiée par <8h>.
- L'instruction **dup;inROM?** (adresse **#1A2DAh**) donne un booléen permettant de savoir si l'objet du niveau 1 a son adresse en ROM, entre les adresses #0h et #7FFFFh (voir tome 1, page 350).
- L'instruction **?:dq/sk** (adresse **#61993h**): si le booléen vaut *True*, on évalue l'objet qui suit dans le Rpl en cours, en même temps qu'on sort de ce Rpl. Sinon on "saute" cet objet et on continue après (traduire "dq" par "*do and quit*" et "sk" par "*skip*").
- L'instruction **→tag!(_,st)** (adresse **#5E81h**): elle permet de tagguer l'objet du niveau 2 avec la chaîne placée au niveau 1 (voir tome 1, page 234).

Une application de EXT→ :

Le programme **EXT→** prend comme argument une structure Rpl. Mais à ce stade il ne fait aucune différence entre les programmes (ceux auxquels la commande **TYPE** attribue la valeur 8) et les fonctions ou commandes intégrées (TYPE ⇒ 18 ou 19).

On peut donc appliquer le programme **EXT→** à un objet intégré à la HP48. On obtient alors la liste des adresses utilisées dans cet objet.

Exemple: on place la commande **DUP** au niveau 1 et on appelle **EXT→**.

On obtient la liste **{ #18AA5h #3188h }** ce qui prouve que de façon interne, la commande **DUP** est formée par un appel à l'adresse **#18AA5h** (mnémonique **LibNeed1**: vérifie la présence d'un objet sur la pile, "*marque*" la pile, et retient le nom de la commande exécutée), puis à l'adresse **#3188h** (mnémonique **dup**: c'est la forme interne de **DUP**)

Une amélioration de EXT→ :

On peut demander à **EXT→** de nous fournir le contenu d'une séquence Rpl, cette dernière étant soit placée sur la pile (c'est la version précédente de **EXT→**) soit simplement donnée par son adresse (sous la forme d'un entier binaire).

Si on part d'un entier binaire **#b** au niveau 1, on exécute les instructions suivantes pour placer au niveau 1 le contenu de l'objet situé à l'adresse **#b** (sans l'évaluer):

- **#4003h SYSEVAL** (pour placer l'entier-système <2h> sur la pile).
- **#62B9Ch SYSEVAL** (instruction **get(cp,s)**, pour rappeler le contenu de l'objet sur la pile).

Il suffit donc d'ajouter la liste suivante à celle qui nous a permis de créer la première version de **EXT→** (où **#405Dh** est l'adresse de l'entier-système **<Bh>**, qui permet à l'instruction **ChkArg** d'identifier un entier-binaire):

{ #405Dh { #4003h #62B9Ch EXT→ } }

Ainsi modifiée la liste servant à créer **EXT→** a une taille de 497.5 octets et un CheckSum de **#61F9h**, tandis que le Rpl créé par **→EXT** (et qui devient la nouvelle version de **EXT→**) a une taille de 130 octets et un CheckSum de **#AB3h**.

Remarque: Si vous donnez une adresse qui n'est pas celle d'une structure Rpl, vous allez vers une erreur "*Bad Argument Type*".

Exemple: avec cette version, la séquence **#1E3C2h EXT→** donne la liste:

{ #18EDFh #64C66h #4F539h #4143h #4F598h }

#1E3C2h est l'adresse de l'instruction intégrée **TLINE** (#64C66h est celle de <55h> et #4143h celle de <22h>). On voit ici comment l'adresse **#18EDFh** permet de tester les deux arguments requis et d'aiguiller la suite du programme sur telle ou telle adresse (**#4F598h**, par exemple, dans le cas de deux arguments complexes).

Une modification de EXT→ :

Nous considérons ici la version améliorée de **EXT→**.

Sur l'exemple précédent, on voit que le programme **EXT→** ne "rentre" pas dans les sous-structures Rpl basées en ROM. En effet, l'adresse **#4F598h** ci-dessus (**TLINE** appliquée à deux nombres complexes) désigne une structure Rpl, et celle-ci n'est pas "décortiquée" par le programme **EXT→**.

La raison en est dans la façon dont a été conçu **EXT→**.

Si vous observez attentivement la liste qui nous a permis de le créer, vous constatez que la séquence [**#1A2DAh ..... →ADR**], dont le rôle est de traiter les objets situés en ROM (à l'exception des objets intégrés) vient **avant** la séquence [**#621E7h #61993h EXT→**] dont le rôle est d'appliquer récursivement **EXT→** aux sous-structures Rpl de la structure initiale.

Les sous-structures Rpl basées en ROM ne sont donc pas "scannées".

La solution, si on ne se satisfait pas de ce comportement, consiste bien entendu à intervertir les deux séquences ci-dessus dans la liste servant à créer **EXT→**.

Avec cette nouvelle liste, le programme **→EXT** crée une nouvelle structure Rpl. On obtient ainsi une deuxième version de **EXT→** (si on veut modifier ce nom, pour cette nouvelle version, et utiliser par exemple EXT2→, alors il faut remplacer EXT→ par EXT2→ dans la liste de départ!).

Le problème est que les instructions standards de la HP48 sont programmées de manière modulaire et utilisent souvent un très grand nombre de sous-programmes. A tel point que la séquence **#1E3C2h EXT→** utilisée précédemment échoue sur une HP48 ayant encore 27K de mémoire vive, sur une erreur "*Out Of Memory*" !!!!.

Il n'en va pas toujours de même heureusement (mais ce problème fait tout de même pencher la balance en faveur de la version précédente de **EXT→**).

Exemple: Si on place la commande **STOF** (ou son adresse **#1C67Fh**) au niveau 1, l'exécution de **EXT→** donne le résultat suivant:

```
Avec la première version de EXT→ :
{  #18ECEh  #4021h  #1C6A2h  #405Dh  #1C6E3h  }
```

```
Avec la deuxième version de EXT→ :
{  #18ECEh
   #4021h  { #54AFh  { #4003h  #3D4Eh  }
           #61993h   #18CA2h   #18F9Dh   #64DCEh
           { #1C6F7h
             { #1C731h  { #64B80h  #53784h  #61A2Ch  #112ECh  } }
           }
         }
   #405Dh   { #1C731h  { #64B80h  #53784h  #61A2Ch  #112ECh } }
}
```

UN VECTEUR D'OBJETS QUELCONQUES

On sait que la façon dont sont codés les tableaux en mémoire (voir tome 1, page 51) permet de concevoir des tableaux formés d'objets de type quelconque (chaînes, expressions, etc...), et ayant plus de 2 dimensions. La seule obligation est que tous les objets qui y figurent aient le même type.

Malheureusement, la ROM de la HP48 ne contient pratiquement aucun sous-programme permettant de créer, de modifier, ou d'utiliser de tels tableaux, ce qui enlève un peu d'intérêt à la création de tels objets.

Néanmoins, la HP48 utilise des vecteurs de chaînes de caractères pour stocker les messages de ses librairies (cf tome 1, page 53 et suivantes).

Dans ce paragraphe, nous allons voir une méthode pour créer des vecteurs quelconques. Mis à part le plaisir intellectuel d'arriver à de telles constructions, on peut trouver à ce projet une utilité dans la création de la table des messages d'une librairie (voir tome 1, page 114).

Il n'est pas beaucoup plus difficile de créer des tableaux quelconques ayant plusieurs dimemsions, mais celà ne présente guère d'intérêt.

Le programme développé ici s'appelle '**→VEC**'. Son schéma fonctionnel est:

1:Liste ---[**→VEC**]---▸ **1:Vecteur**

La liste initiale doit être non vide et formée d'éléments ayant tous le même type. Ces éléments sont alors combinés (en conservant l'ordre dans lequel ils apparaissent), pour former le vecteur final.

En gros, la logique du programme est la suivante:

- Vérifier que le niveau 1 contient bien une liste non vide (les exemples précédents nous ont habitués à ce genre de contrôle).
  On utilise les instructions **need1** et **ChkArg** (adresses **#18AB2h** et **#18FB2h**).
- S'assurer que tous les objets de la liste sont bien du même type. C'est l'affaire de la première des deux boucles **start...next** dans le listing ci-dessous.
  L'outil essentiel est l'adresse **#1CB90h** (Mnémonique **type!**) qui représente la forme interne de l'instruction **TYPE**. La seule différence est que **type!** laisse son argument au niveau 2 (ce qui oblige à le détruire par la suite).
  La raison pour laquelle on n'utilise pas l'instruction **TYPE** est qu'elle "*marque*" la pile, ce marquage se substituant à celui opéré par l'instruction **need1** initiale.
- Former un entier binaire (au départ vide) pour y stocker les différentes composantes du vecteur (par des concaténations successives).
- Transformer l'entier binaire obtenu en un tableau, par une simple modification de son adresse-préfixe.

Le programme →VEC

Au format →EXT:

Vous placez la liste suivante au niveau 1, et vous lui appliquez le programme →EXT. Vous obtenez une structure Rpl que vous stockez sous le nom '**→VEC**'.

La commande **BYTES** donne les résultats suivants:

- Pour cette liste: Taille **764 octets**, CheckSum **#4DD0h**.
- Pour la structure Rpl: Taille **165 octets**, CheckSum **#CDFAh**.

N.B: 10830 est transformé par **→EXT** en **<2A4Eh>** (adresse préfixe des entiers binaires) et 10728 en **<29E8h>** (adresse-préfixe des tableaux).

```
{  #18AB2h     #18FB2h
   #4021h
   {  #3188h     #55B7h
      #63B05h     #54AFh
      #5E35Ch     #32C2h
      #1CB90h     #6284Bh
      #73DBh
         #3223h     #1CB90h
         #60F9Bh     #32C2h
         #2A8CCh     #63B05h
      #7334h
      #627A7h     #5E401h
      #3FF9h     #5E401h
      #55D5h
      #6E97h
      {  #6657h     #3188h
         #5902h     #6258Ah
         #6657h     #3223h
         #631B9h     #5089h
         10830     #5ACCh
      }
      #634CFh     #611FEh
      #613B6h     #6F8Eh
      #3FF9h     #4021h
      #5815h     #518Ah
      #3223h     #73DBh
         #5182Fh     #612F3h
         #613B6h     #6F8Eh
         #402Bh     #32C2h
         #5616h     #5815h
         #518Ah
      #7334h
      #7497h
      10728     #5ACCh
   }
}
```

```
begin     need1     ChkArg
<5h>
begin     dup     ={}?
?:BAV!     breaks(cp)
dupB     over
type!     3rolld;drop
start(1,s)
   swap     type!
   drop(2)     over
   <>(2r)?     ?:BAV!
next
drop;dup     pushB[n]
<1h>     pushB[n]
Null#
≈
begin     newob     dup
   nibbles(_)     5+(s)
   newob     swap
   2->list     get1(cp)
   <2A4Eh>     ModType(_,s)!
end
Ldef''     3pick
1Lget     eval
<1h>     <5h>
sub(b,2s)     conc(2b)
swap     start(1,s)
   BeforeStop     roll(s+1)
   1Lget     eval
   <6h>     over
   quart(b)     sub(b,2s)
   conc(2b)
next
Lpop
<29E8h>     ModType(_,s)!
end
end
```

Au format EXTERNPC:

Voici le texte à utiliser avec **EXTERNPC**.

```
begin
   need1   ChkArg    <5h>
   begin
      dup   ={}?   ?:BAV!
      breaks(cp)   dupB   over   type!   3rolld;drop
      start(1,s)
         swap   type!   drop(2)   over   <>(2r)?   ?:BAV!
      next
      drop;dup   pushB[n]   <1h>   pushB[n]   Null#
      ≈
      begin
         newob   dup   nibbles(_)   5+(s)   newob   swap
         2->list      get1(cp)   <2A4Eh>   ModType(_,s)!
      end
      Ldef''   3pick   1Lget   eval
      <1h>   <5h>   sub(b,2s)   conc(2b)
      swap   start(1,s)
         BeforeStop   1Lget   eval
         <6h>   over   quart(b)   sub(b,2s)   conc(2b)
      next
      Lpop   <29E8h>   ModType(_,s)!
   end
end
```

Commentaires:

Les listings précédents font apparaître un certain nombre d'adresses que nous n'avons pas encore utilisées. C'est le cas en particulier pour deux adresses agissant sur des "***méta-objets***" (ou encore "***blocs***": voir tome 1, pages 208 et suivantes).

Rappelons qu'un bloc est une succession de **n** objets (**n** entier positif ou nul) **objn,...,obj2,obj1**, suivis de la longueur **<n>** de ce bloc (sous-forme d'entier-système), ces différents objets occupant des positions successives sur la pile (la longueur **<n>** étant placée au niveau le plus bas du bloc).

L'instruction **breaks(cp)** (adresse **#54AFh**) qui libère sur la pile un objet composé (voir tome 1, pages 181 et suivantes) est le moyen le plus couramment employé pour créer un "bloc".

Voici donc les deux instructions employées ici:

- **#5E3Ch** (mnémonique **dupB**): duplique le bloc situé au niveau 1.
- **#5E401h** (mnémonique **pushB[n]**): ajoute l'objet du niveau 1 au bloc situé au niveau 2, cet objet prenant la dernière position dans ce bloc.

Le programme →VEC

Il faut maintenant, pour comprendre la deuxième partie du programme (notamment la deuxième boucle **start...next**), revenir sur la façon dont un vecteur de **n** objets doit être codé en mémoire.

Symboliquement, la forme de ce codage est:

[8E920][Taille][Type][10000][n][....éléments....]

où les 5 premiers champs occupent chacun 5 quartets.

- Le champ **[n]** contient par exemple **90000** si on crée un vecteur de **9** termes.
- Le champ **[10000]** signifie qu'il y a un seul indice dans le tableau.
- Le champ **[Type]** contient l'adresse-préfixe des éléments du tableau (par exemple **[C2A20]** s'il s'agit d'un vecteur de chaînes de caractères).
- Le champ **[Taille]** contient la taille totale de l'objet, à l'exception des 5 quartets occupés par le champ **[8E920]** représentant l'adresse-préfixe des tableaux.
- Le champ **[...éléments...]** contient le codage des différents éléments du tableau, à l'exception de leur adresse-préfixe (devenue inutile car présente une fois pour toutes dans le champ **[Type]**).

La méthode employée dans le programme **→VEC** consiste à former, par des concaténations successives, l'entier binaire:

[E4A20][Taille][Type][10000][n][....éléments....],

puis à effectuer un changement d'adresse-préfixe pour obtenir le tableau voulu (adresse **#5ACCh**, mnémonique **ModType(_,s)!**).

- On part pour celà de l'entier binaire vide (adresse **#55D5h**, mnémonique **Null#**). Les concaténations successives actualisent à chaque étape le contenu du champ **[Taille]**.
- On doit "copier" chaque élément du tableau sur un entier binaire. C'est le rôle du sous-programme suivant:

```
begin
      newob    dup    nibbles(_)    5+(s)    newob    swap
      2->list    get1(cp)    <2A4Eh>    ModType(_,s)!
end
```

- Ce sous-programme est placé dans une variable locale (instruction **Ldef''**, adresse **#634CFh**), car il doit être utilisé de deux manières différentes. Pour l'évaluer, il suffit de le rappeler par **1Lget** (adresse **#613B6h**) et d'exécuter **eval** (adresse **#6F8Eh**).

- On copie l'adresse-préfixe d'un des éléments de la liste initiale dans un entier binaire de longueur 5, avant de le concaténer avec l'entier binaire en cours. C'est l'affaire des instructions suivantes:

  3pick 1Lget eval <1h> <5h> sub(b,2s) conc(2b)

- Pour prévoir la formation des champs **[10000]** et **[n]** (précédant le codage des éléments du tableau), on ajoute symboliquement les entiers-système **<1h>** et **<n>** en tête de la liste initiale.

  C'est le rôle des instructions:

  drop;dup pushB[n] <1h> pushB[n]

- On copie dans un entier binaire chacun des éléments devant ensuite figurer dans le tableau, à l'exception de leur adresse-préfixe. C'est le rôle de la deuxième boucle **start...next.**

  On voit dans cette boucle comment l'adresse-préfixe est éliminée: l'entier binaire représentant un élément du tableau étant formé, on n'en garde que les quartets à partir de la position <6h>. Tel est le rôle des instructions:

  <6h> over quart(b) sub(b,2s)

Voilà pour les commentaires sur le fonctionnement du programme **→VEC**. Si certaines adresses qui y figurent n'ont pas été décrites dans ces lignes, c'est qu'elles apparaissent dans au moins un des programmes précédents de ce chapitre.

Exemples:

Le programme **→VEC** permet de créer des vecteurs de nombres réels ou complexes, bien que ça ne soit pas la meilleure méthode pour celà (mais c'est une façon de voir qu'il fonctionne correctement).

```
{ 1  5.3  2  -4.8 }  -[→VEC]-▸  [ 1  5.5  2  -4.8 ]
```

Il est plus intéressant de demander à **→VEC** de former des tableaux plus "exotiques" (des vecteurs de chaînes, d'entiers-système, etc....):

```
{ "ABC"  "DEFG"  "H" }  -[→VEC]-▸  Array of String

{ <3h> <2h> <5h> <0h> }  -[→VEC]-▸  Array of System Binary

{ 'ABC'  'DEFG'  'H' }  -[→VEC]-▸  Array of Global Name
```

Remarque:

Il est inutile (et risqué) de vouloir éditer ces tableaux en ligne de commande! Si vous effectuez **OBJ→** sur ces vecteurs, vous verrez que la HP48, désorientée, les considère comme des tableaux de nombres complexes... (voir le tome 1, pages 53 et 54).

L'OPÉRATION INVERSE DE →VEC

On se propose ici de créer un programme, baptisé **VEC→**, dont le rôle est de transformer un vecteur d'objets quelconques (tel que créé par le programme **→VEC**) en la liste des éléments de ce tableau.

Les deux programmes **→VEC** et **VEC→** sont donc inverses l'un de l'autre, et le schéma fonctionnel de **VEC→** est:

1:Vecteur ---[VEC→]---▸ 1:Liste

Ce programme présente un peu moins de difficulté que le précédent, notamment par le fait que la structure de vecteur est, au départ, plus *sûre* que la structure de liste, et demande moins de contrôles.

Comme d'habitude, on vérifie d'abord que le niveau 1 de la pile contient bien un tableau. Il faut vérifier en outre que ce tableau n'est pas une matrice. Celà est réalisé par l'adresse suivante:

- **#357A8h** (mnémonique **matrix(ar)?**): si le tableau du niveau 1 est une matrice, alors répond **3:<L> 2:<C> 1:True** (où **<C>** et **<L>** sont deux entiers-système indiquant le nombre de colonnes et de lignes). Si ce tableau est un vecteur, la réponse est **2:<N> 1:False**, où **<N>** est un entier-système précisant le nombre de termes du vecteur.

A la suite de ce test, il faut prévoir une erreur "*Bad Argument Value*" si le tableau est une matrice. Ceci est réalisé par l'adresse **#63B05h** (mnémonique **?:BAV!**):

Il faut alors en venir aux choses sérieuses. Sans entrer, comme pour l'exemple précédent, dans des explications trop détaillées, l'algorithme utilise encore des manipulations d'entiers binaires (le tableau initial lui-même est d'ailleurs rapidement transformé en un entier binaire).

Une petite difficulté réside dans le fait qu'il faut associer l'unique adresse-préfixe apparaissant dans le corps du tableau (et donnant le type des éléments), avec chacun des élements de ce vecteur.

Il n'est pas non plus très commode de "lire" individuellement les différents éléments du tableau, dont on ne connaît pas à priori les dimensions (variables dans le cas d'un vecteur de chaînes, par exemple, mais qui sont fixes dans le cas d'un vecteur d'entiers-système).

Mais trêve d'explications. Voici les deux listings, comme d'habitude, qui permettent de créer le programme **VEC→**.

Au format →EXT:

Vous placez la liste suivante au niveau 1, et vous lui appliquez le programme **→EXT**. Vous obtenez une structure Rpl que vous stockez sous le nom **'VEC→'**.

La commande **BYTES** donne les résultats suivants:

- Pour cette liste: Taille **462.5 octets**, CheckSum **#FFFEh**.
- Pour la structure Rpl: Taille **100 octets**, CheckSum **#7EB0h**.

```
{ #18AB2h     #18FB2h
  #4017h<4h>
  {  #3188h     #357A8h
     #63B05h
     #3223h     10830
     #5AB3h     #63A9Ch
     #4021h     #5815h
     #3223h     #408Fh
     #32C2h     #5616h
     #5815h     #611FEh
     #73DBh
        #31ACh     #518Ah
        #4003h     #62B9Ch
        #6657h
        #61380h     #5902h
        #6260Ah     #32C2h
        #5616h     #5815h
        #3223h     #6109Eh
     #7334h
     #3258h     #5459h
  }
}
```

```
begin   need1   ChkArg

begin    dup    matrix(ar)?
   ?:BAV!
   swap    <2A4Eh>
   ModType(_,s)    dup;<1h>
   <5h>    sub(b,2s)
   swap    <10h>
   over    quart(b)
   sub(b,2s)    3pick
   start(1,s)
      dup2    conc(2b)
      <2h>    get(cp,s)
      newob
      swap;over   nibbles(_)
      4-(s)    over
      quart(b)    sub(b,2s)
      swap    4rolld
   next
   drop2    ->list(s)
end
end
```

Au format EXTERNPC:

Voici le texte à utiliser avec **EXTERNPC**.

```
begin
  need1   ChkArg   <4h>
  begin
     dup   matrix(ar)?   ?:BAV!
     swap   <2A4Eh>    ModType(_,s)   dup;<1h>
     <5h>   sub(b,2s)   swap   <10h>   over
     quart(b)   sub(b,2s)   3pick
     start(1,s)
        dup2   conc(2b)   <2h>   get(cp,s)   newob
        swap;over   nibbles(_)   4-(s)   over
        quart(b)   sub(b,2s)   swap   4rolld
     next
     drop2    ->list(s)
  end
end
```

Exemple:

Les deux programmes **→VEC** et **VEC→** sont bien réciproques l'un de l'autre, comme le montre l'exemple suivant.

```
{ "ABC"  "DEFG"  "H" }  --[→VEC]-->  Array Of String
                        --[VEC→]-->  { "ABC"  "DEFG"  "H" }
```

N.B: le temps d'exécution de **→VEC** et de **VEC→**, sur l'exemple précédent, est inférieur à 1/10ème de seconde...

PERSONNALISER LES MESSAGES DE LA HP48

Nous allons maintenant utiliser les programmes →**VEC** et **VEC**→ pour "*customiser*" les messages de la calculatrice. Plus précisément, il s'agit de modifier les vecteurs de chaînes de caractères dans lesquels sont stockés les messages des différentes librairies intégrées (voir tome 1, pages 55 et suivantes).

Ces vecteurs sont situés en ROM. Les modifier "*in situ*" n'est donc pas possible. La seule possibilité est de les recopier en mémoire vive, de les y modifier (en partie ou en totalité), puis de faire en sorte que la HP48 utilise ces nouveaux vecteurs plutôt que les anciens.

L'adresse essentielle pour assurer cette "déviation" est:

- **#764Eh** (Mnémonique **SetMessLib(s,ar)**): Son rôle est de faire du tableau **ar** (qui doit être un vecteur de chaînes) la nouvelle table de messages pour la librairie n°**s**. Pour que cette modification soit opérationnelle, il faut que la librairie soit attachée. Si elle est ensuite détachée, elle revient à sa table de message initiale.

LIRE LA TABLE DES MESSAGES D'UNE LIBRAIRIE:

Dans un premier temps, voyons comment prendre connaissance, d'une manière lisible, de la table des messages d'une librairie donnée.

Le programme **VMS**→ fonctionne de la manière suivante:

1:n ---[**VMS**→]---▸ **1:Liste**

où **n** est un réel (entier en principe, mais qui sera arrondi à l'entier positif ou nul le plus proche), et où "**Liste**" est la liste des messages de la librairie de numéro **n**.

Le programme **VMS**→ s'interromp sur une erreur "*Bad Argument Value*" si la librairie de numéro **n** n'existe pas, ou s'il ne lui correspond aucune table de messages.

Pour le programme **VMS**→, il n'est pas nécessaire que la librairie dont on recherche le vecteur des messages soit "attachée". Il suffit qu'elle soit installée dans l'un des ports la calculatrice ayant été éteinte, puis rallumée.

Le programme **VMS**→ utilise deux adresses ayant un rapport avec les librairies:

- **#8E15h** (Mnémonique **RclKnownLibs**): donne la liste des numéros (sous forme d'entiers-système) de toutes les librairies connues (tome 1, page 334
- **#8130h** (Mnémonique **MessLib(s)?**): examine si la librairie n°**s** possède une table de messages. Dans la négative la réponse est **1:False**. Sinon la réponse est **2:AccessObject 1:True**, où **AccessObject** est un objet de type **EXT2**, permettant d'accéder au tableau, quelque soit le port dans lequel il se trouve (voir tome 1, page 331)

- L'adresse **#71646h** (mnémonique: **eval(ap)** permet d'évaluer l'objet référencé par l' "Access-Pointer" placé au niveau 1 (tome 1, page 343). Ici le tableau sera posé sur la pile.

- L'adresse **#643EFh** (Mnémonique **InCp(_,cp)?**): teste si l'objet du niveau 2 fait partie de l'objet composé du niveau 1. Réponse par True ou False (tome 1, page 183).

Voici la liste sur laquelle appliquer le programme **→EXT** (et qui conduit à une structure Rpl que vous stockerez sous le nom **VMS→**). En parralèle, dans la partie droite du tableau, vous trouvez le texte à utiliser avec le programme **EXTERNPC** (et qui sert également de commentaire à la liste):

```
{#18AB2h    #18FB2h    #3FF9h
 {
   #62CE1h    #8E15h
   #643EFh    #63B19h
   #8130h    #63B19h
   #71646h    VEC->
 }
}
```

```
begin   need1   ChkArg   <1h>
   begin
     r=>s;dup    RclKnownLibs
     InCp(_,cp)?    ?#:BAV!
     MessLib(s)?    ?:#BAV!
     eval(pa)    VEC->
   end
end
```

N.B: la commande **BYTES**, donne les résultats suivants:
Pour la liste: Taille 147.5 octets, CheckSum #B2D4h
Pour la structure Rpl: Taille 42.5 octets, CheckSum #EB06h

Rappels:

Dans le tome 1, on trouve aux pages 55 à 58, le contenu des vecteurs de messages des différentes librairies intégrées.

Rappelons qu'une librairie de numéro **n**, peut posséder de 0 à 255 messages, et que ceux-ci portent des numéros successifs, à partir du numéro **256*n+1**.

Ainsi la librairie n°10 (10=#Ah) possède 9 messages, dont les numéros varient de #A01h à #A09h.

Exemple:

La séquence **10 VMS→** place la liste suivante sur la pile:
{ "Bad Guess(es)" "Constant?" "Interrupted" "Zero" "Sign Reversal" "Extremum" "Left" "Right" "Expr" }

MODIFIER LA TABLE DES MESSAGES D'UNE LIBRAIRIE:

Nous allons maintenant considérer le programme **→VMS** qui attend un nom au niveau 2, et un réel **n** (entier) au niveau 1. Le nom doit être celui d'une variable **contenant un vecteur de chaînes** (tel que formé par **→VEC**). **→VMS** fait des chaînes de caractères de ce vecteur les nouveaux messages de la librairie de numéro **n**.

2:Nom 1:n ---[→VMS]---▸ 1:....

Le programme →VMS

Le fait qu'il faille que le vecteur de chaînes de caractères soit le contenu d'une variable est important. Celà assure en effet (tant que vous ne purgez pas cette variable, tant que vous ne la déplacez pas, tant que vous ne modifiez pas l'ordonnance des autres variables) que le vecteur reste à une adresse fixe en mémoire.

Sans celà la librairie dont vous venez de modifier les messages ne saurait plus où les trouver (le risque de *Memory Clear* semble cependant très faible).

Le mieux serait en fait que le vecteur de chaînes que vous utilisez soit à une adresse fixe en ROM (sur une carte d'extension protégée en écriture). A défaut, placez de préférence cette variable en tête dans le répertoire **HOME**.

Voici, de part et d'autre du tableau suivant, la liste à utiliser avec **→EXT**, et le texte à utiliser avec **EXTERNPC**. Le programme **→EXT**, appliqué à la liste, forme une structure Rpl que vous stockez sous le nom **'→VMS'**.

A titre indicatif, la commande **BYTES** donne:

Pour la liste: Taille 205 octets, CheckSum #9CD0h.

Pour la structure Rpl: Taille 47.5 octets, CheckSum #8648h.

```
{#18A8Dh   #18FB2h
 #64C98h
 {  #62CE1h
    #8E15h   #643EFh
    #63B19h   #61380h
    #4003h   #620EBh
    #61AD8h
    #3244h   #7709h
    #20B81h
    #764Eh
 }
}
```

```
begin   need2   ChkArg
    <61h>
    begin    r=>s;dup
       RclKnownLibs   InCp(_,cp)?
       ?#:BAV!    swap;over
       <2h>    over;=(2s)?
       pifte
       drop   HomeAttach(s)!
       rcl(nm)
       SetMessLib(s,ar)
    end
end
```

Remarque:

On n'est pas tenu de remplacer le vecteur des messages d'une librairie par un vecteur de même longueur. Utiliser un vecteur plus long, dans le cas d'une librairie intégrée, n'est pas très utile dans la mesure où les codes d'erreurs choisis par la HP48 restent les mêmes. Si on utilise un vecteur plus court, les messages devenus manquants sont considérés comme des messages vides.

Exemple: On peut remplacer la table des 8 messages de la librairie intégrée n°2, par la table constituée des 4 chaînes suivantes:

{ "Tu manques d'arguments" "Quelle salade !"
"Revois ta copie, mec" "Ce nom m'est inconnu" }

On applique le programme **→VEC** à cette liste, de manière à former un "*Array Of String*". On place ensuite ce vecteur dans une variable, nommée par exemple **'TAB'**. On exécute enfin la séquence **'TAB' 2 →VMS**.

Il suffit alors de provoquer une erreur comme **CLEAR DUP**, pour lire le message:

DUP Error:
Tu manques d'arguments

MODIFIER UN MESSAGE EN PARTICULIER:

Il peut être préférable de modifier un ou plusieurs des messages d'une librairie.

Il y a pour celà une première possibilité consistant à exhiber la liste de tous les messages de cette librairie (au moyen du programme **VMS→**), puis à modifier une ou plusieurs des chaînes de cette liste. La liste modifiée est alors transformée en "*Array Of String*" par le programme **→VEC**. Il reste à stocker ce tableau dans une variable, avant d'utiliser le programme **→VMS**.

Une deuxième solution consiste à utiliser le programme **→MES** dont voici le schéma fonctionnel:

2:String 1:k --[→MES]--▸ 1:....

Ici, *String* est une chaîne de caractères représentant le nouveau message, et **k** est un réel (entier) donnant le numéro de ce message. L'entier **(k mod 256)** représente le numéro de message au sein de la librairie, et ne doit pas être supérieur au nombre de messages de celle-ci.

Le message de numéro **k** appartient, on le sait, à la librairie de numéro **n = (k div 256)**. Le programme **→MES** extrait la liste des messages de cette librairie. Il y effectue le remplacement voulu, convertit cette liste en un "*Array Of String*", puis le stocke dans une variable nommée **'TABn'** (exemple 'TAB0', 'TAB1',...), dans le répertoire **HOME**, avant d'appeler le programme **→VMS**.

Voici donc le programme **→MES**, présenté de la manière habituelle. Le programme **→EXT** transforme la liste en une struture Rpl que vous stockez ensuite sous le nom global **'→MES'**.

A titre indicatif, la commande **BYTES** donne les résultats suivants:

Pour la liste: Taille 313.5 octets, CheckSum #EDF3h.

Pour la structure Rpl: 93 octets, CheckSum #84BEh.

```
{
  #18A8Dh    #18FB2h
  #64B44h
  {
     #18CEAh    #4DD7h
     #18DBFh     #3188h
     VMS→     #3223h
     #6109Eh     #611F9h
     #56B6h     #61AD8h
     #3244h     #18CA2h
     #1DC00h
     →VEC    "TAB"
     #611FEh    #35287h
     #5193h    #5B15h
     #61099h
     #2E9E6h    →VMS
  }
}
```

```
begin
need2    ChkArg
<31h>
begin
   r+=>s      mdv(s,256)
   s=>r      dup
   VMS->       swap
   4rolld     dup;3pick
   get(cp,s)?     pifte
   drop     doerr203
   put(_,s,li)
   ->VEC      "TAB"
   3pick      ->StdStr(r)
   +(2st)!     st=>gn
   dup;4rolld
   HomeSto(_,gn)     ->VMS
end
end
```

Commentaires:

Le listing précédent fait apparaître un certain nombre d'adresses non évidentes, qui n'ont pas encore été utilisées:

- **#4DD7h** (Mnémonique **mdv(s,256)**): part d'un entier-système **s** et donne le résultat **2:(s mod 256) 1:(s div 256)**. Voir tome 1, page 159.

- **#56B6h** (Mnémonique **get(cp,s)?**): tente d'extraire l'objet en position **s** dans l'objet-composé du niveau 2. La réponse est **2:Obj 1:True** si c'est possible, et **1:False** sinon (c'est un bon moyen de tester la validité de l'entier **s**).

- **#1DC00h** (Mnémonique **put(_,s,li)**): place l'objet du niveau 3 dans la liste du niveau 1, en position définie par l'entier-système **s**. Ne teste pas la.validité de **s** (cf tome 1, page 182).

- **#35287h** (Mnémonique **−>StdStr(r)**): transforme un réel en chaîne de caractères, comme si on était en mode standard, mais sans modifier le mode d'affichage des réels (cf tome 1, page 201).

Remarque:

Souvenez-vous que la redirection de la table des messages vers un vecteur de chaînes stocké en RAM cesse de fonctionner correctement dès lors que la position de ce vecteur est modifiée (ne serait-ce que lorsque vous créez ou détruisez des variables n'ayant aucun rapport avec celle-ci).

Il est recommandé de placer en début de **HOME** les variables contenant des vecteurs de chaînes servant à rediriger des messages. De cette manière, la position de ces variables est à peu près stable (sauf si vous modifiez le port 0).

Exemple:

Le mot **Error:** qui figure sur la première ligne de tous les messages d'erreurs est en fait le message n°#133h=#307d.

On peut modifier ce mot en exécutant, par exemple:

"Incident:" 307 →MES

Cette séquence a pour effet de placer, en début du répertoire **HOME**, la variable **'TAB1'** contenant le vecteur des messages de la librairie n°1.

Dans ces conditions, une erreur telle que l'appui sur la touche + alors qu'aucun objet n'est sur la pile, provoquera le message d'erreur:

+ Incident:
Too Few Arguments

REVENIR AUX MESSAGES STANDARDS:

Une méthode très simple permettant de revenir aux messages standards de la HP48 est d'effectuer un appui simultané sur les touches **ON** et **C**, pour provoquer un *Reset*. Celà a cependant le désavantage, entre autres, de vous faire perdre le contenu de la pile et celui de **PICT**.

Vous pouvez tout autant utiliser le programme baptisé **STDM**, et créé à partir du listing suivant, comme d'habitude:

Le programme **→EXT** transforme la liste ci-dessous en une structure Rpl que vous pouvez stocker sous le nom **'STDM'**.

La commande **BYTES** donne les résultats suivants:
Pour la liste: Taille 174 octets, CheckSum #BB34h
Pour la structure Rpl: Taille 37.5 octets, CheckSum #CBDBh

```
{ #8E15h                          begin    RclKnownLibs
  #63498h                            dup;start(1,szcp)
     #3188h     #7221h                 dup    index
     #62D1Dh                           get(cp,s);dup
     #76AEh                            HomeDetach(s)
     #4003h     #620EBh                <2h>    over;=(2s)?
     #61AD8h                           pifte
     #7709h     #3244h                 HomeAttach(s)!   drop
  #7334h                             next
  #3244h                             drop
}                                 end
```

Le principe du programme **STDM** est de détacher du répertoire **HOME** toutes les librairies connues (qu'elles soient attachés ou pas, d'ailleurs), en prenant soin tout de même de réattacher immédiatement la librairie intégrée de numéro **2**, qui est essentielle au fonctionnement de la HP48.

Le programme précédent utilise une adresse qui n'a pas encore été utilisée dans un des exemples de ce chapitre:

- **#63498h** (Mnémonique **dup;start(1,szcp)**): Part d'un objet-composé présent au niveau 1, le duplique, puis démarre une boucle du type **start...next**, en prenant **<1h>** pour indice initial et la taille de l'objet-composé pour indice final ("**szcp**" est l'abréviation de *size cp*, **cp** étant lui-même l'abréviation souvent utilisée dans ce livre pour désigner un objet-composé).

Ceci termine les variations consacrées à la personnalisation des messages d'erreurs de la HP48. Les plus courageux pourront donc les franciser entièrement. Les plus farceurs trouveront des tournures plus amusantes que les messages traditionnels...

REMPLACEMENTS DANS UN OBJET COMPOSÉ

Rappelons que les objets composés désignent dans ce livre les objets de l'un des types suivants: Structure Rpl, Liste, Expression Algébrique, ou Objet-Unité (voir le tome 1, pages 181 et suivantes).

Le programme **REPLA**, présenté ici, permet de procéder à des remplacements multiples dans un objet composé. Il y a deux schémas fonctionnels possibles, suivant le nombre de remplacements que l'on désire effectuer.

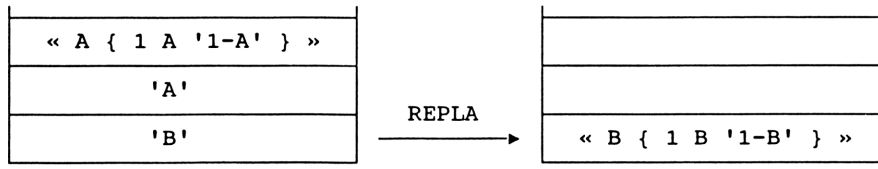
Dans un premier temps, le programme **'REPLA'** permet de remplacer, dans un objet-composé **Obj3**, toutes les occurrences d'un objet **Obj2** par autant d'exemplaires d'un objet **Obj1**, le résultat de ces remplacements étant un objet **Obj4**.

En reprenant ces notations, le schéma fonctionnel est:

3:Obj3 2:Obj2 1:Obj1 ---[REPLA]---▸ 1:Obj4

Attention: l'objet composé **Obj3** est formé d'éléments divers. Il est très important de noter que si ces derniers sont eux-mêmes des objets composés, alors le programme **REPLA** va les "scanner" eux aussi, à la recherche d'exemplaires de **Obj2**, et ainsi de suite, de manière récursive.

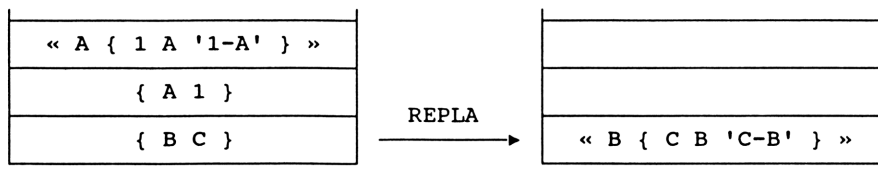
Voici un exemple du fonctionnement attendu de **REPLA**:



Le schéma fonctionnel de **REPLA** peut être enrichi puisqu'il autorise la présence de listes aux niveaux 2 et 1. Ces deux listes **List2** et **List1** doivent avoir la même longueur.

Le programme **REPLA** remplacera, dans l'objet **Obj3**, toutes les occurrences de n'importe quel objet **Obj2** de **List2** par l'objet **Obj1** qui lui correspond (qui a la même position) dans la liste **List1**.

Voici un exemple d'une telle utilisation:



On peut très bien imaginer une solution utilisant le programme **→STRI** (on transforme **Obj3** en chaîne de caractères, et dans celle-ci on remplace toutes les occurrences de la chaîne représentant **Obj2** par autant d'exemplaires de la chaîne représentant **Obj1**. Une fois ces remplacements terminés, on utilise le programme **→OBJ** pour obtenir l'objet final **Obj4**.

Cette méthode a l'avantage de simplifier le problème lié aux imbrications d'objets composés. Elle a pour inconvénient de n'être pas sûre à 100%. La chaîne représentant **Obj3** a par ailleurs une taille mémoire double de celle de **Obj3** lui-même.

La méthode suivie ici est la suivante (en reprenant les notations ci-dessus, et en se plaçant dans le cas du deuxième schéma fonctionnel):

- On "développe" l'objet composé **Obj3** sur la pile.
- On examine chacun des éléments ainsi apparus:
  - Si l'élément considéré est un élément **Obj2** de **List2**, on le remplace par l'élément correspondant **Obj1** dans **List1**.
  - S'il est composé, on lui applique récursivement le programme **REPLA**.
  - Dans les autres cas, cet élément n'est pas modifié.
- On "reconstruit" enfin l'objet composé initial (dont le contenu a donc été, éventuellement, modifié).

La liste permettant de construire le programme **REPLA** se trouve page suivante. Elle contient quelques adresses qu'il n'est pas inutile de commenter:

- **#6E97h** (Mnémonique ≈): Place sur la pile l'objet qui suit dans le Rpl en cours, sans l'évaluer.
- **#63EC5h** (Mnémonique **dup;=0(s)?;pifte**): Imaginons que cette instruction est suivie des objets **Obj1** et **Obj2** dans le Rpl en cours. Cette instruction examine l'entier-système du niveau 1, après l'avoir dupliqué. S'il est non nul, on évalue **Obj1**, sinon on évalue **Obj2**. Dans les deux cas, on poursuit après **Obj2**.
- **#618A8h** (Mnémonique **dup;=0(s)?:drdq/sk)**: Comme le précédent, sauf que si l'entier est nul, alors on l'élimine, puis on exécute **Obj1** en même temps que l'on quitte le Rpl en cours (**drdq** est l'abréviation de **drop;do;quit**). Si cet entier est non nul, on "***saute***" **Obj1** pour continuer au niveau de **Obj2** (sk="***skip***").
- **#7249h** (Mnémonique **StoIndex**): Dans une boucle Start, c'est la valeur de l'entier-système immédiatement supérieur à la dernière valeur admise pour l'indice de la boucle.
- **#52D26h** (Mnémonique **{4[""]}**: C'est l'adresse d'une liste de 4 éléments, tous égaux à l'adresse du nom local vide.

Voici maintenant le listing du programme **REPLA**. Vous saisissez la liste ci-après et lui appliquez le programme **→EXT**. La structure Rpl obtenue doit être stockée dans la variable **'REPLA'**.

La commande **BYTES** donne les résultats suivants:

- Pour la liste: Taille 747 octets, CheckSum #4871h
- Pour la structure Rpl: Taille 180 octets, CheckSum #FF78h

Le programme REPLA

Listing de **REPLA**:

```
{ #18A68h
  #4003h    #73DBh
     #3223h    #62211h
     #712Ah    #23EEDh
  #7334h
  #63231h    #611FEh
  #567Bh     #3D4Eh    #63B05h
  #6E97h
  {  #62C41h
     #63EC5h     #3244h
     {  #73DBh
           #7249h     #3325h
           #613B6h     #6F8Eh
           #3223h
        #7334h
     }
  }
  #6E97h
  {  #61438h    #32C2h
     #644A3h
     #618A8h
     {  #18FB2h
        #403Fh
        {  #1A2DAh    #61A3Bh
           #632E5h    #5445h
        }
        #4049h
        {
           #632E5h     #546Dh
        }
        #4021h
        {
           #632E5h     #5459h
        }
        #407Bh
        {
           #632E5h     #5481h
        }
        #3FEFh     #6E8Eh
     }
     #6140Eh    #60F0Eh
     #62B9Ch
  }
  #52D26h     #74D0h
  #613B6h     #6F8Eh
  #7497h
}
```

```
begin   need3
  <2h>     start(1,s)
     swap    dup;li?
     ?:skip     1->list
  next
  dup;ssize(cp)    3pick
  ssize(cp)    <>(2s)?     ?:BAV!
  ≈
  begin    break;dup
     dup;=0(s)?;pifte    drop
     begin    start(1,s)
        StopIndex     roll(s)
        1Lget     eval
        swap
     next
     end
  end
  ≈
  begin    4Lget    over
     spos(cp,_)
     dup;=0(s)?;drdq/sk
     begin    ChkArg
        <8h>
        begin    dup;InRom?    ?:q
            2Lget;eval    ->rpl(s)
        end
        <9h>
        begin
          2Lget;eval    ->expr(s)
        end
        <5h>
        begin
          2Lget;eval    ->list(s)
        end
        <Eh>
        begin
          2Lget;eval     ->unit(s)
        end
        <0h>    nop
     end
     3Lget    drop(3);swap
     get(cp,s)
  end
  {4['']}    Ldef(_,1i)
  1Lget    eval
  Lpop
end
```

On peut, au vu du listing précédent, analyser le fonctionnement du programme **REPLA**. Voici les opérations réalisées successivement:

- On commence par vérifier la présence de trois arguments sur la pile.

- On s'assure que les éléments présents aux niveaux 2 et 1 sont des listes, au besoin en leur appliquant l'instruction **1→LIST**.

- On vérifie que la taille de ces deux listes est la même (sinon il y a une erreur "***Bad Argument Value***"). Ces deux listes peuvent être vides, mais il n'y aura dans ce cas aucun remplacement...

- On place alors, sur la pile, deux structures Rpl, que nous noterons par exemple **Rpl2** et **Rpl1**:
  La première (**Rpl2**) est celle qui est chargée de développer sur la pile un objet composé et de générer une boucle consistant à "examiner" chacun des éléments ainsi apparus.
  La seconde (**Rpl1**) consiste précisément en l'examen d'un tel élément.

- Dans cet "examen", on effectue un remplacement immédiat (par l'élément de substition prévu à cet effet) de l'élément en cours d'examen s'il appartient à la liste des éléments recherchés.
  Sinon, on étudie si l'élément examiné est un objet composé. Dans chacun des cas possibles (structure Rpl, expressions, listes, objets-unités) on fait un appel récursif à la procédure **Rpl1**.
  Dans les autres cas, l'élément examiné n'est pas modifié (mnémonique "**nop**").

- On crée quatre variables locales, qui sont ensuite rappelables par les instructions **1Lget** (pour la structure **Rpl1**) **2Lget** (pour la structure **Rpl2**) **3Lget** (pour la liste **List1** des éléments de substitution **Obj1**) et **4Lget** (pour la liste **List2** des éléments à rechercher **Obj2**).

- A ce moment, seul reste sur la pile l'objet **Obj3** (au sens des notations initiales). Il suffit de lancer la procédure de recherche-remplacement par les instructions **1Lget eval** (on démarre par l'"examen" de l'objet **Obj3**.

On observe en passant que la structure Rpl **Rpl1** appelle éventuellement la structure Rpl **Rpl2** (si l'objet qui lui est soumis est un objet composé) et qu'inversement **Rpl2** appelle systématiquement **Rpl1**. On est donc en présence d'un cas intéressant de récursivité croisée.

N.B: **REPLA** sera utilisé dans les programmes **→LIB** et **LIB→**.

CRÉER UNE LIBRAIRIE

Vous avez remarqué que les exemples de programmes en Externals, présentés dans cette partie du livre, suivent à peu près une courbe ascendante de complexité. Nous voici donc arrivé à un sommet avec le programme qui va suivre, baptisé **→LIB**, et dont le rôle est de vous permettre de créer vos propres librairies, sans quitter la HP48.

Quand on sait la complexité de la structure d'une librairie, il s'agit d'un problème épineux. Cette structure est décrite, dans le détail, aux pages 110 à 120 du tome 1. On ne peut que recommander au lecteur de revoir ce chapitre.

Le programme **→LIB** (si on y ajoute les trois programmes **→VEC** **→STRI** et **REPLA**, développés dans les pages précédentes, et qui sont appelés par **→LIB**) possède une taille à peine supérieure à 1500 octets, ce qui est raisonnable au regard de la difficulté de la tâche. Enfin, ce qui ne gâche rien, le programme **→LIB** est assez rapide (beaucoup plus que s'il était écrit en Rpl-utilisateur, beaucoup moins que s'il était écrit en assembleur n'est-ce-pas **Stéphane**?).

Le schéma fonctionnel de **→LIB** est le suivant:

1:Name ---[**→LIB**]---▸ **1:....**

Ici, *Name* est le nom d'une variable dont le contenu doit être un sous-répertoire du répertoire en cours. Les programmes **→LIB**, **→VEC**, **→STRI** et **REPLA** doivent être "visibles" dans le répertoire en cours (c'est-à-dire lui appartenir ou appartenir à un des répertoires du chemin ascendant vers **HOME**).

A la sortie de **→LIB** (si tout s'est bien passé), le nom *Name* ne contient plus un répertoire, mais l'objet-librairie résultant de la traduction de ce répertoire.

Principes de fonctionnement:

Comme il vient d'être dit, la variable contenant au départ le sous-répertoire à traduire contient, à la fin, l'objet-librairie résultant de cette traduction.

Tout au long de l'exécution de **→LIB**, les variables du sous-répertoire sont purgées au fur et à mesure des besoins (et de leur conversion en objets de la future librairie). L'avantage principal de cette méthode est de limiter au maximum la taille-mémoire nécessaire pour le fonctionnement de **→LIB**.

Dans la pratique, le programme **→LIB** pourra fonctionner correctement si la mémoire disponible au départ est au moins égal à 3 fois la taille de la plus grosse des variables du sous-répertoire à convertir. Si vous avez suffisamment de place, vous pouvez bien sûr préalablement dupliquer le sous-répertoire dans une autre variable du répertoire en cours....

Le programme **→LIB** est assez complet, notamment en ce qui concerne la gestion des erreurs possibles. Par conséquent, même si vous ne respectez pas toutes les consignes précisées ci-après, vous ne devriez pas avoir de mauvaise surprise.

Conditions requises et variables réservées:

Un certain nombre de conditions doivent impérativement être réunies si l'on veut que le programme **→LIB** fonctionne. D'autre part certains noms de variables (obligatoires ou non) jouent un rôle spécial. Elles doivent appartenir au sous-répertoire à traduire, et leur nom commence toujours par le caractère **$** (que l'on peut obtenir par les touches α-(shift left)-4).

- Le nom **'Name'** initial doit être celui d'un sous-répertoire du répertoire en cours. Ce sous-répertoire ne doit pas lui-même contenir de sous-répertoire.
- Le sous-répertoire à traduire doit contenir une variable nommée **$ROMID** contenant le numéro de la librairie. Ce numéro doit être un réel compris entre 256 et 2047.
- Vous pouvez préciser le titre de la librairie dans une variable nommée **$TITLE**: ce titre doit être une chaîne d'au plus 255 caractères. Si la variable **$TITLE** est absente, la librairie prendra pour titre le nom *Name* du sous-répertoire en cours de conversion.
- Vous pouvez associer à la future librairie une table de messages. Pour celà vous placez une liste de chaînes de caractères dans une variable **$MESSAGE**. Cette liste sera traduite en un vecteur de chaînes par le programme **→VEC** (vu précédemment) qui est appelé par **→LIB**. Vous avez droit à au plus 255 messages.

  Si la librairie porte le numéro N, la chaîne n°k de cette liste devient l'erreur de numéro 256*N+k.
- Vous pouvez indiquer, dans une variable **$HIDDEN**, la liste des variables qui doivent rester cachées dans la librairie. Toutes les variables réservées citées ici (**$ROMID, $TITLE, $MESSAGE, $HIDDEN**) ne sont pas traduites en objets de la librairie par le programme **→LIB**. Il n'y a donc pas lieu de les inclure dans la liste placée dans **$HIDDEN**.
- Vous pouvez placer dans une variable **$CONFIG** un objet de configuration. Il s'agit d'un programme qui sera exécuté lors de la prise de connaissance de la librairie par la HP48 (une fois que vous aurez stocké la librairie dans un port, puis éteint et rallumé la calculatrice).

  L'application principale de l'objet de configuration est de rendre une librairie auto-attachable. Si la librairie porte par exemple le numéro 1789, vous la rendez auto-attachable en plaçant le programme « **1789 ATTACH** » dans la variable **$CONFIG**.

  Le contenu de **$CONFIG** est automatiquement traduit en un objet caché de la librairie. Inutile donc de le faire figurer dans le contenu de **$HIDDEN**.

- **ATTENTION**: l'objet de configuration doit laisser la pile comme il la trouve et il ne doit pas être susceptible de provoquer une erreur, sous peine de *"Memory Clear"*.

Les erreurs possibles:

Plusieurs messages d'erreur peuvent interrompre le fonctionnement de **→LIB**:

- "***Undefined Name***":
  Si le nom initialement au niveau 1 n'existe pas.
  Si la variable **$ROMID** n'existe pas dans le sous-répertoire à transformer.
  Si la variable **→VEC** n'est pas visible dans le répertoire initial (uniquement en cas de présence de **$MESSAGE**).
  Si **→STRI** ou **REPLA** n'est pas visible dans le répertoire initial.
- "***Bad Argument Type***":
  Si le nom initial ne désigne pas un sous-répertoire.
  Si le contenu de **$ROMID** n'est pas un réel.
  Si le contenu éventuel de **$TITLE** n'est pas une chaîne.
  Si le contenu éventuel de **$MESSAGE** n'est pas une liste de chaînes.
  Si le contenu éventuel de **$CONFIG** n'est pas un programme.
  Si le contenu éventuel de **$HIDDEN** n'est pas une liste de noms.
- "***Bad Argument Value***":
  Si le numéro de la librairie (dans **$ROMID**) est illégal.
- "***Directory Not Allowed***":
  Si le répertoire à transformer contient au moins un sous-répertoire.

Comme il a été dit, les données du sous-répertoire sont purgées au fur et à mesure de leur traduction. Il s'ensuit que certaines erreurs peuvent se produire alors que des données ont déjà été supprimées. Celà ne concerne que les variables réservées décrites ci-dessus (purgées dans l'ordre suivant: **$ROMID, $TITLE, $MESSAGE, $HIDDEN**).

Il faut faire une exception pour l'erreur "***Insufficient Memory***", qui peut survenir à tout moment si vous lancez **→LIB** sans avoir suffisamment de mémoire disponible. Dans ce cas, il se peut que des données du sous-répertoire aient disparu. Il faut être prévoyant.

Les messages de →LIB:

Le programme **→LIB** affiche un certain nombre de messages pour renseigner l'utilisateur sur la progression de la conversion du sous-répertoire en librairie.

Voici quels sont ces messages:

- "1: Processing $TITLE"
- "2: Processing $MESSAGE"
- "3: Processing $HIDDEN"
- "4: Building hash-table"
- "5: Building link-table"
- "6: Collecting pieces"
- "7: The job is over"

Dans les listings qui vont suivre, les instructions (adresses) relatives à ces messages sont imprimées en caractères soulignés. Vous êtes libres de ne pas conserver ces instructions. Les CheckSums seront d'ailleurs fournis pour les deux possibilités.

Format d'une librairie créée par →LIB:

Bien que celà n'ait qu'un intérêt théorique, les librairies formées par le programme **→LIB** obéissent à la règle suivante:

Dans le corps de la librairie on trouve successivement: La *Hash-Table*, la table des messages, la *Link-Table*, et la zone des objets. L'objet de configuration, s'il existe, est le premier des objets cachés de la librairie (dans l'ordre des numéros **XLIB**).

Enfin les objets (non cachés) apparaîtront dans la librairie dans le même ordre que dans le sous-répertoire initial.

Objets algébriques ou non:

Le programme **→LIB** permet de créer des objets algébriques (qui peuvent être utilisés dans des expressions) dans la librairie en cours de formation.

Pour que le contenu d'une variable soit traduit comme objet algébrique, il est nécessaire que ce contenu soit un programme, et que ce programme débute:

- Soit par la séquence: « → «
- Soit par la séquence: « → '

Il s'agit donc des programmes dont la première instruction est une création de variables locales.

Le listing de →LIB:

Pour présenter le programme **→LIB**, nous allons procéder en plusieurs étapes, chacune d'elle correspondant à un moment particulier de la construction d'une librairie.

Les tableaux en deux parties, auxquels vous êtes maintenant habitués, seront à nouveau utilisés. La partie gauche contient les listes d'adresses à saisir sur la HP48. La partie droite contient les mnémoniques qui correspondent à ces adresses (autant pour servir de commentaires à la liste que pour indiquer le texte à saisir si vous voulez utiliser le programme **EXTERNPC**).

Pour ce qui est des commentaires extérieurs à ces tableaux, ils seront limités au minimum. On se contentera d'une brève description de l'algorithme.

Première étape:

Vous saisissez la liste ci-dessous, puis vous lui appliquez le programme **→EXT**. La structure Rpl obtenue sera placée dans une variable nommée **'ETAP1'**. Vous pouvez bien sûr conserver la liste dans une variable, mais souvenez vous que le programme **EXT→** vous permet de la récupérer à partir de la structure Rpl.

Cette première étape correspond aux opérations suivantes:

- On vérifie que les variables **→STRI** et **REPLA** sont bien "visibles".
- On vérifie que le nom présent au niveau 1 est bien celui d'une variable du répertoire en cours, et que cette variable contient un sous-répertoire.
- On entre dans ce sous-répertoire. On teste qu'il ne contient aucun sous répertoire.
- On contrôle l'existence de **$ROMID**, et le fait qu'elle contient un réel "correct".
- Si la variable **$TITLE** est présente, on vérifie qu'elle contient une chaîne (qui peut être vide), puis on la purge. Sinon on fait du nom initial une chaîne.
- On combine la chaîne de caractères obtenue avec l'entier contenu dans **$ROMID**, dans un même entier binaire. Tout au long des étapes à suivre, le programme **→LIB** formera ainsi des entiers binaires, dont la concaténation conduira, à un changement d'adresse-préfixe près, à la librairie finale.

La commande BYTES donne les résultats suivants:

- Pour la liste: CheckSum #AE39h, Taille 745.
- Pour le Rpl produit par **→EXT**: CheckSum #CE1Eh, Taille 209.5 octets.
  (Rappel: vous stockez ce Rpl dans la variable '**ETAP1**')

```
{"→STRI"   #5B15h   #20B81h
 "REPLA"   #5B15h   #20B81h
 #3258h
 #3188h    #1936Ch
 #621BDh   #63B46h   #8D08h
 #1CC85h   #186E8h   #55B7h
 #712Ah    #11026h
 "$ROMID"   #5B15h
 #20962h   #3188h
 #8C4Ah    #214A9h
 #8DF7h    #5853Eh   #63B05h
 #59CCh    #3FF9h    #400Dh
 #5815h    #3223h
 "1: Processing $TITLE"
 #5046Ah   #1245Bh
 "$TITLE"   #5B15h
 #1853Bh   #61AD8h
 {  #62154h   #63B46h
    #3188h    #8C4Ah   #60F9Bh
 }
 #5BE9h    #64E32h   #63259h
 #627BBh   #63EC5h
 #5A75h
 {  #5A75h    #61380h
    #52EEh    #3223h
 }
 #525Bh    #32C2h    #518Ah
}
```

```
"->STRI"    st=>gn   rcl(nm)
"REPLA"    st=>gn   rcl(nm)
drop2
dup   dup;RclInVars(nm)
dup;di?   ?#:BAT!  SetCurDir(di)
15   tvars(r)   ={}?
?:skip   doerr12A
"$ROMID"   st=>gn
RclRealInVars(nm)   dup
purge(_)   ChkN°Lib(r)
<7FFh>   over;<(2s)?   ?:BAV!
s=>b   <1h>   <3h>
sub(b,2s)   swap
"1: Processing $TITLE"
cllcd   1disp(st)
"$TITLE"   st=>gn
RclInVars(nm)?   pifte
begin   dup;st?   ?#:BAT!
   dup   purge(_)   drop(2)
end
nm=>st!   <100h>   sub(st,1,s-1)
dup;ssize(st)   dup;=0(s)?;pifte
s=>ch
begin   s=>ch   swap;over
  +(st,ch)   swap
end
prep(st,ch)   over   conc(2b)
```

Deuxième étape:

Dans cette deuxième étape, le programme **→LIB** exécute les opérations suivantes:

- On place dans une variable locale un sous-programme qui sera appelé par trois fois (dans les étapes 2,4 et 5).
  Le fait que cette variable locale "courre" sur 3 étapes successives montre qu'il est hors de question d'exécuter individuellement les structures Rpl correspondant à ces étapes.

- On teste l'existence de la variable **$MESSAGE**. Dans l'affirmative, on transforme son contenu en vecteur par **→VEC** (dont on contrôle d'abord l'existence).
  On s'assure ensuite que le vecteur obtenu est bien composé de chaînes de caractères. Ce vecteur est alors transformé en un entier binaire.

- Si la variable **$MESSAGE**, n'existe pas, on laisse un entier binaire vide sur la pile.
- On purge finalement la variable **$MESSAGE**.

N.B: Le réel 10728 sera transformé en l'entier-système <29E8h> (adresse-préfixe des tableaux) par le programme **→EXT**.

Les caractéristiques obtenues par **BYTES** sont les suivantes:
Liste: Taille: 490.5 octets. CheckSum: #7FA7h
Rpl produit par **→EXT**: Taille: 149 octets. CheckSum: #E9D5h
(Rappel: vous stockez la structure Rpl dans la variable '**ETAP2**')

```
{ #6E97h
  {
    #59CCh    #32C2h    #5616h
    #6258Ah    #59CCh    #3295h
    #518Ah    #518Ah
  }
  #634CFh
  "2: Processing $MESSAGE"
  #1246Bh
  "$MESSAGE"    #5B15h
  #18536h    #61AD8h
  {
    "→VEC"    #5B15h    #797Bh
    #712Ah    #18C92h    #6F8Eh
    #3FF9h    #32C2h    #371Dh
    #3244h    #62159h    #63B46h
    #55D5h    #518Ah
    10728     #613B6h    #6F8Eh
  }
  #55D5h
  #3223h    #1854Fh
}
```

```
≈
begin
   s=>b      over      quart(b)
   5+(s)      s=>b      rot
   conc(2b)    conc(2b)
end
Ldef''
"2: Processing $MESSAGE"
2disp(st)
"$MESSAGE"    st=>gn
dup;RclInVars(nm)?    pifte
begin
   "->VEC"    st=>gn    rcl(nm)?
   ?:skip    doerr204    eval
   <1h>    over    get(s,ar)!
   drop    st?    ?#:BAT!
   Null#    conc(2b)
   <29E8h>    1Lget    eval
end
Null#
swap    purge(nm)
```

Troisième étape:

Cette liste correspond aux opérations suivantes:

- On teste l'existence de la variable **$CONFIG**. Si elle est présente, on s'assure que son contenu est bien une structure Rpl.
- Idem avec **$HIDDEN** qui, si elle existe doit contenir une liste de noms, puis on la purge (son contenu reste sur la pile).
- On ajoute le nom **$CONFIG** en tête de la liste contenue dans **$HIDDEN** (à la liste vide si **$HIDDEN** n'existait pas). On obtient ainsi la liste des variables qui doivent (sous réserve de la vérification suivante) rester cachées.
- On retire de la liste obtenue par **VARS** les noms appartenant à la liste précédente. En même temps on élimine dans cette précédente liste les doublons éventuels ou les noms qui ne se trouveraient même pas dans la liste **VARS**.
- On obtient ainsi deux listes disjointes, dont la réunion est la liste obtenue par **VARS**. La liste obtenue au niveau 2 est formée par les noms devant réellement rester cachés. La liste du niveau 1 est celle des noms devant rester visibles.

Les caractéristiques obtenues par **BYTES** sont les suivantes:
Liste: Taille: 486 octets. CheckSum: #2508h
Rpl produit par →**EXT**: Taille: 150 octets. CheckSum: #EAE5h
(Rappel: vous stockez la structure Rpl dans la variable '**ETAP3**')

```
{ "$CONFIG"   #5B15h
  #18536h   #619BCh
  {  #621ECh   #63B46h  }
  "3: Processing $HIDDEN"
  #1247Bh
  "$HIDDEN"   #5B15h
  #1853Bh   #61AD8h
  {
     #20BE0h
     #3188h
     #62C69h   #8C4Ah
  }
  #55E9h
  #1AC93h   #18779h   #631F5h
  #61305h   #55E9h   #60F33h
  #73DBh
     #5182Fh   #61318h
     #31ACh   #644A3h
     #63EC5h   #3258h
     {  #63001h   #52FAh
        #60FACh   #E4DEh
     }
  #7334h
}
```

```
"$CONFIG"   st=>gn
dup;RclInVars(nm)?   pift
begin   rpl?   ?#:BAT!   end
"3: Processing $HIDDEN"
3disp(st)
"$HIDDEN"   st=>gn
RclInVars(nm)?   pifte
begin
   dup;ListName?#:BAT!
   dup
   newob;swap   purge(_)
end
{}
+(_,cp)   vars   swap;break
dup;roll(s+2)   {}   swap(1,3)
start(1,s)
   BeforeStop   roll(s+2)
   dup2   spos(cp,_)
   dup;=0(s)?;pifte   drop2
   begin   4roll;rot   +(cp,_)
    3rolld   del(li,s)
   end
next
```

Quatrième étape:

Voici venue l'étape cruciale (et difficile) de la création de la Hash-Table. Voyons succintement les actions réalisées par le programme →**LIB** dans cette quatrième étape.

On part de la liste des noms visibles, au niveau 1, et on en déduit la construction de la Hash-Table, sous la forme d'un entier binaire (comme il a déjà été dit, tous les éléments de la librairie sont construits sous cette forme, pour être concaténés à la fin).

- Si cette liste est vide, on se contente de placer un entier binaire vide sur la pile, et la Hash-Table est obtenue...
- Dans le cas contraire, on crée 16 variables locales (contenant au départ des binaires vides). Chacun de ces 16 binaires est progressivement augmenté du codage des noms visibles ayant une longueur déterminée.

  Par exemple l'un de ces entiers binaires reçoit le codage des noms visibles ayant exactement 5 caractères. L'une des variables reçoit le codage des noms visibles ayant 16 caractères ou plus.

  Cette opération est réalisée par la première boucle de la liste.
- On construit (sous la forme d'entiers binaires) les offsets vers les zones de noms de longueur donnée (voir structure Hash-Table, tome 1 page 114) en se basant sur la taille des 16 variables locales précédentes (opération réalisée par la deuxième boucle).
- On concatène ces 16 offsets ainsi que le binaire indiquant la longueur totale de la zone des noms (3ème boucle). On obtient ainsi un entier binaire #b1.
- On concatène les 16 variables locales évoquées ci-dessus (c'est l'affaire de la quatrième boucle). On obtient un entier binaire #b2.
- Le résultat de cette concaténation est dupliqué et sa copie est transformée en chaîne de caractères (par →**STRI**). De cette manière, il est plus facile de calculer les offsets permettant d'accéder individuellement à chacun des noms visibles dans la zone correspondant à son nombre de caractères (ces calculs sont réalisés dans la 5ème boucle).
- Ces offsets sont obtenus sous la forme d'entiers binaires (il y en a autant que de noms visibles dans la librairie) et sont concaténés ensemble pour obtenir un entier binaire #b3.

  On concatène alors les entiers binaires #b1,#b2,#b3.

  Ces différentes opérations sont réalisées par la 6ème boucle.
- L'entier binaire obtenu #b représente exactement la Hash-Table. Il reste à le copier sur un autre entier binaire, en vue de la concaténation finale.

Les caractéristiques obtenues par **BYTES** sont les suivantes:

Liste: Taille: octets. CheckSum: #h

Rpl produit par →**EXT**: Taille: octets. CheckSum: #h

(Rappel: vous stockez la structure Rpl dans la variable '**ETAP4**')

N.B: Le réel 10830 (vers la fin de la liste) est transformé par →**EXT** en l'entier-système <2A4Eh> (adresse préfixe des entiers binaires).

Le programme →LIB

```
{ "4: Building hash-table"
  #1248Bh
  #3188h   #55B7h
  #61AD8h   #55D5h
  {  #3188h   #55D5h
     #408Fh   #5E370h
     #6E97h   #34D30h
     #61CE9h
     #63498h
        #7221h  #31ACh  #62B9Ch
        #5BE9h   #627BBh
        #628EBh   #4099h
        #624BAh  #3188h  #75A5h
        #62001h   #5A75h
        #525Bh  #7221h  #3E0Eh
        #59CCh  #3FF9h  #400Dh
        #5815h   #518Ah
        #61099h   #518Ah
        #3223h   #75E9h
        #60F33h   #1DC00h
     #7334h
     #64C66h   #40A3h   #4003h
     #73F7h
        #7221h   #75A5h
        #5616h   #62266h
        #61AD8h  #3FEFh  #32C2h
        #59CCh   #60FACh
        #3DBCh   #6261Ah
     #7334h
     #59CCh   #408Fh
     #73DBh   #518Ah   #7334h
     #613E7h   #40A3h   #400Dh
     #73F7h
        #7221h  #75A5h  #518Ah
     #7334h
     #7497h
     #62CA5h   →STRI
     #631F5h   #634CAh
     #61305h   #3223h
     #73C3h
        #3188h   #613B6h
        #61318h   →STRI
        #4053h   #63281h
        #15EF6h   #5622h
        #62794h   #3DEFh
        #7221h  #4021h  #3EC2h
        #3DBCh  #59CCh  #3223h
     #7334h
     #3244h   #634B6h   #3DEFh
     #73DBh   #518Ah   #7334h
     10830   #613B6h   #6F8Eh
  }
  #6109Eh   #3223h   #521Fh
}
```

```
"4: Building hash-table"
4disp(st)
dup   ={}?
pifte   Null#
begin   dup   Null#
  <10h>   conB(_,s)
  ≈   Local''
  LBlockDef(_,s,ln)
  dup;start(1,szcp)
     index   dup2   get(cp,s)
     nm=>st!   dup;ssize(st)
     dup;1+(s)   <11h>
     min(2s)   dup   Lget(s)
     swap2     s=>ch
     prep(st,ch)  index  1-(s)
     s=>b    <1h>   <3h>
     sub(b,2s)   conc(2b)
     dup;4rolld   conc(2b)
     swap   Lput(_,s)
     swap(1,3)   put(_,s,li)
  next
  <55h>   <12h>   <2h>
  start(s1,s2-1)
     index   Lget(s)
     quart(b)   dup;=0(s)?
     pifte   <0h>   over
     s=>b    3rolld
     +(2s)   5-(s)
  next
  s=>b     <10h>
  start(1,s)   conc(2b)   next
  2Lget   <12h>   <3h>
  start(s1,s2-1)
     index   Lget(s)   conc(2b)
  next
  Lpop
  rot;over     ->STRI
  swap;break   dup;Ldef''
  dup;roll(s+2)   swap
  start(0,s-1)
     dup     1Lget
     roll(s+2)     ->STRI
     <Ah>   sub(st,s+1,∞)
     spos(2st)   over;ssize(st)
     swap;-(2s)   1+(s)
     index   <5h>   *(2s)
     +(2s)    s=>b    swap
  next
  drop   1Lget;Lpop   1+(s)
  start(1,s)   conc(2b)   next
  <2A4Eh>   1Lget   eval
end
4rolld   swap   +(2cp)
```

Cinquième étape:

Cette cinquième étape est comme la précédente, cruciale et difficile. Il s'agit en effet de créer deux zones fondamentales de la librairie: la Link-table et la zone des contenus des objets. Ces deux zones seront obtenues sous la forme d'entiers binaires.

Précisons tout d'abord que cette étape attend:

- au niveau 5: un entier binaire de trois quartets indiquant le numéro de la librairie (cet objet a été créé lors de la première étape).
- au niveau 1: la liste de tous les noms de la librairie (d'abord les noms devant rester visibles, puis les noms à cacher). Cette liste a été obtenue tout à la fin de la quatrième étape: les noms y apparaissent dans l'ordre croissant de leurs futurs numéros XLIB.

Voici brièvement le "script" de cette 5ème étape.

- Si la liste des noms est vide, on se contente de placer deux entiers binaires vides sur la pile (la librairie ne contiendra aucun nom...) et c'est fini.
- Dans le cas contraire, après la création de quelques variables locales utiles, on entre dans une boucle qui couvre la presque totalité de cette étape.

  Ces variables locales peuvent être appelées au moyen des instructions codées **1Lget**, **2Lget**,..., **5Lget**. En particulier les mnémoniques **5Lget** et **3Lget** permettent de rappeler la liste des noms du répertoire initial et la liste des noms XLIB qui vont leur correspondre.
- Lors d'un passage dans cette boucle, on extrait un des noms (appelons-le 'name1') de la liste initiale. On rappelle le contenu de ce nom sur la pile.
  - On remplace dans ce contenu toutes les occurrences des noms de variables par les noms XLIB correspondants (utilisation du programme **REPLA** présenté précédemment).
  - Si le contenu de 'name1' est une structure Rpl, on vérifie si elle est de nature algébrique ou non (voir le tome 1, page 113), ce qui permet de terminer sur la pile le codage du contenu initial de 'name1'.
  - On purge le nom 'name1' devenu inutile.
  - Le nouveau contenu de ce nom est transformé en entier binaire et concaténé aux entiers binaires représentant le codage des contenus déjà obtenus.
  - En fonction de la taille de ce dernier contenu obtenu, on en déduit, dans la link-table (qui elle aussi, partie d'un binaire nul, augmente peu à peu), l'offset qui permettra d'accéder à cet objet.
- A la sortie de cette boucle infernale, les deux zones (la link-table et la zone des contenus des objets) sont sur la pile.

Les caractéristiques obtenues par **BYTES** sont les suivantes:

Liste: Taille: 1278.5 octets. CheckSum: #218Ah
Rpl produit par **→EXT**: Taille: 312 octets. CheckSum: #CEACh
(Rappel: vous stockez la structure Rpl dans la variable **'ETAP5'**)

N.B: pendant cette 5ème étape, un message renseigne en permanence sur le nombre d'entrées de la librairie qui restent à traiter.

```
{ "5: Building link-table"
#1249Bh
#55D5h  #3188h
#611FEh  #6103Ch
#32C2h  #55B7h  #61AD8h
#3258h
{ #3188h  #5A03h
  #55E9h  #6121Ch  #567Bh
  #73C3h
     #32C2h  #7221h
     #7E50h  #52FAh
  #7334h
  #60F9Bh  #400Dh  #EDE1h
  #403Fh  #59CCh
  #3FF9h  #3188h  #5815h
  #5A75Ch  #74D0h
  #6145Ch  #567Bh  #73DBh
     #5182Fh
     #167E4h  "Still "
     #622EFh  " Name(s)"
     #5193h  #124ABh
     #6145Ch  #7221h  #62B9Ch
     #1853Bh  #627A7h
     #6145Ch  #6140Eh
     REPLA  #3223h  #8C4Ah
     #621E7h  #3223h
     #6657h  #3188h  #5902h
     #6258Ah  #6657h  #3223h
     #631B9h  #5089h
     10830  #5ACCh
     #3223h  #61AD8h
     {
        #5DEEBh
        #613B6h  #613E7h
     }
     #613B6h
     #61438h  #518Ah
     #7221h  #3E0Eh  #59CCh
     #3FF9h  #400Dh  #5815h
     #518Ah  #60EE7h
     #518Ah  #62CA5h  #5616h
     #5182Fh  #4021h  #3EC2h
     #3DBCh  #59CCh  #518Ah
     #60F33h  #518Ah
  #7334h
  #7497h  #3223h  10830
  #634B6h  #6F8Eh  #3223h
}
}
```

```
"5: Building link-table"
5disp(st)
Null#  dup
3pick  8roll
over  ={}?  pifte
drop2
begin   dup    b=>s
   {}  4pick  ssize(cp)
   start(0,s-1)
      over  index
      2s=>xn   +(cp,_)
   next
   drop(2)  <3h>  makeb(s)
   <8h>   s=>b
   <1h>  dup  sub(b,2s)
   \C57A5  Ldef(_,li)
   5Lget  ssize(cp)  start(1,s)
   BeforeStop
      s=>st  "Still "
      swap;+(2st)!  " Name(s)"
      +(2st)!  6disp(st)
      5Lget  index  get(cp,s)
      RclInVars(nm)?   drop;dup
      5Lget  3Lget
      REPLA  swap  purge(_)
      dup;rpl?  swap
      newob  dup  nibbles(_)
      5+(s)  newob  swap
      2->list    get1(cp)
      <2A4Eh>  ModType(_,s)!
      swap  pifte
      begin
         alg(_)?:dq/skdq
         1Lget  2Lget
      end
      1Lget
      4Lget  conc(2b)
      index  1-(s)    s=>b
      <1h>  <3h>  sub(b,2s)
      conc(2b)  swap(2,3)
      conc(2b)  rot;over  quart(b)
      BeforeStop  <5h>  *(2s)
      +(2s)   s=>b  conc(2b)
      swap(1,3)  conc(2b)
   next
   Lpop  swap  <2A4Eh>
   1Lget;Lpop  eval  swap
end
```

N.B: La longue chaîne de caractères doit être considérée comme appartenant aux deux parties du tableau.

Cette chaîne sert à déterminer si une structure Rpl est ou n'est pas de nature algébrique.

Pour bien saisir le fonctionnement de la sixième étape, il faut savoir qu'elle trouve la pile dans l'état suivant:

Au niveau 7: le nom initial du sous-répertoire.
Au niveau 6: Un binaire contenant le nom puis le N° de librairie.
Au niveau 5: Un binaire contenant la Hash-Table.
Au niveau 4: Un binaire contenant la table des messages.
Au niveau 3: La liste { noms visibles + noms cachés }.
Au niveau 2: Un binaire contenant la Link-Table.
Au niveau 1: Un binaire contenant le corps des objets de la librairie.

6ème étape:

Cette étape correspond aux opérations suivantes:

- On calcule les offsets (devant être placés vers la tête de la librairie) et qui permettent d'accéder respectivement à la Hash-Table, la table des messages et la Link-Table. Ce travail est réalisé par la première boucle. On y utilise la taille des 3 objets précités, qui ont été obtenus au cours des étapes 2, 4, et 5.

- On vérifie la présence éventuelle du nom **$CONFIG** parmi ceux du répertoire initial. Dans l'affirmative la position **n** de **$CONFIG** dans la liste {variables visibles, variables cachées} signifie que **$CONFIG** correspond maintenant au nom XLIB de numéro **n – 1** dans la librairie.
  Connaissant l'entier **(n – 1)**, on trouve la position du corps de **$CONFIG** dans la zone des contenus des objets (en utilisant la link-table créée à l'étape précédente). Cette position permet de calculer l'offset permettant d'accéder directement, depuis le début de la librairie, au programme **$CONFIG**.

- Tous les zones de la future librairie sont maintenant obtenues sous formes d'entiers binaires (sauf les quatre derniers quartets, contenant le CheckSum). Il reste à concaténer ces différents entiers, ce qui est réalisé par la 2ème boucle.

- A l'issue de cette sixième étape, la pile contient le nom initial du sous-répertoire (au niveau 2), et un entier binaire correspondant à l'image de la librairie (à l'adresse-préfixe près et au CheckSum près).

Les caractéristiques obtenues par **BYTES** sont les suivantes:
Liste: Taille: 697 octets. CheckSum: #BFC1h
Rpl produit par **→EXT**: Taille: 172 octets. CheckSum: #BBAAh
(Rappel: vous stockez la structure Rpl dans la variable **'ETAP6'**)

```
{ "6: Collecting pieces"
  #124ABh
  #3295h   #634CFh
  #60FD8h   #40B7h
  #400Dh   #73DBh
     #5182Fh
     #611D2h   #5616h
     #3295h   #6364Bh
     #61AD8h   #3FEFh
     #611FEh   #59CCh
     #518Ah   #60FACh
     #3DBCh   #6261Ah
  #7334h
  #6121Ch   #5616h
  #3DE0h   #634B6h
  "$CONFIG"   #5B15h
  #644A3h   #62266h
  #61AD8h   #60F9Bh
  {
     #3E2Dh   #4021h
     #3EC2h   #6123Ah
     #32C2h   #6260Ah
     #611FEh   #5815h
     #5A03h   #60FACh
     #3DBCh   #6261Ah
     #3DBCh
  }
  #59CCh   #518Ah
  #610C4h   #4017h
  #73DBh   #518Ah
  #7334h
}
```

```
"6: Collecting pieces"
6disp(st)
rot   Ldef''
5roll   <14h>
<3h>   start(1,s)
   BeforeStop
   pick(s+3)   quart(b)
   rot   over;=0(s)?
   pifte   <0h>
   3pick   s=>b
   conc(2b)   3rolld
   +(2s)   5-(s)
next
4pick   quart(b)
-(2s)   1Lget;Lpop
"$CONFIG"   st=>gn
spos(cp,_)   dup;=0(s)?
pifte   drop(2)
begin
   2+(s)   <5h>
   *(2s)   5pick
   over   4-(s)
   3pick   sub(b,2s)
   b=>s   3rolld
   +(2s)   5-(s)
   +(2s)
end
s=>b   conc(2b)
5rolld   <4h>
start(1,s)   conc(2b)
next
```

7ème étape:

On approche de la fin... Il ne reste plus qu'à réaliser les opérations suivantes:

- Calculer la somme de contrôle de la librairie, sous la forme d'un entier binaire de quatre chiffres hexadécimaux.
- Placer ce CheckSum à la fin de l'entier binaire du niveau 1.
- Par une simple modification d'adresse-préfixe, transformer l'entier binaire obtenu en l'objet librairie tant cherché.
- Remonter dans le répertoire parent (celui d'où on était parti, car toutes les opérations se sont réalisées dans le sous-répertoire à traduire).
- Stocker l'objet-librairie sous le nom initial, à la place du sous-répertoire qu'il contenait au départ.

Le calcul du CheckSum doit s'effectuer par une routine écrite en assembleur (pour des raisons de rapidité, et on a hâte d'en finir!).

Nous allons utiliser le point d'entrée **#05981h** (voir dans le dernier chapitre de ce livre, avec le label **Crc0A**) qui utilise le circuit spécialisé de la HP48.

L'instruction assembleur **GOSBVL #5981h** calcule le checksum de la zone débutant à l'adresse pointée par **D0**, sur une longueur de **A:***A* ("**A** champ *A*") quartets. Le résultat va dans le champ *A* du registre **A**. Pour finir, le registre **D0** pointe sur le premier quartet qui suit la zone parcourue.

Pour l'instant la pile contient, au niveau 1, un entier binaire qui est identique à la future librairie à deux détails près:

- L'adresse-préfixe n'est pas la bonne....
- Il manque à la fin les quatre quartets sur lesquels on doit coder le checksum de la librairie.

Le CheckSum de la librairie doit être calculé sur tous les quartets, depuis le premier à indiquer la longueur de l'objet (juste après l'adresse-préfixe, cette longueur tenant compte de la présence des 4 quartets finaux de checksum), jusqu'au dernier quartet précédant le CheckSum.

La première opération que nous allons réaliser consiste à concaténer à l'entier binaire qui se trouve sur la pile un entier binaire nul de quatre chiffres héxadécimaux, et ce pour deux raisons:

- C'est sur ces 4 quartets nuls que va être écrit le checksum.
- La taille de l'entier binaire est maintenant celle que possèdera la librairie (et cette taille est utilisée dans le calcul du checksum).

Nous allons profiter de la routine en assembleur pour changer l'adresse-préfixe, de manière à transformer l'entier binaire en librairie (l'adresse préfixe des librairies est #02B40h).

Nous allons donc utiliser un objet-code. Pour le placer sur la pile, exécutez le programme suivant (dont la taille est de 100.5 octets, le CheckSum étant égal à #3B6Dh):

```
« "CCD20A30008FB9760"  "1471343404B20144164"  +
  "142818F838F1895015838D34150"  +  →OBJ
»
```

Vous constatez que ce programme forme une chaîne de caractères, qui est ensuite transformée en un ***Code Object*** par le programme **→OBJ** (qui a été présenté précédemment).

Vous placez cet objet-code dans une variable (baptisée **'CODE'**, par exemple), et vous la laissez au chaud pour l'instant (mais ne l'évaluez pas!).

Le programme →LIB

Voici le code source de cette routine en langage machine. Chaque ligne de ce tableau fait apparaître, successivement:

- Les quartets de l'objet-code, instruction par instruction.
- Les mnémoniques correspondant à ces instructions.
- Des commentaires succints indiquant le contenu de tel ou tel registre.

```
8FB9760       GOSBVL   #679B        Sauve B,D,D0 et D1.
147           C=DAT1   A            C:A Pointe sur l'entier-système.
134           D0=C                  Idem avec D0.
3404B20       LC(5)    #2B40        (C:A) adr-préfixe des librairies
144           DAT0=C   A            Transforme le #..h en librairie.
164           D0=D0+   5            Pointe sur le champ "longueur".
142           A=DAT0   A            Place dans A:A cette longueur.
818F83        A=A-4    A            Longueur de la zone à lire.
8F18950       GOSBVL   #5981        Calcule le CRC (checksum).
1583          DAT0=A   4            Ecrit ce CRC en fin de lib.
8D34150       GOVLNG   #5143        Récupère B,D,D0,D1 et retour Rpl.
```

Nous voici en mesure de réaliser la 7ème étape de la fabrication du programme **→LIB**, à l'aide de la liste ci-dessous et du programme **→EXT**.

Les caractéristiques obtenues par **BYTES** sont les suivantes:
Liste: Taille: 140.5 octets. CheckSum: #D4EAh
Rpl produit par **→EXT**: Taille: 77.5 octets. CheckSum: #779Eh
(Rappel: vous stockez la structure Rpl dans la variable '**ETAP7**')

- Pour former cette liste, vous placez tous ses éléments sur la pile, avant d'effectuer une instruction **8 →LIST**.
- Pour placer l'objet-code sur la pile (et pas le nom 'Code' évidemment!), vous rappelez le contenu de la variable dans laquelle vous avez placé cet objet-code (mais vous ne l'évaluez pas!).

```
{ #4017h      #61C1Ch                 <4h>      add0(b,s)
  Code                                \....
  "7: The job is over..."             "7: The job is over..."
  #124BBh       #3FDD1h               7disp(st)      biiip
  #1A16Fh       #1955Bh               updir      sto(nm,_)
}
```

Remarque:

Dans la partie droite du tableau précédent, l'abréviation \... remplace les codes qu'il faut inclure à cet endroit-là dans le texte que vous voulez utiliser avec **EXTERNPC**. D'une façon précise, rappelons que des quartets peuvent être directement spécifiés dans ce texte, à condition d'être précédés du signe \ (anti-slash). Ainsi le texte à soumettre à **EXTERNPC** peut-il contenir (la longueur des séquences est laissée à votre appréciation):

\CCD20A30008FB9760
\1471343404B20144164
\142818F848F1895015838D34150

8ème et dernière étape !:

Les efforts précédents vont maintenant être récompensés. Les étapes 1 à 7 nous ont permis de créer 7 sous-programmes, placés dans les variables **'ETAP1'**, **'ETAP2'**, ..., **'ETAP7'**.

Nous voulons réunir (concaténer) ces 7 sous-programmes, pour obtenir le programme **→LIB**. Il faut prévoir (on en a pris l'habitude) de vérifier la présence d'un argument sur la pile et le fait que cet argument est un nom global (**need1**, **ChkArg**, **<6h>**, d'adresses respectives **#18AB2h**, **#18FB2h**, **#402Bh**).

Le programme qui suit va maintenant concaténer les 7 structures Rpl ainsi obtenues, et introduire la vérification initiale sur le type d'argument. La taille de ce programme est de **219.5 octets**, et son CheckSum est **#BC0Ah**:

```
« { #18AB2h   #18FB2h   #402Bh }   →EXT
  { ETAP1   ETAP2   ETAP3   ETAP4   ETAP5   ETAP6   ETAP7 }
  OBJ→   DROP   #55FDh   SYSEVAL
  8   2   FOR   k
     k   ROLL   RCL   #521Fh   SYSEVAL
  -1   STEP
  #63FE7h   SYSEVAL   #521Fh   SYSEVAL
»
```

Quand vous aurez exécuté ce programme, vous obtiendrez une structure Rpl au niveau 1 de la pile. Stockez la sous le nom **→LIB**, et :

C'EST FINI !

Vous disposez maintenant d'un programme vous permettant de convertir vos sous-répertoires en librairies. Il faut signaler que la commande **BYTES** fournit les résultats suivants, pour le contenu de **→LIB**: Taille 1369 octets; CheckSum #FE6Eh.

Rappelez vous bien la nécessité de prévoir la variable **$ROMID** dans le sous-répertoire à traduire, et le fait qu'il faut environ (mais ça dépend d'autres facteurs) une taille-mémoire libre égale au moins à 3 fois la taille de la plus grosse des variables du sous-répertoire.

Le sous-répertoire ne doit pas lui-même contenir de sous-répertoire, et il doit être "autosuffisant" (c'est-à-dire qu'aucune des variables qui le composent ne doit utiliser des variables d'autres répertoires).

La vitesse de fonctionnement de **→LIB** dépend de la taille et du nombre des variables du sous-répertoire. A titre d'exemple, un sous-répertoire de 5K-octets peut être traduit en environ 25 secondes (l'étape la plus longue est la création de la Link-Table).

DÉCOMPILER VOS LIBRAIRIES

Le programme →**LIB** vous permet de créer vos propres librairies. Vous savez qu'un tel type d'objet ne peut pas être modifié facilement. La seule solution est de revenir au sous-répertoire qui vous a permis de former cette librairie, de le modifier, et de le transformer à nouveau au moyen de →**LIB**.

Si vous n'avez pas conservé le sous-répertoire initial, le programme **LIB→** présenté maintenant va vous être d'un grand secours. Il permet en effet de "*décompiler*" votre librairie, c'est-à-dire de la transformer "*à rebours*" en le sous-répertoire qui lui a donné naissance.

Pour être décompilée par **LIB→**, votre librairie doit être installée dans l'un des ports 0, 1 ou 2 (la calculatrice ayant été éteinte puis rallumée pour qu'elle en prenne connaissance). Il n'est pas nécessaire que cette librairie soit attachée.

Le schéma fonctionnel de **LIB→** est le suivant:

1: n ----[**LIB→**]----▸ **1:**.....

où **n** est un entier indiquant le numéro de la librairie.

A la sortie de **LIB→**, le contenu de la librairie est stocké, sous forme d'un sous-répertoire, dans le répertoire en cours. Le nom de ce sous-répertoire est par exemple **'L1234'** si le numéro de votre librairie est **1234**. Ce sous-répertoire contient les entrées suivantes, éventuellement:

- Tous les objets visibles de la librairie sont stockés sous leur nom "*en clair*".
- Les objets invisibles de la librairie sont stockés sous des noms commençant par **'XL'**. Par exemple le nom **'XL26'** désignera le nom de l'objet initialement caché et dont le numéro XLIB était 26 dans la librairie initiale.
- La variable **$CONFIG** contient le contenu de l'objet de configuration.
- La variable **$HIDDEN** contient la liste des noms des objets qui étaient cachés dans la librairie (voir ci-dessus).
- La variable **$MESSAGE** contient la liste des chaînes de caractères constituant les différents messages de la librairie. Cette liste est obtenue, à partir du vecteur de chaînes contenu dans la librairie, par utilisation du programme **VEC→**, déjà présenté dans ce chapitre.
- La variable **$TITLE** contient une chaîne indiquant le nom de la librairie.
- La variable **$ROMID** contient le numéro de la librairie. C'est la seule variable qui apparait obligatoirement.

Le principe fondamental du programme **LIB→** est de vous permettre d'obtenir, à partir d'une de vos librairies créées par →**LIB**, un sous-répertoire qui, s'il est à nouveau transformé par →**LIB**, fournira la librairie initiale (→**LIB** et **LIB→** sont donc inverses l'un de l'autre).

Néanmoins, le sous-répertoire formé par **LIB→** n'est pas exactement semblable au sous-répertoire initial, à cause de la traduction qui doit être faite des noms des objets cachés (si vous déclarez comme caché un objet de nom **'Name'** dans le sous-répertoire que vous transformez par →**LIB**, ce nom disparaît de la librairie obtenue).

Remarques:

Une des tâches principales du programme **LIB→** consiste à remplacer, dans le corps des différents objets de la librairie, les appels de noms XLIB par des appels de noms "*en clair*". Cette opération peut prendre un certain temps, notamment si votre librairie contient de nombreuses entrées ou des programmes de taille importante.

Ces opérations de remplacement sont effectuées par le programme **REPLA**, déjà présenté dans ce chapitre, et qui doit donc être "visible" dans le sous-répertoire en cours. A ce propos, il ne faut pas que la librairie que vous décompilez contienne d'entrées dont le nom soit précisément **REPLA**, sous peine d'erreurs aux conséquences imprévisibles.

De même, il faut éviter que la librairie contienne des objets visibles dont les noms soient du type **XLn** (où n est un entier), à cause de la confusion qui en résulte avec les noms créés par **LIB→** pour désigner les objets cachés de la librairie).

La progression dans les opérations de remplacement est signalée à l'utilisateur par un message du type "**Still n name(s)**", où **n** varie en décroissant et désigne le nombre d'entrées qui restent à décompiler.

Contrairement au programme **→LIB**, qui purgeait le sous-répertoire au fur et à mesure de sa traduction, le programme **LIB→** ne détruit pas la librairie qui doit être décompilée. Vous devez donc disposer de suffisamment de mémoire disponible.

Le programme **LIB→** est beaucoup plus court que le programme **→LIB**. La raison en est que la structure de librairie (sur laquelle travaille **LIB→**) est beaucoup plus "*sûre*" que la structure de sous-répertoire (sur laquelle travaille **→LIB**). Il n'est par exemple plus nécessaire de calculer de CheckSum, ni de former la *Link-Table* ou la *Hash-Table*.

D'autre part, il est clair que les adresses que l'on trouve à la page 331 du tome 1 sont ici particulièrement utiles.

Un certain nombre de messages d'erreurs peuvent interrompre le fonctionnement du programme **LIB→**. Les voici:

- "***Bad Argument Value***" si le numéro de librairie est incorrect.
- "***Undefined Name***" si les programmes **REPLA** et **VEC→** ne sont pas visibles dans le répertoire en cours (pour **VEC→**, seulement si la librairie possède un vecteur de messages; pour **REPLA**, dans le cas où la librairie contient au moins une entrée, ce qui est tout de même le cas général).

Le listing du programme **LIB→** se trouve à la page suivante. Comme d'habitude, si vous utilisez le programme **→EXT**, il faut saisir la liste de la moitié gauche du tableau et lui appliquer **→EXT**. On obtient alors une structure Rpl que l'on peut placer dans la variable nommée '**LIB→**'.

La commande **BYTES** donne les résultats suivants:

- Pour la liste: Taille 1655 octets, CheckSum #F26h
- Pour la structure Rpl: Taille 479 octets, CheckSum #AEC3h

Le programme LIB→

```
{#18AB2h  #18FB2h  #3FF9h
{#214A9h  #3188h
   #8199h  #63B19h
   "Uncompiling Lib "
   #611FEh  #167E4h  #5193h
   #5046Ah  #1245Bh  #5BE9h
   #8DF7h  #77E4h
   #611FEh  #167E4h  #6551Ah
   #525Bh  #5B15h  #61380h
   #8696h  #6F8Eh  "$TITLE"
   #5B15h  #8696h  #3188h
   #8130h  #619BCh
   { #71646h  "VEC→"
     #5B15h  #20B81h
     #6F8Eh  "$MESSAGE"
     #5B15h  #8696h
   }
   #3188h  #813Ch  #61AD8h
   { #71646h  #5616h
     #4021h  #3EF7h  #60F9Bh
   }
   #3FEFh  #63EC5h  #3258h
   { #32C2h  #3F4FBh  #55E9h
     #611FEh  #73C3h
        #6121Ch  #7221h
        #7E50h  #52FAh
     #7334h
     #55E9h  #6121Ch  #73C3h
        #6123Ah  #7221h  #7E50h
        #81FBh  #712A
        {  "XL"  #7221h
          #167E4h  #5193h
          #5B15h
        }
        #52FAh
     #7334h
     #3295h  #630C9h  #63EB1h
     {  #62D31h  #3DEFh
       #6123Ah  #5821h
       "$HIDDEN"  #5B15h
       #8696h
     }
     #3244h  "REPLA"  #5B15h
     #20B81h  #3244h
     #6121Ch  #8157h  #619BCh
     {  #63C68h  REPLA
       "$CONFIG"  #5B15h
        #8696h
     }
     #3295h  #73C3h
        #5182Fh  #3188h
        #167E4h  "Still "
        #622EFh  " Name(s)"
        #5193h  #1248Bh  #6121Ch
        #32C2h  #3E0Eh  #7E50h
        #7E99h  #3244h  #63FBAh
        REPLA  #611FEh  #3295h
        #62B9Ch  #8696h
        #7334h
     #60F4Bh
   }
   #1A16Fh  "The job is over..."
   #1248Bh  #3FDD1h
}}
```

```
begin  need1  ChkArg  <1h>
begin  ChkLib(r)  dup
  NameLib(s)?  ?#:BAV!
  "Uncompiling Lib "
  3pick  s=>st  +(2st)!
  cllcd  ldisp(st)  nm=st!
  <7FFh>  MakeDirWithLib(s)
  3pick  s=>st  $L
  prep(st,ch)  st=>gn  swap;over
  NewSto(_,nm)  eval  "$TITLE"
  st=>gn  NewSto(_,nm)  dup
  MessLib(s)?  pift
  begin  eval(pa)  "VEC->"
    st=>gn  rcl(nm)
    eval  "$MESSAGE"
    st=>gn  NewSto(_,nm)
  end
  dup  LinkLib(s)?  pifte
  begin  eval(pa)  quart(b)
    <5h>  mdv(2s)  drop(2)
  end
  <0h>  dup;=0(s)?;pifte  drop2
  begin  over  NumEntryLib(s)  {}
    3pick  start(0,s-1)
       4pick  index
       2s=>xn  +(cp,_)
    next
    {}  4pick  star(0,s-1)
       5pick  index    2s=>xn
       name(xn)?  ?:skip
       begin  "XL"  index
         s=>st  +(2st)!
         st=>gn
       end
       +(cp,_)
    next
    rot  4pick;over  >(2s)?;pifte
    begin  over;swap  1+(s)
       5pick  sub(cp,2s)
       "$HIDDEN"  st=>gn
       NewSto(_,nm)
    end
    drop  "REPLA"  st=>gn
    rcl(nm)  drop
    4pick  ConfigLib(s)?  pift
    begin  pick(2-3)  REPLA
       "$CONFIG"   st=>gn
       NewSto(_,nm)
    end
    rot  start(0,s-1)
       BeforeStop  dup
       s=>st  "Still "
       swap;+(2st)!  " Name(s)"
       +(2st)!  4disp(st)  4pick
       over  1-(s)   2s=>xn
       rcl(xn)?  drop  pick(3-4)
       REPLA  3pick  rot
       get(cp,s)  NewSto(_,nm)
    next
    3dropn
  end
  updir  "The job is over..."
  4disp(st)  biiip
end  end
```

DEUXIEME PARTIE

Programmation en Assembleur

LE MICRO-PROCESSEUR SATURN

Cette deuxième partie s'ouvre sur une présentation des caractéristiques du micro-processeur Saturn (registres, mode héxadécimal ou BCD, sélecteurs de champ, arithmétiques signée ou non signée, etc...).

La parfaite connaissance de ces notions est un préalable indispensable si on veut se lancer dans la programmation en assembleur.

◆ INTRODUCTION

Le lecteur sait qu'il peut programmer sa HP48 sous plusieurs langages différents:

- **Le RPL-utilisateur**: c'est le langage de plus haut niveau, le plus simple, le plus courant, celui par lequel commencent tous les utilisateurs de la HP48. C'est aussi un langage "*sûr*": il est en principe impossible de "*planter*" sa calculatrice en l'utilisant.

  Cette sécurité se paye évidemment d'une (relative) lenteur, dûe aux tests multiples protégeant les données de votre calculatrice de toute erreur involontaire. Et puis il peut être frustrant, à la longue, de se limiter à ce qui est seulement permis.

- **Le RPL-système**: c'est le langage dans lequel est écrit la plus grande partie de la ROM de la HP48. Il constitue un sur-ensemble du "*Rpl-utilisateur*" (il recèle beaucoup plus de possibilités, mais en même temps il est moins "sûr").

  La structure de ce langage a été longuement développée dans la première partie de ce livre. Les "*adresses*" (ou "*externals*"), qui sont autant d'instructions nouvelles de ce langage, ont fait l'essentiel du tome 1.

- **Le langage-machine**: c'est le seul directement compréhensible par le microprocesseur "**Saturn**", qui est le cerveau de notre HP48.

  On peut dire que chacune des instructions de ce langage est une combinaison précise de "bits" (0 ou 1). Le défaut de ce langage est son caractère illisible. Sa qualité principale est sa rapidité d'exécution.

- **L'assembleur**: pour être précis, un assembleur "*Saturn*" est un programme (traitement de texte éventuel, mais en tout cas compilateur) qui permet de produire des séquences écrites en langage-machine, à partir de codes lisibles (les "*mnémoniques*").

  Le "*code-source*" (la succession logique de ces mnémoniques) est traduite par le compilateur en un "*code-objet*" (écrit en langage-machine et directement exécutable). Bien entendu, il faut s'entendre une fois pour toutes sur les mnémoniques à utiliser. Vous trouverez ici ceux que reconnait officiellement **Hewlett-Packard**.

  L'intérêt est double: conserver la rapidité du code-machine, et permettre une programmation "*lisible*". Celà reste cependant un langage ardu (manque de clarté, difficultés de "*débogage*") qui demande une longue pratique et une non moins longue patience (que celui qui n'a jamais connu de "*Memory Lost*" me jette le premier octet).

La raison essentielle pour laquelle j'ai entrepris d'écrire les deux tomes des "**Secrets de la HP48**" n'était pas de décrire le microprocesseur "*Saturn*" et son langage-machine. Elle était plutôt de vous présenter les possibilités du langage "*Rpl-Système*". Je pense (modestement tout de même) avoir produit là quelque chose d'original. C'est en tout cas le fruit d'un travail personnel et patient.

Néanmoins, un tel ouvrage (qui s'adresse à des utilisateurs motivés) ne serait pas complet s'il ne décrivait pas le langage-machine de la HP48. Je ne prétends pas, comme dans le reste de cet ouvrage, à une grande originalité. Les mnémoniques officiels du "*Saturn*" sont connues. Elles ont été publiées aux USA. Les caractéristiques du microprocesseur ont également été décrites outre-atlantique.

Je me propose simplement d'expliquer au mieux (la pédagogie, c'est tout de même mon métier) ce qu'est la programmation en assembleur. Les deux programmes **ASM48** et **ASMPC** vous permettront de voler de vos propres ailes. Le lecteur jugera de lui-même. Quant à ceux qui s'y connaissent déjà un peu, il pourront survoler ici où là quelques inévitables généralités.

◆ LE MICRO-PROCESSEUR "SATURN"

Le microprocesseur "Saturn" est le cerveau de la HP48. Il est le dernier né d'une famille qui a d'abord équipé le HP71, puis la HP28, enfin la HP48. A chacune de ces étapes, il s'est enrichi de nouvelles instructions.

Un certain nombre de points importants doivent être dégagés dès maintenant:

▸ **Les registres:**

Ce sujet ne peut qu'être effleuré ici. Il sera développé un peu plus loin. Disons simplement que le microprocesseur "Saturn" possède des "*emplacements-mémoire*" particuliers destinés à être lus, modifiés, utilisés, etc... Ces élements de mémoire ne doivent pas être confondus avec ce que nous appelons en général la mémoire (morte ou vive, ROM ou RAM) de la HP48.

Ces *registres*, car c'est ainsi qu'on les appelle, sont de tailles variables, et chacun d'eux a un rôle particulier. Sans rentrer dans le détail, disons qu'il existe des registres de type "*pointeur*" (pour désigner un point particulier de la mémoire), des registres "*de travail*" (pour contenir et manipuler des données) et des registres "*de flags*" (décrivant le contexte en cours).

▸ **Les quartets:**

L'unité logique d'information, pour le "Saturn", est le **quartet** (**nibble**, en anglais), qui consiste en quatre bits successifs. Un quartet permet donc de coder un entier naturel de 0=#0000b à 15=#1111b.

Une unité de mesure plus courante en informatique est l'**octet**, qui est la juxtaposition de deux quartets. Sur un octet (8 bits), on peut coder un entier de 0 à #11111111b=2^8-1=255. On peut également y coder un "*caractère*" (le code du caractère "A" est 65=#41h=#01000001b).

Rappelons également qu'un **Kilo-Octet** est égal à #400h=1024 octets.

▸ **L'adressage:**

Le microprocesseur "Saturn" exécute séquentiellement des instructions. Celles-ci sont placées "*quelque part en mémoire*". de la même manière, il lui est souvent nécessaire de lire ou d'écrire des données "*à telle ou telle adresse*".

Les adresses auxquelles peut accéder le "Saturn" sont tous les entiers pouvant être codés sur 20 bits. Un tel entier vaut au minimum 0, et au maximum #FFFFFh.

A chaque adresse, se trouve un quartet. On comprend dès lors que l'espace mémoire total auquel peut accéder le "Saturn" est long de #100000h*5 bits (#100000h quartets, #80000h octets, #200 Ko, c'est-à-dire 512 Kilo-Octets, ou encore un demi "Méga-Octet").

▸ **Héxadécimal ou Binaire Codé Décimal:**

Une des originalités du "Saturn" est de pouvoir effectuer des opérations ou bien en héxadécimal, ou bien en décimal.

Les opérations en "*héxadécimal*" prennent pleinement en compte les différents "bits" des données (il s'agit de numérotation binaire, l'aspect héxadécimal ne correspondant qu'à notre façon de noter ces opérations). Autrement dit les quartets utilisés peuvent prendre toutes les valeurs de 0 à #Fh=15. Dans un tel mode, la somme de #4827h et de #7858h est égale à #C07Fh.

Le mode **BCD** (*binaire codé décimal*) consiste à ne considérer que des quartets limités aux valeurs de 0 à 9, et à effectuer les opérations en tenant compte des retenues *décimales*. Dans un tel mode, la somme de #4827h et de #7858h est égale à #12685h.

▸ **Ecriture et lecture "à l'envers":**

La longueur des principaux registres du "Saturn" est toujours un nombre entier de quartets (au minimum 1, au maximum 16). Dans un registre long de **n** quartets (4*n bits), ceux-ci sont (implicitement) numérotés de 0 (le quartet de poids le plus faible) à n-1 (le quartet de poids le plus fort).

Ainsi, si l'on dit d'un registre de 20 bits (5 quartets) qu'il contient la valeur #FD45Bh, c'est que le quartet n°0 contient #Bh et que le quartet n°4 contient #Fh.

Les transferts de données s'effectuent toujours par groupes de quartets. Dans les opérations de lecture ou d'écriture (d'un registre vers la mémoire, ou le contraire), les quartets de poids faible du registre sont écrits (ou lus) avant les quartets de poids fort. Il s'ensuit une impression de lecture-écriture "à l'envers".

Prenons un exemple:

Si le contenu d'un registre de 5 quartets est #FD45Bh et si ce contenu est écrit à l'adresse #7E000h, alors:

A l'adresse #7E000h, on trouvera le quartet #Bh.
A l'adresse #7E001h, on trouvera le quartet #5h.
....
A l'adresse #7E004h, on trouvera le quartet #Fh.

Réciproquement, si on trouve successivement les quartets #7h, #Fh, #8h, #3h, et #Dh à partir de l'adresse #7F000h et qu'on les charge dans ce même registre, #7h va dans le quartet n°0 du registre, #Fh va dans le quartet n°1, ..., et #Dh va dans le quartet n°4: le registre contient alors #D38F7h.

◆ LES REGISTRES DU "SATURN"

Le microprocesseur **SATURN** est "équipé" de:

- **4 registres de travail**: **A**,**B**,**C** et **D**.
  Leur longueur est de 64 bits (c-à-d 16 quartets).
  Ils sont essentiellement utilisés pour manipuler des données.
  Les registres **A** et **C** sont également sollicités dans les échanges (lecture et/ou écriture) avec la mémoire de la HP48.
  N.B: Dans le fonctionnement par défaut de la HP48, les registres **B** et **D** se voient attribuer un rôle particulier: **B** contient la première adresse de la mémoire libre (juste après la pile des retours du RPL-Système). **D** contient la quantité de mémoire disponible, exprimée en blocs de 5 quartets (adresses).

- **5 registres de sauvegarde**: **R0**,**R1**,**R2**,**R3**,**R4**.
  Leur longueur est de 64 bits (16 quartets) et ils servent surtout de sauvegarde temporaire pour le contenu des registres de travail **A**,**B**,**C** ou **D** (beaucoup plus rapidement que s'il fallait écrire puis relire en mémoire centrale les contenus des registres à sauvegarder).

- **2 registres "pointeurs de données"**: **D0** et **D1**.
  Leur longueur est de 20 bits (5 quartets). Ils sont surtout utilisés pour désigner une adresse particulière de la mémoire.
  Les opérations de lecture/écriture s'effectuent à l'adresse qu'ils désignent, les registres de travail servant à "véhiculer" l'information.
  N.B: Dans le fonctionnement par défaut de la HP48, le registre **D1** pointe sur le niveau 1 de la pile-utilisateur, les niveaux suivants étant accessibles aux adresses **D1 + 5**, **D1 + 10**, etc...

- **1 registre "pointeur de champ"**: **P**.
  Sa longueur est de 4 bits (un quartet). Il peut donc contenir un entier de 0 à 15.
  Son rôle est principalement de désigner l'un des 16 quartets des registres de travail **A**,**B**,**C**, et **D** (qui sont numérotés de 0 à 15, du quartet de poids le plus faible au quartet de poids le plus fort).
  Le registre **P** peut également servir de compteur de boucle (dans la mesure où les valeurs 0 à 15 sont suffisantes). Les opérations d'incrémentation ou de décrémentation du registre **P** sont en effet particulièrement rapides.

- **1 registre "pointeur d'instruction"**: **PC** (Program Counter)
  Sa longueur est de 20 bits (5 quartets). Il contient l'adresse de la prochaine instruction à exécuter. Chaque opération réalisée par le microprocesseur modifie donc en conséquence la valeur de **PC**. L'utilisateur peut être amené à le modifier plus brutalement pour effectuer un saut inconditionnel à une adresse déterminée.

- **2 registres d'entrée-sortie**: **IN** et **OUT**.
  Ils sont dévolus aux opérations d'entrée-sortie du système.
  Le registre **IN** (registre des entrées) est long de 16 bits (4 quartets) et le registre **OUT** (registre des sorties) a une longueur de 12 bits (3 quartets).

- **La pile des retours**: **RSTK**
  Il ne s'agit pas d'un registre au même titre que les précédents, mais d'une pile permettant de mémoriser les adresses de retours des sous-programmes en langage machine. Elle permet de stocker au maximum 8 adresses (de 5 quartets chacune bien sûr).
  Le principe est toujours le même. La dernière adresse "*empilée*" sera la première "*dépilée*" (pile de type LIFO: "*Last IN, First OUT*").
  On ne confondra pas cette pile avec la pile des retours de sous-programmes en RPL système (largement évoquée dans la première partie de ce livre).

- **Le bit CARRY**:
  Il s'agit d'un bit isolé au sein du microprocesseur "Saturn", et dont le rôle est de signaler les retenues (par *overflow*) survenant lors d'opérations arithmétiques sur certains registres (une retenue place ce bit à 1), et de décrire les résultat des tests logiques (si un test entre deux registres donne la valeur logique *VRAI*), alors le bit **CARRY** est armé (positionné à 1), sinon il est désarmé.
  Alors que **CARRY** est un bit isolé, d'autres bits sont groupés en deux registres.

- **Le registre des "Status Bits"**: **ST**
  Il sont au nombre de 16, numérotés de 0 à 15. Les bits 12 à 15 (les 4 bits de poids fort) sont réservés à l'usage interne de la HP48.
  Les 12 bits numérotés de 0 à 11 sont à la disposition de l'utilisateur. Ils jouent le rôle d'indicateur, permettant ainsi d'orienter le flux des instructions selon les besoins.

- **Le registre des "Hardware Status Bits"**: **HS**.
  Ils sont armés par des événements matériels. Ils doivent être désarmés manuellement avant que d'être à nouveau testés pour détecter de tels événements.
  Ils sont au nombre de 4:
  MP (bit 3): "*Module Pulled*"
  SR (bit 2): "*Service Request*"
  SB (bit 1): "*Sticky Bit*"
  XM (bit 0): "*External Module Missing*"
  Seul le registre **SB** semble pouvoir être utilisé par un programmeur novice. Il est armé lorsqu'une opération de décalage à droite (*Right Shift*) sur un registre de travail, se traduit par la sortie, à l'extrémité droite de ce registre, d'un bit à 1.

◆ LES CHAMPS DES REGISTRES DE TRAVAIL

Quand on désire utiliser le contenu d'un registre de travail (**A**, **B**, **C** ou **D**) pour le modifier "*in situ*", ou pour écrire ce contenu en mémoire, ou pour y charger un nouveau contenu (en provenance de la mémoire ou d'un autre registre de travail), il n'est pas nécessaire d'opérer sur la longueur totale du registre.

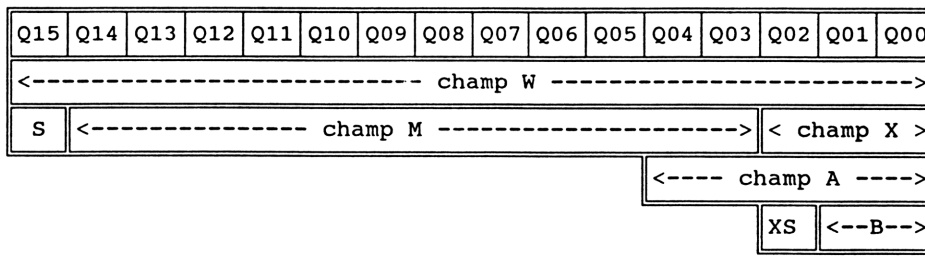
Une des propriétés les plus importantes du "Saturn" est qu'il nous permet en effet d'effectuer de telles opérations sur une partie seulement du ou des registres de travail concernés. Cette partie est toujours constituée d'un ou de plusieurs quartets consécutifs.

Une telle "*portion*" d'un registre est appelée un **champ**. De nombreuses instructions du "Saturn" doivent être accompagnées d'une identification du champ utilisé. Seuls certains champs sont reconnus par la syntaxe du "Saturn".

Chaque instruction qui nécessite la spécification d'un champ est codée différemment selon le champ utilisé (au niveau du "*code objet*" lui-même). Pour ce qui est du "*code source*", le mnémonique de cette instruction doit être suivi d'un "*sélecteur de champ*".

Voyons d'abord quels sont les différents champs possibles d'un registre de travail (**A**, **B**, **C** ou **D**: tous ont une longueur de 16 quartets).

La première ligne ci-dessous représente un registre de travail. La position des 16 quartets est désignée de Q00 (quartet n°0, celui qui a le poids le plus faible) à Q15 (quartet n°15, celui qui a le poids le plus fort). Les lignes suivantes indiquent les noms des différents champs possibles, ainsi que leur amplitude.



On distingue donc:

- Le champ **W** (*word*): quartets 0 à 15 (la totalité du registre).
- Le champ **S** (*mantissa sign*): uniquement le quartet n°15.
- Le champ **M** (*mantissa*): les 12 quartets n°3 à n°14.
- Le champ **X** (*exponent*): les 3 quartets de poids le plus faible.
- Le champ **XS** (*exponent sign*): uniquement le quartet n°2.
- Le champ **B** (*byte*): les deux quartets de poids le plus faible.
- Le champ **A** (*address*): les 5 quartets de poids le plus faible.

Les champs précédents sont fixes. Il existe encore deux spécifications de champ variable, qui dépendent de la valeur en cours du registre **P** (dont on rappelle qu'il est long d'un quartet, et permet donc de représenter tous les entiers de 0 à 15):

- Le champ **P**: c'est le quartet désigné par la valeur du registre P.
- Le champ **WP**: depuis le quartet n°0 jusqu'au quartet n°P (inclus).

Remarques:

Les champs **S**, **M**, **X**, et **XS** tirent leur nom de la façon dont les nombres réels étaient codés sur le HP71.

On ne confondra pas, bien sûr, le champ **A** d'un registre de travail, et le registre **A** lui-même. Il en est de même pour le champ **B** et le registre **B**, ou encore pour le champ **P** et le registre **P** (encore que dans ce dernier cas, le rapport soit évident).

Le champ **A** est, de loin, le plus utilisé. De nombreuses instructions sont d'ailleurs codées d'une manière très particulière quand ce champ doit être utilisé.

Le champ **A** tire son nom du fait que son amplitude (5 quartets) le désigne tout naturellement à recevoir des *adresses* (elles aussi codées sur 5 quartets). Quant au champ **B**, on rappelle qu'un octet se dit ***byte*** en anglais, et que la taille d'un octet est de deux quartets...

◆ TABLES DE SÉLECTION DE CHAMP

Nous allons voir que la syntaxe du "Saturn" (c-à-d la manière dont sont représentées les instructions dans le "code-objet") permet d'identifier, quand c'est nécessaire, le champ utilisé. Pour celà on utilise deux tables qui établissent une correspondance entre les différents champs possibles et les différentes valeurs d'un quartet.

Considérons par exemple l'instruction qui consiste à mettre à zéro le contenu d'un champ du registre de travail **C**.

Dans le "*code source*", l'instruction s'écrit, par exemple:

C=0 A (pour annuler le champ **A** du registre **C**)
C=0 W (pour annuler la totalité du contenu de **C**)
C=0 S (pour annuler le champ **S**, c-à-d le quartet n°15)
C=0 B (pour annuler les deux octets de poids faible de **C**)
etc...

Voici comment s'écrivent ces 4 instructions, dans le "*code-objet*":

C=0 A est codée **D2** (codage particulier pour le champ A)
C=0 W est codée **AF2** ||
C=0 S est codée **AC2** || Les codages utilisés ne diffèrent que
C=0 B est codée **AE2** || par **un** quartet de code-machine.

Considérons maintenant l'instruction qui consiste à ajouter le contenu du registre **A** au contenu du registre **C**, en se limitant (pour ces deux registres) à un champ particulier, le résultat étant stocké (toujours dans le même champ) dans le registre **C**.

Dans le "*code source*", l'instruction s'écrit, par exemple:

C=C+A A (somme des "*champs A*" de **A** et **C**)
C=C+A W (addition de la totalité des registres **A** et **C**)
C=C+A S (addition limitée au quartet n°15)
C=C+A B (addition limitée aux deux octets de poids faible)
etc...

Voici comment s'écrivent ces 4 instructions, dans le "*code-objet*":

C=C+A	**A**	est codée **C2**	(codage spécifique au champ A)
C=C+A	**W**	est codée **A72**	|| Les codages utilisés ne
C=C+A	**S**	est codée **A42**	|| diffèrent que par **un**
C=C+A	**B**	est codée **A62**	|| quartet de code-machine.

On constate, sur les exemples précédents, que la valeur d'un seul quartet, au sein d'un groupe de quartets correspondant au codage d'une instruction en langage-machine, peut permettre d'identifier le champ utilisé.

Cependant il n'y a que 8 champs possibles (si on excepte le champ **A**, pour lequel il y a le plus souvent des codages spécifiques) alors que le quartet en question peut prendre 16 valeurs. Il y a donc la possibilité (et on a vu ci-dessus qu'elle est utilisée) de coder les variantes de **deux** instructions différentes (en l'occurence **C=0** et **C=C+A**) avec un même quartet "sélecteur".

Tout celà ne doit rien au hasard. Les codes-machine du "Saturn" utilisent deux tables permettant d'identifier un champ par la valeur d'un quartet. Voici ces deux tables:

Quartet (a) du code-machine	Quartet (b) du code-machine	Champ qui correspond	Quartets concernés dans le registre
0	8	P	n°P
1	9	WP	n°0 à n°P
2	A	XS	n°2
3	B	X	n°0 à n°2
4	C	S	n°15
5	D	M	n°3 à n°14
6	E	B	n°0 à n°1
7	F	W	n°0 à n°15
F	---	A	n°0 à n°4

Voici comment il conviendra d'interpréter ces tables. Dans le chapitre suivant, nous verrons la liste de toutes les instructions du "Saturn" (les codes-machine ainsi que les mnémoniques associés). Reprenons les deux instructions **C=0** et **C=C+A** utilisées précédemment. Voici leur description, permettant de retrouver rapidement le code-machine correspondant à un champ donné.

Mnémonique	Description
C=0 *field*	Annule le contenu du champ "*field*" du registre **C** Champ A => code-machine: **D2** Autre champ => code-machine: **Ab2**
C=C+A *field*	Ajoute les champs "*field*" de **C** et **A**. Résultat dans le champ "*field*" de **C** Champ A => code-machine: **C2** Autre champ => code-machine: **Aa2**

On verra dans le chapitre suivant que l'utilisation du champ **A** (de loin la plus fréquente) est souvent codée de manière spécifique, comme sur les deux exemples ci-dessus. Il en va de même (mais plus rarement) pour le champ **B**.

On peut cependant constater, dans la table des sélecteurs de champ décrite précédemment, que le quartet **(a)** peut éventuellement désigner, s'il prend la valeur **F**, le champ **A**.

Cette situation ne se produit que lorsqu'une instruction du "Saturn", dans laquelle une spécification de champ est nécessaire, ne possède pas de code-machine spécifique pour le champ **A**.

C'est l'exemple de l'instruction suivante:

Mnémonique	Description
C=C&A *field*	**"ET logique"** entre les champs "*field*" de **A** et **C**. Le résultat va dans le champ "*field*" de **C**. Champ quelconque => code-machine: **0Ea2**

♦ Encore un sélecteur de champ...

Nous avons vu les différents champs sur lequels on peut agir, dans ou depuis un registre de travail.

Il existe, pour certaines instructions, une possibilité qui n'a pas été évoquée jusqu'ici. Elle consiste à indiquer, par la donnée d'un quartet **q** (compris donc entre 0 et #Fh=15) le champ allant du quartet n°**0** du registre de travail au quartet n°**q** de ce registre.

La longueur de ce champ est donc de **q+1** quartets. Tout se passe dans ce cas comme si l'on utilisait le champ **WP** après avoir donné la valeur **q** au registre **P** (sauf qu'ici le registre **P** n'est pas modifié).

Le quartet **q** est effectivement présent dans le code-machine correspondant à cette instruction. Dans le mnémonique, il est d'usage de faire apparaître le nombre de quartets de ce champ, c'est-à-dire l'entier **q+1**.

Voici un exemple d'instruction utilisant ce principe:

Mnémonique	Description
A=DAT1 *q+1*	Charge **q+1** quartets dans le registre **A** (0≤q≤15) Ces quartets sont lus à partir de l'adresse pointée par le registre **D1**, et sont stockés dans les quartets **0** à **q** du registre **A**. Le code machine est: **15Bq**

◆ Arithmétique non signée (en héxadécimal)

Imaginons un "champ" de n+1 bits consécutifs, numérotés de b_0, pour le bit de poids le plus faible, à b_n, pour le bit de poids le plus fort.

Considérons le mot formé par la juxtaposition $b_n b_{n-1} \ldots b_2 b_1 b_0$ de ces n+1 bits.

Ce mot permet de coder tous les entiers positifs ou nuls, depuis 0 jusqu'à $2^{n+1}-1$. Dans ce cas, en effet, le bit b_k correspond (s'il est armé), à la valeur 2^k.

Le nombre ainsi codé est: $\mathbf{n = b_n*2^n + b_{n-1}*2^{n-1} \ldots + b_2*2^2 + b_1*2 + b_0}$

La valeur maximum est bien: $\mathbf{2^n + 2^{n-1} + \ldots + 2^2 + 2 + 1 = 2^{n+1}-1}$.

Par exemple, un champ de 4 bits (un quartet) permet de coder tous les entiers positifs de 0 à $2^4-1=15$.

De même un champ de 8 bits (2 quartets, un octet) permet de coder les entiers de 0 à $2^8-1=255$, et, par exemple, le nombre représenté par les bits `10110011` vaut $2^7+2^5+2^4+2+1=179$.

Si on fixe le nombre n+1 de bits des "mots" considérés, il est alors aisé d'effectuer une addition ou une soustraction sur deux nombres positifs codés de cette manière. Une telle opération s'accompagne éventuellement d'une retenue finale (dans le cas d'une addition si la somme est supérieure ou égale à 2^{n+1}, et dans le cas d'une soustraction si la différence est négative).

Exemples: (sur 8 bits). Le bit entre parenthèses indique la retenue finale.

```
   01101110          11101110          10110001          01110111
+  01011011       +  11011011       -  01110111       -  10110001
-----------       -----------       -----------       -----------
(0)11001001       (1)11001001       (0)00111010       (1)11000110
```

Pour comparer deux nombres **m** et **n** codés de cette manière, **m** est strictement inférieur à **n** si la différence **m-n** conduit à une retenue finale.

La manipulation d'entiers positifs représentés de la façon ci-dessus s'appelle ***arithmétique non signée***. Toutes les opérations héxadécimales du "Saturn" où il s'agit de comparer le même champ de deux registres de travail, ou de les ajouter, les retrancher, etc... se font en arithmétique non signée.

Autrement dit, pour le "Saturn", le contenu d'un champ d'un registre de travail est toujours positif ou nul.

Quand une opération sur un champ d'un registre de travail conduit à une retenue, celle-ci est stockée dans le bit CARRY. En tout cas cette retenue ne se propage pas (dans le registre de travail) au-delà du champ spécifié.

Le caractère "autonome" de chaque champ d'un des registres **A**, **B**, **C** ou **D** se retrouve également dans les opérations de décalage (ou de rotation) de bit, vers la gauche ou vers la droite. Quand on décale d'un bit vers la droite le contenu du champ XS (quartet n°2) du registre **A**, par exemple, le bit le plus à droite de ce champ ne va pas dans le champ B (son voisin immédiat sur la droite).

◆ Arithmétique signée (en héxadécimal)

La méthode précédente ne permet que de coder des entiers positifs ou nuls.

Pour représenter un nombre positif ou négatif sur un champ de n+1 bits consécutifs $b_n b_{n-1}....b_2 b_1 b_0$, on attribue au bit de poids le plus fort b_n le rôle de bit de signe: s'il est armé (resp. désarmé) l'entier est négatif (resp. positif). Les autres bits conservent la même signification qu'en arithmétique non signée.

De cette manière, l'entier n codé par $b_n b_{n-1}....b_2 b_1 b_0$ a la valeur suivante:

$$n = -b_n*2^n + b_{n-1}*2^{n-1} .... + b_2*2^2 + b_1*2 + b_0$$

Si le bit de signe b_n est désarmé, on code ainsi tous les entiers de 0 jusqu'à 2^n-1. Si le bit de signe b_n est armé, on code tous les entiers de -2^n à $-2^n+(2^n-1)=-1$.

Par exemple, sur 8 bits (2 quartets), on obtient tous les entiers de `10000000`=-128 à `01111111`=127, en passant par `10000001`=-127, `10000010`=-126, ..., `11111110`=-2, `11111111`=-1, `00000000`=0, etc...

Sur 3 quartets, on obtient tous les nombres de #800h=-2048 à #7FFh=2047.

Sur 4 quartets, on obtient tous les nombres de #8000h=-32768 à #7FFFh=32767.

Pour prendre l'opposé d'un nombre codé de cette manière, il ne suffit pas d'inverser le bit de signe; il faut procéder comme suit: inverser l'état de tous les bits, puis ajouter 1 au résultat.

Par exemple, sur huit bits: le nombre `01110101` vaut 117. Son opposé est obtenu en inversant tous les bits et en ajoutant 1: il vaut `10001011`=-117.

Ce codage des entiers positifs ou négatifs s'appelle le mode "***complément à 2***", ou encore "***arithmétique signée***".

Les opérations arithmétiques du "Saturn" se font toujours dans le mode "arithmétique non signée". Seuls les offsets (décalages) nécessaires aux opérations de branchement relatif (de type *Goto* ou *Gosub*) sont codés sur 2, 3 ou 4 quartets, en mode "complément à 2".

Il existe néanmoins des opérations qui permettent de prendre l'opposé du contenu d'un champ d'un registre de travail.

Considérons par exemple les instructions "**A=−A** *B*" et "**A=−A−1** *B*" appliquées au champ B (les 2 quartets de poids faible) du registre **A**.

L'opération "**A=−A** *B*" effectue le complément à 2 du champ B (on inverse de tous les bits, puis on ajoute 1: celà revient à considérer le contenu de ce champ comme un nombre signé sur deux quartets, et à en prendre l'opposé).

L'opération "**A=−A−1** *B*" effectue une inversion de tous les bits du champ B (on parle de "*complémentation à 1*"). On comprend dès lors, d'après ce qui précède, l'origine du mnémonique. Peut être un mnémonique du type "**A=NOT(A)** *B*" serait-il plus explicite.

◆ Arithmétique BCD

Par défaut, les opérations arithmétiques du "Saturn" s'effectuent en héxadécimal. Tous les bits d'un champ sont alors utilisés de la même manière. Le microprocesseur de notre HP48 est cependant capable d'effectuer très rapidement des calculs en base 10 (c'est le mode ***BCD***: *binaire codé décimal*).

L'instruction **SETDEC** permet de passer du mode héxa au mode BCD, et l'instruction **SETHEX** permet de revenir au mode par défaut.

En BCD, on ne peut raisonner que par quartet (groupement de 4 bits). Un tel quartet contiendra un nombre de 0 à 9=1001 (codé comme si on était en héxadécimal).

La spécificité du mode BCD (hormis le fait qu'on limite chaque quartet du champ spécifié à ne contenir qu'un nombre de 0 à 9) tient dans la façon dont sont effectués les opérations arithmétiques et les reports de retenue.

Tout se passe, très simplement, "comme sur le papier".

Une retenue est générée dès que la somme de deux quartets est supérieure à 9 (et la retenue va sur le quartet suivant), ou dès qu'une différence de deux quartets conduit théoriquement à un résultat négatif.

Par exemple, sur 2 quartets (la retenue finale est entre parenthèses):

```
   01100111      c'est à dire        67
+  10010110       en décimal     +   96
-----------          ==>         -----
(1)01100011                     (1)63
```

Il y a donc une propogation éventuelle de retenues. La retenue finale (si on travaille sur un champ spécifié d'un registre de travail) n'est pas répercutée sur le quartet suivant, mais va dans le registre CARRY.

Le mode BCD conduit également à des comportements particuliers pour les instructions du type "*complément à 2*" ou "*complément à 1*", qui ont été évoquées précédemment. Il faut plutôt parler ici de "*complément à 10*" ou de "*complément à 9*".

Exemple: Supposons que le champ B du registre A contienne #62h.

- L'instruction "**A=−A−1** *B*" (complément à 9), effectue le complément à 9 des deux quartets du champ. On obtient alors #37h. Cette opération est involutive, puisqu'appliquée à nouveau, elle remet le champ B de A dans son état initial.

- L'instruction "**A=−A** *B*" (complément à 10), effectue le même travail que l'instruction précédente, mais ajoute 1 ensuite. D'une façon plus simple, le quartet de poids le plus faible se voit complémenté à 10 (avec une retenue s'il est nul), les autres quartets restant complémentés à 9. On passe ainsi de #62h à #38h, puis de #38h à #62h. Là encore l'opération est involutive.

Plus généralement quand un champ de *n* quartets (en BCD) est complémenté à 9, la somme de l'ancien et du nouveau champ est égale à 99...9 (*n* fois). Quand il est complémenté à 10, cette somme est égale à 00...0 (*n* fois, avec une retenue finale).

Remarque: le mode héxadécimal/décimal n'est lisible dans aucun registre. Pour connaître le mode en cours, exécutez par exemple la séquence suivante, qui arme le bit CARRY si et seulement si on est en BCD: **LA(1)** ***9***, **A=A+1** ***P***.

◆ Quelques conseils:

Quand on programme en langage-machine, il faut être très attentif aux conditions de sortie du Rpl ambiant (vers le code-machine) ou de retour à ce Rpl. Un certain nombre de précautions doivent être de mise, notamment si vous faites appel à des routines de la ROM de la HP48.

- Pensez à sauvegarder les registres sensibles **B**,**D**,**D0** et **D1** en tout début de code, avant de les restaurer à la fin. Pensez également que certaines routines de la ROM procèdent elles-mêmes à ces opérations (comme les sauvegardes se font à adresses fixes, seule la dernière compte).
- Soyez certain de quitter votre code-machine en mode **HEX**, notamment si vous utilisez des routines en ROM qui agissent sur des nombres réels.
- Vérifiez également que lorsque vous rendez la main au Rpl-Système, le contenu du registre **P** est égal à 0. Les erreurs dues à un mauvais contrôle de **P** sont également fréquentes avec les instructions **LA** et **LC** (chargement de constantes dans les registres **A** et **C**).
- Souvenez vous que le registre **RSTK** ne comporte que 8 niveaux. Deux de ces niveaux peuvent être utilisés à tout moment par la 48 pour gérer les interruptions. N'abusez donc pas des **GOSUB** emboîtés, ou de l'instruction **RSTK=C** (qui empile le contenu du champ ***A*** de **C** sur le registre **RSTK**). Au mieux vous obtiendriez un arrêt-système.
- Parmi les différents champs des registres de travail, le champ ***A*** est sans aucun doute le plus utilisé (notamment parce qu'il produit un code plus compact et donc plus rapide). Pensez cependant que les différents champs sont "autonômes": si vous stockez un octet dans le champ ***B*** de **C**, par exemple (quartets n°0 et 1), puis si vous utilisez le champ ***A*** de **C**, n'oubliez pas les quartets n°2,3,4...
- Utilisez les bits 0 à 11 du registre **ST** pour stocker les résultats de tests que vous pourriez être amené à utiliser plus tard dans votre code. Le bit **Carry** a en effet un caractère assez fugitif qui le réserve aux tests suivis immédiatement du traitement correspondant.

LES INSTRUCTIONS DU SATURN

Ce chapitre est consacré aux mnémoniques du Saturn (instructions que vous utiliserez dans vos programmes-source). Ce sont, en principe, les mnémoniques reconnus par Hewlett-Packard.

J'ai cherché à produire une présentation claire et efficace de ces instructions, classées ici suivant le type de registre concerné et suivant la nature de l'opération réalisée.

Ce chapitre se termine par un dictionnaire alphabétique, ordonné d'abord dans l'ordre des mnémoniques, puis dans celui des codes-machine.

◆ INTRODUCTION

Dans ce chapitre, nous allons passer en revue toutes les instructions du microprocesseur "Saturn" (autant les codes-machine associés à ces instructions que les mnémoniques qui leur correspondent).

Le nombre de ces instructions est très important (notamment si l'on tient compte des variantes découlant des spécifications de champ, dans le cas d'opérations sur les registres de travail).

Il a donc fallu adopter quelques principes de présentation.

On pouvait par exemple imaginer de présenter toutes les instructions par ordre alphabétique du mnémonique (ce qui est commode pour celui qui veut programmer "à la main" en assembleur), ou par ordre "alphabétique" des codes-machine (utile pour qui veut désassembler une succession de codes-hexadécimaux).

Ces deux méthodes n'ont pu être retenues dans la mesure où elles sont mal adaptées au lecteur novice, pour lequel une approche plus didactique est nécessaire.

On peut présenter les instructions en fonction du registre sur lequel elles s'appliquent, ou selon le type de fonction exécutée. C'est une combinaison de ces deux méthodes qui a été choisie ici.

Pour répondre à tous les besoins, on a fait figurer en annexe le dictionnaire de toutes les instructions du Saturn, classées par ordre alphabétique du mnémonique, puis par ordre alphabétique du code-machine.

Les mnémoniques présentés seront utilisés par les programmes **ASM48** (assembleur en Externals) et **ASMPC** (assembleur pour compatibles PC), qui seront présentés dans les deux chapitres suivants.

◆ Bit CARRY et nombre de cycles:

Pour chacune des instructions du Saturn, nous indiquerons si elle affecte ou non le bit **CARRY** (très utile pour les opérations de branchement qui pourraient suivre) et quel est sa durée d'exécution, exprimée en nombre de cycles du microprocesseur.

Il faut savoir que le "Saturn" fonctionne avec une fréquence d'environ 2 Mhz, c'est à dire à un rythme de deux millions de cycles par seconde.

Si une instruction s'exécute en 20 cycles, par exemple, celà signifie que la durée de cette instruction est de 1/100000ème de seconde. Ces informations seront utiles à ceux qui recherchent la plus grande vitesse d'exécution possible.

◆ Opérations sur les registres de travail

Mise à Zéro d'un champ d'un registre de travail (A,B,C ou D)
Le registre CARRY n'est pas modifié
Temps d'exécution: 7 cycles pour le champ A
3+n cycles pour un autre champ de n quartets
Rappel des spécifications de champ "b":
b = (8,9,A,B,C,D,E,F) ◄==► *field* = (P,WP,XS,X,S,M,B,W)

Mnémonique		Description	Code
A=0	*field*	Met à zéro le champ "*field*" du registre **A** *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **D0** **A*b*0**
B=0	*field*	Met à zéro le champ "*field*" du registre **B** *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **D1** **A*b*1**
C=0	*field*	Met à zéro le champ "*field*" du registre **C** *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **D2** **A*b*2**
D=0	*field*	Met à zéro le champ "*field*" du registre **D** *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **D3** **A*b*3**

Décrémentation d'un champ d'un registre de travail (A,B,C ou D)
Le registre CARRY est affecté: il est armé si le contenu initial du champ est zéro, désarmé sinon.
Temps d'exécution: 7 cycles pour le champ A
3+n cycles pour un autre champ de n quartets
Rappel des spécifications de champ "a":
a = (0,1,2,3,4,5,6,7) ◄==► *field* = (P,WP,XS,X,S,M,B,W)

Mnémonique		Description	Code
A=A-1	*field*	Décrémente le champ "*field*" du registre **A** *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **CC** **A*a*C**
B=B-1	*field*	Décrémente le champ "*field*" du registre **B** *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **CD** **A*a*D**
C=C-1	*field*	Décrémente le champ "*field*" du registre **C** *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **CE** **A*a*E**
D=D-1	*field*	Décrémente le champ "*field*" du registre **D** *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **CF** **A*a*F**

Incrémentation d'un champ d'un registre de travail (A,B,C ou D)
Le registre CARRY est affecté: il est armé si le nouveau contenu du champ est zéro, désarmé sinon.
Temps d'exécution: 7 cycles pour le champ A
3+n cycles pour un autre champ de n quartets
Rappel des spécifications de champ "a":
a = (0,1,2,3,4,5,6,7) ◄==► *field* = (P,WP,XS,X,S,M,B,W)

Mnémonique		Description	Code
A=A+1	*field*	Incrémente le champ "*field*" du registre **A**	
		field = A -->	**E4**
		field = P,WP,XS,X,S,M,B,W -->	**Ba4**
B=B+1	*field*	Incrémente le champ "*field*" du registre **B**	
		field = A -->	**E5**
		field = P,WP,XS,X,S,M,B,W -->	**Ba5**
C=C+1	*field*	Incrémente le champ "*field*" du registre **C**	
		field = A -->	**E6**
		field = P,WP,XS,X,S,M,B,W -->	**Ba6**
D=D+1	*field*	Incrémente le champ "*field*" du registre **D**	
		field = A -->	**E7**
		field = P,WP,XS,X,S,M,B,W -->	**Ba7**

Complément à 2 (en hexadécimal) ou à 10 (en décimal) d'un champ d'un registre de travail.
Le registre CARRY est affecté: il est armé si le contenu initial du champ n'est pas égal à zéro, désarmé sinon.
Temps d'exécution: 7 cycles pour le champ A
3+n cycles pour un autre champ de n quartets
Rappel des spécifications de champ "b":
b = (8,9,A,B,C,D,E,F) ◄==► *field* = (P,WP,XS,X,S,M,B,W)

Mnémonique		Description	Code
A=-A	*field*	Complémente le champ "field" du registre **A**	
		field = A -->	**F8**
		field = P,WP,XS,X,S,M,B,W -->	**Bb8**
B=-B	*field*	Complémente le champ "field" du registre **B**	
		field = A -->	**F9**
		field = P,WP,XS,X,S,M,B,W -->	**Bb9**
C=-C	*field*	Complémente le champ "field" du registre **C**	
		field = A -->	**FA**
		field = P,WP,XS,X,S,M,B,W -->	**BbA**
D=-D	*field*	Complémente le champ "field" du registre **D**	
		field = A -->	**FB**
		field = P,WP,XS,X,S,M,B,W -->	**BbB**

Complément à 1 (en hexadécimal) ou à 9 (en décimal) d'un champ d'un registre de travail.
Le registre CARRY est systématiquement désarmé
Temps d'exécution: 7 cycles pour le champ A
3+n cycles pour un autre champ de n quartets
Rappel des spécifications de champ "b":
b = (8,9,A,B,C,D,E,F) ◄==► *field* = (P,WP,XS,X,S,M,B,W)

Mnémonique	Description	Code
A=-A-1 *field*	Complémente le champ "field" du registre **A** *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **FC** **B***b***C**
B=-B-1 *field*	Complémente le champ "field" du registre **B** *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **FD** **B***b***D**
C=-C-1 *field*	Complémente le champ "field" du registre **C** *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **FE** **B***b***E**
D=-D-1 *field*	Complémente le champ "field" du registre **D** *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **FF** **B***b***F**

Échange des deux mêmes champs de deux registres de travail.
Possible uniquement sur les couples (A,B), (B,C), (C,A), (D,C).
Le bit CARRY n'est pas affecté.
Temps d'exécution: 7 cycles pour le champ A
3+n cycles pour un autre champ de n quartets
Rappel des spécifications de champ "b":
b = (8,9,A,B,C,D,E,F) ◄==► *field* = (P,WP,XS,X,S,M,B,W)

Mnémonique	Description	Code
ABEX *field*	Echange les champs "field" des registres **A,B** *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **DC** **A***b***C**
BCEX *field*	Echange les champs "field" des registres **B,C** *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **DD** **A***b***D**
CAEX *field*	Echange les champs "field" des registres **A,C** *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **DE** **A***b***E**
DCEX *field*	Echange les champs "field" des registres **C,D** *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **DF** **A***b***F**

Remarque:

Dans le tableau précédent, il aurait été possible d'inclure des mnémoniques **BAEX**, **CBEX**, **ACEX**, et **CDEX**, synonymes des précédents.

Tout dépend de l'assembleur que vous utilisez. Sachez que les programmes **ASM48** et **ASMPC**, présentés dans les deux chapitres suivants se limitent à la seule possibilité retenue ici (pour des questions de taille du programme, essentiellement).

Quand deux registres de travail différents sont arguments d'une même opération, j'ai toujours privilégié l'ordre suivant:

(A,B), (B,C) (C,A), (D,C).

Copie d'un champ d'un registre de travail sur le même champ d'un autre registre de travail.
Les seules directions possibles de cette copie sont:
A ◄=► B, B ◄=► C, C ◄=► D, et A ◄=► C
Le bit CARRY n'est pas affecté.
Temps d'exécution: 7 cycles pour le champ A
3+n cycles pour un autre champ de n quartets
Rappel des spécifications de champ "b":
b = (8,9,A,B,C,D,E,F) ◄==► *field* = (P,WP,XS,X,S,M,B,W)

Mnémonique	Description	Code
A=B *field*	Copie le champ **B:***field* sur le champ **A:***field* *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **D4** **Ab4**
B=C *field*	Copie le champ **C:***field* sur le champ **B:***field* *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **D5** **Ab5**
C=A *field*	Copie le champ **A:***field* sur le champ **C:***field* *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **D6** **Ab6**
D=C *field*	Copie le champ **C:***field* sur le champ **D:***field* *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **D7** **Ab7**
B=A *field*	Copie le champ **A:***field* sur le champ **B:***field* *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **D8** **Ab8**
C=B *field*	Copie le champ **B:***field* sur le champ **C:***field* *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **D9** **Ab9**
A=C *field*	Copie le champ **C:***field* sur le champ **A:***field* *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **DA** **AbA**
C=D *field*	Copie le champ **D:***field* sur le champ **C:***field* *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **DB** **AbB**

Addition d'un champ d'un registre de travail au même champ d'un autre (ou du même) registre de travail.
Les seules directions possibles de cette addition sont:
A◂=▸A, B◂=▸B, C◂=▸C, D◂=▸D, A◂=▸B, B◂=▸C, C◂=▸D, et A◂=▸C
Le bit CARRY est armé il y a un "*overflow*", désarmé sinon.
Temps d'exécution: 7 cycles pour le champ A
3+n cycles pour un autre champ de n quartets
Rappel des spécifications de champ "a":
a = (0,1,2,3,4,5,6,7) ◂==▸ *field* = (P,WP,XS,X,S,M,B,W)

Mnémonique	Description	Code
A=A+B *field*	Additionne le champ **B:***field* au champ **A:***field*	
	field = A -->	**C0**
	field = P,WP,XS,X,S,M,B,W -->	**Aa0**
B=B+C *field*	Additionne le champ **C:***field* au champ **B:***field*	
	field = A -->	**C1**
	field = P,WP,XS,X,S,M,B,W -->	**Aa1**
C=C+A *field*	Additionne le champ **A:***field* au champ **C:***field*	
	field = A -->	**C2**
	field = P,WP,XS,X,S,M,B,W -->	**Aa2**
D=D+C *field*	Additionne le champ **C:***field* au champ **D:***field*	
	field = A -->	**C3**
	field = P,WP,XS,X,S,M,B,W -->	**Aa3**
A=A+A *field*	Double le champ A:*field*. *field* = A -->	**C4**
	field = P,WP,XS,X,S,M,B,W -->	**Aa4**
B=B+B *field*	Double le champ **B:***field*. *field* = A -->	**C5**
	field = P,WP,XS,X,S,M,B,W -->	**Aa5**
C=C+C *field*	Double le champ **C:***field*. *field* = A -->	**C6**
	field = P,WP,XS,X,S,M,B,W -->	**Aa6**
D=D+D *field*	Double le champ **D:***field*. *field* = A -->	**C7**
	field = P,WP,XS,X,S,M,B,W -->	**Aa7**
B=A+B *field*	Additionne le champ **A:***field* au champ **B:***field*	
	field = A -->	**C8**
	field = P,WP,XS,X,S,M,B,W -->	**Aa8**
C=B+C *field*	Additionne le champ **B:***field* au champ **C:***field*	
	field = A -->	**C9**
	field = P,WP,XS,X,S,M,B,W -->	**Aa9**
A=C+A *field*	Additionne le champ **C:***field* au champ **A:***field*	
	field = A -->	**CA**
	field = P,WP,XS,X,S,M,B,W -->	**AaA**
C=D+C *field*	Additionne le champ **D:***field* au champ **C:***field*	
	field = A -->	**CB**
	field = P,WP,XS,X,S,M,B,W -->	**AaB**

Soustraction des deux mêmes champs de deux registres de travail.
Le bit CARRY est armé s'il y a une retenue (différence négative).
Temps d'exécution: 7 cycles pour le champ A
3+n cycles pour un autre champ de n quartets
Rappel des spécifications de champ "a":
a = (0,1,2,3,4,5,6,7) ◄==► *field* = (P,WP,XS,X,S,M,B,W)

Mnémonique	Description	Code
A=A-B *field*	(**A:***field*)-(**B:***field*) va dans le champ **A:***field* Soustraction de registres *field* = A --> **E0** *field* = P,WP,XS,X,S,M,B,W --> **Ba0**
B=B-C *field*	(**B:***field*)-(**C:***field*) va dans le champ **B:***field* *field* = A --> **E1** *field* = P,WP,XS,X,S,M,B,W --> **Ba1**
C=C-A *field*	(**C:***field*)-(**A:***field*) va dans le champ **C:***field* *field* = A --> **E2** *field* = P,WP,XS,X,S,M,B,W --> **Ba2**
D=D-C *field*	(**D:***field*)-(**C:***field*) va dans le champ **D:***field* *field* = A --> **E3** *field* = P,WP,XS,X,S,M,B,W --> **Ba3**
B=B-A *field*	(**B:***field*)-(**A:***field*) va dans le champ **B:***field* *field* = A --> **E8** *field* = P,WP,XS,X,S,M,B,W --> **Ba8**
C=C-B *field*	(**C:***field*)-(**B:***field*) va dans le champ **C:***field* *field* = A --> **E9** *field* = P,WP,XS,X,S,M,B,W --> **Ba9**
A=A-C *field*	(**A:***field*)-(**C:***field*) va dans le champ **A:***field* *field* = A --> **EA** *field* = P,WP,XS,X,S,M,B,W --> **BaA**
C=C-D *field*	(**C:***field*)-(**D:***field*) va dans le champ **C:***field* *field* = A --> **EB** *field* = P,WP,XS,X,S,M,B,W --> **BaB**
A=B-A *field*	(**B:***field*)-(**A:***field*) va dans le champ **A:***field* *field* = A --> **EC** *field* = P,WP,XS,X,S,M,B,W --> **BaC**
B=C-B *field*	(**C:***field*)-(**B:***field*) va dans le champ **B:***field* *field* = A --> **ED** *field* = P,WP,XS,X,S,M,B,W --> **BaD**
C=A-C *field*	(**A:***field*)-(**C:***field*) va dans le champ **C:***field* *field* = A --> **EE** *field* = P,WP,XS,X,S,M,B,W --> **BaE**
D=C-D *field*	(**C:***field*)-(**D:***field*) va dans le champ **D:***field* *field* = A --> **EF** *field* = P,WP,XS,X,S,M,B,W --> **BaF**

"ET logique" ou "OU logique" (bit à bit) d'un champ d'un registre de travail sur le même champ d'un autre registre de travail.
Directions possibles: A◄=►B, B◄=►C, A◄=►C, et C◄=►D
Le bit CARRY n'est pas affecté par cette opération.
Temps d'exécution: 4+n cycles (n = nombre de quartets du champ)
Rappel: *a* = (0,1,2,3,4,5,6,7,F) ◄==► *field* = (P,WP,XS,X,S,M,B,W,A)

Mnémonique	Description	Code
A=A&B *field*	"ET" du champ **B:***field* sur le champ **A:***field* *field* = P,WP,XS,X,S,M,B,W,A -->	**0Ea0**
B=B&C *field*	"ET" du champ **C:***field* sur le champ **B:***field* *field* = P,WP,XS,X,S,M,B,W,A -->	**0Ea1**
C=C&A *field*	"ET" du champ **A:***field* sur le champ **C:***field* *field* = P,WP,XS,X,S,M,B,W,A -->	**0Ea2**
D=D&C *field*	"ET" du champ **C:***field* sur le champ **D:***field* *field* = P,WP,XS,X,S,M,B,W,A -->	**0Ea3**
B=A&B *field*	"ET" du champ **A:***field* sur le champ **B:***field* *field* = P,WP,XS,X,S,M,B,W,A -->	**0Ea4**
C=B&C *field*	"ET" du champ **B:***field* sur le champ **C:***field* *field* = P,WP,XS,X,S,M,B,W,A -->	**0Ea5**
A=C&A *field*	"ET" du champ **C:***field* sur le champ **A:***field* *field* = P,WP,XS,X,S,M,B,W,A -->	**0Ea6**
C=D&C *field*	"ET" du champ **D:***field* sur le champ **C:***field* *field* = P,WP,XS,X,S,M,B,W,A -->	**0Ea7**
A=A!B *field*	"OU" du champ **B:***field* sur le champ **A:***field* *field* = P,WP,XS,X,S,M,B,W,A -->	**0Ea8**
B=B!C *field*	"OU" du champ **C:***field* sur le champ **B:***field* *field* = P,WP,XS,X,S,M,B,W,A -->	**0Ea9**
C=C!A *field*	"OU" du champ **A:***field* sur le champ **C:***field* *field* = P,WP,XS,X,S,M,B,W,A -->	**0EaA**
D=D!C *field*	"OU" du champ **C:***field* sur le champ **D:***field* *field* = P,WP,XS,X,S,M,B,W,A -->	**0EaB**
B=A!B *field*	"OU" du champ **A:***field* sur le champ **B:***field* *field* = P,WP,XS,X,S,M,B,W,A -->	**0EaC**
C=B!C *field*	"OU" du champ **B:***field* sur le champ **C:***field* *field* = P,WP,XS,X,S,M,B,W,A -->	**0EaD**
A=C!A *field*	"OU" du champ **C:***field* sur le champ **A:***field* *field* = P,WP,XS,X,S,M,B,W,A -->	**0EaE**
C=D!C *field*	"OU" du champ **D:***field* sur le champ **C:***field* *field* = P,WP,XS,X,S,M,B,W,A -->	**0EaF**

Opérations sur les registres de travail

Voici maintenant les instructions qui permettent d'ajouter ou de retrancher une constante aux registres **A**,**B**,**C**, et **D** (autre que 1, pour laquelle il y a des codes spécifiques).

Cette constante doit être codée sur un quartet, ce qui limite les valeurs à [2..16].

Notez qu'exceptionnellement, certaines spécifications de champ ne sont pas permises ici (les champs **P**, **WP**, **XS**, **S**).

Addition ou soustraction d'une constante (comprise entre 2 et 16) à un champ d'un registre de travail (A,B,C ou D).
Le bit CARRY est affecté par cette opération (armé si celle-ci conduit à une retenue finale, et désarmé sinon).
Temps d'exécution: 6+n cycles (n = nombre de quartets du champ)
Attention: Ici le code *a* de spécification de champ est limité à:
a = (3,5,6,7,F) ◄==► *field* = (X,M,B,W,A)
Dans les notations ci-dessous, *n* est un quartet, ou encore une valeur comprise entre 0 et 15. Pour prendre un exemple, l'instruction **C=C+7 A** (codée **818F26**) a pour effet d'ajouter 7 au champ A du registre **C**.

Mnémonique	Description	Code
A=A+*(n+1)* *field*	Additionne *(n+1)* au champ **A:***field* *field* = X,S,M,B,W,A -->	**818***a***0***n*
B=B+*(n+1)* *field*	Additionne *(n+1)* au champ **B:***field* *field* = X,S,M,B,W,A -->	**818***a***1***n*
C=C+*(n+1)* *field*	Additionne *(n+1)* au champ **C:***field* *field* = X,S,M,B,W,A -->	**818***a***2***n*
D=D+*(n+1)* *field*	Additionne *(n+1)* au champ **D:***field* *field* = X,S,M,B,W,A -->	**818***a***3***n*
A=A-*(n+1)* *field*	Soustrait *(n+1)* au champ **A:***field* *field* = X,S,M,B,W,A -->	**818***a***8***n*
B=B-*(n+1)* *field*	Soustrait *(n+1)* au champ **B:***field* *field* = X,S,M,B,W,A -->	**818***a***9***n*
C=C-*(n+1)* *field*	Soustrait *(n+1)* au champ **C:***field* *field* = X,S,M,B,W,A -->	**818***a***A***n*
D=D-*(n+1)* *field*	Soustrait *(n+1)* au champ **D:***field* *field* = X,S,M,B,W,A -->	**818***a***B***n*

Chargement d'une constante, dans l'un des registres A ou C
Les instructions **LA(*n+1*)** et **LC(*n+1*)** permettent de charger jusqu'à 16 quartets dans le registre **A** ou **C**. Le chargement s'effectue à partir du quartet numéro *p*, où *p* est le contenu du registre **P**.
Le chargement se poursuit sur les quartets *p+1*,*p+2*, de **A** ou **C**.
Au besoin, le chargement de **A** ou de **C** se poursuit par un retour au quartet de position 0 dans le registre.
Le bit CARRY n'est pas affecté par cette opération.
Durée d'exécution: 4+n cycles pour **LC(*n+1*)**,6+n cycles pour **LA(*n+1*)** où n est le nombre de quartets chargés, diminué de 1.

Mnémonique	**LC(*n+1*)** $h_n...h_0$
Code	**3***n*$h_0...h_n$
Description	Charge successivement les n+1 quartets h_0, $h_1...h_n$, (0≤n≤#Fh), dans le registre **C**, à partir de la position définie par le contenu p du registre **P**. Le quartet h_0 va dans le quartet N°p du registre, puis h1 va dans le registre N°(p+1), etc... avec retour éventuel à la position N°0.
Exemple	Si le contenu initial de C est 1111111111111111 et le contenu de P est #Dh=13, alors l'instruction **LC(7) 9ABCDEF** (codée **36FEDCBA9**) transforme le contenu de C en DEF1111111119ABC
Mnémonique	**LA(*n+1*)** $h_n...h_0$
Code	**8082***n*$h_0...h_n$
Description	Voir description de **LC(*n+1*)**, en remplaçant **C** par **A**.
Exemple	Si le contenu initial de **A** est 0000000000000000 et le contenu de **P** est 1, alors l'instruction **LA(4) CDEF** (codée **80823FEDC**) transforme le contenu de **A** en 00000000000CDEF0

Positionnement d'un bit du registre A ou du registre C. Les seuls bits accessibles sont les bits n°0 à 15 (soit quatre quartets)
Le registre CARRY n'est pas affecté.

Mnémonique		Description	Code
ABIT=0	*n*	Désarme le bit *n* du registre **A** (0≤*n*≤#Fh)	**8084***n*
ABIT=1	*n*	Arme le bit *n* du registre **A** (0≤*n*≤#Fh)	**8085***n*
CBIT=0	*n*	Désarme le bit *n* du registre **C** (0≤*n*≤#Fh)	**8088***n*
CBIT=1	*n*	Arme le bit *n* du registre **C** (0≤*n*≤#Fh)	**8089***n*

Décalage d'un quartet vers la gauche (produit par 16) ou vers la droite (division par 16) d'un champ d'un registre de travail.
Le quartet sortant (à gauche ou à droite) est perdu. Le quartet entrant est nul.
Dans un décalage à gauche, le "Sticky Bit" **SB** n'est pas affecté.
Dans le cas d'un décalage à droite (division par 16) le bit **SB** est armé si le quartet sortant est non nul, et inchangé sinon.
Le bit CARRY n'est pas affecté.
Temps d'exécution: 7 cycles pour le champ A
3+n cycles pour un autre champ de n quartets
Rappel des spécifications de champ "b":
b = (8,9,A,B,C,D,E,F) ◄==► *field* = (P,WP,XS,X,S,M,B,W)

Mnémonique		Description	Code
ASL	*field*	Multiplie par 16 le contenu du champ **A:***field* *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **F0** **Bb0**
BSL	*field*	Multiplie par 16 le contenu du champ **B:***field* *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **F1** **Bb1**
CSL	*field*	Multiplie par 16 le contenu du champ **C:***field* *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **F2** **Bb2**
DSL	*field*	Multiplie par 16 le contenu du champ **D:***field* *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **F3** **Bb3**
ASR	*field*	Divise par 16 le contenu du champ **A:***field* *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **F4** **Bb4**
BSR	*field*	Divise par 16 le contenu du champ **B:***field* *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **F5** **Bb5**
CSR	*field*	Divise par 16 le contenu du champ **C:***field* *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **F6** **Bb6**
DSR	*field*	Divise par 16 le contenu du champ **D:***field* *field* = A --> *field* = P,WP,XS,X,S,M,B,W -->	 **F7** **Bb7**

Rotation d'un quartet vers la gauche, ou vers la droite, sur la totalité d'un registre de travail (A,B,C ou D)
Dans une rotation vers la gauche, le "sticky bit" **SB** est inchangé.
Dans le cas d'une rotation vers la droite, le bit **SB** est armé si le quartet qui passe de la position 0 à la position 15 est non nul, et il est inchangé sinon.
Le bit CARRY n'est pas affecté.
Temps d'exécution: 21 cycles

Mnémonique	Description	Code
ASLC	Rotation du registre **A**, d'un quartet à gauche	**810**
BSLC	Rotation du registre **B**, d'un quartet à gauche	**811**
CSLC	Rotation du registre **C**, d'un quartet à gauche	**812**
DSLC	Rotation du registre **D**, d'un quartet à gauche	**813**
ASRC	Rotation du registre **A**, d'un quartet à droite	**814**
BSRC	Rotation du registre **B**, d'un quartet à droite	**815**
CSRC	Rotation du registre **C**, d'un quartet à droite	**816**
DSRC	Rotation du registre **D**, d'un quartet à droite	**817**

Décalage d'un bit vers la droite (division par 2) de la totalité d'un registre de travail (A,B,C ou D).
Le bit sortant (à droite) est perdu. Si ce bit est égal à 1, alors le "Sticky Bit" **SB** est armé. Sinon le bit **SB** est inchangé.
Le bit entrant (à gauche) est égal à 0.
Le bit CARRY n'est pas affecté.
Pour un décalage à gauche (multiplication par 2) on utilisera les instructions du type "A=A+A W", etc...
Temps d'exécution: 20 cycles

Mnémonique	Description	Code
ASRB	Décalage, d'un bit à droite, du registre **A**	**81C**
BSRB	Décalage, d'un bit à droite, du registre **B**	**81D**
CSRB	Décalage, d'un bit à droite, du registre **C**	**81E**
DSRB	Décalage, d'un bit à droite, du registre **D**	**81F**

Décalage d'un bit vers la droite (division par 2) d'un champ d'un registre de travail (A,B,C ou D).
Le bit sortant (à droite) est perdu. Si ce bit est égal à 1, alors le "Sticky Bit" **SB** est armé. Sinon le bit **SB** est inchangé.
Le bit entrant (à gauche) est égal à 0.
Le bit CARRY n'est pas affecté.
Pour un décalage à droite sur la totalité du registre (champ W), on utilisera les instructions précédentes **ASRB**, **BSRB**, etc...
Pour un décalage à gauche (multiplication par 2), on utilisera les instructions du type "**A=A+A** *field*", etc...
Rappel des spécifications de champ "a":
a = (0,1,2,3,4,5,6,F) ◄==► *field* = (P,WP,XS,X,S,M,B,A)

Mnémonique	Description	Code
ASRB.F *field*	Décale d'un bit à droite le champ **A:***field* *field* = P,WP,XS,X,S,M,B,A -->	**819a0**
BSRB.F *field*	Décale d'un bit à droite le champ **B:***field* *field* = P,WP,XS,X,S,M,B,A -->	**819a1**
CSRB.F *field*	Décale d'un bit à droite le champ **C:***field* *field* = P,WP,XS,X,S,M,B,A -->	**819a2**
DSRB.F *field*	Décale d'un bit à droite le champ **D:***field* *field* = P,WP,XS,X,S,M,B,A -->	**819a3**

◆ Opérations sur les registres de sauvegarde

Les registres **R0**, **R1**, **R2**, **R3**, **R4**, tous de longueur 16 quartets (64 bits) sont capables d'accueillir le contenu complet d'un registre de travail. Comme eux, ils sont accessibles en totalité ou suivant une spécification de champ.

Pour autant, leur utilisation est limitée à des échanges ou des transferts (dans un sens ou dans l'autre) avec les registres **A** et **C**. On retrouvera cette même situation pour ce qui est des instructions de lecture ou d'écriture en mémoire (à des adresses pointées par **D0** ou **D1**).

En particulier, il n'est pas possible de charger directement l'un de ces registres avec une constante, ou d'agir directement sur son contenu.

Les registres **R0**, **R1**, **R2**, **R3**, **R4** sont bien sûr un bon moyen de sauvegarder temporairement un registre de travail. Les routines de la ROM de la HP48 en font un usage constant (de façon plus fréquente avec **R0** et **R1** qu'avec les autres).

Si vous faites appel à de tels points d'entrée depuis votre code, soyez attentif aux sauvegardes que vous auriez pu effectuer dans ces registres...

Copie de la totalité d'un registre de travail (A ou C) vers un registre de sauvegarde (R0,R1,R2,R3,R4).
Le bit CARRY n'est pas affecté.
Temps d'exécution: 19 cycles

Mnémonique	Description	Code
R0=A	Copie du registre **A** dans le registre **R0**	**100**
R1=A	Copie du registre **A** dans le registre **R1**	**101**
R2=A	Copie du registre **A** dans le registre **R2**	**102**
R3=A	Copie du registre **A** dans le registre **R3**	**103**
R4=A	Copie du registre **A** dans le registre **R4**	**104**
R0=C	Copie du registre **C** dans le registre **R0**	**108**
R1=C	Copie du registre **C** dans le registre **R1**	**109**
R2=C	Copie du registre **C** dans le registre **R2**	**10A**
R3=C	Copie du registre **C** dans le registre **R3**	**10B**
R4=C	Copie du registre **C** dans le registre **R4**	**10C**

Comme ci-dessus, la copie se limitant au champ spécifié.
Le bit CARRY n'est pas affecté.
Rappel des spécifications de champ "a":
a = (0,1,2,3,4,5,6,F) ◄==► *field* = (P,WP,XS,X,S,M,B,A)

Mnémonique		Description	Code
R0=A.F	*field*	Copie le champ **A:***field* dans **R0:***field*	**81Aa00**
R1=A.F	*field*	Copie le champ **A:***field* dans **R1:***field*	**81Aa01**
R2=A.F	*field*	Copie le champ **A:***field* dans **R2:***field*	**81Aa02**
R3=A.F	*field*	Copie le champ **A:***field* dans **R3:***field*	**81Aa03**
R4=A.F	*field*	Copie le champ **A:***field* dans **R4:***field*	**81Aa04**
R0=C.F	*field*	Copie le champ **C:***field* dans **R0:***field*	**81Aa08**
R1=C.F	*field*	Copie le champ **C:***field* dans **R1:***field*	**81Aa09**
R2=C.F	*field*	Copie le champ **C:***field* dans **R2:***field*	**81Aa0A**
R3=C.F	*field*	Copie le champ **C:***field* dans **R3:***field*	**81Aa0B**
R4=C.F	*field*	Copie le champ **C:***field* dans **R4:***field*	**81Aa0C**

Opérations sur les registres de sauvegarde

Copie de la totalité d'un registre de sauvegarde (R0,R1,R2,R3,R4) dans un registre de travail (A ou C).
Le bit CARRY n'est pas affecté.
Temps d'exécution: 19 cycles

Mnémonique	Description	Code
A=R0	Copie du registre **R0** dans le registre **A**	**110**
A=R1	Copie du registre **R1** dans le registre **A**	**111**
A=R2	Copie du registre **R2** dans le registre **A**	**112**
A=R3	Copie du registre **R3** dans le registre **A**	**113**
A=R4	Copie du registre **R4** dans le registre **A**	**114**
C=R0	Copie du registre **R0** dans le registre **C**	**118**
C=R1	Copie du registre **R1** dans le registre **C**	**119**
C=R2	Copie du registre **R2** dans le registre **C**	**11A**
C=R3	Copie du registre **R3** dans le registre **C**	**11B**
C=R4	Copie du registre **R4** dans le registre **C**	**11C**

Comme ci-dessus, la copie se limitant au champ spécifié.
Le bit CARRY n'est pas affecté.
Rappel des spécifications de champ "a":
a = (0,1,2,3,4,5,6,F) ◄==► *field* = (P,WP,XS,X,S,M,B,A)

Mnémonique		Description	Code
A=R0.F	*field*	Copie le champ **R0:***field* dans **A:***field*	**81A*a*10**
A=R1.F	*field*	Copie le champ **R1:***field* dans **A:***field*	**81A*a*11**
A=R2.F	*field*	Copie le champ **R2:***field* dans **A:***field*	**81A*a*12**
A=R3.F	*field*	Copie le champ **R3:***field* dans **A:***field*	**81A*a*13**
A=R4.F	*field*	Copie le champ **R4:***field* dans **A:***field*	**81A*a*14**
C=R0.F	*field*	Copie le champ **R0:***field* dans **C:***field*	**81A*a*18**
C=R1.F	*field*	Copie le champ **R1:***field* dans **C:***field*	**81A*a*19**
C=R2.F	*field*	Copie le champ **R2:***field* dans **C:***field*	**81A*a*1A**
C=R3.F	*field*	Copie le champ **R3:***field* dans **C:***field*	**81A*a*1B**
C=R4.F	*field*	Copie le champ **R4:***field* dans **C:***field*	**81A*a*1C**

Echange du contenu total d'un registre de travail (A ou C) avec celui d'un registre de sauvegarde (R0,R1,R2,R3,R4).
Le bit CARRY n'est pas affecté.
Temps d'exécution: 19 cycles

Mnémonique	Description	Code
AR0EX	Echange du registre **A** et du registre **R0**	**120**
AR1EX	Echange du registre **A** et du registre **R1**	**121**
AR2EX	Echange du registre **A** et du registre **R2**	**122**
AR3EX	Echange du registre **A** et du registre **R3**	**123**
AR4EX	Echange du registre **A** et du registre **R4**	**124**
CR0EX	Echange du registre **C** et du registre **R0**	**128**
CR1EX	Echange du registre **C** et du registre **R1**	**129**
CR2EX	Echange du registre **C** et du registre **R2**	**12A**
CR3EX	Echange du registre **C** et du registre **R3**	**12B**
CR4EX	Echange du registre **C** et du registre **R4**	**12C**

Comme ci-dessus, l'échange se limitant au champ spécifié.
Le bit CARRY n'est pas affecté.
Rappel des spécifications de champ "a":
a = (0,1,2,3,4,5,6,F) ◄==► *field* = (P,WP,XS,X,S,M,B,A)

Mnémonique		Description	Code
AR0EX.F	*field*	Echange les champs **A:***field* et **R0:***field*	**81Aa20**
AR1EX.F	*field*	Echange les champs **A:***field* et **R1:***field*	**81Aa21**
AR2EX.F	*field*	Echange les champs **A:***field* et **R2:***field*	**81Aa22**
AR3EX.F	*field*	Echange les champs **A:***field* et **R3:***field*	**81Aa23**
AR4EX.F	*field*	Echange les champs **A:***field* et **R4:***field*	**81Aa24**
CR0EX.F	*field*	Echange les champs **C:***field* et **R0:***field*	**81Aa28**
CR1EX.F	*field*	Echange les champs **C:***field* et **R1:***field*	**81Aa29**
CR2EX.F	*field*	Echange les champs **C:***field* et **R2:***field*	**81Aa2A**
CR3EX.F	*field*	Echange les champs **C:***field* et **R3:***field*	**81Aa2B**
CR4EX.F	*field*	Echange les champs **C:***field* et **R4:***field*	**81Aa2C**

◆ Opérations sur les registres D0 et D1

Les registres **D0** et **D1** sont les pointeurs qui vous permettent d'accéder à une adresse quelconque, pour y lire ou y écrire des données. Là encore les registres **A** et **C** jouent un rôle particulièrement important puisqu'ils servent de registres de stockage de ces données.

Naturellement, il est possible de charger les registres **D0** et **D1** avec telle ou telle adresse. Celà peut se faire de plusieurs manières:

- ▸ En chargeant directement une constante de deux, quatre ou cinq quartets (le chargement s'effectuant à partir du quartet n° 0 du registre).
  Cette méthode est notamment utile pour accéder à certains points précis de la ROM de la HP48. La possibilité de ne charger que les quartets faibles d'une adresse est particulièrement aprréciable.
- ▸ En chargeant dans **D0** ou **D1** le contenu du registre **A** ou **C** (sur les quatre ou cinq quartets de poids faible). L'opération inverse est d'ailleurs possible.
- ▸ En échangeant (toujours sur les quatre ou cinq quartets de poids faible) les contenus de **D0** ou **D1** d'une part, et de **A** ou **C** d'autre part. C'est un bon moyen de modifier le contenu de **D0** ou **D1** tout en sauvegardant temporairement sa valeur (restaurée ensuite par le même échange).

Enfin il est possible d'incrémenter ou de décrémenter le contenu de **D0** ou **D1** (d'au maximum 16 quartets en une fois), ce qui est indispensable pour "balayer" des zones assez vastes.

Chargement d'une constante dans le registre D0 ou D1, sur les 2 ou 4 quartets de poids faible, ou sur la totalité du registre.
Le bit CARRY n'est pas affecté.
Temps d'exécution: respectivement 4,6, et 7 cycles

Mnémonique	Description	Code
D0=(2) *ab*	Charge les 2 quartets de poids faible de **D0** avec la constante hexa *#ab*	**19***ba*
D0=(4) *abcd*	Charge les 4 quartets de poids faible de **D0** avec la constante hexa *#abcd*	**1A***dcba*
D0=(5) *abcde*	Charge la totalité du registre **D0** avec la constante hexadécimale *#abcde*	**1B***edcba*
D1=(2) *ab*	Charge les 2 quartets de poids faible de **D1** avec la constante hexa *#ab*	**1D***ba*
D1=(4) *abcd*	Charge les 4 quartets de poids faible de **D1** avec la constante hexa *#abcd*	**1E***dcba*
D1=(5) *abcde*	Charge la totalité du registre **D1** avec la constante hexadécimale *#abcde*	**1F***edcba*

Addition ou soustraction d'une constante (comprise entre 1 et 16) au registre D0 ou D1.
Le bit CARRY est armé s'il y a retenue, et désarmé sinon.
Temps d'exécution: 7 cycles
Dans les notations ci-dessous, *n* est un quartet, ou encore une valeur comprise entre 0 et 15. Pour prendre un exemple, l'instruction **D0=D0+ 9** (codée **168**) a pour effet d'ajouter 9 au contenu du registre **D0**.
Attention: ces opérations sont toujours effectuées en mode héxa.

Mnémonique	Description	Code
D0=D0+ *(n+1)*	Additionne *(n+1)* au registre **D0**.	**16***n*
D1=D1+ *(n+1)*	Additionne *(n+1)* au registre **D1**.	**17***n*
D0=D0- *(n+1)*	Soustrait *(n+1)* au registre **D0**.	**18***n*
D1=D1- *(n+1)*	Soustrait *(n+1)* au registre **D1**.	**1C***n*

Copie du champ A d'un registre de travail (A ou C) dans un des deux registres D0 ou D1.
Le bit CARRY n'est pas affecté. Temps d'exécution: 8 cycles.

Mnémonique	Description	Code
D0=A	Copie le contenu de **A** (*champ A*) dans **D0**	**130**
D1=A	Copie le contenu de **A** (*champ A*) dans **D1**	**131**
D0=C	Copie le contenu de **C** (*champ A*) dans **D0**	**134**
D1=C	Copie le contenu de **C** (*champ A*) dans **D1**	**135**

Copie des 4 quartets de poids faible d'un registre A ou C sur les 4 quartets de poids faible d'un registre D0 ou D1.
Le quartet de poids fort de D0 ou D1 n'est pas modifié.
Le bit CARRY n'est pas affecté. Temps d'exécution: 7 cycles.

Mnémonique	Description	Code
D0=AS	Copie les 4 quartets de poids faible de **A** sur les 4 quartets de poids faible de **D0**	**138**
D1=AS	Copie les 4 quartets de poids faible de **C** sur les 4 quartets de poids faible de **D1**	**139**
D0=CS	Copie les 4 quartets de poids faible de **A** sur les 4 quartets de poids faible de **D0**	**13C**
D1=CS	Copie les 4 quartets de poids faible de **C** sur les 4 quartets de poids faible de **D1**	**13D**

Echange du champ A d'un registre de travail (A ou C) et du contenu d'un des deux registres D0 ou D1.
Le bit CARRY n'est pas affecté.
Temps d'exécution: 8 cycles.

Mnémonique	Description	Code
AD0EX	Echange le contenu de **A** (*champ A*) et celui de **D0**	**132**
AD1EX	Echange le contenu de **A** (*champ A*) et celui de **D1**	**133**
CD0EX	Echange le contenu de **C** (*champ A*) et celui de **D0**	**136**
CD1EX	Echange le contenu de **C** (*champ A*) et celui de **D1**	**137**

Echange des 4 quartets de poids faible d'un registre A ou C et des 4 quartets de poids faible d'un registre D0 ou D1.
Le quartet de poids fort de D0 ou D1 n'est pas modifié.
Le bit CARRY n'est pas affecté.
Temps d'exécution: 7 cycles.

Mnémonique	Description	Code
AD0XS	Echange les 4 quartets de poids faible de **A** et les 4 quartets de poids faible de **D0**	**13A**
AD1XS	Echange les 4 quartets de poids faible de **A** et les 4 quartets de poids faible de **D1**	**13B**
CD0XS	Echange les 4 quartets de poids faible de **C** et les 4 quartets de poids faible de **D0**	**13E**
CD1XS	Echange les 4 quartets de poids faible de **C** et les 4 quartets de poids faible de **D1**	**13F**

Nous voyons maintenant comment les registres **D0** et **D1** permettent, en conjonction avec les registres **A** et **C**, de lire ou d'écrire en mémoire.

On peut remarquer que pour les instructions qui suivent, il existe un code-machine particulier pour le champ ***B***, ce qui est commode pour les opérations de lecture ou d'écriture dans des chaînes de caractères (un caractère = un octet = deux quartets c'est-à-dire la longueur du champ B).

Lecture en mémoire, à partir de la position pointée par D0 ou D1, d'un nombre de quartets égal à la longueur du champ spécifié. Le résultat est placé dans le champ spécifié du registre A ou C.
Temps d'exécution: 18 (resp. 15) cycles pour le champ A (resp. B)
Pour une lecture dans un autre champ, de n quartets: 17+n cycles
Le bit CARRY n'est pas affecté.
Rappel des spécifications de champ "a":
a = (0,1,2,3,4,5,7) ◄==► *field* = (P,WP,XS,X,S,M,W)

Mnémonique	Description	Code
A=DAT0 *field*	Le(s) quartet(s) lu(s) à l'adresse pointée par **D0** viennent remplir le champ **A:***field*	
	field = A -->	**142**
	field = B -->	**14A**
	field = P,WP,XS,X,S,M,W -->	**152***a*
A=DAT1 *field*	Le(s) quartet(s) lu(s) à l'adresse pointée par **D1** viennent remplir le champ **A:***field*	
	field = A -->	**143**
	field = B -->	**14B**
	field = P,WP,XS,X,S,M,W -->	**153***a*
C=DAT0 *field*	Le(s) quartet(s) lu(s) à l'adresse pointée par **D0** viennent remplir le champ **C:***field*	
	field = A -->	**146**
	field = B -->	**14E**
	field = P,WP,XS,X,S,M,W -->	**156***a*
C=DAT1 *field*	Le(s) quartet(s) lu(s) à l'adresse pointée par **D1** viennent remplir le champ **C:***field*	
	field = A -->	**147**
	field = B -->	**14F**
	field = P,WP,XS,X,S,M,W -->	**157***a*

Lecture en mémoire, à partir de la position pointée par D0 ou D1, d'un nombre de quartets spécifié (et compris entre 1 et 16). Le résultat va dans le registre A ou C, à partir du quartet n°0.
Temps d'exécution: 16+q cycles, où q est le nombre de quartets lus
Le bit CARRY n'est pas affecté.

Mnémonique	Description	Code
A=DAT0 *(n+1)*	Lit (n+1) quartets (0≤n≤#Fh), à l'adresse pointée par **D0**. Le résultat va dans **A**	**15A***n*
A=DAT1 *(n+1)*	Lit (n+1) quartets (0≤n≤#Fh), à l'adresse pointée par **D1**. Le résultat va dans **A**	**15B***n*
C=DAT0 *(n+1)*	Lit (n+1) quartets (0≤n≤#Fh), à l'adresse pointée par **D0**. Le résultat va dans **C**	**15E***n*
C=DAT1 *(n+1)*	Lit (n+1) quartets (0≤n≤#Fh), à l'adresse pointée par **D1**. Le résultat va dans **C**	**15F***n*

Ecriture en mémoire, à partir de la position pointée par D0 ou D1, d'un nombre de quartets égal à la longueur du champ spécifié.
Les quartets sont ceux du champ spécifié du registre A ou C.
Temps d'exécution: 17 (resp. 14) cycles pour le champ A (resp. B); et pour un autre champ, de n quartets: 16+n cycles
Le bit CARRY n'est pas affecté.
Rappel des spécifications de champ "a":
a = (0,1,2,3,4,5,7) ◄==► *field* = (P,WP,XS,X,S,M,W)

Mnémonique	Description	Code
DAT0=A *field*	Ecrit le(s) quartet(s) du champ **A:***field* à partir de l'adresse pointée par **D0**	
	field = A -->	**140**
	field = B -->	**148**
	field = P,WP,XS,X,S,M,W -->	**150***a*
DAT1=A *field*	Ecrit le(s) quartet(s) du champ **A:***field* à partir de l'adresse pointée par **D1**	
	field = A -->	**141**
	field = B -->	**149**
	field = P,WP,XS,X,S,M,W -->	**151***a*
DAT0=C *field*	Ecrit le(s) quartet(s) du champ **C:***field* à partir de l'adresse pointée par **D0**	
	field = A -->	**144**
	field = B -->	**14C**
	field = P,WP,XS,X,S,M,W -->	**154***a*
DAT1=C *field*	Ecrit le(s) quartet(s) du champ **C:***field* à partir de l'adresse pointée par **D1**	
	field = A -->	**145**
	field = B -->	**14D**
	field = P,WP,XS,X,S,M,W -->	**155***a*

Ecriture en mémoire, à partir de la position pointée par D0 ou D1, d'un nombre de quartets spécifié (et compris entre 1 et 16).
Les quartets sont ceux du registre A ou C, à partir du quartet n°0
Temps d'exécution: 15+q cycles, où q = nombre de quartets écrits
Le bit CARRY n'est pas affecté.

Mnémonique	Description	Code
DAT0=A *(n+1)*	Ecrit les *n+1* quartets (0≤n≤#Fh) de poids faible de **A**, à l'adresse pointée par **D0**	**158***n*
DAT1=A *(n+1)*	Ecrit les *n+1* quartets (0≤n≤#Fh) de poids faible de **A**, à l'adresse pointée par **D1**	**159***n*
DAT0=C *(n+1)*	Ecrit les *n+1* quartets (0≤n≤#Fh) de poids faible de **C**, à l'adresse pointée par **D0**	**15C***n*
DAT1=C *(n+1)*	Ecrit les *n+1* quartets (0≤n≤#Fh) de poids faible de **C**, à l'adresse pointée par **D1**	**15D***n*

◆ LES BRANCHEMENTS

Rappelons que le registre PC (*Program Counter*) contient toujours l'adresse du prochain code-machine à exécuter.

Il faut distinguer plusieurs types de branchements. Chacun d'eux est une manière de quitter provisoirement ou définitivement une séquence donnée de codes-machine.

▸ **Les branchements inconditionnels**:

Ils s'effectuent par une modification, sans aucune condition préalable, du contenu du registre **PC**.

Dans cette catégorie, il faut distinguer les branchements du type *GOTO* de ceux qui sont du type *GOSUB* (Appel de sous-programme).

Dans le cas d'un appel de sous-programme, le contenu du registre **PC** est stocké dans la pile **RSTK** (dont on se souviendra qu'elle ne contient que 8 niveaux, deux d'entre eux devant rester libres pour les interruptions). Le registre **PC** est ensuite chargé avec l'adresse où débute ce sous-programme. La fin de celui-ci est marquée par une instruction du type *RETURN*, qui dépile du registre **RSTK** le contenu de **PC**, permettant un retour à la séquence initiale.

▸ **Les branchements conditionnels**:

A la différence des précédents, le branchement n'est effectué que si le résultat d'un test (immédiatement antérieur) retourne la valeur logique *VRAI*, ou selon le contenu du registre **CARRY** (armé ou désarmé).

Comme on le verra dans les pages qui suivent, les codes-machine qui définissent l'amplitude d'un branchement conditionnel consécutif à un test doivent suivrent immédiatement les codes-machine définissant ce test, à tel point que l'on considère habituellement leur juxtaposition comme une seule instruction.

Un cas particulier de branchement conditionnel est l'instruction **RTNYES** (code **00**) qui provoque un retour de sous-programme si le test précédent a renvoyé la valeur *VRAI*.

▸ **Les branchements relatifs ou absolus**:

L'adresse du code-machine vers lequel on veut "*dérouter*" le flux des instructions peut être mesurée de manière absolue (on précise la valeur exacte de l'adresse-cible) ou de manière relative (on précise le décalage existant entre l'adresse où se situe le code-machine en cours d'exécution et l'adresse où se trouve celui vers lequel on désire se brancher).

Les branchements relatifs peuvent être de type *court* ou *long*, suivant l'amplitude du décalage évoqué ci-dessus. L'utilisation de branchements relatifs est indispensable pour mettre au point un code "*relogeable*" (c'est-à-dire indifférent de l'adresse à laquelle il est placé pour exécution).

Les branchements absolus seront utilisés, en principe, pour accéder à des portions de code figurant dans la ROM de la HP48. Une "*bibliothèque d'adresses*" est alors nécessaire.

▸ **Les Labels:**

Celui qui programme sa HP48 en langage-machine, mais qui ne dispose pas d'un assembleur, est tenu d'utiliser directement les codes-machines du microprocesseur "Saturn". Une des difficultés qui l'attend tient au calcul des décalages nécessaires au calcul des branchements du type *GOTO* ou *GOSUB*.

Les labels (ou "étiquettes") sont des points nommés par l'utilisateur dans son code-source. L'assembleur chargé de traduire ce "source" en un "objet" exécutable, et qui rencontre un label, ne produit à cette occasion aucun code-machine. Il mémorise seulement l'adresse qui correspond à ce label. Si celui-ci est invoqué dans une instruction de branchement, l'assembleur calcule lui-même le décalage et produit le code correspondant.

Exemple:

Labels	Code	Mnémoniques		Commentaires
	20	P=0		Initialise P=0
LOOP				
	1527	A=DAT0	W	Lit 16 quartets (adr. D0)
	1517	DAT1=A	W	Les écrit à l'adresse D1
	16F	D0=D0+	16	Incrémente D0 de 16.
	17F	D1=D1+	16	Idem avec D1
	0C	P=P+1		Incrémente P
	5FE	GONC	LOOP	Continuer si P non nul.

La séquence précédente écrit, à partir de l'adresse pointée par **D1**, les 16*16 quartets trouvés à partir de l'adresse pointée par **D0**.

Celui qui dispose d'un assembleur se contentera de saisir les mnémoniques (et le label *LOOP*), avant de demander à son assembleur de produire le code-objet.

Celui qui ne dispose pas d'assembleur devra saisir les codes-machine (2ème colonne) et calculer lui-même le déplacement permettant d'exécuter la boucle une nouvelle fois.

Ce déplacement, comme on va le voir ci-dessous, est obtenu en calculant la différence *adr1-adr2*, où *adr1* est l'adresse où l'on désire se brancher, et où *adr2* est l'adresse du deuxième quartet de l'instruction "**GONC**". Peu importe combien valent réellement ces adresses, seul compte leur différence, égale ici à: -17 (qui se code **#EFh** sur deux quartets en complément à 2).

Les décalages permettant de calculer l'adresse de destination (dans le cas de branchements relatifs) sont calculés sur 2, 3 ou 4 quartets, en mode "*complément à 2*" (mode que l'on désigne encore sous le nom d'*arithmétique signée*). Le bit de poids fort du quartet de poids fort sert ici de bit de signe (il est à 1 pour une valeur négative et à zéro pour une valeur positive). Ces questions ont été évoquées à la fin du chapitre précédent...

Voici successivement toutes les instructions permettant un branchement à une adresse spécifiée (le code-machine utilisant, dans le cas d'un branchement relatif, un décalage pour calculer l'adresse destination).

Branchement RELATIF court, suivant l'état du bit CARRY.
Le décalage *#ab* est codé sur 2 quartets, en mode complément à 2.
Il s'obtient en calculant *Adresse(label)-Adresse(décalage)*.
Le décalage doit donc être compris entre -128=#80h et +127=#7Fh.
Le bit CARRY n'est pas modifié par cette instruction.
Temps d'exécution: 10 cycles s'il y a branchement, 3 cycles sinon.

Mnémonique	Description	Code
GOC *label*	Branchement si le registre CARRY est à 1	**4***ba*
GONC *label*	Branchement si le registre CARRY est à 0	**5***ba*

Branchement RELATIF court, inconditionnel.
Le décalage *#abc* est codé sur 3 quartets, en mode complément à 2.
Il s'obtient en calculant *Adresse(label)-Adresse(décalage)*.
Il doit donc être compris entre -2048=#800h et 2047=#7FFh.
Le bit CARRY n'est pas modifié par cette instruction.
Temps d'exécution: 11 cycles.

Mnémonique	Description	Code
GOTO *label*	Branche à l'adresse représentée par *label*	**6***cba*

Branchement RELATIF long, inconditionnel.
Le décalage *#abcd* est codé sur 4 quartets, en mode complément à 2.
Il s'obtient en calculant *Adresse(label)-Adresse(décalage)*.
Il doit donc être compris entre -32768=#8000h et 32767=#7FFFh.
Le bit CARRY n'est pas modifié par cette instruction.
Temps d'exécution: 14 cycles.

Mnémonique	Description	Code
GOLONG *label*	Branche à l'adresse représentée par *label*	**8C***dcba*

Branchement ABSOLU, inconditionnel (*GOto Very LoNG*)
Le bit CARRY n'est pas modifié par cette instruction.
Temps d'exécution: 14 cycles.

Mnémonique	Description	Code
GOVLNG *label*	Branchement à l'adresse *label* = *#abcde*	**8D***edcba*

Les branchements

Comme nous allons le voir maintenant, il y a d'autres façons de forcer un branchement à une adresse donnée. Il suffit par exemple de charger le registre **PC** avec les 5 quartets de poids faible de A (ou de **C**), ou encore d'échanger ces mêmes quartets avec les 5 quartets du registre **PC**...

Branchement à l'adresse définie DIRECTEMENT par le registre A ou C
Si le contenu du champ A du registre **A** (ou **C**) est *#abcde*, alors il y a branchement à l'adresse *#abcde*.
Le bit CARRY n'est pas modifié par cette instruction.

Mnémonique	Description	Code
PC=A	Branchement à l'adresse définie par les 5 quartets de poids faible de **A**.	**81B2**
PC=C	Branchement à l'adresse définie par les 5 quartets de poids faible de **A**.	**81B3**

Branchement à l'adresse définie DIRECTEMENT par le registre A ou C et sauvegarde du contenu de PC dans A ou C.
Si le contenu du champ A du registre **A** (ou **C**) est *#abcde*, alors il y a branchement à l'adresse *#abcde*, et le contenu actuel du registre **PC** est sauvegardé dans le registre **A** (ou **C**).
Le bit CARRY n'est pas modifié par cette instruction.

Mnémonique	Description	Code
APCEX	Echange les 5 quartets de poids faible du registre **A** et le contenu du registre **PC**	**81B6**
CPCEX	Echange les 5 quartets de poids faible du registre **C** et le contenu du registre **PC**	**81B7**

Branchement à l'adr. définie INDIRECTEMENT par le registre A ou C.
Si le contenu du champ A du registre **A** (ou **C**) est *#abcde* et si les 5 quartets situés à cette adresse sont *u,t,z,y,x* alors il y a branchement à l'adresse *#xyztu*.
Le bit CARRY n'est pas modifié par cette instruction.
Temps d'exécution: 23 cycles.

Mnémonique	Description	Code
PC=(A)	Branchement à l'adresse lue à l'adresse définie par les 5 quartets de poids faible du registre A.	**808C**
PC=(C)	Branchement à l'adresse lue à l'adresse définie par les 5 quartets de poids faible du registre C.	**808E**

Les deux instructions suivantes, bien que n'opérant pas un branchement immédiat, ont leur place ici car elles permettent de sauvegarder le contenu du registre **PC** (pour permettre ultérieurement un retour à l'instruction qui suit immédiatement).

Sauvegarde du registre PC dans l'un des registres A ou C.
Le bit CARRY n'est pas modifié par cette instruction.

Mnémonique	Description	Code
A=PC	Copie le contenu du registre **PC** sur les 5 quartets de poids faible de **A.**	**81B4**
C=PC	Copie le contenu du registre **PC** sur les 5 quartets de poids faible de **C.**	**81B5**

Venons-en maintenant aux appels de sous-programmes...

Appel de sous-programme (branchement RELATIF court).
Le décalage *#abc* est codé sur 3 quartets, en mode complément à 2.
Il vaut *Adresse(label) - Adresse(InstructionSuivante)*.
Il doit donc être compris entre -2048=#800h et 2047=#7FFh.
Le bit CARRY n'est pas modifié par cette instruction.
Temps d'exécution: 12 cycles.

Mnémonique	Description	Code
GOSUB *label*	Appel du sous-programme à l'adresse *label*	**7***cba*

Appel de sous-programme (branchement RELATIF long). *(GOSUB Long)*
Le décalage *#abcd* est codé sur 4 quartets, en mode complément à 2.
Il vaut *Adresse(label)-Adresse(InstructionSuivante)*.
Il doit donc être compris entre -32768=#8000h et 32767=#7FFFh.
Le bit CARRY n'est pas modifié par cette instruction.
Temps d'exécution: 15 cycles.

Mnémonique	Description	Code
GOSUBL *label*	Appel du sous-programme à l'adresse *label*	**8E***dcba*

Appel de sous-programme (branchement ABSOLU) *(GOSuB Very Long)*
Le bit CARRY n'est pas modifié par cette instruction.
Temps d'exécution: 15 cycles.

Mnémonique	Description	Code
GOSBVL *label*	Appel ss-prg à l'adresse *label=#abcde*	**8F***edcba*

Les branchements

Voici les instructions qui effectuent des retours de sous-programme. Leur action commune est de dépiler l'adresse de retour qui avait été placée sur la pile du registre **RSTK**, et de placer cette adresse dans le registre **PC**. Ces instructions se différencient par l'action qu'elle réalisent simultanément à ce retour.

Retours de sous-programme
Temps d'exécution: 9 cycles

Mnémonique	Description	Code
RTN	Retour de sous-programme	**01**
RTNSC	*Return & Set CARRY*: Retour de sous-prog et arme le bit CARRY	**02**
RTNCC	*Return & Clear CARRY*: Retour de sous-prog et désarme le bit CARRY	**03**
RTI	Retour de sous-programme et autorise les interruptions	**0F**
RTNSXM	*Return & Set XM bit*: Retour de sous-prog & arme le bit **XM** (*External Module Missing*)	**00**

Ici le retour de sous-programme est conditionné par le bit **CARRY**.

Retours conditionnels de sous-programme, selon le bit CARRY.
Temps d'exécution: 10 cycles s'il y a retour, 3 cycles sinon.
Le bit CARRY n'est pas modifié.

Mnémonique	Description	Code
RTNC	*Return If CARRY*: retour de sous-programme si le bit CARRY est armé (est à 1)	**400**
RTNNC	*Return If Not CARRY*: retour de sous-prog si le bit CARRY est désarmé (est à 0)	**500**

Nous allons maintenant passer en revue les différents tests que reconnait le microprocesseur "Saturn". Chacun de ces tests doit être suivi d'une instruction du type **GOYES** ou **RNTYES** provoquant (dans le cas où le test a renvoyé la valeur VRAI) un branchement (relatif court) à une adresse donnée, ou un retour de sous-programme.

◆ INSTRUCTIONS DE TEST

Le code-machine des instructions de test que nous allons décrire maintenant obéit toujours au format suivant: <*Test*> <*Complément*>.

- <*Test*> indique bien sûr quel est le test effectué et il est en général codé sur 3 quartets. Le résultat de ce test est stocké dans le bit CARRY (armé pour désigner la valeur VRAI, et désarmé pour désigner la valeur FAUX).
- <*Complément*> indique le branchement à effectuer dans le cas où le test a renvoyé la valeur VRAI, et est codé sur deux quartets.

Si <*Complément*>=00: le résultat positif du test entraîne l'exécution d'une instruction de retour de sous-programme.

Si <*Complément*> est différent de 00, il est interprété comme un décalage (*offset*) à appliquer à l'adresse de <Complément> pour obtenir l'adresse à laquelle on doit se brancher si le test est positif. Ce décalage est codé en mode "*complément à deux*", et est donc compris entre #80h= −128 et #7F=127.

Si l'on utilise un assembleur (où si on compte assembler soi-même les mnémoniques "à la main"), les instructions de tests s'écriront:

Dans le cas où <*Complément*> est égal à 00

<*Mnémonique_du_Test*>
RNTYES

Dans le cas où <*Complément*> est différent de 00

<*Mnémonique_du_Test*>
GOYES *Label*

Exemples: Considérons la séquence suivante.

Adresse	Labels	Code	Mnémoniques	
914B4		137	CD1EX	
914B7		06	RSTK=C	
914B9		137	CD1EX	
914BC	LOOP	143	A=DAT1	A
914BF		174	D1=D1+	5
914C2		8AC	?A#0	A
914C5		7F	GOYES	LOOP
914C7		1C4	D1=D1-	5
914CA		07	C=RSTK	
914CC		137	CD1EX	

Le test effectué ci-dessus consiste à vérifier si le contenu du champ *A* du registre **A** est non nul. Si c'est le cas, on se branche à nouveau à l'adresse définie par le label *LOOP*. Le label *LOOP* est à l'adresse #914BCh. L'adresse de l'instruction GOYES est #914C5h. La valeur du décalage est donc:

#914BCh-#914C5h=#FF...FFFF7h, c-à-d #F7h= −9 sur deux quartets.

On observe bien sûr l'inversion, en mémoire, des deux quartets représentant ce décalage, mais on commence à avoir l'habitude...

Commençons par décrire les instructions **GOYES** et **RNTYES**, compléments indispensables des instructions de tests qui vont suivre.

Compléments GOYES et RNTYES des instructions de tests.
Dans le cas d'une instruction GOYES, le décalage #xy représente, sur deux quartets, et en complément à 2, la différence:
Adresse(*Instruction_Destination*)-Adresse(*Instruction_GOYES*)
Les instructions GOYES ou RNTYES ne modifient pas le bit CARRY.
Temps d'exécution: voir celui des instructions de test.

Mnémonique	Description	Code
GOYES *label*	Branchement à l'adresse label si le test précédent a renvoyé la valeur VRAI	**yx**
RTNYES	Retour de sous-programme si le test qui précède a renvoyé la valeur VRAI	**00**

Commence alors une longue série de tests... Nous nous limiterons ici à ceux qui portent sur le contenu des registres de travail **A**, **B**, **C** ou **D**. Les autres tests, qui s'appliquent aux registres **P**, **ST**, **HS**, seront évoqués plus loin.

Tester si le champ spécifié d'un registre de travail est nul.
Le bit CARRY est armé si le résultat est VRAI, et désarmé sinon.
Cette instruction doit obligatoirement être suivie par un complément du type GOYES ou RNTYES.
Temps d'exécution: *13+n* ou *6+n* cycles, selon que le test renvoie la valeur VRAI (et il y a branchement) ou la valeur FAUX.
n est ici le nombre de quartets du champ spécifié.
Rappel des spécifications de champ "a":
a = (0,1,2,3,4,5,6,7) ◄==► *field* = (P,WP,XS,X,S,M,B,W)

Mnémonique	Description	Code
?A=0 *field*	Le contenu du champ **A:***field* est-il nul ?	
	field = A -->	**8A8**
	field = P,WP,XS,X,S,M,B,W -->	**9a8**
?B=0 *field*	Le contenu du champ **B:***field* est-il nul ?	
	field = A -->	**8A9**
	field = P,WP,XS,X,S,M,B,W -->	**9a9**
?C=0 *field*	Le contenu du champ **C:***field* est-il nul ?	
	field = A -->	**8AA**
	field = P,WP,XS,X,S,M,B,W -->	**9aA**
?D=0 *field*	Le contenu du champ **C:***field* est-il nul ?	
	field = A -->	**8AB**
	field = P,WP,XS,X,S,M,B,W -->	**9aB**

Tester si le champ spécifié d'un registre de travail est non nul.
Le bit CARRY est armé si le résultat est VRAI, et désarmé sinon.
Cette instruction doit obligatoirement être suivie par un complément du type GOYES ou RNTYES.
Temps d'exécution: *13+n* ou *6+n* cycles, selon que le test renvoie la valeur VRAI (et il y a branchement) ou la valeur FAUX.
n est ici le nombre de quartets du champ spécifié.
Rappel des spécifications de champ "a":
a = (0,1,2,3,4,5,6,7) ◄==► *field* = (P,WP,XS,X,S,M,B,W)

Mnémonique		Description	Code
?A#0	*field*	Le contenu du champ **A**:*field* est-il non nul?	
		field = A -->	**8AC**
		field = P,WP,XS,X,S,M,B,W -->	**9aC**
?B#0	*field*	Le contenu du champ **B**:*field* est-il non nul?	
		field = A -->	**8AD**
		field = P,WP,XS,X,S,M,B,W -->	**9aD**
?C#0	*field*	Le contenu du champ **C**:*field* est-il non nul?	
		field = A -->	**8AE**
		field = P,WP,XS,X,S,M,B,W -->	**9aE**
?D#0	*field*	Le contenu du champ **C**:*field* est-il non nul?	
		field = A -->	**8AF**
		field = P,WP,XS,X,S,M,B,W -->	**9aF**

Les instructions qui suivent permettent de tester individuellement l'état de chacun des 16 bits de poids faible du registre **A** ou **C** (c'est-à-dire les bits du champ **A** de ces registres).

Elles doivent être rapprochées des instructions **ABIT=0** *n*, etc..., qui permettent d'armer ou de désarmer l'un quelconque de ces 16 bits.

Tester l'état d'un bit du registre A ou C (limité aux bits 0 à 15)
Le bit CARRY est armé si le résultat est VRAI, et désarmé sinon.
Cette instruction doit obligatoirement être suivie par un complément du type GOYES ou RNTYES.

Mnémonique		Description	Code
?ABIT=0	*n*	Le bit *n* du registre **A** est-il désarmé ?	**8086***n*
?ABIT=1	*n*	Le bit *n* du registre **A** est-il armé ?	**8087***n*
?CBIT=0	*n*	Le bit *n* du registre **C** est-il désarmé ?	**808A***n*
?CBIT=1	*n*	Le bit *n* du registre **C** est-il armé ?	**808B***n*

Tests sur les registres A,B,C,D

Les champs spécifiés de deux registres de travail sont-ils égaux ?
Le bit CARRY est armé si le résultat est VRAI, et désarmé sinon.
Cette instruction doit obligatoirement être suivie par un complément du type GOYES ou RNTYES.
Temps d'exécution: *13+n* ou *6+n* cycles, selon que le test renvoie la valeur VRAI (et il y a branchement) ou la valeur FAUX.
n est ici le nombre de quartets du champ spécifié.
Rappel des spécifications de champ "a":
a = (0,1,2,3,4,5,6,7) ◄==► *field* = (P,WP,XS,X,S,M,B,W)

Mnémonique	Description	Code
?A=B *field*	Les champs **A:***field* et **B:***field* sont ils égaux?	
	field = A -->	**8A0**
	field = P,WP,XS,X,S,M,B,W -->	**9a0**
?B=C *field*	Les champs **B:***field* et **C:***field* sont ils égaux?	
	field = A -->	**8A1**
	field = P,WP,XS,X,S,M,B,W -->	**9a1**
?C=A *field*	Les champs **C:***field* et **A:***field* sont ils égaux?	
	field = A -->	**8A2**
	field = P,WP,XS,X,S,M,B,W -->	**9a2**
?D=C *field*	Les champs **D:***field* et **C:***field* sont ils égaux?	
	field = A -->	**8A3**
	field = P,WP,XS,X,S,M,B,W -->	**9a3**

Les champs spécifiés de 2 registres de travail sont-ils distincts?
Le bit CARRY est armé si le résultat est VRAI, et désarmé sinon.
Cette instruction doit obligatoirement être suivie par un complément du type GOYES ou RNTYES.
Temps d'exécution: voir le tableau ci-dessus
Rappel des spécifications de champ "a":
a = (0,1,2,3,4,5,6,7) ◄==► *field* = (P,WP,XS,X,S,M,B,W)

Mnémonique	Description	Code
?A#B *field*	Les champs **A:***field*,**B:***field* sont ils distincts?	
	field = A -->	**8A4**
	field = P,WP,XS,X,S,M,B,W -->	**9a4**
?B#C *field*	Les champs **B:***field*,**C:***field* sont ils distincts?	
	field = A -->	**8A5**
	field = P,WP,XS,X,S,M,B,W -->	**9a5**
?C#A *field*	Les champs **C:***field*,**A:***field* sont ils distincts?	
	field = A -->	**8A6**
	field = P,WP,XS,X,S,M,B,W -->	**9a6**
?D#C *field*	Les champs **D:***field*,**C:***field* sont ils distincts?	
	field = A -->	**8A7**
	field = P,WP,XS,X,S,M,B,W -->	**9a7**

Comparaison (INEGALITE STRICTE) des champs spécifiés de 2 registr.
Il s'agit d'une comparaison non signée.
Le bit CARRY est armé si le résultat est VRAI, et désarmé sinon.
Cette instruction doit obligatoirement être suivie par un complément du type GOYES ou RNTYES.
Temps d'exécution: *13+n* ou *6+n* cycles, selon que le test renvoie la valeur VRAI (et il y a branchement) ou la valeur FAUX.
n est ici le nombre de quartets du champ spécifié.
Rappel des spécifications de champ "b":
b = (8,9,A,B,C,D,E,F) ◄==► *field* = (P,WP,XS,X,S,M,B,W)

Mnémonique		Description	Code
?A>B	*field*	Le contenu du champ **A:***field* est-il supérieur	
	field	strictement à celui du champ **B:***field* ?	
		field = A -->	**8B0**
		field = P,WP,XS,X,S,M,B,W -->	**9***b***0**
?B>C	*field*	Le contenu du champ **B:***field* est-il supérieur	
		strictement à celui du champ **C:***field* ?	
		field = A -->	**8B1**
		field = P,WP,XS,X,S,M,B,W -->	**9***b***1**
?C>A	*field*	Le contenu du champ **C:***field* est-il supérieur	
		strictement à celui du champ **A:***field* ?	
		field = A -->	**8B2**
		field = P,WP,XS,X,S,M,B,W -->	**9***b***2**
?D>C	*field*	Le contenu du champ **D:***field* est-il supérieur	
		strictement à celui du champ **C:***field* ?	
		field = A -->	**8B3**
		field = P,WP,XS,X,S,M,B,W -->	**9***b***3**
?A<B	*field*	Le contenu du champ **A:***field* est-il inférieur	
		strictement à celui du champ **B:***field* ?	
		field = A -->	**8B4**
		field = P,WP,XS,X,S,M,B,W -->	**9***b***4**
?B<C	*field*	Le contenu du champ **B:***field* est-il inférieur	
		strictement à celui du champ **C:***field* ?	
		field = A -->	**8B5**
		field = P,WP,XS,X,S,M,B,W -->	**9***b***5**
?C<A	*field*	Le contenu du champ **C:***field* est-il inférieur	
		strictement à celui du champ **A:***field* ?	
		field = A -->	**8B6**
		field = P,WP,XS,X,S,M,B,W -->	**9***b***6**
?D<C	*field*	Le contenu du champ **D:***field* est-il inférieur	
		strictement à celui du champ **C:***field* ?	
		field = A -->	**8B7**
		field = P,WP,XS,X,S,M,B,W -->	**9***b***7**

Comparaison (INEGALITE LARGE) des champs spécifiés de 2 registres
Il s'agit d'une comparaison non signée.
Le bit CARRY est armé si le résultat est VRAI, et désarmé sinon.
Cette instruction doit obligatoirement être suivie par un complément du type GOYES ou RNTYES.
Temps d'exécution: *13+n* ou *6+n* cycles, selon que le test renvoie la valeur VRAI (et il y a branchement) ou la valeur FAUX.
n est ici le nombre de quartets du champ spécifié.
Rappel des spécifications de champ "b":
b = (8,9,A,B,C,D,E,F) ◄==► *field* = (P,WP,XS,X,S,M,B,W)

Mnémonique	Description	Code
?A>=B *field*	Le contenu du champ **A:***field* est-il supérieur ou égal à celui du champ **B:***field* ?	
	field = A -->	**8B8**
	field = P,WP,XS,X,S,M,B,W -->	**9***b***8**
?B>=C *field*	Le contenu du champ **B:***field* est-il supérieur ou égal à celui du champ **C:***field* ?	
	field = A -->	**8B9**
	field = P,WP,XS,X,S,M,B,W -->	**9***b***9**
?C>=A *field*	Le contenu du champ **C:***field* est-il supérieur ou égal à celui du champ **A:***field* ?	
	field = A -->	**8BA**
	field = P,WP,XS,X,S,M,B,W -->	**9***b***A**
?D>=C *field*	Le contenu du champ **D:***field* est-il supérieur ou égal à celui du champ **C:***field* ?	
	field = A -->	**8BB**
	field = P,WP,XS,X,S,M,B,W -->	**9***b***B**
?A<=B *field*	Le contenu du champ **A:***field* est-il inférieur ou égal à celui du champ **B:***field* ?	
	field = A -->	**8BC**
	field = P,WP,XS,X,S,M,B,W -->	**9***b***C**
?B<=C *field*	Le contenu du champ **B:***field* est-il inférieur ou égal à celui du champ **C:***field* ?	
	field = A -->	**8BD**
	field = P,WP,XS,X,S,M,B,W -->	**9***b***D**
?C<=A *field*	Le contenu du champ **C:***field* est-il inférieur ou égal à celui du champ **A:***field* ?	
	field = A -->	**8BE**
	field = P,WP,XS,X,S,M,B,W -->	**9***b***E**
?D<=C *field*	Le contenu du champ **D:***field* est-il inférieur ou égal à celui du champ **C:***field* ?	
	field = A -->	**8BF**
	field = P,WP,XS,X,S,M,B,W -->	**9***b***F**

◆ LE REGISTRE P

Le registre **P** définit les champs *P* et *WP* d'un des registres de travail **A**,**B**,**C** ou **D**. Il est également utilisé comme compteur de boucle (si les valeurs 0 à 15 suffisent).

Opérations sur le registre P
Les opérations P=P+1, P=P-1, C+P+1 se font en héxadécimal (ceci sans changer le mode en cours: décimal ou héxadécimal).
Le bit CARRY est affecté par: P=P+1, P=P-1, C+P+1, ?P# *n*, ?P= *n*.
Les deux tests "?P= *n*" et "?P# *n*" doivent être suivis d'une instruction de type GOYES ou RNTYES. Le temps d'exécution de l'instruction qui en résulte est alors de 13 cycles si le test renvoie la valeur VRAI (il y a branchement) et 6 cycles sinon.
Dans les notations ci-dessous, *n* est un quartet (0≤n≤#Fh=15)

Mnémonique	Description	Code
P= *n*	Donne au registre **P** la valeur *n* (2 cycles)	**2***n*
P=P+1	Incrémente le contenu du registre **P** (3 cycles)	**0C**
P=P-1	Décrémente le contenu du registre **P** (3 cycles)	**0D**
?P# *n*	Le contenu de **P** est-il différent de *n* ?	**88***n*
?P= *n*	Le contenu de **P** est-il égal à *n* ?	**89***n*
C=P *n*	Copie **P** sur le quartet *n* de **C** (6 cycles)	**80C***n*
P=C *n*	Copie dans **P** le quartet *n* de **C** (6 cycles)	**80D***n*
C+P+1 **C=C+P+1**	Ajoute au contenu du champ A du registre **C** le contenu de **P**, plus 1 (durée 8 cycles)	**809**
CPEX *n*	Echange **P** et le quartet *n* de **C** (6 cycles)	**80F***n*

◆ LE REGISTRE RSTK

Le registre **RSTK** est une pile hardware de 8 niveaux, qui sert à mémoriser provisoirement les adresses de retour de sous-programmes.

Instructions s'appliquant au registre RSTK
Temps d'exécution: 8 cycles. Le bit CARRY n'est pas affecté

Mnémonique	Description	Code
RSTK=C	Empile dans le registre **RSTK** l'adresse qui est contenue dans le champ A du registre **C**	**06**
C=RSTK	Dépile l'adresse la plus récente du registre **RSTK** et la place dans le champ A de **C**.	**07**

◆ LE REGISTRE ST (Status Bits)

Les 16 bits du registre **ST** (numérotés de 0 à 15) sont des indicateurs. Seuls les bits n°0 à 11 sont utilisables (les 4 bits de poids fort sont utilisés par la HP48, pour ses routines internes). Ces 12 bits sont au langage-machine ce que sont les flags-utilisateur au langage RPL usuel.

Opérations sur le registre ST (*Status Bits*)
Seules les instructions de test "?ST=0 *n*" et "?ST=1 *n*" affectent le bit CARRY (armé si test VRAI, désarmé sinon).
Ces deux instructions doivent être complétées par un GOYES ou par un RNTYES. La durée d'exécution est alors de 14 cycles s'il y a branchement (le test est VRAI) et de 7 cycles sinon.
Dans les notations ci-dessous, *n* est un quartet ($0 \le n \le \#Fh=15$)

Mnémonique	Description	Code
ST=0 *n*	Désarme le Status Bit *n* (4 cycles)	**84***n*
ST=1 *n*	Arme le Status Bit *n* (4 cycles)	**85***n*
?ST=0 *n*	Le Status Bit *n* est-il désarmé ?	**86***n*
?ST=1 *n*	Le Status Bit *n* est-il armé ?	**87***n*
CLRST	Désarme les Status bits 0 à 11 (6 cycles)	**08**
C=ST	Copie les bits 0 à 11 du registre **ST** sur les bits 0 à 11 (champ X) du registre **C** (6 cycles)	**09**
ST=C	Copie les bits 0 à 11 du registre **C** (champ X) sur les bits 0 à 11 du registre **ST** (6 cycles)	**0A**
CSTEX	Echange les bits 0 à 11 du registre **ST** et les bits 0 à 11 (champ X) du registre **C** (6 cycles)	**0B**

◆ LES REGISTRES IN et OUT

Le registre **IN** (registre des entrées) est long de 16 bits (4 quartets). Le registre **OUT** (registre des sorties) est long de 12 bits (3 quartets).

Opérations sur les registres IN et OUT (bit CARRY non affecté)

Mnémonique	Description	Code
OUT=CS	Quartet 0 de **C** va sur quartet 0 de **OUT** (4 cycl)	**800**
OUT=C	**OUT** est chargé avec le champ X de **C** (6 cycles)	**801**
A=IN	Copie **IN** sur les quartets 0 à 3 de **A** (7 cycles)	**802**
C=IN	Copie **IN** sur les quartets 0 à 3 de **C** (7 cycles)	**803**

◆ LE REGISTRE HS (Hardware Status bits)

Les instructions qui suivent permettent de positionner ou de tester les 4 bits du registre HS: le bit **MP** (*Module Pulled*), le bit **SR** (*Service Request*), le bit **SB** (*Sticky Bit*),et le bit **XM** (*External Module Missing*).

Opérations sur le registre HS (*Hardware Status bits*)
Seules les instructions de test affectent le bit CARRY (armé si le test renvoie la valeur VRAI, désarmé sinon).
Ces tests doivent être complétés par un GOYES ou par un RNTYES.

Mnémonique	Description	Code
XM=0	Désarme le bit **XM** (3 cycles)	**821**
SB=0	Désarme le bit **SB** (3 cycles)	**822**
SR=0	Désarme le bit **SR** (3 cycles)	**824**
MP=0	Désarme le bit **MP** (3 cycles)	**828**
CLRHST	Désarme les 4 Hardware Status bits (3 cycles)	**82F**
HS=0 *n*	Désarme un ou plusieurs des Hard. Status bits en utilisant le masque n (3 cycles) *n* = 8**mp* + 4**sr* + 2**sb* + *xm*	**82***n*
?XM=0	Le bit **XM** est-il désarmé ? (13 ou 6 cycles)	**831**
?SB=0	Le bit **SB** est-il désarmé ? (13 ou 6 cycles)	**832**
?SR=0	Le bit **SR** est-il désarmé ? (13 ou 6 cycles)	**834**
?MP=0	Le bit **MP** est-il désarmé ? (13 ou 6 cycles)	**838**
?HS=0 *n*	Teste un ou plusieurs des Hardware Status bits en utilisant le masque *n* (13 ou 6 cycles) *n* = 8**mp* + 4**sr* + 2**sb* + *xm*	**83***n*

◆ INSTRUCTIONS VIDES

Voici trois instructions qui ne font rien! Elles n'en sont pas moins très utiles dans la phase de mise au point d'un programme, où dans les programmes qui s'auto-modifient.

Mnémonique	Description	Code
NOP3	Instruction vide, occupant 3 quartets	**420**
NOP4	Instruction vide, occupant 4 quartets	**6300**
NOP5	Instruction vide, occupant 5 quartets	**64000**

♦ INSTRUCTIONS OPÉRANT SUR LE "BUS"

Voici quelques instructions plutôt rares dans la ROM de la HP48. Les instructions CONFIG et UNCNFG sont utilisées pour accéder à la "ROM cachée" (voir le chapitre consacré à ce sujet dans le tome 1 des "Secrets"). L'instruction RESET apparaît dans le traitement de l'arrêt-système par ON-C...

Mnémonique	Description	Code
UNCNFG	Déconfigure le chip dont l'adresse est spécifiée par le champ A de **C** (12 cycles).	**804**
CONFIG	Copie le contenu du champ A de **C** dans le registre de config. du chip à configurer (11 cycles)	**805**
C=ID	Stocke dans le champ A de **C** le registre de config du chip qui doit être configuré (11 cycles)	**806**
SHUTDN	Envoie la commande "*ShutDown*" sur le bus-système. L'horloge est arrêtée. (5 cycles)	**807**
RESET	Applique un reset à chaque chip (6 cycles)	**80A**
BUSCC	Envoie la commande "**C**" sur le bus du système	**80B**
SREQ?	Arme le bit **SR** si un chip envoie une requête.	**80E**
BUSCB	Envoie la commande "**B**" sur le bus du système	**8083**
BUSCD	Envoie la commande "**D**" sur le bus du système	**808D**

♦ LES INTERRUPTIONS

Mnémonique	Description	Code
INTON	Autorise les interruptions clavier (5 cycles)	**8080**
INTOFF	Interdit les interruptions clavier (5 cycles)	**808F**
RSI	Réinitialise les interruptions (6 cycles)	**80810**

♦ CHOIX DU MODE (Héxa/BCD)

Mnémonique	Description	Code
SETDEC	Sélectionne le mode décimal (3 cycles)	**05**
SETHEX	Sélectionne le mode héxadécimal (3 cycles)	**04**

DICTIONNAIRE ALPHABÉTIQUE DES MNÉMONIQUES

Dictionnaire des mnémoniques

Voici la liste des mnémoniques du "Saturn", classés par ordre alphabétique croissant, et suivis du code-machine qui leur correspond. On a utilisé des caractères en italiques pour désigner les parties variables dans chaque mnémonique (et donc dans chaque code-machine), comme par exemple:

- Les spécifications de champ: *A*,*B*, ou *f*. Les champs *A* ou *B*, lorsqu'ils conduisent à un code-machine spécifique sont toujours mentionnés. Le champ *f* est l'un des autres champs possibles (*P*,*WP*,*XS*,*X*,*S*,*M*,*B*,*W*).
- Les quartets sélecteurs de champs *a* ou *b*. Rappelons leur signification:
  a = (0,1,2,3,4,5,6,7) ◄==► *f(ield)* = (P,WP,XS,X,S,M,B,W)
  b = (8,9,A,B,C,D,E,F) ◄==► *f(ield)* = (P,WP,XS,X,S,M,B,W)
- Les offsets *#pq* (en héxadécimal sur deux quartets, en mode complément à deux) nécessaires pour compléter les instructions de test (toutes celles dont le mnémonique commence par ?)

  Rappelons que l'octet *pq*=00 caractérise l'instruction **RNTYES**.
- Les offsets *#pq*, *#pqr*, *#pqrs*, utilisés dans les instructions de branchement relatif (Gosub ou Goto).
- Le quartet *n* utilisé parfois pour contenir une valeur entre 0 et #Fh=15.

Les instructions **GOYES** et **RNTYES** ne figurent pas dans ce dictionnaire (elles sont en effet associées aux instructions de test).

Dans le cas où deux registres jouent un rôle symétrique, une même instruction pouvait être décrite par plusieurs mnémoniques différents. C'est le cas par exemple, des instructions **A=A!B** et **A=B!A**. Dans ce cas, on n'a fait figurer ici qu'une seule variante.

Le lecteur verra qu'il y a tout de même une logique, qui consiste à associer les registres de travail suivant les couples (A,B), (B,C), (C,A) et (D,C).

?A#0	*A*	8AC*qp*	?B#0	*f*	9aD*qp*
?A#0	*f*	9aC*qp*	?B#C	*A*	8A5*qp*
?A#B	*A*	8A4*qp*	?B#C	*f*	9a5*qp*
?A#B	*f*	9a4*qp*	?B<=C	*A*	8BD*qp*
?A<=B	*A*	8BC*qp*	?B<=C	*f*	9bD*qp*
?A<=B	*f*	9bC*qp*	?B<C	*A*	8B5*qp*
?A<B	*A*	8B4*qp*	?B<C	*f*	9b5*qp*
?A<B	*f*	9b4*qp*	?B=0	*A*	8A9*qp*
?A=0	*A*	8A8*qp*	?B=0	*f*	9a9*qp*
?A=0	*f*	9a8*qp*	?B=C	*A*	8A1*qp*
?A=B	*A*	8A0*qp*	?B=C	*f*	9a1*qp*
?A=B	*f*	9a0*qp*	?B>=C	*A*	8B9*qp*
?A>=B	*A*	8B8*qp*	?B>=C	*f*	9b9*qp*
?A>=B	*f*	9b8*qp*	?B>C	*A*	8B1*qp*
?A>B	*A*	8B0*qp*	?B>C	*f*	9b1*qp*
?A>B	*f*	9b0*qp*	?C#0	*A*	8AE*qp*
?ABIT=0	*n*	8086*nqp*	?C#0	*f*	9aE*qp*
?ABIT=1	*n*	8087*nqp*	?C#A	*A*	8A6*qp*
?B#0	*A*	8AD*qp*	?C#A	*f*	9a6*qp*

```
?C<=A   A       8BEqp
?C<=A   f       9bEqp
?C<A    A       8B6qp
?C<A    f       9b6qp
?C=0    A       8AAqp
?C=0    f       9aAqp
?C=A    A       8A2qp
?C=A    f       9a2qp
?C>=A   A       8BAqp
?C>=A   f       9bAqp
?C>A    A       8B2qp
?C>A    f       9b2qp
?CBIT=0 n       808Anqp
?CBIT=1 n       808Bnqp
?D#0    A       8AFqp
?D#0    f       9aFqp
?D#C    A       8A7qp
?D#C    f       9a7qp
?D<=C   A       8BFqp
?D<=C   f       9bFqp
?D<C    A       8B7qp
?D<C    f       9b7qp
?D=0    A       8ABqp
?D=0    f       9aBqp
?D=C    A       8A3qp
?D=C    f       9a3qp
?D>=C   A       8BBqp
?D>=C   f       9bBqp
?D>C    A       8B3qp
?D>C    f       9b3qp
?HS=0   n       83nqp
?MP=0           838qp
?P#     n       88nqp
?P=     n       89nqp
?SB=0           832qp
?SR=0           834qp
?ST=0   n       86nqp
?ST=1   n       87nqp
?XM=0           831qp
A=-A    A       F8
A=-A    f       Bb8
A=-A-1  A       FC
A=-A-1  f       BbC
A=0     A       D0
A=0     f       Ab0
A=A!B   A       0EF8
A=A!B   f       0Ea8
A=A&B   A       0EF0
A=A&B   f       0Ea0
A=A+(n+1)  A    818F0n
A=A+(n+1)  f    818a0n
A=A+1   A       E4
A=A+1   f       Ba4
A=A+A   A       C4
A=A+A   f       Aa4
A=A+B   A       C0
A=A+B   f       Aa0
A=A-(n+1)  A    818F8n
A=A-(n+1)  f    818a8n
A=A-1   A       CC
A=A-1   f       AaC
A=A-B   A       E0
A=A-B   f       Ba0
A=A-C   A       EA
A=A-C   f       BaA
A=B     A       D4
A=B     f       Ab4
A=B-A   A       EC
A=B-A   f       BaC
A=C     A        DA
A=C     f        AbA
A=C!A   A       0EFE
A=C!A   f       0EaE
A=C&A   A       0EF6
A=C&A   f       0Ea6
A=C+A   A       CA
A=C+A   f       AaA
A=DAT0  (n+1)   15An
A=DAT0  A       142
A=DAT0  B       14A
A=DAT0  f       152a
A=DAT1  (n+1)   15Bn
A=DAT1  A       143
A=DAT1  B       14B
A=DAT1  f       153a
A=IN            802
A=PC            81B4
A=R0            110
A=R0.F  A       81AF10
A=R0.F  f       81Aa10
A=R1            111
A=R1.F  A       81AF11
A=R1.F  f       81Aa11
A=R2            112
A=R2.F  A       81AF12
A=R2.F  f       81Aa12
A=R3            113
A=R3.F  A       81AF13
A=R3.F  f       81Aa13
A=R4            114
A=R4.F  A       81AF14
A=R4.F  f       81Aa14
ABEX    A       DC
ABEX    f       AbC
ABIT=0  n       8084n
ABIT=1  n       8085n
AD0EX           132
AD0XS           13A
AD1EX           133
AD1XS           13B
APCEX           81B6
AR0EX           120
AR0EX.F A       81AF20
AR0EX.F f       81Aa20
AR1EX           121
AR1EX.F A       81AF21
```

```
AR1EX.F   f     81Aa21
AR2EX           122
AR2EX.F   A     81AF22
AR2EX.F   f     81Aa22
AR3EX           123
AR3EX.F   A     81AF23
AR3EX.F   f     81Aa23
AR4EX           124
AR4EX.F   A     81AF24
AR4EX.F   f     81Aa24
ASL       A     F0
ASL       f     Bb0
ASLC            810
ASR       A     F4
ASR       f     Bb4
ASRB            81C
ASRB.F    A     819F0
ASRB.F    f     819a0
ASRC            814
B=-B      A     F9
B=-B      f     Bb9
B=-B-1    A     FD
B=-B-1    f     BbD
B=0   A         D1
B=0   f         Ab1
B=A   A         D8
B=A   f         Ab8
B=A!B   A       0EFC
B=A!B   f       0EaC
B=A&B   A       0EF4
B=A&B   f       0Ea4
B=A+B   A       C8
B=A+B   f       Aa8
B=B!C   A       0EF9
B=B!C   f       0Ea9
B=B&C   A       0EF1
B=B&C   f       0Ea1
B=B+(n+1)   A   818F1n
B=B+(n+1)   f   818a1n
B=B+1   A       E5
B=B+1   f       Ba5
B=B+B   A       C5
B=B+B   f       Aa5
B=B+C   A       C1
B=B+C   f       Aa1
B=B-(n+1)   A   818F9n
B=B-(n+1)   f   818a9n
B=B-1   A       CD
B=B-1   f       AaD
B=B-A   A       E8
B=B-A   f       Ba8
B=B-C   A       E1
B=B-C   f       Ba1
B=C   A         D5
B=C   f         Ab5
B=C-B   A       ED
B=C-B   f       BaD
BCEX   A        DD
BCEX   f        AbD
BSL   A         F1
BSL   f         Bb1
BSLC            811
BSR   A         F5
BSR   f         Bb5
BSRB            81D
BSRB.F   A      819F1
BSRB.F   f      819a1
BSRC            815
BUSCB           8083
BUSCC           80B
BUSCD           808D
C+P+1           809
C=-C   A        FA
C=-C   f        BbA
C=-C-1   A      FE
C=-C-1   f      BbE
C=0   A         D2
C=0   f         Ab2
C=A   A         D6
C=A   f         Ab6
C=A-C   A       EE
C=A-C   f       BaE
C=B   A         D9
C=B   f         Ab9
C=B!C   A       0EFD
C=B!C   f       0EaD
C=B&C   A       0EF5
C=B&C   f       0Ea5
C=B+C   A       C9
C=B+C   f       Aa9
C=C!A   A       0EFA
C=C!A   f       0EaA
C=C&A   A       0EF2
C=C&A   f       0Ea2
C=C+(n+1)   A   818F2n
C=C+(n+1)   f   818a2n
C=C+1   A       E6
C=C+1   f       Ba6
C=C+A   A       C2
C=C+A   f       Aa2
C=C+C   A       C6
C=C+C   f       Aa6
C=C-(n+1)   A   818FAn
C=C-(n+1)   f   818aAn
C=C-1   A       CE
C=C-1   f       AaE
C=C-A   A       E2
C=C-A   f       Ba2
C=C-B   A       E9
C=C-B   f       Ba9
C=C-D   A       EB
C=C-D   f       BaB
C=D   A         DB
C=D   f         AbB
C=D!C   A       0EFF
C=D!C   f       0EaF
```

```
C=D&C    A       0EF7
C=D&C    f       0Ea7
C=D+C    A       CB
C=D+C    f       AaB
C=DAT0   (n+1)   15En
C=DAT0   A       146
C=DAT0   B       14E
C=DAT0   f       156a
C=DAT1   (n+1)   15Fn
C=DAT1   A       147
C=DAT1   B       14F
C=DAT1   f       157a
C=ID             806
C=IN             803
C=P   n          80Cn
C=PC             81B5
C=R0             118
C=R0.F   A       81AF18
C=R0.F   f       81Aa18
C=R1             119
C=R1.F   A       81AF19
C=R1.F   f       81Aa19
C=R2             11A
C=R2.F   A       81AF1A
C=R2.F   f       81Aa1A
C=R3             11B
C=R3.F   A       81AF1B
C=R3.F   f       81Aa1B
C=R4             11C
C=R4.F   A       81AF1C
C=R4.F   f       81Aa1C
C=RSTK           07
C=ST             09
CAEX   A         DE
CAEX   f         AbE
CBIT=0   n       8088n
CBIT=1   n       8089n
CD0EX            136
CD0XS            13E
CD1EX            137
CD1XS            13F
CLRHST           82F
CLRST            08
CONFIG           805
CPCEX            81B7
CPEX   n         80Fn
CR0EX            128
CR0EX.F  A       81AF28
CR0EX.F  f       81Aa28
CR1EX            129
CR1EX.F  A       81AF29
CR1EX.F  f       81Aa29
CR2EX            12A
CR2EX.F  A       81AF2A
CR2EX.F  f       81Aa2A
CR3EX            12B
CR3EX.F  A       81AF2B
CR3EX.F  f       81Aa2B
CR4EX            12C
CR4EX.F  A       81AF2C
CR4EX.F  f       81Aa2C
CSL    A         F2
CSL    f         Bb2
CSLC             812
CSR    A         F6
CSR    f         Bb6
CSRB             81E
CSRB.F   A       819F2
CSRB.F   f       819a2
CSRC             816
CSTEX            0B
D=-D   A         FB
D=-D   f         BbB
D=-D-1   A       FF
D=-D-1   f       BbF
D=0    A         D3
D=0    f         Ab3
D=C    A         D7
D=C    f         Ab7
D=C-D    A       EF
D=C-D    f       BaF
D=D!C    A       0EFB
D=D!C    f       0EaB
D=D&C    A       0EF3
D=D&C    f       0Ea3
D=D+(n+1)    A   818F3n
D=D+(n+1)    f   818a3n
D=D+1    A       E7
D=D+1    f       Ba7
D=D+C    A       C3
D=D+C    f       Aa3
D=D+D    A       C7
D=D+D    f       Aa7
D=D-(n+1)    A   818FBn
D=D-(n+1)    f   818aBn
D=D-1    A       CF
D=D-1    f       AaF
D=D-C    A       E3
D=D-C    f       Ba3
D0=(2)     pq    19qp
D0=(4)     pqrs  1Asrqp
D0=(5)     pqrst 1Btsrqp
D0=A             130
D0=AS            138
D0=C             134
D0=CS            13C
D0=D0+   (n+1)   16n
D0=D0-   (n+1)   18n
D1=(2)     pq    1Dqp
D1=(4)     pqrs  1Esrqp
D1=(5)     pqrst 1Ftsrqp
D1=A             131
D1=AS            139
D1=C             135
D1=CS            13D
D1=D1+   (n+1)   17n
```

D1=D1-	*(n+1)*	1C*n*
DAT0=A	*(n+1)*	158*n*
DAT0=A	*A*	140
DAT0=A	*B*	148
DAT0=A	*f*	150*a*
DAT0=C	*(n+1)*	15C*n*
DAT0=C	*A*	144
DAT0=C	*B*	14C
DAT0=C	*f*	154*a*
DAT1=A	*(n+1)*	159*n*
DAT1=A	*A*	141
DAT1=A	*B*	149
DAT1=A	*f*	151*a*
DAT1=C	*(n+1)*	15D*n*
DAT1=C	*A*	145
DAT1=C	*B*	14D
DAT1=C	*f*	155*a*
DCEX	*A*	DF
DCEX	*f*	A*b*F
DSL	*A*	F3
DSL	*f*	B*b*3
DSLC		813
DSR	*A*	F7
DSR	*f*	B*b*7
DSRB		81F
DSRB.F	*A*	819F3
DSRB.F	*f*	819*a*3
DSRC		817
GOC	*label*	4*qp*
GOLONG	*label*	8C*srqp*
GONC	*label*	5*qp*
GOSBVL	*label*	8F*tsrqp*
GOSUB	*label*	7*rqp*
GOSUBL	*label*	8E*srqp*
GOTO	*label*	6*rqp*
GOVLNG	*label*	8D*tsrqp*
HS=0	*n*	82*n*
INTOFF		808F
INTON		8080
LA(*n+1*)	$h_n..h_0$	8082*n*$h_0..h_n$
LC(*n+1*)	$h_n..h_0$	3*n*$h_0..h_n$
MP=0		828
NOP3		420
NOP4		6300
NOP5		64000
OUT=C		801
OUT=CS		800
P=	*n*	2*n*
P=C	*n*	80D*n*
P=P+1		0C
P=P-1		0D
PC=(A)		808C
PC=(C)		808E
PC=A		81B2
PC=C		81B3
R0=A		100
R0=A.F	*A*	81AF00
R0=A.F	*f*	81A*a*00
R0=C		108
R0=C.F	*A*	81AF08
R0=C.F	*f*	81A*a*08
R1=A		101
R1=A.F	*A*	81AF01
R1=A.F	*f*	81A*a*01
R1=C		109
R1=C.F	*A*	81AF09
R1=C.F	*f*	81A*a*09
R2=A		102
R2=A.F	*A*	81AF02
R2=A.F	*f*	81A*a*02
R2=C		10A
R2=C.F	*A*	81AF0A
R2=C.F	*f*	81A*a*0A
R3=A		103
R3=A.F	*A*	81AF03
R3=A.F	*f*	81A*a*03
R3=C		10B
R3=C.F	*A*	81AF0B
R3=C.F	*f*	81A*a*0B
R4=A		104
R4=A.F	*A*	81AF04
R4=A.F	*f*	81A*a*04
R4=C		10C
R4=C.F	*A*	81AF0C
R4=C.F	*f*	81A*a*0C
RESET		80A
RSI		80810
RSTK=C		06
RTI		0F
RTN		01
RTNC		400
RTNCC		03
RTNNC		500
RTNSC		02
RTNSXM		00
SB=0		822
SETDEC		05
SETHEX		04
SHUTDN		807
SR=0		824
SREQ?		80E
ST=0	*n*	84*n*
ST=1	*n*	85*n*
ST=C		0A
UNCNFG		804
XM=0		821

DICTIONNAIRE ALPHABÉTIQUE DES CODES-MACHINE

Dictionnaire des code-machine

Voici la liste des codes-machine du "Saturn", classés par ordre alphabétique croissant, et suivis du mnémonique qui leur correspond.

On a utilisé des caractères en italiques pour désigner les parties variables dans chaque code-machine (et donc dans chaque mnémonique), comme par exemple:

- Les quartets sélecteurs de champs *a* ou *b*. Rappelons leur signification:
  a = (0,1,2,3,4,5,6,7) ◂==▸ *f(ield)* = (P,WP,XS,X,S,M,B,W)
  b = (8,9,A,B,C,D,E,F) ◂==▸ *f(ield)* = (P,WP,XS,X,S,M,B,W)
- Les spécifications de champ: *A*,*B*, ou *f*. Les champs *A* ou *B*, lorsqu'ils conduisent à un code-machine spécifique sont toujours mentionnés. Le champ *f* est l'un des autres champs possibles (*P*,*WP*,*XS*,*X*,*S*,*M*,*B*,*W*).
- Les offsets *#pq* (en héxadécimal sur deux quartets, en mode complément à deux) nécessaires pour compléter les instructions de test (toutes celles dont le mnémonique commence par ?)

  Rappelons que l'octet *pq*=00 caractérise l'instruction **RNTYES**.
- Les offsets *#pq*, *#pqr*, *#pqrs*, utilisés dans les instructions de branchement relatif (Gosub ou Goto).
- Le quartet *n* utilisé parfois pour contenir une valeur entre 0 et #Fh=15.

Les instructions **GOYES** et **RNTYES** ne figurent pas dans ce dictionnaire (elles sont en effet associées aux instructions de test).

Dans le cas où deux registres jouent un rôle symétrique, une même instruction pouvait être décrite par plusieurs mnémoniques différents. C'est le cas par exemple, des instructions **A=A!B** et **A=B!A**. Dans ce cas, on n'a fait figurer ici qu'une seule variante.

Le lecteur verra qu'il y a tout de même une logique, qui consiste à associer les registres de travail suivant les couples (A,B), (B,C), (C,A) et (D,C).

00	RTNSXM		0Ea4	B=A&B	*f*
01	RTN		0Ea5	C=B&C	*f*
02	RTNSC		0Ea6	A=C&A	*f*
03	RTNCC		0Ea7	C=D&C	*f*
04	SETHEX		0Ea8	A=A!B	*f*
05	SETDEC		0Ea9	B=B!C	*f*
06	RSTK=C		0EaA	C=C!A	*f*
07	C=RSTK		0EaB	D=D!C	*f*
08	CLRST		0EaC	B=A!B	*f*
09	C=ST		0EaD	C=B!C	*f*
0A	ST=C		0EaE	A=C!A	*f*
0B	CSTEX		0EaF	C=D!C	*f*
0C	P=P+1		0EF0	A=A&B	*A*
0D	P=P-1		0EF1	B=B&C	*A*
0Ea0	A=A&B	*f*	0EF2	C=C&A	*A*
0Ea1	B=B&C	*f*	0EF3	D=D&C	*A*
0Ea2	C=C&A	*f*	0EF4	B=A&B	*A*
0Ea3	D=D&C	*f*	0EF5	C=B&C	*A*

0EF6	A=C&A	*A*	141	DAT1=A	*A*
0EF7	C=D&C	*A*	142	A=DAT0	*A*
0EF8	A=A!B	*A*	143	A=DAT1	*A*
0EF9	B=B!C	*A*	144	DAT0=C	*A*
0EFA	C=C!A	*A*	145	DAT1=C	*A*
0EFB	D=D!C	*A*	146	C=DAT0	*A*
0EFC	B=A!B	*A*	147	C=DAT1	*A*
0EFD	C=B!C	*A*	148	DAT0=A	*B*
0EFE	A=C!A	*A*	149	DAT1=A	*B*
0EFF	C=D!C	*A*	14A	A=DAT0	*B*
0F	RTI		14B	A=DAT1	*B*
100	R0=A		14C	DAT0=C	*B*
101	R1=A		14D	DAT1=C	*B*
102	R2=A		14E	C=DAT0	*B*
103	R3=A		14F	C=DAT1	*B*
104	R4=A		150*a*	DAT0=A	*f*
108	R0=C		151*a*	DAT1=A	*f*
109	R1=C		152*a*	A=DAT0	*f*
10A	R2=C		153*a*	A=DAT1	*f*
10B	R3=C		154*a*	DAT0=C	*f*
10C	R4=C		155*a*	DAT1=C	*f*
110	A=R0		156*a*	C=DAT0	*f*
111	A=R1		157*a*	C=DAT1	*f*
112	A=R2		158*n*	DAT0=A	*(n+1)*
113	A=R3		159*n*	DAT1=A	*(n+1)*
114	A=R4		15A*n*	A=DAT0	*(n+1)*
118	C=R0		15B*n*	A=DAT1	*(n+1)*
119	C=R1		15C*n*	DAT0=C	*(n+1)*
11A	C=R2		15D*n*	DAT1=C	*(n+1)*
11B	C=R3		15E*n*	C=DAT0	*(n+1)*
11C	C=R4		15F*n*	C=DAT1	*(n+1)*
120	AR0EX		16*n*	D0=D0+	*(n+1)*
121	AR1EX		17*n*	D1=D1+	*(n+1)*
122	AR2EX		18*n*	D0=D0-	*(n+1)*
123	AR3EX		19*qp*	D0=(2)	*pq*
124	AR4EX		1A*srqp*	D0=(4)	*pqrs*
128	CR0EX		1B*tsrqp*	D0=(5)	*pqrst*
129	CR1EX		1C*n*	D1=D1-	*(n+1)*
12A	CR2EX		1D*qp*	D1=(2)	*pq*
12B	CR3EX		1E*srqp*	D1=(4)	*pqrs*
12C	CR4EX		1F*tsrqp*	D1=(5)	*pqrst*
130	D0=A		2*n*	P= *n*	
131	D1=A		3*n*h_0...h_n	LC(*n+1*)	h_n..h_0
132	AD0EX		400	RTNC	
133	AD1EX		420	NOP3	
134	D0=C		4*qp*	GOC	*label*
135	D1=C		500	RTNNC	
136	CD0EX		5*qp*	GONC	*label*
137	CD1EX		6300	NOP4	
138	D0=AS		64000	NOP5	
139	D1=AS		6*rqp*	GOTO	*label*
13A	AD0XS		7*rqp*	GOSUB	*label*
13B	AD1XS		800	OUT=CS	
13C	D0=CS		801	OUT=C	
13D	D1=CS		802	A=IN	
13E	CD0XS		803	C=IN	
13F	CD1XS		804	UNCNFG	
140	DAT0=A	*A*	805	CONFIG	

806	C=ID		81Aa01	R1=A.F	*f*
807	SHUTDN		81Aa02	R2=A.F	*f*
8080	INTON		81Aa03	R3=A.F	*f*
80810	RSI		81Aa04	R4=A.F	*f*
8082*n*h_0...h_n	LA(*n*+1)	h_n...h_0	81Aa08	R0=C.F	*f*
8083	BUSCB		81Aa09	R1=C.F	*f*
8084*n*	ABIT=0	*n*	81Aa0A	R2=C.F	*f*
8085*n*	ABIT=1	*n*	81Aa0B	R3=C.F	*f*
8086*nqp*	?ABIT=0	*n*	81Aa0C	R4=C.F	*f*
8087*nqp*	?ABIT=1	*n*	81Aa10	A=R0.F	*f*
8088*n*	CBIT=0	*n*	81Aa11	A=R1.F	*f*
8089*n*	CBIT=1	*n*	81Aa12	A=R2.F	*f*
808A*nqp*	?CBIT=0	*n*	81Aa13	A=R3.F	*f*
808B*nqp*	?CBIT=1	*n*	81Aa14	A=R4.F	*f*
808C	PC=(A)		81Aa18	C=R0.F	*f*
808D	BUSCD		81Aa19	C=R1.F	*f*
808E	PC=(C)		81Aa1A	C=R2.F	*f*
808F	INTOFF		81Aa1B	C=R3.F	*f*
809	C+P+1		81Aa1C	C=R4.F	*f*
80A	RESET		81Aa20	AR0EX.F	*f*
80B	BUSCC		81Aa21	AR1EX.F	*f*
80C*n*	C=P	*n*	81Aa22	AR2EX.F	*f*
80D*n*	P=C	*n*	81Aa23	AR3EX.F	*f*
80E	SREQ?		81Aa24	AR4EX.F	*f*
80F*n*	CPEX	*n*	81Aa28	CR0EX.F	*f*
810	ASLC		81Aa29	CR1EX.F	*f*
811	BSLC		81Aa2A	CR2EX.F	*f*
812	CSLC		81Aa2B	CR3EX.F	*f*
813	DSLC		81Aa2C	CR4EX.F	*f*
814	ASRC		81AF00	R0=A.F	*A*
815	BSRC		81AF01	R1=A.F	*A*
816	CSRC		81AF02	R2=A.F	*A*
817	DSRC		81AF03	R3=A.F	*A*
818*a*0*n*	A=A+(*n*+1)	*f*	81AF04	R4=A.F	*A*
818*a*1*n*	B=B+(*n*+1)	*f*	81AF08	R0=C.F	*A*
818*a*2*n*	C=C+(*n*+1)	*f*	81AF09	R1=C.F	*A*
818*a*3*n*	D=D+(*n*+1)	*f*	81AF0A	R2=C.F	*A*
818*a*8*n*	A=A-(*n*+1)	*f*	81AF0B	R3=C.F	*A*
818*a*9*n*	B=B-(*n*+1)	*f*	81AF0C	R4=C.F	*A*
818*a*A*n*	C=C-(*n*+1)	*f*	81AF10	A=R0.F	*A*
818*a*B*n*	D=D-(*n*+1)	*f*	81AF11	A=R1.F	*A*
818F0*n*	A=A+(*n*+1)	*A*	81AF12	A=R2.F	*A*
818F1*n*	B=B+(*n*+1)	*A*	81AF13	A=R3.F	*A*
818F2*n*	C=C+(*n*+1)	*A*	81AF14	A=R4.F	*A*
818F3*n*	D=D+(*n*+1)	*A*	81AF18	C=R0.F	*A*
818F8*n*	A=A-(*n*+1)	*A*	81AF19	C=R1.F	*A*
818F9*n*	B=B-(*n*+1)	*A*	81AF1A	C=R2.F	*A*
818FA*n*	C=C-(*n*+1)	*A*	81AF1B	C=R3.F	*A*
818FB*n*	D=D-(*n*+1)	*A*	81AF1C	C=R4.F	*A*
819*a*0	ASRB.F	*f*	81AF20	AR0EX.F	*A*
819*a*1	BSRB.F	*f*	81AF21	AR1EX.F	*A*
819*a*2	CSRB.F	*f*	81AF22	AR2EX.F	*A*
819*a*3	DSRB.F	*f*	81AF23	AR3EX.F	*A*
819F0	ASRB.F	*A*	81AF24	AR4EX.F	*A*
819F1	BSRB.F	*A*	81AF28	CR0EX.F	*A*
819F2	CSRB.F	*A*	81AF29	CR1EX.F	*A*
819F3	DSRB.F	*A*	81AF2A	CR2EX.F	*A*
81Aa00	R0=A.F	*f*	81AF2B	CR3EX.F	*A*

81AF2C CR4EX.F *A*
81B2 PC=A
81B3 PC=C
81B4 A=PC
81B5 C=PC
81B6 APCEX
81B7 CPCEX
81C ASRB
81D BSRB
81E CSRB
81F DSRB
821 XM=0
822 SB=0
824 SR=0
828 MP=0
82F CLRHST
82*n* HS=0 *n*
831*qp* ?XM=0
832*qp* ?SB=0
834*qp* ?SR=0
838*qp* ?MP=0
83*nqp* ?HS=0 *n*
84*n* ST=0 *n*
85*n* ST=1 *n*
86*nqp* ?ST=0 *n*
87*nqp* ?ST=1 *n*
88*nqp* ?P# *n*
89*nqp* ?P= *n*
8A0*qp* ?A=B *A*
8A1*qp* ?B=C *A*
8A2*qp* ?C=A *A*
8A3*qp* ?D=C *A*
8A4*qp* ?A#B *A*
8A5*qp* ?B#C *A*
8A6*qp* ?C#A *A*
8A7*qp* ?D#C *A*
8A8*qp* ?A=0 *A*
8A9*qp* ?B=0 *A*
8AA*qp* ?C=0 *A*
8AB*qp* ?D=0 *A*
8AC*qp* ?A#0 *A*
8AD*qp* ?B#0 *A*
8AE*qp* ?C#0 *A*
8AF*qp* ?D#0 *A*
8B0*qp* ?A>B *A*
8B1*qp* ?B>C *A*
8B2*qp* ?C>A *A*
8B3*qp* ?D>C *A*
8B4*qp* ?A<B *A*
8B5*qp* ?B<C *A*
8B6*qp* ?C<A *A*
8B7*qp* ?D<C *A*
8B8*qp* ?A>=B *A*
8B9*qp* ?B>=C *A*
8BA*qp* ?C>=A *A*
8BB*qp* ?D>=C *A*
8BC*qp* ?A<=B *A*
8BD*qp* ?B<=C *A*
8BE*qp* ?C<=A *A*
8BF*qp* ?D<=C *A*
8C*srqp* GOLONG *label*
8D*tsrqp* GOVLNG *label*
8E*srqp* GOSUBL *label*
8F*tsrqp* GOSBVL *label*
9a0*qp* ?A=B *f*
9a1*qp* ?B=C *f*
9a2*qp* ?C=A *f*
9a3*qp* ?D=C *f*
9a4*qp* ?A#B *f*
9a5*qp* ?B#C *f*
9a6*qp* ?C#A *f*
9a7*qp* ?D#C *f*
9a8*qp* ?A=0 *f*
9a9*qp* ?B=0 *f*
9aA*qp* ?C=0 *f*
9aB*qp* ?D=0 *f*
9aC*qp* ?A#0 *f*
9aD*qp* ?B#0 *f*
9aE*qp* ?C#0 *f*
9aF*qp* ?D#0 f
9b0*qp* ?A>B *f*
9b1*qp* ?B>C *f*
9b2*qp* ?C>A *f*
9b3*qp* ?D>C *f*
9b4*qp* ?A<B *f*
9b5*qp* ?B<C *f*
9b6*qp* ?C<A *f*
9b7*qp* ?D<C *f*
9b8*qp* ?A>=B *f*
9b9*qp* ?B>=C *f*
9bA*qp* ?C>=A *f*
9bB*qp* ?D>=C *f*
9bC*qp* ?A<=B *f*
9bD*qp* ?B<=C *f*
9bE*qp* ?C<=A *f*
9bF*qp* ?D<=C *f*
Aa0 A=A+B *f*
Aa1 B=B+C *f*
Aa2 C=C+A *f*
Aa3 D=D+C *f*
Aa4 A=A+A *f*
Aa5 B=B+B *f*
Aa6 C=C+C *f*
Aa7 D=D+D *f*
Aa8 B=A+B *f*
Aa9 C=B+C *f*
AaA A=C+A *f*
AaB C=D+C *f*
AaC A=A-1 *f*
AaD B=B-1 *f*
AaE C=C-1 *f*
AaF D=D-1 *f*
Ab0 A=0 *f*
Ab1 B=0 *f*
Ab2 C=0 *f*
Ab3 D=0 *f*

Ab4	A=B f	CE	C=C-1 A
Ab5	B=C f	CF	D=D-1 A
Ab6	C=A f	D0	A=0 A
Ab7	D=C f	D1	B=0 A
Ab8	B=A f	D2	C=0 A
Ab9	C=B f	D3	D=0 A
AbA	A=C f	D4	A=B A
AbB	C=D f	D5	B=C A
AbC	ABEX f	D6	C=A A
AbD	BCEX f	D7	D=C A
AbE	CAEX f	D8	B=A A
AbF	DCEX f	D9	C=B A
Ba0	A=A-B f	DA	A=C A
Ba1	B=B-C f	DB	C=D A
Ba2	C=C-A f	DC	ABEX A
Ba3	D=D-C f	DD	BCEX A
Ba4	A=A+1 f	DE	CAEX A
Ba5	B=B+1 f	DF	DCEX A
Ba6	C=C+1 f	E0	A=A-B A
Ba7	D=D+1 f	E1	B=B-C A
Ba8	B=B-A f	E2	C=C-A A
Ba9	C=C-B f	E3	D=D-C A
BaA	A=A-C f	E4	A=A+1 A
BaB	C=C-D f	E5	B=B+1 A
BaC	A=B-A f	E6	C=C+1 A
BaD	B=C-B f	E7	D=D+1 A
BaE	C=A-C f	E8	B=B-A A
BaF	D=C-D f	E9	C=C-B A
Bb0	ASL f	EA	A=A-C A
Bb1	BSL f	EB	C=C-D A
Bb2	CSL f	EC	A=B-A A
Bb3	DSL f	ED	B=C-B A
Bb4	ASR f	EE	C=A-C A
Bb5	BSR f	EF	D=C-D A
Bb6	CSR f	F0	ASL A
Bb7	DSR f	F1	BSL A
Bb8	A=-A f	F2	CSL A
Bb9	B=-B f	F3	DSL A
BbA	C=-C f	F4	ASR A
BbB	D=-D f	F5	BSR A
BbC	A=-A-1 f	F6	CSR A
BbD	B=-B-1 f	F7	DSR A
BbE	C=-C-1 f	F8	A=-A A
BbF	D=-D-1 f	F9	B=-B A
C0	A=A+B A	FA	C=-C A
C1	B=B+C A	FB	D=-D A
C2	C=C+A A	FC	A=-A-1 A
C3	D=D+C A	FD	B=-B-1 A
C4	A=A+A A	FE	C=-C-1 A
C5	B=B+B A	FF	D=-D-1 A
C6	C=C+C A		
C7	D=D+D A		
C8	B=A+B A		
C9	C=B+C A		
CA	A=C+A A		
CB	C=D+C A		
CC	A=A-1 A		
CD	B=B-1 A		

UN ASSEMBLEUR EN EXTERNALs

Il est tout à fait illusoire de vouloir écrire en langage-machine si on ne dispose pas d'un assembleur. Ce chapitre est consacré à la présentation d'un tel programme.

L'assembleur que nous allons considérer est entièrement écrit en Externals. Celà montre qu'il est vain de vouloir opposer le "Rpl-Système" et le langage-machine. Ces deux langages sont faits pour s'entendre...

Ce chapitre est le point d'orgue de la programmation en Rpl-Système que j'ai voulu présenter dans ce livre. Les commentaires pour les différents programmes sont ici très détaillés. Vous n'aurez plus d'excuses pour ne pas vous lancer, vous aussi...

◆ L'ASSEMBLEUR ASM48

Ce chapitre est donc consacré à un assembleur, que nous appelerons **ASM48**. Il s'agit en fait d'un ensemble de routines, parmi lesquelles le programme **ASM48** lui-même est le programme principal.

Il s'agit en fait d'un programme qui vous permettra un peu plus que d'assembler des mnémoniques du Saturn. Le programme **ASM48** autorise en effet la création de toutes sortes d'objets utilisables sur la HP48 et en particulier: des objets-code, des structures Rpl contenant des objets-code, des objets de type "***Linked Array***"...

Tous les listings de ce chapitre sont donnés au format nécessaire à l'utilisation du programme **EXTERNPC**. Un mini-dictionnaire des mnémoniques utilisés, présenté dans les pages suivantes, vous permet de traduire ces listings dans le format utilisé par le programme **→EXT** (en même temps qu'il vous indique, si vous utilisez **EXTERNPC**, ce que vous devez mettre dans votre fichier de mnémoniques).

Il faut tout d'abord noter que cet assembleur fait appel à trois programmes qui ont déjà été présentés dans ce livre:

- Le programme **→STRI**, qui transforme un objet en une chaîne de caractères représentant l'image de cet objet en mémoire.
- Le programme **→OBJ**, qui réalise l'opération inverse de **→STRI**. Une chaîne de caractères représentant un objet est transformée en cet objet.
- Le programme **→ADR**, qui donne, sous la forme d'un entier-binaire, l'adresse de l'objet situé au niveau 1 de la pile.

Il va de soi que vous devez d'abord créer ces programmes avant de vouloir utiliser l'assembleur **ASM48**.

Le schéma fonctionnel du programme **ASM48** est le suivant:

- On place au niveau 1 de la pile une chaîne de caractères contenant le code à assembler, sous forme de mnémoniques, et à raison d'une seule instruction par ligne.
- On appelle le programme **ASM48**, qui transforme cette chaîne en un objet de type code, ou en une structure Rpl (ou en autre chose) selon ce que vous aurez mis dans la chaîne initiale.
- Si votre chaîne-source contient des erreurs de syntaxe, ou d'autres erreurs rendant impossible la création du code objet, le programme **ASM48** vous rend la main en laissant sur la pile la ligne (extraite de la chaîne initiale) où s'est produite l'erreur.

Le programme **ASM48** reconnaît tous les mnémoniques du **Saturn**, telles qu'ils ont été donnés dans le chapitre précédent, à quelques exceptions près cependant. Si vous avez un doute, il vous est conseillé de consulter les contenus des variables **ST3**, **ST4**, **ST5**, **ST56**, **ST6** et **ST7** (voir plus loin) qui indiquent quels sont les mnémoniques utilisables.

Il est des cas où plusieurs mnémoniques synonymes pourraient s'appliquer à une même opération. On pense par exemple à une instruction du type **A=A&B**, qu'on pourrait tout aussi bien écrire **A=B&A**. L'assembleur **ASM48** ne connaît qu'un seul mnémonique pour chaque instruction.

◆ PARTICULARITÉS DE SYNTAXE

Tout assembleur, outre sa fonction principale de transformer des mnémoniques en codes exécutables, doit régler un certain nombre de problèmes liés à l'écriture du code-source, comme les labels, les commentaires, les arguments acceptés par telle ou telle instruction, les possibilités de lier le fichier principal à d'autres fichiers secondaires (déjà assemblés ou non), etc...

Ces opérations nécessitent une syntaxe particulière dans le code-source. Voici ce qu'il en est pour le programme **ASM48**:

LA STRUCTURE DE LA LIGNE

Chaque ligne de la chaîne initiale ne peut contenir au plus qu'une seule instruction. Les espaces qui débutent éventuellement la ligne sont sans importance. Une ligne vide, ou constituée exclusivement d'espaces, est tout simplement ignorée.

Les différentes instructions du **Saturn** se groupent en deux catégories suivant qu'elles s'écrivent en une seule partie, ou en deux parties successives.

Le sous-programme **NETLINE** a pour rôle de "nettoyer" une ligne pour en extraire les deux premières parties.

- Les espaces de tête sont ignorés.
- La première partie de la ligne s'arrête au premier espace après le premier mot (ou au saut de ligne).
- La seconde partie de la ligne commence au premier caractère du deuxième mot (s'il existe) et s'arrête au premier espace rencontré dans ce mot (ou au saut de ligne).
- Il y a une exception quand le deuxième mot de la ligne commence par le caractère ' ou le caractère /. Dans ce cas, tout ce qui suit dans cette ligne est conservé, même s'il y a des espaces.

Quand une instruction en un seul mot est reconnue, tout ce qui suit dans la ligne est ignoré. De la même manière est ignoré tout ce qui vient après les deux mots d'une instruction en deux parties.

LES COMMENTAIRES

Vous pouvez insérer des commentaires de plusieurs manières. Comme on vient de le voir, vous pouvez les placer à la suite de l'instruction qui débute la ligne, que cette instruction soit en un mot ou en deux (mais évidemment pas entre les deux parties d'une même instruction).

Vous pouvez de même réserver une ligne entière comme commentaire. Il vous suffit de la faire débuter (après éventuellement quelques espaces sans importance), par le caractère * (astérique). Celà permet aussi de soustraire une ligne, au moins provisoirement, au processus d'assemblage.

LES LABELS

Les labels, ou étiquettes, sont des noms désignant tel ou tel point de votre code (de manière à favoriser les branchements, et à améliorer la lisibilité de votre programme). Un label peut être déclaré sans être utilisé, mais le contraire n'est évidemment pas vrai.

Pour déclarer un label, vous ferez débuter la ligne par le caractère : (deux points) et vous ferez suivre ce caractère du nom de ce label. Celui-ci doit suivre la syntaxe des noms de variable de la HP48 (et par exemple ne pas commencer par un chiffre).

Une dernière remarque, évidente: vous ne pouvez pas utiliser deux fois le même nom, pour désigner deux points différents.

LES BRANCHEMENTS

On sait qu'on distingue les branchements absolus (vers une adresse déterminée en mémoire, en principe en ROM) des branchements relatifs (en principe d'un point de votre code à un autre).

Les branchements relatifs doivent ici tous s'effectuer vers un label. Il se peut que ce label ne soit pas encore déclaré: ce problème est géré par le programme **ASM48**.

Bien entendu, une erreur finale interviendra si vous avez utilisé un label comme but d'un branchement, sans jamais positionner ce label dans votre code-source.

LES VARIABLES LOCALES

Il est possible de donner un nom à certaines constantes, de manière à favoriser la relecture et la portabilité de votre code. Pour définir une variable, vous devez faire débuter la ligne par un caractère = (égal).

Ce caractère doit être suivi immédiatement du nom de la variable (syntaxe des noms de la HP48). La deuxième partie de la liste contient alors la constante à placer dans cette variable. Cette constante peut être un entier-binaire ou un réel.

Ce nom (créé par le programme **ASM48** comme variable locale) pourra alors être utilisé par les instructions de votre code-source, en lieu et place d'un argument qui aurait du être de type réel ou de type entier binaire. Vous ne pouvez pas déclarer un même nom (comme variable ou comme label) à deux endroits différents de votre code-source (le programme **ASM48** vous signalerait aussitôt l'erreur).

LIER UN FICHIER SECONDAIRE

Vous pouvez à tout moment inclure des fichiers-source secondaires à votre fichier-source principal. Pour celà, il vous suffit de faire débuter la ligne par le symbole **[** (crochet ouvrant) suivi immédiatement du nom de la variable contenant ce que vous voulez inclure.

Le nom de cette variable peut être une chaîne ou un objet-code.

- Si c'est une chaîne, elle est interprétée comme une séquence d'instructions, et est immédiatement assemblée par le programme **ASM48**. Tout se passe donc comme si cette chaîne venait s'insérer dans le code-source principal.

  Il est tout à fait possible que le fichier secondaire fasse lui-même appel à d'autres fichiers, sans limitation autre que la mémoire. Evitez évidemment les appels récursifs!
- Si c'est un objet-code, il est immédiatement concaténé à ce que le programme **ASM48** a déjà assemblé. Il est ainsi possible d'utiliser des outils déjà assemblés, sans procéder à nouveau à leur compilation.

ARGUMENTS PARTICULIERS

Dans certaines instructions il est possible d'utiliser des arguments de différents types. Si par exemple une instruction attend un argument entier, vous pourrez donner cet entier sous forme réelle, sous forme d'entier binaire, sous forme d'un nom local (désignant une variable locale précédemment déclarée) ou sous la forme d'une expression (cette dernière étant alors évaluée, et pouvant faire référence à des noms de variables).

Il est par exemple tout à fait possible d'envisager une instruction qui s'écrirait:
LC(5) 'table + 5'

Il est difficile de donner le détail de toutes les possibilités. Il est préférable que vous en preniez connaissance, pour chaque instruction qui vous intéresse, dans les commentaires qui accompagneront le listing du programme **ASM48** et de ses sous-programmes).

◆ INSTRUCTIONS SUPPLÉMENTAIRES

Le programme **ASM48** vous permet d'utiliser un certain nombre d'instructions supplémentaires, qui faciliteront la création de vos codes-objets.

INSTRUCTION NUL

Suivie d'un argument qui doit être un entier **n** (réel ou binaire) ou s'évaluer sur un tel entier, elle crée **n** quartets nuls dans le code-objet. Celà vous permet de créer des tables à l'intérieur de votre programme, qui peut alors s'auto-modifier.

INSTRUCTION HEX

Elle doit être suivie d'une séquence de caractères héxas (de 0 à 9 et/ou de A à F), elle ajoute alors cette séquence de caractères (chacun d'eux occupera un quartet) dans le code-objet en cours d'assemblage.

L'instruction "**HEX ABC**" inclut donc les 3 quartets **ABC** dans le code-objet.

INSTRUCTION ASC

Elle doit être suivie d'une séquence quelconque de caractères. Si le premier d'entre eux est le caractère /, il sera enlevé mais c'est le moyen d'inclure des espaces dans cette séquence. Cette séquence de caractères est alors ajoutée au code-objet déjà assemblé: chaque caractère est codé sur deux quartets.

L'instruction "**HEX ABC**" inclut donc les 6 quartets **142434** dans le code-objet (le code Ascii de A est #41h, celui de B est #42, celui de C est #43).

INSTRUCTION EXE

Elle peut être suivie d'un entier binaire, ou d'un nom (local, global, ou nom XLIB) ou d'un objet intégré à la HP48. L'entier binaire est interprété comme une adresse et ajouté, sur cinq quartets, au code-objet déjà assemblé.

Les noms sont également ajoutés (sous la forme qui est la leur en mémoire) sans être remplacés par leur contenu éventuel. Les objets-intégrés à la HP48 sont ajoutés, sur 5 quartets, sous la forme de leur adresse en ROM.

Exemples:

- "**EXE ABC**" inclut les quartets **84E2030142434** dans le code-objet (si la variable ABC est connue comme nom local, ce sont les quartets **D6E2030142434** qui sont ajoutés).
- "**EXE #3188**" inclut les quartets **88130** dans le code-objet (dans une structure Rpl, c'est l'adresse de la forme interne de l'instruction **DUP**).
- "**EXE DUP**" inclut les quartets **78BF1** dans le code-objet (l'adresse en ROM de l'instruction **DUP** est effectivement #1FB87h)

INSTRUCTIONS INC ET inc

Elles permettent d'ajouter le contenu d'un objet (tel qu'il se présenterait en mémoire) au code-objet déjà assemblé. La différence entre **INC** et **inc** est que l'instrucion **inc** élimine les 5 quartets représentant l'adresse-préfixe de l'objet à inclure.

L'argument de ces deux instructions peut-être quelconque. Si c'est un nom et si ce nom est connu, alors il est remplacé par son contenu.

L'argument de ces deux instructions est interprété comme une ligne de commande. Il peut contenir des espaces à condition que vous le fassiez débuter par le caractère / (qui est d'ailleurs éliminé avant traitement de cet argument).

Exemple: l'instruction "**INC {1}**" inclut les quartets **47A209C2A2B2130** dans le code-objet. L'instruction "**inc {1}**" n'inclut que les quartets **9C2A2B2130**.

INSTRUCTIONS REL2, REL3, REL4, REL5

Elles permettent d'inclure, dans le code-objet en cours d'assemblage, des offsets (décalages) codés sur 2, 3, 4 ou 5 quartets. L'argument de ces instructions doit être un label. Le décalage est calculé en faisant la différence entre l'adresse de ce label, et l'adresse du quartet où se situe l'instruction **REL**.

Ceci est particulièrement utile pour coder des objets dont la structure contient des offsets ou des longueurs (par exemple les objets-code dont les quartets 6 à 10 contiennent la longueur de l'objet hors adresse-préfixe). C'est également un moyen pratique, dans un objet-code, de créer des tables d'adresses relatives (utile pour gérer des tests multiples).

Exemple: Pour créer un objet-code avec **ASM48**, vous utiliserez une chaîne ayant la forme suivante:

```
....                    Ce qui précède l'objet-code
HEX     CCD20           Adresse préfixe des objets-code
REL5    fin             Offset sur 5 quartets
....                       Le contenu de l'objet-code
:fin                    Label indiquant la fin de l'objet
....                    Ce qui suit l'objet-code
```

DEUX EXEMPLES SIMPLES

Imaginons que vous ayez placé dans une variable '**ADR**' la chaîne de caractères suivante:

```
=SaveRegs  #0679B       Adresse pour sauver registres
=SafeExit  #05143       Les récupère et retour Rpl
```

Le programme suivant (qui est tranformé par **ASM48** en objet-code) attend un entier-système au niveau 1, et lui ajoute 1. Evidemment l'intérêt est ici proche de zéro: c'est juste un exemple...

```
[ADR                    Lit déclarations de var locales
HEX     CCD20           Adresse-préfixe de l'objet-code
REL5    fin             Génère la longueur hors préfixe
GOSBVL  SaveRegs        Sauvegarde registres B,D,D0,D1
A=DAT1  A               Lit adresse du niveau 1
D1=A                    Pointe sur objet du niveau 1
D1=D1+  5               Saute l'adresse-préfixe
A=DAT1  A               Lit le contenu de l'entier-système
A=A+1   A               Ajoute 1 à ce contenu
DAT1=A  A               Ecrit le nouveau contenu
GOVLNG  SafeExit        Récupère registres et retour Rpl
:fin                    Label de fin de l'objet-code
```

Ce deuxième exemple montre une structure Rpl dans laquelle est insérée un objet-code. Il s'agit de calculer la somme des différents chiffres de la mantisse d'un nombre réel, présent au niveau 1 de la pile (sur une idée de **Robert PULLUARD**, que je salue au passage, pour la revue **Haute-Performance**).

Contrairement au programme précédent qui n'effectuait aucun test sur la présence d'un argument correct sur la pile, le programme suivant vérifie qu'un réel se trouve bien au niveau 1.

Le réel utilisé pour calculer la somme est éliminé. Le résultat est placé au niveau 1, sous la forme d'un entier-système.

```
HEX       D9D20          Préfixe des structures RPL
EXE       #18AB2         Instruction need1
EXE       #18FB2         Instruction ChkArg
HEX       1192010000     Entier-système <1h> (test réel)
HEX       CCD20          Préfixe de objets-code
REL5      fin            Longueur hors préfixe
A=DAT1    A              Adresse de l'objet du niveau 1
D1=D1+    5              Pointe sur niveau 2
D=D+1     A              5 quartets de + de mémoire
GOSBVL    #679B          Sauve registres
D1=A                     Pointe sur ex-objet du niveau 1
D1=D1+    8              Pointe sur début mantisse
P=        4              Prépare compteur de boucle
A=0       A              Nettoie A
C=0       A              Nettoie C
:Loop                    Label de début de boucle
C=DAT1    1              Lit un quartet, qui va dans C
A=A+C     A              L'ajoute au contenu de A
D1=D1+    1              Position suivante
P=P+1                    Incrémente compteur
GONC      Loop           Continuer si P<>0
GOVLNG    #357Ch         Sortie et <A> sur la pile
:fin                     Label de fin de l'objet-code
HEX       B2130          Suffixe des structures Rpl
```

Le programme précédent est compilé par le programme **ASM48** en 5 secondes. L'objet obtenu est une structure Rpl dont l'image sur la pile est la suivante:

```
External   External   <1h>   Code
```

AVANT D'ALLER PLUS LOIN

Nous allons maintenant découvrir **ASM48**. La taille de l'ensemble des programmes appelés par le programme principal est de 7000 octets. Difficile de faire plus court pour un tel projet. Enfin, **ASM48** compile une chaîne-source à la vitesse approximative de 8 lignes par seconde.

◆ LES MNÉMONIQUES UTILISÉS

Le programme **ASM48** et tous ses sous-programmes font appel à un grand nombre d'Externals. Pour des raisons de place, mais aussi de clarté, il n'a pas semblé utile de présenter les listings sous la forme de listes d'adresses (telles que celles sur lesquelles agit le programme **→EXT**). Les listings ne seront donc donnés que sous le format attendu par le programme **EXTERNPC**.

Pour ne pas léser les lecteurs qui, ne disposant pas d'un PC, sont contraints d'utiliser le programme **→EXT**, la liste ci-dessous contient tous les mnémoniques appelés par le programme **ASM48** et ses sous-programmes. De cette manière, vous pourrez reconstituer les listings nécessaires à l'utilisation du programme **→EXT** (il vous faudra cependant une bonne dose de courage).

Cette liste est également utile aux lecteurs qui utiliseront le programme **EXTERNPC**. Elle permet en effet de constituer le fichier de mnémoniques minimum pour créer le programme **ASM48** et tous ses sous-programmes (pour les entiers-système, on rappelle que le programme **EXTERNPC** les traduit lui-même, ce qui ne rend pas nécessaire l'usage d'un mnémonique).

```
$# ..................................#65433h
$' ..................................#6947Ch
$/ ..................................#6545Dh
$a ..................................#65583h
$spc ................................#65686h
'd ..................................#5A665h
'r ..................................#5A670h
+(2cp) ..............................#0521Fh
+(2s) ...............................#03DBCh
+(2st)! .............................#05193h
+(2st)!;swap ........................#63F6Ah
+(st,ch) ............................#052EEh
+(st,RC) ............................#63191h
-(2s) ...............................#03DE0h
-(2s);dup ...........................#627F8h
<>0(s)? .............................#03CC7h
<0h> ................................#03FEFh
<0h>;false ..........................#6351Fh
<0h>;swap ...........................#62E3Ah
<10h> ...............................#0408Fh
<14h> ...............................#040B7h
<18h> ...............................#040DFh
<1Ah> ...............................#040F3h
<1Ch> ...............................#04107h
<1Fh> ...............................#04125h
<1h> ................................#03FF9h
<20h> ...............................#0412Fh
<22h> ...............................#04143h
<23h> ...............................#0414Dh
<24h> ...............................#04157h
<28h> ...............................#0417Fh
<29h> ...............................#04189h
<2Ah> ...............................#04193h
<2Dh> ...............................#64B1Ch
<2Eh> ...............................#64B26h
<2h> ................................#04003h
<30h> ...............................#64B3Ah
<31h> ...............................#64B44h
<32h> ...............................#64B4Eh
<36h> ...............................#64B76h
<37h> ...............................#64B80h
<38h> ...............................#64B8Ah
<39h> ...............................#64B94h
<3Ah> ...............................#64B9Eh
<3Bh> ...............................#64BA8h
<3Ch> ...............................#64BB2h
<3Dh> ...............................#64BBCh
<3Eh> ...............................#64BC6h
<3Fh> ...............................#64BD0h
<3h> ................................#0400Dh
<40h> ...............................#64BDAh
<41h> ...............................#64BE4h
<4Ah> ...............................#64C20h
<4h> ................................#04017h
<51h> ...............................#64C3Eh
<53h> ...............................#64C52h
<56h> ...............................#64C70h
<57h> ...............................#64C7Ah
<5h> ................................#04021h
<60h> ...............................#64C8Eh
<62h> ...............................#64CA2h
<65h> ...............................#64CB6h
<6h> ................................#0402Bh
<7h> ................................#04035h
<80h> ...............................#64D10h
<8h> ................................#0403Fh
<9h> ................................#04049h
<Ah> ................................#04053h
<Bh> ................................#0405Dh
<Ch> ................................#04067h
<Eh> ................................#0407Bh
<Fh> ................................#04085h
=(2s)? ..............................#03D19h
=(2s)?:skip .........................#6336Ch
=(2s)?;pifte ........................#62C2Dh
=0(s)?;pifte ........................#63E89h
=1(s)?:dq/sk ........................#63D26h
>(2s)? ..............................#03D83h
>(2s)?;pifte ........................#63EB1h
?:dq/sk .............................#61993h
?:drdq/sk ...........................#618F7h
?:q .................................#61A3Bh
?:skip ..............................#0712Ah
( ...................................#02A74h
('d'r) ..............................#5A660h
() ..................................#055E9h
) ...................................#0312Bh
1+(r) ...............................#50262h
1+(s) ...............................#03DEFh
1-(s) ...............................#03E0Eh
2*(s) ...............................#03E6Fh
2+(s) ...............................#03E2Dh
2-(s) ...............................#03E4Eh
3->list .............................#631CDh
```

3pick ..... #611FEh
3pick;+(2s) ..... #63740h
3rolld ..... #60FACh
3rolld;drop ..... #6284Bh
4-(s) ..... #6260Ah
4pick ..... #6121Ch
4roll ..... #60FBBh
4rolld ..... #6109Eh
5pick ..... #6123Ah
5roll ..... #60FD8h
6+(s) ..... #6259Ah
6roll ..... #61002h
8*(s) ..... #62674h
8+(s) ..... #625BAh
and(2?) ..... #03B46h
b=>s ..... #05A03h
begin ..... #02D9Dh
break;drop ..... #62B88h
ChkArg ..... #18FB2h
ChkArg! ..... #18F9Dh
CutRC(st) ..... #127A7h
del1(st) ..... #0516Ch
do ..... #071A2h
doerr(s) ..... #6383Ah
doerr(st) ..... #15048h
doerr001 ..... #04FB6h
doerr002 ..... #04FC2h
doerr003 ..... #04FCEh
doerr004 ..... #05016h
doerr006 ..... #04FAAh
doerr008 ..... #04FDAh
doerr009 ..... #04FE6h
drop ..... #03244h
drop(2) ..... #60F9Bh
drop(2-3) ..... #6112Ah
drop(3) ..... #60F21h
drop;"" ..... #04D3Eh
drop;dup ..... #627A7h
drop;rot ..... #62FC5h
drop2 ..... #03258h
dropn(s) ..... #0326Eh
DropRet ..... #06FB7h
dup ..... #03188h
dup;""=? ..... #63209h
dup;=0(r)? ..... #63BAAh
dup;=0(s)?:dq/sk ..... #61891h
dup;=0(s)?:drdq/sk ..... #618A8h
dup;=0(s)?;pifte ..... #63EC5h
dup;1+(s) ..... #628EBh
dup;2+(s) ..... #626F7h
dup;b? ..... #6213Fh
dup;ex? ..... #621D2h
dup;gn? ..... #62035h
dup;index ..... #63411h
dup;known(nm)? ..... #62A2Fh
dup;li? ..... #62211h
dup;ln? ..... #62115h
dup;r? ..... #62169h
dup;rcl(nm)? ..... #62C05h
dup;ssize(st) ..... #627BBh
dup;start(1,szcp) ..... #63498h
dup;StdObj? ..... #61FA9h
dup;≠ ..... #63925h
dup2 ..... #031ACh
dup2;+(2s) ..... #63704h
end ..... #0312Bh
endwhile ..... #071E5h
error! ..... #04ED1h
eval ..... #06F8Eh
eval(cp) ..... #06F9Fh
false ..... #03AC0h
get(cp,s) ..... #62B9Ch
get(cp,s);dup ..... #62D1Dh
get(cp,s)? ..... #056B6h
gn? ..... #6203Ah
HardSto(_,nm) ..... #07D27h
hex ..... #53C37h

IfError ..... #04E5Eh
InCp(_,cp)? ..... #643EFh
Ldef(_,li) ..... #074D0h
Ldef(_,s) ..... #074E4h
makeb(s) ..... #0EDE1h
MapStr(b) ..... #15E83h
max(2s) ..... #624C6h
mdv(2s) ..... #03EF7h
need1 ..... #18AB2h
nm=>ln ..... #05AEDh
Null"" ..... #055DFh
Null"";swap ..... #62D59h
or(2?) ..... #03B75h
over ..... #032C2h
over;+(2s) ..... #6372Ch
over;<(2s)? ..... #5853Eh
over;=(2s)? ..... #620EBh
over;=(2s)?:drdq/sk ..... #618D3h
over;=(2st)?:drdq/sk ..... #2856Ch
over;>(2s)? ..... #636DCh
over;ssize(st) ..... #05622h
pift ..... #619BCh
pifte ..... #61AD8h
pos(2st) ..... #1CAD7h
prep(st,ch) ..... #0525Bh
r+=>s ..... #18CEAh
r=>b ..... #543F9h
rcl(nm) ..... #20B81h
rcl(nm)? ..... #07978h
rclerr ..... #04CE6h
repeat ..... #071EEh
repl(st,r,st) ..... #4FAF7h
rot ..... #03295h
rot;dup ..... #62775h
s=>r ..... #18DBFh
same? ..... #03B97h
sdepth ..... #0314Ch
snum(st,s) ..... #30805h
space ..... #65254h
spos(_,cp) ..... #644BCh
spos(2st) ..... #15EF6h
spos(cp,_) ..... #644A3h
ssize(st) ..... #05636h
st=>ch ..... #050EDh
st? ..... #62159h
step(s) ..... #073A5h
sub(cp,2s) ..... #05821h
sub(st,1,s-1) ..... #63259h
sub(st,2s) ..... #05733h
sub(st,s+1,∞) ..... #63281h
sub(st,s,∞) ..... #6326Dh
sub(st,s2,s1-1) ..... #63245h
sub(st,s2,s1-1) ..... #63245h
sub(st,s2,s1-1) ..... #63245h
sub(st,s2,s1-1) ..... #63245h
swap ..... #03223h
swap(1,3) ..... #60F33h
swap(2,3) ..... #60EE7h
swap;+(2st)! ..... #622EFh
swap;-(2s) ..... #62794h
swap;<1h> ..... #63AB0h
swap;1+(s) ..... #62904h
swap;dup ..... #62747h
swap;false ..... #2164Ch
swap;over ..... #61380h
swap;true ..... #4F1D8h
swap2 ..... #62001h
SyntaxOK(st)? ..... #238A4h
ThenError ..... #04EB8h
true ..... #03A81h
true;≠ ..... #63989h
until ..... #071C8h
while ..... #071A2h
≠ ..... #06E97h

◆ LE PROGRAMME ASM48

C'est ici le programme principal. Comme vous pouvez le constater, il est relativement court. Voici comment il fonctionne:

▸a On vérifie que le niveau 1 de la pile est occupé par une chaîne.
▸b On passe en mode héxadécimal.
▸c On crée deux variables locales nommées **'d** et **'r**. La variable locale **'r** jouera le rôle d'un "*drapeau*" booléen indiquant si l'instruction précédemment assemblée est une instruction de test (et doit donc être suivie d'un **GOYES** ou d'un **RNTYES**). **'d** contiendra la ligne en cours d'assemblage.
▸d On crée les conditions permettant de démarrer l'assemblage: une chaîne vide (destinée à recevoir les quartets du code) est placée au niveau 2 de la pile, juste au-dessus de la chaîne de caractères qui va être assemblée.
▸e On entre ensuite dans le vif du sujet. Le coeur du programme est contrôlé par une structure **IfError...ThenError**, dont nous reparlerons plus loin.

▸f La chaîne est découpée au niveau de son premier saut de ligne par l'instruction **CutRC(st)**. La pile a alors l'aspect suivant:
3: Chaîne contenant le code déjà assemblé.
2: Ce qui reste de la chaîne à assembler.
1: La ligne qui doit maintenant être assemblée.

▸g La ligne à assembler est placée dans la variable locale **'d**. Elle est "*nettoyée*" par le programme **NETLINE**, qui sera décrit ci-après, mais qui laisse la pile dans l'état suivant:
4: Chaîne contenant le code déjà assemblé.
3: Ce qui reste de la chaîne à assembler.
2: Le premier champ de la ligne à assembler.
1: Le deuxième champ (éventuellement vide) de cette ligne.

▸h Si le premier champ obtenu est vide, c'est que la ligne à assembler était vide (ou formée uniquement d'espaces). On se contente alors de détruire les deux champs vides des niveaux 1 et 2.
▸i Sinon on appelle le sous-programme **ASSLINE** qui est chargé de vérifier la syntaxe des deux champs obtenus, et de les remplacer par le code correspondant, qui sera ajouté à la fin de la chaîne située en haut de la pile.
▸j A ce stade, la pile à l'allure suivante:
2: Chaîne contenant le code déjà assemblé.
1: Ce qui reste de la chaîne à assembler.
▸k On recommence cette séquence d'opérations, depuis ▸f, à moins que la chaîne restante soit vide, ce qui signifie que l'assemblage est terminé. On détruit alors la chaîne vide résiduelle et on appelle le programme **→OBJ**, qui transforme la chaîne en l'objet souhaité (objet code, structure Rpl, etc...).

Voici maintenant le listing du programme **ASM48**, à utiliser avec **EXTERNPC**:

```
begin
  need1   ChkArg   <3h>
  begin
     hex   <0h>;false   {'d'r}   Ldef(_,li)   Null"";swap
     IfError
     begin
        do
           CutRC(st)   dup;≈   'd   HardSto(_,nm)   NETLINE
           over;ssize(st)   =0(s)?;pifte   drop2   ASSLINE
           dup;""=?
        until
     drop   ->OBJ
     end
     ThenError
     begin
        sdepth   dropn(s)   'd   LSTERR   rclerr
        get(cp,s)?   pifte   doerr(st)   error!
     end
  end
end
```

◆ LE TRAITEMENT DES ERREURS

L'essentiel du fonctionnement du programme s'effectue sous le contrôle d'une structure **IfError...ThenError**, dont le rôle est de rediriger les erreurs qui peuvent se produire ici ou là.

Si une situation d'erreur est détectée lors de l'assemblage (instruction inconnue, mauvaise syntaxe, duplication d'un label, etc...), une instruction du type **doerr** est générée avec un numéro compris entre 1 et 9.

La clause qui suit **ThenError** est alors exécutée. Elle consiste à extraire de la liste stockée dans **LSTERR** le message correspondant à la situation, et à provoquer une erreur-utilisateur sur la base de ce message.

Avant ce message d'erreur, la pile est vidée et on place au niveau 1 la ligne dans laquelle le problème a été rencontré.

Par mesure de précaution, si une erreur se produit avec un numéro supérieur à 9 (ce qui ne pourrait provenir que d'un événement imprévu), cette erreur est exécutée normalement.

Deux erreurs peuvent cependant échapper au contrôle par la structure **IfError...ThenError**:

- Une erreur lors de l'évaluation d'une expression utilisée dans un argument (voir le sous-programme **GETARG**).
- Une erreur lors de l'exécution du programme **→OBJ** (c'est le cas si un label utilisé lors d'un saut n'est défini nulle part).

Voici le contenu de la liste **LSTERR**: Notez les sauts de ligne dans la quatrième chaîne ("***Argument...ou absent***") et la 6ème chaîne ("***Champ...absent***") de cette liste:

```
{  "Instruction inconnue"
      "Erreur de syntaxe"
      "Nom déjà utilisé"
      "Argument incorrect
         ou absent"
      "Saut trop long"
      "Champ incorrect ou
         absent"
      "Label incorrect"
      "Link incorrect"
      "RTN/GOYES attendu"
}
```

◆ LE SOUS-PROGRAMME NETLINE

Son rôle est de nettoyer la ligne qui doit maintenant être assemblée. Cette ligne est vidée de ses espaces éventuels de tête.

Elle est ensuite coupée à hauteur du premier espace éventuel qui suit le premier mot de la ligne.

La deuxième partie de la ligne subit alors le même traitement (destruction des espaces de tête, puis coupe au niveau du premier espace suivant le premier mot). On ne garde cependant que le premier mot de cette deuxième partie de ligne.

A la sortie, les niveaux 1 et 2 de la pile sont occupés de la manière suivante:

2: ***Champ1***
1: ***Champ2***

où *Champ2* et *Champ1* sont deux chaînes représentant respectivement le premier et le second mot de la ligne initiale.

Voici quelques exemples qui permettront de mieux comprendre **NETLINE**:

```
1:"     "       ==▸    2:""  1:""
1:" RTN    "   ==▸    2:"RTN"  1:""
1:"A=A+1   X"    ==▸    2:"A=A+1"  1:"X"
1:"GOSUB   SAVREG"     ==▸     2:"GOSUB"    1:"SAVREG"
1:"GOSUB   SAVREG    sauvegarde D0,D1,B,D"
                       ==▸     2:"GOSUB"    1:"SAVREG"
```

Voici donc le listing du sous-programme **NETLINE**:

```
begin
  ≈  begin
        while   dup   st=>ch   $spc   same?
        repeat   del1(st)   endwhile
     end
  ≈  begin
        dup   space   spos(2st)   dup;=0(s)?;pifte   drop;""
        begin
           dup2   sub(st,1,s-1)   3rolld   sub(st,s+1,∞)
        end
     end
  rot   3pick   eval   over   eval   4roll   eval   dup
  st=>ch   { $/ $' }   InCp(_,cp)?   ?:dq/sk   drop(3)
  drop;rot   eval   drop
end
```

Dans le programme ci-dessus, on remarque deux structures Rpl, qui sont placées sur la pile sans être exécutées (grâce à l'instruction ≈). La première de ces deux structures est la routine qui élimine les espaces de tête. La seconde coupe la chaîne au premier espace rencontré.

N.B: Si le deuxième mot de la ligne commence par le caractère / (marquant le début d'une chaîne, voir les instructions **ASC** ou **LC** ou **LA**) ou par le caractère ' (début d'une expression), alors ce deuxième mot n'est pas coupé à hauteur de son (éventuel premier espace).

Par exemple:

```
1: "ASC   abcdef  ghij"   ==▸   2:"ASC"   1:"abcdef"
1: "ASC   /abcdef  ghij"  ==▸   2:"ASC"   1:"/abcdef  ghij"
1: "LC(5)   'table + 8"   ==▸   2:"LC(5)  1:"'table + 8"
```

◆ LE SOUS-PROGRAMME ASSLINE

ASSLINE est chargé essentiellement de diriger le processus d'assemblage de la ligne en cours d'examen, suivant le résultat d'un examen rapide du premier champ de cette ligne.

Rappelons qu'à l'entrée dans ce programme, la pile a l'aspect suivant (les quatre niveaux sont occupés par des chaînes de caractères).

4: Chaîne contenant les quartets déjà assemblés
3: Ce qui reste de la chaîne initiale
2: Le premier champ de la ligne en cours
1: Le deuxième champ de la ligne en cours

Voici quelles sont les opérations réalisées successivement par **ASSLINE.**

- Permuter les niveaux 1 et 2. Le premier champ de la ligne étudiée passe donc au niveau 1. On procède ensuite à une série de tests successifs.
- On calcule le code Ascii du 1er caractère **C** du champ 1 (=► un entier système **S**).

Si **S = <2Ah>**, c'est-à-dire si **C = "*"**: la ligne est donc un commentaire, et ne doit pas être prise en compte. La seule action **d'ASSLINE** est donc de détruire les chaînes des niveaux 1 et 2 (les deux champs de la ligne étudiée).

- On entame ensuite une série de tests qui conduisent, en cas de succès, à l'appel d'un sous-programme particulier, chargé de produire le code correspondant à l'instruction formée par les champs des niveaux 1 et 2. Ces deux champs doivent être remplacés par la chaîne des codes de cette instruction.

Si le contenu de la variable locale **'r** est égal à ***True***, c'est que l'instruction de test qui précède était un test. On appelle donc le sous-programme **IFTEST**.

Si **S = <3Ah>**, c'est-à-dire si **C = ":"**. La ligne contient une déclaration de label, et on appelle le programme **ETIQ**.

Si **S = <5Bh>**, c'est-à-dire si **C = "["**, la ligne contient une directive d'inclusion de fichier externe. On appelle le sous-programme **LINK**.

Si **S = <3Dh>**, c'est-à-dire si **C = "="**, la ligne contient une déclaration de variable. On appelle donc le sous-programme **EQUATE**.

Si **S = <47h>**, c'est-à-dire si **C = "G"**, on considère qu'il s'agit d'une instruction de saut et on appelle le sous-programme **SAUT**.

Si **S = <4Ch>**, c'est-à-dire si **C = "L"**, on considère qu'il s'agit d'une des deux instructions **LC** ou **LA** (charger une constante dans l'un des deux registres **A** ou **C**), et on appelle le sous-programme **LOADAC**.

Si **S = <3Fh>**, c'est-à-dire si **C = "?"**, on considère que l'instruction étudiée est une instruction de test, et on place la valeur booléenne **True** dans la variable locale **'r**, en prévision de la prochaine instruction à assembler. Aucun sous-programme n'est cependant appelé pour traiter cette instruction de test.

- A ce stade, le code **S** n'est plus utile et a été éliminé. On se penche maintenant sur la longueur du premier champ de l'instruction (la chaîne qui est au niveau 1).

Si cette longueur est comprise entre 2 et 7 (inclus), alors on appelle le sous-programme qui correspond à cette longueur (de **ASS3** à **ASS7**).

Si la longueur du premier champ est inférieure à 2 ou supérieure à 7, alors il y a une erreur de code 1 (caractérisée par le message "***Instruction Inconnue***").

- Comme on le constate, un sous-programme est appelé pour traiter chacune des situations possibles. Si ce traitement n'est pas interrompu par une erreur, ce sous-programme laisse au niveau 1 la chaîne contenant les codes de l'instruction (en remplacement des deux champs initiaux).

Le contenu de la pile est alors le suivant:

3: Chaîne contenant les quartets déjà assemblés
2: Ce qui reste de la chaîne initiale
1: Chaîne des codes de l'instruction.

La chaîne du niveau 1 est alors ajoutée à la chaîne du niveau 3.

Le terrain est alors dégagé pour retourner au programme principal **ASM48**, qui devra maintenant traiter la ligne suivante de la chaîne-source initiale.

Toutes ces explications devraient rendre le listing de **ASSLINE** assez clair:

```
begin
  swap;dup   <1h>   snum(st,s)
  <2Ah>   over;=(2s)?:drdq/sk   drop2
  begin
     'r   ?:drdq/sk   IFTEST
     <3Ah>   over;=(2s)?:drdq/sk   ETIQ
     <5Bh>   over;=(2s)?:drdq/sk   LINK
     <3Dh>   over;=(2s)?:drdq/sk   EQUATE
     <47h>   over;=(2s)?:drdq/sk   SAUT
     <4Ch>   over;=(2s)?:drdq/sk   LOADAC
     <3Fh>   =(2s)?   pift
     begin   true;≈   'r   HardSto(_,nm)   end
     { ASS3  ASS4  ASS5  ASS6  ASS7 }
     over;ssize(st)   2-(s)   <1h>   max(2s)   get(cp,s)?
     pifte   eval   doerr001
  end
  swap(2,3)   +(2st)!;swap
end
```

◆ LE SOUS-PROGRAMME IFTEST

Rappelons qu'il est appelé lorsque l'instruction précédemment assemblée était un test (à cette occasion, le programme **ASSLINE** avait placé la valeur ***True*** dans la variable locale **'r**).

Le sous-programme **IFTEST** commence d'ailleurs par replacer la valeur ***False*** dans la variable locale **'r**.

On s'attend bien sûr à ce que le premier champ de l'instruction (qui est au niveau 1) soit "**RNTYES**" ou **GOYES**".

- Si ce premier champ est "**RNTYES**", on l'élimine (ainsi que le deuxième champ, tout à fait inutile), et on place la chaîne "**00**" au niveau 1 (**00** est le code de **RNTYES**).
- Si le premier champ n'est ni "**RTNYES**" ni "**GOYES**", il y a une erreur de code 9 ("***RTN/GOYES attendu***").
- Si le premier champ est "**GOYES**", on place l'entier-système <2h> au niveau 3 (car la longueur du saut sera codée sur deux quartets), et la longueur de la chaîne déjà assemblée (pour représenter l'adresse où placer les deux quartets du saut) aux niveaux 2 et 1.

  On appelle alors le programme **OFFSET**, dont le rôle est d'examiner le deuxième champ de l'instruction, pour déterminer la longueur du saut (ou pour traiter le cas où le saut demandé s'effectue vers un label non encore déclaré).

Voici donc le listing du sous-programme **IFTEST**:

```
begin
  false   ≈   'r   HardSto(_,nm)
  "RTNYES"   over;=(2st)?:drdq/sk
  begin   drop   "00"   end
  "GOYES"   same?   ?:skip   doerr009
  <2h>   4pick   ssize(st)   dup   OFFSET
end
```

◆ LE SOUS-PROGRAMME GETARG

C'est sans doute le sous-programme le plus souvent appelé. Son rôle est d'examiner la syntaxe du deuxième champ d'une instruction. Ce deuxième champ (c'est pour l'instant une chaîne de caractères) est, au départ, placé au niveau 1.

Pour vérifier cette syntaxe, on utilise l'instruction **SyntaxOK(st)?** qui est en fait la routine chargée de vérifier le contenu de la ligne de commande de la HP48. Cette routine attend une chaîne au niveau 1, et elle renvoie:

- Au niveau 1: **True** si la syntaxe est correcte, **False** sinon.
- Au niveau 2, si la syntaxe est correcte: l'objet résultat de l'examen de cette chaîne de caractères. Ce peut être un nom, un réel, un programme, etc...

Si l'objet obtenu est un nom: il s'agit d'un nom local uniquement si ce nom est celui d'une variable locale en cours d'existence. Cette particularité nous permet de savoir, notamment dans le cas d'un saut vers un label, si ce label a déjà été déclaré ou non.

Si l'objet obtenu est une expression, cette expression est évaluée par le sous-programme **GETARG**. Signalons qu'alors peuvent se produire des erreurs d'exécution, et que ces erreurs ne sont pas contrôlées par la structure ***IfError...ThenError*** du programme principal **ASM48**. Dans le cas où une telle erreur se produit, le contenu de la pile peut être tout à fait quelconque...

Dans le cas où l'exécution de **GETARG** dévoile une erreur de syntaxe dans la chaîne traitée, il y bien sûr exécution d'une erreur de code 2 (message "***Erreur de syntaxe***").

Voici le listing, très court, de **GETARG**:

```
begin
  SyntaxOK(st)?   ?:skip   doerr002
  dup;ex?   pift   eval(cp)
end
```

◆ LE SOUS-PROGRAMME OFFSET

Son rôle est de calculer les sauts relatifs (correspondant aux instructions **GOYES, GOC, GONC, GOTO, GOLONG, GOSUB,** et **GOSUBL**). Il est donc appelé par les sous-programmes **IFYES** (évoqué ci-dessus) et **SAUT** (voir plus loin).

La pile, lors de l'appel d'**OFFSET**, doit avoir l'aspect suivant:
4: Deuxième champ de l'instruction
3: **<n>** = nombre de quartets pour coder le saut (2,3,4, ou 5).
2: **<adr>** = adresse où stocker les n quartets du saut.
1: **<adr'>** = adresse à partir de laquelle on doit calculer le saut.

<n>, **<adr>** et **<adr'>** doivent être des entiers-système. Les adresses contenues dans **<adr>** et **<adr'>** sont relatives au début de la chaîne contenant ce qui est déjà assemblé (le premier caractère de cette chaîne ayant pour valeur 0, par convention).

La distinction entre **<adr>** et **<adr'>** est importante, notamment pour la différence qui existe entre les sauts **GOSUB, GOSUBL**, et les autres:

Pour les premiers, on aura **<adr'> = <adr> + <n>** (en effet le déplacement, lors d'un saut de type **GOSUB**, est calculé entre l'adresse où on doit se brancher et l'adresse qui suit immédiatement les **n** quartets du saut.

Pour les autres, on aura **<adr> = <adr'>**, car le déplacement est calculé entre l'adresse où on doit se brancher et l'adresse où débutent les **n** quartets du saut.

La création de la pile telle qu'il vient d'être dit est du ressort des programmes qui appellent **OFFSET**, à savoir **IFTEST, SAUT**, et **ASS4** (ce dernier pour les instructions **REL2** à **REL5**).

Le fonctionnement du programme **OFFSET**, n'est pas des plus simples, notamment parce qu'il y a lieu de prévoir quoi faire si une instruction de branchement utilise un label qui n'est pas encore déclaré (branchement donc vers une adresse supérieure à l'adresse en cours). Essayons donc de résumer ce qui se passe dans **OFFSET**:

- On place au niveau 1 le champ n°2 de l'instruction en cours (instruction **4roll**) puis on appelle le sous-programme **GETARG**, pour tester la syntaxe de ce champ, et pour en extraire l'objet qu'il contient. Il est évident que l'on attend ici le nom d'un label. Le problème est que ce label n'est peut-être pas encore connu.
- Si cet objet est un nom global, c'est précisément qu'il s'agit d'un nom de label non encore déclaré. Dans ce cas, on transforme ce nom en nom local. On crée la variable locale portant ce nom en y plaçant une liste vide (voir ci-après), en laissant une copie du nom au niveau 1.
- On teste si l'objet au niveau 1 est un nom local. Si ce n'en est pas un, alors il y a une erreur de code 4 (message "***Argument incorrect ou absent***").
- On rappelle maintenant sur la pile le contenu de ce nom local. On teste ensuite le type de ce contenu (instruction **ChkArg!**).

- Si le contenu de ce nom local est une liste:
  Celà signifie qu'on a affaire à un label qui n'a pas encore été déclaré (voir ci-dessus). Dans ce cas, on rajoute, à cette liste, les trois entiers-système **<n>,<adr>,<adr'>** dont la signification a été donnée ci-dessus.
  On stocke cette nouvelle liste dans la variable locale. Ainsi, si un label est évoqué une ou plusieurs fois dans des instructions de saut, alors qu'il n'est pas encore déclaré, la variable locale qui lui correspond contient une liste formée de groupes successifs de trois entiers-système (chaque groupe **<n>,<adr>,<adr'>** correspondant à un appel particulier de ce label, pour un saut relatif de **<n>** quartets, à stocker à l'adresse **<adr>** et à calculer à partir de l'adresse **<adr'>**).
  Après avoir ainsi actualisé le contenu de la variable locale, on place une chaîne de **<n>** espaces sur la pile, au niveau 1 (Instructions **makeb(s)**, puis **MapStr(b)**). Cette chaîne sera donc ajoutée à la chaîne contenant le code déjà assemblé, de manière à réserver la place nécessaire pour stocker les **<n>** quartets du déplacement (qui ne pourront être calculés que lorsque le label aura été enfin déclaré).
- Si le contenu du nom local est un entier-système:
  Celà signifie que le label évoqué dans l'instruction a déjà été déclaré. Il s'agit donc d'un branchement en amont. Notons **<L>** l'entier-système trouvé ici. **<L>** est l'adresse du label, relativement à la chaîne contenant les codes déjà assemblés (le premier caractère de cette chaîne ayant l'adresse <0>).
  On élimine (instructions **drop(2)** puis **drop(3)**) le nom local (résiduel au niveau 2) et l'entier-système **<adr>** (voir notation ci-dessus) devenu inutile ici car les **<n>** quartets du déplacement seront ajoutés, on le sait, à la suite de la chaîne contenant le code déjà assemblé.
  L'allure de la pile est alors la suivante:
  5: Chaîne contenant les codes déjà assemblés.
  4: Ce qui reste de la chaîne-source initiale.
  3: **<n>** = nombre de quartets pour coder le déplacement.
  2: **<adr'>** = adresse relative pour calculer le déplacement.
  1: **<L>** = adresse du label évoqué dans l'instruction de saut.
  On calcule la différence **<D> = <L>-<adr'>**, qui est précisément la valeur du déplacement. Il faut maintenant vérifier que ce déplacement **<D>** est codable sur les **<n>** quartets prévus (on vérifie donc que le saut n'est pas trop long).
  Pour celà, on place sur la pile l'entier-système <80h>, <800h>, <8000h>, ou <80000h>, suivant que la valeur de **<n>** est égale à <2h>, <3h>, <4h> ou <5h>. Notons **<M>** l'entier-système ainsi placé au niveau 1.
  Le saut envisagé est trop long si **<M> + <D>** est supérieur à **2*<M>-1** (on laisse au lecteur le soin de comprendre pourquoi il en est ainsi. Rappelons seulement que les comparaisons entre entiers-système se font en mode "non signé"). Il y a dans ce cas une erreur de code 5 (message "***Saut trop long***").
  Si le saut n'est pas trop long, on transforme **<D>** en une chaîne de caractères (appel du programme **→STRI**). On extrait alors de cette chaîne les **<n>** caractères de la position <6h> à la position <6h> + <n>-1 (incluses). La chaîne restante contient donc le codage du déplacement.

Par exemple, et si **<n>** est égal à <3h>: un déplacement égal à **<D>** = <4A9h> est transformé par **→STRI** en **"119209A400"**, et on ne garde que la sous-chaîne **"9A4"**.

- Enfin, si le contenu du nom local (le nom utilisé pour label) n'est pas une liste ni un entier-système, alors on génère une erreur de code 4 (message "***Argument incorrect ou absent***").

Après toutes ces explications, le listing de **OFFSET** vous paraîtra sans doute lisible:

```
begin
  4roll   GETARG
  dup;gn?   pift
  begin
     nm=>ln   dup   {}   swap;<1h>   Ldef(_,s)
  end
  dup;ln?   ?:skip   doerr004
  dup;rcl(nm)?   drop
  ChkArg!
  <5h>
  begin
     5pick   5roll   5roll   3->list   +(2cp)
     swap   HardSto(_,nm)   makeb(s)   MapStr(b)
  end
  <1Fh>
  begin
     drop(2)  drop(3)  swap;-(2s)
     { <80h> <800h> <8000h> <80000h> }
     3pick   1-(s)   get(cp,s);dup   3pick;+(2s)
     swap   2*(s)   1-(s)   >(2s)?   pift
     begin   <5h>   doerr(s)   end
     ->STRI   <6h>   rot   over;+(2s)   sub(st,s2,s1-1)
  end
  <0h>   doerr004
end
```

◆ LE SOUS-PROGRAMME ETIQ:

Il est appelé lorsque le premier caractère de la ligne est ":". Cette ligne est alors interprétée comme la déclaration d'un label (d'une étiquette), dont le nom suit immédiatement le caractère ":". Le deuxième champ de l'instruction est alors inutile.

Voici comment se déroule la prise en compte de ce nom dans le programme **ETIQ**:

- On commence par éliminer (***drop(2)***) le deuxième champ de l'instruction.
- On détruit également le caractère ":" qui débute le premier champ (instruction ***del1(st)***). A ce stade, le niveau 1 contient une chaîne dont le contenu est le nom du label.

- On examine ensuite le contenu de cette chaîne, par un appel au sous-programme **GETARG**. Le résultat est en principe un nom. L'instruction ***ChkArg!*** permet d'orienter le programme **ETIQ** suivant la nature de l'objet obtenu.
- S'il s'agit d'un nom global, c'est que l'on se trouve en présence d'un label déclaré pour la première fois, et qui n'a pas encore été évoqué dans une instruction de branchement précédente.

  Dans ce cas, on transforme ce nom en nom local. On crée la variable locale correspondante en y stockant un entier-système égal à la longueur de la chaîne contenant le code déjà assemblé (cet entier-système représente donc l'adresse relative du label).
- S'il s'agit d'un nom local, on rappelle sur la pile le contenu de la variable locale correspondante, en conservant une copie du nom local sur la pile.
- Si ce contenu est une liste, alors on est en présence d'un label qui a déjà été appelé dans des instructions de saut précédentes (voir le sous-programme **OFFSET**).

  On commence par stocker, dans la variable locale, l'adresse relative qui correspond à ce label (voir ci-dessus).

  On entreprend ensuite d'examiner le contenu de la liste, par groupe de trois entiers-système consécutifs (c'est la boucle qui commence à "***dup;start(1,szcp)***" et se termine à "***<3h> step(s)***".

  Chacun de ces groupes de 3 entiers-système a la forme **<n>,<adr>,<adr'>**, où **<n>** est le nombre de quartets sur lequel on doit coder le déplacement, **<adr>** est l'adresse où doit être écrit ce déplacement, et **<adr'>** est l'adresse à partir de laquelle il doit être calculé (voir les explications dans le programme **OFFSET**).

  On calcule le déplacement **<D> = <L>-<adr'>**, où **<L>** est l'adresse du label, et on vérifie si ce déplacement peut être codé sur **<n>** quartets, en mode complément à 2. Si ce n'est pas le cas, une erreur de code <5h> (message "***Saut trop long***") est générée.

  Si le saut n'est pas trop long, le déplacement est converti en une chaîne de **<n>** quartets (utilisation du programme **→STRI**), et cette chaîne est écrite là où elle doit l'être, dans la chaîne qui contient le code déjà assemblé (on se souvient que le programme **OFFSET** avait écrit **<n>** espaces à cet endroit).
- Si le contenu du nom local n'est pas une liste, alors il y a une erreur de code 3 ("***Nom déjà utilisé***"). Il ne peut en effet s'agir que d'un nom de label qui a déjà été déclaré, ou d'une variable créée grâce au sous-programme **EQUATE**.
- Enfin, si le résultat de l'appel initial à **GETARG** n'est ni un nom global ni un nom local, alors il y a une erreur de code 7 (message "***Label incorrect***").
- On remarquera que, dans le cas où il n'y a pas d'erreur, le programme **ETIQ** dépose pour finir, au niveau 1 de la pile, une chaîne vide (avant que l'on ne revienne au programme appelant **ASSLINE**). C'est que cette instruction ne produit aucun code particulier.

Voici enfin le listing du programme **ETIQ**:

```
begin
  drop(2)   del1(st)   GETARG
  ChkArg!
  <6h>
  begin   nm=>ln   3pick   ssize(st)
     swap;<1h>   Ldef(_,s)   Null""
  end
  <7h>
  begin
     dup;rcl(nm)?   drop   dup;li?   ?:dq/sk
     begin
        4pick   ssize(st)   dup
        4roll   HardSto(_,nm)   swap
        dup;start(1,szcp)
           dup;index   dup;2+(s)   sub(cp,2s)   break;drop
           5pick   swap;-(2s)
           { <80h>  <800h>  <8000h>  <80000h> }
           4pick   1-(s)   get(cp,s)
           dup2;+(2s)    swap   2*(s)   1-(s)
           >(2s)?   pift
           begin   <5h>   doerr(s)   end
           ->STRI   <6h>   4roll   over;+(2s)
           sub(st,s2,s1-1)   6roll   rot   s=>r
           1+(r)   rot   repl(st,r,st)   4rolld
        <3h>   step(s)
        drop2   Null""
     end
     doerr003
  end
  <0h>   begin   <7h>   doerr(s)   end
end
```

◆ LE SOUS-PROGRAMME LINK

Il est appelé quand le premier caractère du premier champ de l'instruction est "[". Cette instruction est interprétée comme une directive d'inclusion de fichier. Le nom du fichier à inclure doit suivre immédiatement le caractère "[" dans le premier champ. Le deuxième champ de l'instruction est donc inutile.

- On commence d'ailleurs par éliminer ce deuxième champ.
- On ôte le premier caractère ("<") du premier champ, puis on appelle le sous-programme **GETARG** pour savoir de quoi est fait le contenu du premier champ. On attend en principe un nom global.
- Si ce n'est pas un nom global, alors on génère une erreur de code 8 ("***Link incorrect***"). On provoque la même erreur si le nom global est inconnu.

- Si le nom global est connu, on examine son contenu. L'instruction **ChkArg!** permet d'orienter la suite du programme **LINK**.

- Si le contenu du nom est une chaîne, on interprète cette chaîne comme une succession d'instructions en assembleur. On lui ajoute un retour chariot final, puis on concatène cette chaîne en tête de ce qui reste de la chaîne source-initiale.
  La suite du programme principal **ASM48** verra donc l'assemblage des instructions ainsi insérées.
  Comme dans ce cas, aucun code n'est produit par l'instruction d'inclusion (le code ne sera produit que lorsque les lignes du fichier inclus seront examinées), une chaîne vide est laissée sur la pile, avant le retour au programme **ASSLINE**.

- Si le contenu du nom est un objet-code, on transforme cet objet en son image en mémoire sous forme de chaîne de caractères (appel du programme **→STRI**), puis on concatène la chaîne obtenue (à l'exception de ses dix premiers caractères qui représentent l'adresse-préfixe d'un objet-code et la longueur de cet objet-code) à la fin de la chaîne qui contient le code déjà assemblé.
  Cette façon de procéder permet donc d'inclure de véritables unités précompilées dans le code en cours d'assemblage.

- Si le contenu du nom n'est ni une chaîne ni un objet-code, alors une erreur de code 8 est provoquée.

Voici le listing, somme toute assez simple, du programme **LINK**:

```
begin
  drop(2)   del1(st)   GETARG   dup;gn?   ?:skip   doerr008
  rcl(nm)?   ?:skip   doerr008
  ChkArg!
  <3h>
  begin   +(st,RC)   swap;+(2st)!   Null""   end
  <7Fh>
  begin   ->STRI   <Bh>   sub(st,s,∞)   end
  <0h>   doerr008
end
```

◆ LE SOUS-PROGRAMME EQUATE

Il est appelé quand le premier caractère du premier champ de l'instruction est " = ". Cette instruction est interprétée comme une directive de définition de variable locale.

Le nom de cette variable doit suivre immédiatement le caractère " = " dans le premier champ. Le deuxième champ de l'instruction doit contenir la constante à placer dans la variable.

- On commence d'ailleurs par éliminer ce deuxième champ.
- On ôte le premier caractère (" = ") du premier champ, puis on appelle le sous-programme **GETARG** pour savoir de quoi est fait le contenu du premier champ. On attend en principe un nom global.
- S'il s'agit d'un nom local, on provoque une erreur de code 3 (message "***Nom déjà utilisé***"). Il s'agit en effet d'une variable qui a déjà été déclarée (ou évoquée) en tant que label, où d'une variable qui a déjà été déclarée dans une instruction de type **EQUATE** précédente.
- S'il ne s'agit pas d'un nom global, on provoque un erreur de code 2 ("***Erreur de syntaxe***").
- S'il s'agit d'un nom global, on le transforme en nom local. On examine ensuite le contenu du deuxième champ de l'instruction au moyen du sous-programme **GETARG**.

  Si ce contenu n'est pas un réel ou un binaire, alors on provoque une erreur de code 4 (message "***Argument incorrect ou absent***").

  Si ce contenu est un réel ou un binaire, on crée une variable locale (avec le nom local évoqué ci-dessus) en y stockant ce réel ou cet entier-binaire.
- On remarquera que, dans le cas où il n'y a pas d'erreur, le programme **EQUATE** dépose pour finir, au niveau 1 de la pile, une chaîne vide (avant que l'on ne revienne au programme appelant **ASSLINE**). C'est que cette instruction ne produit aucun code particulier.

Voici le listing du sous-programme **EQUATE**:

```
begin
  del1(st)   GETARG
  ChkArg!
  <7h>   doerr003
  <6h>
  begin
     nm=>ln   swap   GETARG
     ChkArg!
     <1h>   begin   swap;<1h>   Ldef(_,s)   Null""   end
     <Bh>   begin   swap;<1h>   Ldef(_,s)   Null""   end
     <0h>   doerr004
  end
  <0h>   doerr002
end
```

◆ LE SOUS-PROGRAMME SAUT

Il est appelé lorsque la première lettre du premier champ de l'instruction est la lettre "**G**". Il ne peut s'agir que d'une instruction de saut.

- On commence par vérifier que le premier champ de l'instruction est l'une des 8 mnémoniques permettant de coder un saut (de "**GOC**" à "**GOSBVL**"). Si ce n'est pas le cas, une erreur de code 1 est provoquée (message "***Instruction inconnue***").
- Si le premier champ est correct, on place sur la pile la chaîne de un ou deux caractères représentant la partie invariable du codage de l'instruction de saut (de "**4**" pour **GOC** à "**8F**" pour **GOSBVL**). On ajoute cette chaîne à la chaîne qui contient le code déjà assemblé.
- La position **<p>** qu'occupait le premier champ dans la liste des 8 mnémoniques possibles est un entier-système qui est encore présent sur la pile.
  Si **<p>** est supérieur à <6h>, alors il s'agit d'une instruction du type **GOVLNG** ou **GOSBVL**, en tout cas d'un saut absolu (vers une adresse fixe).
  Si **<p>** est inférieur ou égal à <6h>, il s'agit d'un saut relatif, et le déplacement correspondant doit être calculé.
- Dans le cas d'un saut absolu:
  On examine le contenu du deuxième champ de l'instruction, par un appel au sous-programme **GETARG**. Si le résultat est un nom local, on le remplace par son contenu.
  Si le résultat n'est pas un nombre binaire, il y a une erreur de code 4 ("***Argument incorrect ou absent***"). Sinon, on transforme cet entier-binaire en chaîne par **→STRI**, et on ne conserve que les caractères de position 11 à 15 de cette chaîne.
  Par exemple, le nombre binaire #679Bh est transformé en la chaîne "E4A2051000B9760000000000003" et cette chaîne est tronquée aux cinq caractères "B9760".
- Dans le cas d'un saut relatif, on doit distinguer s'il s'agit ou non d'un branchement vers un sous-programme. Pour celà on compare l'entier-système **<p>** évoqué ci-dessus avec <4h>.
  Si **<p>** est supérieur à <4h> (donc **<p>** = 5 ou 6), alors il s'agit d'un saut vers un sous-programme.
  Dans tous les cas on prépare la pile pour un appel du programme **OFFSET** (qui a déjà été décrit). Ce programme est appelé lorsque les 4 niveaux inférieurs de la pile ont le sens suivant:
  4: Deuxième champ de l'instruction
  3: **<n>** = nombre de quartets pour coder le saut (2,3,4).
  2: **<adr>** = adresse où stocker les n quartets du saut. C'est en fait la taille de la chaîne qui contient le code déjà assemblé.
  1: **<adr'>** = adresse à partir de laquelle on doit calculer le saut.
  Le lecteur vérifiera comment le programme **SAUT** prépare cette pile suivant qu'on a affaire ou non à un saut vers un sous-programme (dans ce cas, on a **<adr'>** = **<adr>** + **<n>**), sinon **<adr'>** = **<adr>**).

Voici le listing du programme **SAUT**:

```
begin
  { "GOC"  "GONC"  "GOTO"  "GOLONG"
    "GOSUB"  "GOSUBL"  "GOVLNG"  "GOSBVL" }
  spos(_,cp)   dup;=0(s)?:drdq/sk   doerr001
  { "4"  "5"  "6"  "8C"  "7"  "8E"  "8D"  "8F" }
  over   get(cp,s)   5roll   swap;+(2st)!   4rolld
  <6h>   over;<(2s)?  ?:drdq/sk
  begin
     GETARG   dup;ln?   pift   rcl(nm)
     dup;b?   ?:skip   doerr004
     ->STRI   <Bh>   <Fh>   sub(st,2s)
  end
  <4h>   over;<(2s)?   pifte
  begin
     2-(s)   4pick   ssize(st)   dup   3pick;+(2s)
  end
  begin
     <2h>   max(2s)   4pick   ssize(st)   dup
  end
  OFFSET
end
```

◆ LE SOUS-PROGRAMME HEXA

Il est appelé pour transformer un entier-système, compris entre **<0h>** et **<Fh>**, en la chaîne de 1 caractère qui lui correspond, de "**0**" à "**F**". Ce sous-programme est particulièrement simple:

```
begin
  "0123456789ABCDEF"   swap;1+(s)   dup   sub(st,2s)
end
```

◆ LE SOUS-PROGRAMME NIBBLE

Il est appelé pour tester la validité d'un argument, dans une instruction, quand cet argument doit être, au final, une valeur entière comprise entre 0 et 15, ou entre 1 et 16.

En entrée la pile doit contenir un booléen au niveau 1, et une chaîne (représentant l'argument à évaluer) au niveau 2.

Le booléen vaut **True** si l'argument est censé être compris entre 0 et 15 (c'est le cas par exemple pour l'entier **n** dans l'instruction **P= n**). Il vaut **False** si cet argument est censé être compris entre 1 et 16 (c'est le cas par exemple pour l'entier **n** dans l'instruction **A=DAT0 n**).

Le résultat du programme **NIBBLE** est la chaîne de 1 caractère représentant l'entier reconnu, de **"0"** à **"F"**, telle qu'elle doit être inscrite dans le code-objet final (dans le cas de l'instruction **A=DAT0 n**, par exemple, l'entier **n** est codé par le quartet **n-1**, alors que dans l'instruction **P= n**, l'entier **n** est codé par le quartet **n**).

Le fonctionnement de **NIBBLE** est assez simple:

- On commence par placer le booléen au niveau 2, puis à évaluer la chaîne de caractères par un appel au sous-programme **GETARG**.
  Si le résultat est un nom local, on le remplace par son contenu.
- L'argument obtenu est alors transformé en un entier système, s'il est de type réel ou de type binaire. dans les autres cas, il y a une erreur de code 4 (*"Argument incorrect ou absent"*).
- On retranche ensuite <1h> à l'entier-système dans le cas où le booléen initial est égal à **True** (dans ce cas l'entier **n** sera bien codé par le quartet **n-1**).
- Si l'entier-système obtenu est supérieur ou égal à <10h>, alors il y a une erreur de code 4. Sinon on transforme l'entier-système en une chaîne de 1 caractère par appel au sous-programme **HEXA**.

Voici le listing du sous-programme **NIBBLE**:

```
begin
  swap   GETARG
  dup;ln?   pift   rcl(nm)
  begin
     ChkArg!
     <1h>   r+=>s
     <Bh>   b=>s
     <0h>   doerr004
  end
  swap   pift   1-(s)   <10h>   over;>(2s)?
  pifte   HEXA   doerr004
end
```

◆ LE SOUS-PROGRAMME LOADAC

Il est appelé lorsque la première lettre du premier champ de l'instruction est **"L"**. Il ne peut s'agir alors que d'une instruction **LC(n)** ou **LA(n)** (chargement dans le registre **A** ou **C** de **n** quartets, à partir de la position indiquée par le registre **P**).

- On commence par vérifier que les trois premiers caractères du premier champ forment la chaîne **"LC("** ou la chaîne **"LA("**, faute de quoi il y a une erreur de code 1 (*"Instruction inconnue"*).
  Sinon on place sur la pile la partie fixe de l'instruction, à savoir **"8082"** pour **"LA("** et **"3"** pour **"LC("**.

- On exécute la même erreur de code 1 si le dernier caractère du premier champ n'est pas la parenthèse fermante "**)**" (code **<29h>**).

  Sinon on extrait la sous-chaîne comprise entre les 2 parenthèses. On interprète cette chaîne comme un entier compris entre 1 et 16, au moyen du programme **NIBBLE** (avec l'arguement **True**, voir ci-dessus).
- Le résultat est alors une chaîne de 1 caractère, de "**0**" (si l'entier valait 1, à "**F**" si cet entier valait 16). On ajoute ce caractère (en en conservant une copie) à la partie fixe "**8082**" ou "**3**", déjà placée sur la pile.
- La copie de ce caractère est ensuite convertie en un entier-système **<n>**. Il faut maintenant lire **<n>** quartets dans le deuxième argument de l'instruction. En fait, on prendra les **<n>** quartets de poids faible de cet argument.
- Si le premier caractère de la chaîne représentant le deuxième argument est "/", on se contente de le supprimer, sans toucher au reste de cette chaîne.
- Sinon, on évalue cette chaîne par le sous-programme **GETARG**. Si le résultat est un nom local, on le remplace par son contenu. Si l'argument est un réel, on le convertit en un entier binaire. Si l'argument final n'est pas un entier binaire, il y a une erreur de code 4 ("***Argument incorrect ou absent***").
- Enfin, on transforme l'argument par le sous-programme →**STRI**). On ne garde de cette chaîne que les caractères de position **<Bh>** à **<Bh>** + **<n>-1**, où **<n>** est l'entier-système évoqué ci-dessus.

On termine en ajoutant cette chaîne tronquée à ce qui a déjà été obtenu pour cette instruction (la partie fixe "**8082**" ou "**3**" suivie d'un caractère de "**0**" à "**F**"). On a ainsi obtenu le codage de notre instruction **LC** ou **LA**.

Exemple: dans l'instruction "LC(16) /abcdefghij", la chaîne "/abcdefghij" est transformée en "abcdefghij" puis en "C2A2091000162636465666768696A6", par →**STRI**. Elle est alors tronquée en "1626364656667686". Au final cette instruction est codée "3F1626364656667686".

Voici enfin le listing du programme **LOADAC**:

```
begin
  { "LA("  "LC(" }   over   <4h>   sub(st,1,s-1)
  spos(cp,_)   dup;=0(s)?:dq/sk   doerr001
  <1h>   =(2s)?;pifte
  "8082"   "3"   over   dup;ssize(st)   snum(st,s)
  <29h>   =(2s)?:skip   doerr001
  swap   <4h>   over;ssize(st)   sub(st,s2,s1-1)
  true   NIBBLE   swap;over
  +(2st)!   swap   <1h>   snum(st,s)   dup   <39h>
  >(2s)?;pifte   <37h>   <30h>   -(2s)   rot;dup
  st=>ch   $/   same?   pifte
  begin   del1(st)   <8h>   makeb(s)   MapStr(b)   +(2st)!   end
  begin
     GETARG   dup;ln?   pift   rcl(nm)   dup;r?   pift
     r=>b   dup;b?   ?:skip   doerr004
  end
  ->STRI   <Bh>   rot   over;+(2s)   sub(st,2s)   +(2st)!
end
```

◆ LE SOUS-PROGRAMME FIELD

Comme son nom l'indique, ce sous-programme est appelé à chaque fois qu'il faut contrôler que le deuxième argument d'une instruction est une spécification de champ.

A l'entrée dans le programme **FIELD**, la pile doit avoir l'aspect suivant:

4: Chaîne correspondant à un registre quelconque
3: Chaîne correspondant au registre **B**, sinon <0h>
2: Chaîne correspondant au registre **A**
1: Deuxième argument de l'instruction.

Les chaînes des niveaux 2, 3, et 4 correspondent à la façon dont doit être codée une instruction dont le deuxième argument est un champ, selon que ce champ est **A** (chaîne du niveau 2), **B** (chaîne du niveau 3) ou un autre champ (chaîne du niveau 4).

Dans le cas où l'instruction considérée ne connait pas de code particulier pour le registre **B**, alors le niveau 3 sera occupé, conventionnellement, par <0h>.

Exemple: pour l'instruction "**C=D** *field*", l'appel au sous-programme **FIELD** se fera avec la pile suivante:

4: "**AbB**" (codage pour les registres diiférents de **A**)
3: **<0h>** (pas de code spécifique pour le registre **B**)
2: "**DB**" (codage de l'instruction **C=D A**)
1: *field* (remplacer par le deuxième argument)

Exemple: pour l'instruction "**DAT0=C** *field*", l'appel au sous-programme **FIELD** se fera avec la pile suivante:

4: "**154a**" (codage pour les registres différents de **A**,**B**)
3: "**14C**" (codage de l'instruction **DAT0=C B**)
2: "**144**" (codage de l'instruction **DAT0=C A**)
1: *field* (remplacer par le deuxième argument)

On voit sur les deux exemples précédents comment on code, en utilisant une convention bien établie, l'instruction dans le cas où le registre est quelconque (le quartet **a** ou **b** jouant le rôle de sélecteur de champ). La "préparation" d'une telle pile sera l'oeuvre des sous-programmes **ASS3**, **ASS4**, **ASS5**, **ASS56**, **ASS6** et **ASS7**, qui appellent tous le sous-programme **FIELD**.

Voici donc comment fonctionne le programme **FIELD**:

- On commence par vérifier que le deuxième argument de l'expression figure bien dans la liste des différents champs. Si ce n'est pas le cas, il y a une erreur de code 6 (message "***Champ incorrect ou absent***").
- Soit **<n>** la position du deuxième argument dans la liste des champs.

Si **<n>=1** (c'est-à-dire s'il s'agit du champ **A**), on ne laisse sur la pile que la chaîne qui correspond à ce champ.

Sinon on remplace **<n>** par **<m>=<n>-2**, qui est compris entre <0h> (pour le champ **P**) et <7h> (pour le champ **W**).

- Si **<m>**=6, c'est-à-dire s'il s'agit du champ **B**, et s'il y a (pour l'instruction en cours de traduction) un code spécifique au champ **B** (on fait le test sur le niveau indiqué ci-dessus: s'il contient une chaîne et non pas <0h>, c'est qu'il y a un code spécifique à **B**), alors on ne laisse sur la pile que la chaîne correspondant au champ **B**.
- Dans les autres cas, on a affaire à un champ quelconque. On examine si le caractère "**a**" ou le caractère "**b**" est présent dans la chaîne représentant le code de l'instruction (l'un des deux doit être présent).

  On place alors, dans cette chaîne, et à la position à laquelle on a lu "**a**" ou "**b**", le caractère qui correspond au registre qui a été spécifié dans l'instruction. On laisse au lecteur le soin de comprendre l'opération qui permet de créer ce caractère (on utilise pour celà le sous-programme **HEXA**, et la valeur de l'entier-système **<m>** évoqué ci-dessus).

  Rappelons simplement les tables qui permettent de traduire un champ en un quartet (nommé **a** ou **b**):

  a=(0,1,2,3,4,5,6,7) ◄==► *field*=(P,WP,XS,X,S,M,B,W)
  b=(8,9,A,B,C,D,E,F) ◄==► *field*=(P,WP,XS,X,S,M,B,W)

Voici le listing du programme **FIELD**:

```
begin
  { "A" "P" "WP" "XS" "X" "S" "M" "B" "W" }
  spos(_,cp)   dup;=0(s)?:dq/sk   doerr006
  <1h>   over;=(2s)?:drdq/sk   drop(2-3)
  drop(2)   2-(s)
  <6h>   over;=(2s)?   3pick   st?   and(2?)
  ?:drdq/sk   drop(2)
  3rolld;drop   dup   "a"   pos(2st)
  dup;=0(r)?   pifte
  begin   drop;dup   "b"   pos(2st)   <8h>   end
  <0h>
  4roll   +(2s)   HEXA   repl(st,r,st)
end
```

◆ LE SOUS-PROGRAMME ASS3

On entre maintenant dans l'étude des sous-programmes chargés de contrôler et de traduire les instructions dont la première partie a une longueur de 3, 4, 5, 6 ou 7 caractères. Commençons par le sous-programme **ASS3**.

La liste des mnémoniques à s caractères est stockée dans la variable **ST3**. Vous devez donc créer cette variable, et y stocker la chaîne suivante (**mais sans les espaces ni les sauts de ligne, qui sont là pour favoriser votre lecture**!)

N.B: la commande **BYTES**, appliquée à cette chaîne, donne les résultats: Taille 107 octets, CheckSum #4B5Bh.

```
"RSI RTN RTI P=0 ?P# ?P= P=C C=P HEX ASC NUL EXE
 INC inc A=0 B=0 C=0 D=0 A=B B=C C=A D=C B=A C=B
 A=C C=D ASL BSL CSL DSL ASR BSR CSR DSR"
```

Le lecteur observateur aura remarqué la présence de la sous-chaîne **P=0**, ce qui ne signifie pas que la seule instruction du type **P= n** permise ici soit **P=0**! L'instruction **P=** est en fait la seule dont la première partie soit longue de deux caractères seulement. L'important, dans la chaîne **ST3**, est que la sous-chaîne **P=** y apparaisse à une position multiple de 3 (en considérant le premier caractère de **ST3** comme ayant la position 0).

Voici maintenant quel est le fonctionnement du programme **ASS3**. Rappelons que les niveaux 1 et 2 de la pile contiennent respectivement le champ 1 et le champ 2 de l'instruction en cours de traduction.

- On commence par évaluer la position **<n>** du champ 1 dans la chaîne **ST3**. Si cette position est nulle, le champ 1 n'est pas reconnu et il y a une erreur de code 1 ("***Instruction inconnue***").
- On considère **<m> = <n>-1**. On vérifie que **<m>** est un multiple de 3, sans quoi il y a une erreur de code 1 (Le champ 1 correspondant en fait à trois caractères consécutifs de **ST3**, mais pas à une position correcte).
- Si **<m>** est bien un multiple de 3, on se retrouve avec, au niveau 1 de la pile, la quantité **<p> = <m>div<3>**, qui mesure le numéro de l'instruction de 3 caractères qui a été reconnue dans la chaîne **ST3** (la première instruction, **RSI**, ayant la position 0). Au niveau 2 de la pile se trouve toujours le champ 2 de l'instruction.
- On commence alors une série de tests sur l'entier **<p>**, de manière à cerner à quelle famille d'instructions il appartient. Les instructions figurant dans **ST3** sont en effet rangées par groupes successifs selon leurs affinités (ce qui permet un traitement plus efficace).
- Le premier test (**<3h> over;>(2s)? ?:dq/sk**) permet de traiter les instructions **RSI, RTN**, et **RTI**.
- Le deuxième test (**<8h> over;>(2s)? ?:dq/sk**) permet de traiter les instructions **P=, ?P#, ?P=, P=C**, et **C=P**. On remarque que pour ces instructions, le programme **ASS3** détermine la partie fixe du code (par exemple "**80D**" pour **P=C**), puis évalue le contenu du second champ de l'instruction, qui doit être un entier compris entre 0 et 16 (l'outil est ici le sous-programme **NIBBLE**, déjà commenté). Le caractère résultant de cette évaluation, de "**0**" à "**F**" est alors ajouté à la partie fixe de l'instruction.
- Le troisième test (**<Eh> over;>(2s)? ?:dq/sk**) permet de traiter les instructions **HEX, ASC, NUL, EXE, INC, inc** (qui sont des instructions spécifiques à cet assembleur). Le traitement de ces instructions est d'ailleurs assuré par un appel à un sous-programme particulier, le programme **DIV3**.

- Les instructions suivantes sont enfin départagées par un dernier test (**<1Ah> over;>(2s)? pifte**), qui permet de traiter d'une part les instructions **A=0** à **C=D** (position **<p>** inférieure à **<1Ah>**), et les instructions **ASL** à **DSR** (position **<p>** supérieure ou égal à **<1Ah>**).

  Ces deux derniers groupes d'instructions se départagent par le code qu'elles nécessitent pour le champ **A** ou pour un autre champ. Elles ont en commun de nécessiter un deuxième argument, qui doit être un champ de registre, et qui est analysé par le sous-programme **FIELD**.

Voici donc le listing du programme **ASS3**:

```
begin
  ST3   swap   spos(2st)
  dup;=0(s)?:dq/sk   doerr001
  1-(s)   <3h>   mdv(2s)   swap
  <>0(s)?   pifte   doerr001
  <3h>   over;>(2s)?   ?:dq/sk
  begin
     drop(2)   <0h>   over;=(2s)?:drdq/sk   "80810"
     =1(s)?:dq/sk   "01"   "0F"
  end
  <8h>   over;>(2s)?   ?:dq/sk
  begin
     { "2" "88" "89" "80D" "80C" }   swap   2-(s)
     get(cp,s)   swap;false   NIBBLE   +(2st)!
  end
  <Eh>   over;>(2s)?   ?:dq/sk   DIV3
  <1Ah>   over;>(2s)?   pifte
  begin   "D"   "Ab"   <Eh>   end
  begin   "F"   "Bb"   <1Ah>   end
  4roll   swap;-(2s)   HEXA
  swap;over   +(2st)!   3rolld   +(2st)!
  <0h>;swap   4roll   FIELD
end
```

◆ LE SOUS-PROGRAMME DIV3

Comme on vient de le voir, il est chargé de traduire, dans le code-objet, les instructions **HEX**, **ASC**, **NUL**, **EXE**, **INC**, et **inc**, qui sont spécifiques à cet assembleur.

Au niveau 2 de la pile se trouve une chaîne représentant le deuxième argument de l'instruction. Au niveau 1 se trouve un entier-système rappelant à quelle position correspondait l'instruction dans la chaîne **ST3**.

Ici **<p>** vaut **<8h>** pour l'instruction **HEX**, **<9h>** pour l'instruction **ASC**,..., et finalement **<Dh>** pour l'instruction **inc**.

- On commence par retirer le caractère "/" s'il débute la chaîne présente au niveau 2 de la pile (c'est utile pour les instructions **ASC** et **EXE**).

- Pour l'instruction **HEX**: on vérifie que le deuxième argument, précédé de #, s'évalue bien en un entier binaire (par un appel à **GETARG**, uniquement pour tester la syntaxe, le résultat étant éliminé).
  Il faut en effet s'assurer que l'argument de **HEX** est une chaîne ne contenant que des caractères "**0**" à "**9**", et "**A**" à "**F**". On se contente ensuite de laisser la chaîne formée par le deuxième argument sur la pile.
  Par exemple: "**HEX D9D20**" laissera la chaîne "**D9D20**" sur la pile.

- Pour l'instruction **ASC**: On transforme cette chaîne en son image en mémoire (sous forme de chaîne), par **→STRI**, et on ne conserve que les caractères de position supérieure ou égale à **<Bh>** de cette chaîne.
  Par exemple: Dans l'instruction "**ASC abcde**", la chaîne "abcde" devient "C2A20F00001626364656", et on ne laisse sur la pile que "**1626364656**".

- Pour l'instruction **NUL**: on évalue la chaîne par **GETARG**. Si le résultat est un réel, on le transforme en entier-binaire. Si le résultat n'est pas alors un entier-binaire, il y a une erreur de code 4.
  On transforme cet entier-binaire en entier-système **<s>**, on crée un entier-binaire de **<s>** quartets nuls, que l'on transforme en chaîne par **→STRI**. On ne garde enfin que les caractères de position ≥ 11 dans cette chaîne.
  Par exemple: L'instruction "**NUL 7**" donne la chaîne "**0000000**" de 7 "0".

- Pour l'instruction **EXE**: Cette instruction n'a vraiment de sens que dans une structure Rpl. On évalue le deuxième argument de l'instruction par **GETARG**. Si le résultat est un objet intégré de la HP48, on le transforme en son adresse en ROM au moyen du sous-programme **→ADR**.
  On le transforme le résultat, au moyen de **→STRI**, en son image sous forme de chaîne de caractères dans le cas où l'objet obtenu est un nom (global, local, ou nom XLIB) ou une structure Rpl.
  S'il s'agit d'un entier binaire, on le transforme en chaîne par **→STRI**, mais on ne garde de cette chaîne que les caractères de position 11 à 15.
  Exemples: "EXE #3188" donne "88310". "EXE ABC" donne "84E2030142434".
  "EXE \DUP 1 +" donne "D9D2078BF19C2A276BA1B2310"
  "EXE DUP" donne "78BF1".

- Restent les deux instructions **INC** et **inc**.
  Pour ces deux instructions, on évalue la chaîne par **GETARG**. Si le résultat est un nom local (ou global) connu, on le remplace par son contenu.
  Le résultat est alors transformé en chaîne par **→STRI**. Cette chaîne est laissée telle quelle dans le cas de l'instruction **INC**. Dans le cas de l'instruction **inc**, on ne garde que les caractères de position ≥ 6 (pour supprimer l'adresse-préfixe).
  Exemple: L'instruction "**INC 1789**" donne "339203000000000098710".
  L'instruction "**inc 1789**" donne "3000000000098710".

Voici enfin le listing du sous-programme **DIV3**:

```
begin
  swap;dup   st=>ch   $/   same?   pift   del1(st)   swap
  <8h>   over;=(2s)?:drdq/sk
  begin   dup   $#   prep(st,ch)   GETARG   drop   end
  <9h>   over;=(2s)?:drdq/sk
  begin   ->STRI   <Bh>   sub(st,s,∞)   end
  <Ah>   over;=(2s)?:drdq/sk
  begin   GETARG   dup;r?   pift   r=>b   dup;b?
     ?:skip   doerr004   b=>s  makeb(s)
     ->STRI   <Bh>   sub(st,s,∞)
  end
  <Bh>   over;=(2s)?:drdq/sk
  begin
     GETARG   dup;StdObj?   pift   ->ADR   ChkArg!
     <6h>   ->STRI   <7h>   ->STRI   <8h>   ->STRI
     <Bh>
     begin   ->STRI   <Bh>   <10h>   sub(st,s2,s1-1)   end
     <Fh>   ->STRI   <0h>   doerr004
  end
  <Ch>  =(2s)?  swap  GETARG  dup;ln?  over  gn?  or(2?)
  pift   begin   dup;known(nm)?   pift   drop(2)   end
  ->STRI   swap   ?:q
  <6h>   sub(st,s,∞)
end
```

◆ LE SOUS-PROGRAMME ASS4

Il est chargé de traiter les instructions dont la première partie est un mot de 4 lettres.

Les différentes instructions possibles sont lues dans une chaîne, contenue dans la variable **'ST4'**. Vous devez donc d'abord stocker la chaîne ci-dessous (**sans les espaces et les sauts de ligne**) dans la variable **'ST4'**:

N.B: la commande **BYTES**, appliquée au contenu de cette chaîne, donne les résultats suivants: Taille 377 octets, CheckSum #A597h.

```
"REL2 REL3 REL4 REL5 ?A=B ?B=C ?C=A ?D=C ?A#B ?B#C ?C#A ?D#C
 ?A=0 ?B=0 ?C=0 ?D=0 ?A#0 ?B#0 ?C#0 ?D#0 ?A>B ?B>C ?C>A ?D>C
 ?A<B ?B<C ?C<A ?D<C ABEX BCEX CAEX DCEX A=-A B=-B C=-C D=-D
 CPEX HS=0 ST=0 ST=1 R0=A R1=A R2=A R3=A R4=A R0=C R1=C R2=C
 R3=C R4=C A=R0 A=R1 A=R2 A=R3 A=R4 C=R0 C=R1 C=R2 C=R3 C=R4
 D0=A D1=A D0=C D1=C A=IN C=IN C=ID ASLC BSLC CSLC DSLC ASRC
 BSRC CSRC DSRC ASRB BSRB CSRB DSRB XM=0 SB=0 SR=0 MP=0 PC=A
 PC=C A=PC C=PC C=ST ST=C RTNC NOP3 NOP4 NOP5"
```

Voici brièvement résumée l'action du programme **ASS4**. Rappelons qu'à l'entrée dans **ASS4**, les niveaux 1 et 2 contiennent (sous forme de chaîne de caractères) la première et la seconde partie de l'instruction en cours de traduction. On notera ces deux chaînes **Ch1** et **Ch2**.

- On commence par vérifier la présence de la chaîne **Ch1**, dans la chaîne **ST4**, à une place normale (sous peine d'une erreur de code 1). On place alors sur la pile l'entier **<p>** égal au numéro de l'instruction ainsi repérée (de **<p>** = <0> pour **REL2**, à **<p>** = <5Dh> pour **NOP5**).
  On entreprend ensuite une série de tests sur la valeur de **<p>**.

- Le premier test concerne le cas où **<p>** est inférieur à 4 (instructions **REL2** à **REL5**: ce sont des instructions spécifiques à cet assembleur).
  Dans ce cas, on prépare la pile pour un appel à **OFFSET**, qui est chargé d'examiner la deuxième partie **Ch2** de l'instruction (qui doit être un label). On se reportera aux commentaires sur le programme **OFFSET** pour les deux cas où le label invoqué a déjà été créé ou non.

- On teste ensuite si **<p>** est inférieur à <24h> (instructions **?A=B** à **D=-D**). On entre alors dans une nouvelle série de tests. Tous se rejoignent pour un appel à **HEXA** (chargé de compléter la partie fixe du code, tant pour le registre **A** que pour les autres registres) et un appel à **FIELD** (chargé d'examiner la chaîne **Ch2**, qui doit contenir un nom de registre, puis de laisser sur la pile le code correspondant à l'instruction pour ce registre).

- Le test suivant isole le cas où **<p>** est inférieur à <28h> (instructions **CPEX** à **ST=1**). La partie fixe de l'instruction est d'abord placée sur la pile. Elle est complétée par une caractère héxa, de "**0**" à "**F**", résultant de l'examen de la chaîne **Ch2** (qui doit contenir un entier compris entre 0 et 15), par **NIBBLE**.

- A ce stade, le mnémonique **drop(2)** montre que la chaîne **Ch2** est devenue inutile (les instructions restant ne nécessitent pas de deuxième partie).

- On teste maintenant si l'entier-système **<p>** est inférieur à <53h> (instructions **R0=A** à **MP=0**). Un test permet encore de séparer ce cas en les deux sous-cas où <p> est inférieur à <40h> (instructions **R0=A** à **D1=C**), et où <p> est supérieur ou égale à <40h> (instructions **A=IN** à **MP=0**).

- Le dernier test permet de séparer le cas p<#57h (instructions **PC=A** à **C=PC**) du cas p≥#57h (instructions **C=ST** à **NOP5**).

Voici le listing du programme **ASS4**:

```
begin
  ST4   swap   spos(2st)
  dup;=0(s)?:dq/sk   doerr001
  1-(s)   <4h>   mdv(2s)   swap
  <>0(s)?   pift   doerr001
  <4h>   over;>(2s)?   ?:dq/sk
  begin
     2+(s)   4pick   ssize(st)   dup   OFFSET
  end
  <24h>   over;>(2s)?   ?:dq/sk
  begin
     begin
        <14h>   over;>(2s)?   ?:dq/sk
        begin   "8A"   "9a"   <4h>   end
        <1Ch>   over;>(2s)?   ?:dq/sk
        begin   "8B"   "9b"   <14h>   end
        <20h>   over;>(2s)?   pifte
        begin   "D"   "Ab"   <10h>   end
        begin   "F"   "Bb"   <18h>   end
     end
     4roll   swap;-(2s)   HEXA   swap;over   +(2st)!
     3rolld   +(2st)!   <0h>;swap   4roll   FIELD
  end
  <28h>   over;>(2s)?   ?:dq/sk
  begin
     { "80F" "82" "84" "85" }   swap   <23h>
     -(2s)   get(cp,s)   swap;false   NIBBLE   +(2st)!
  end
  drop(2)
  <53h>   over;>(2s)?   ?:dq/sk
  begin
     <40h>   over;>(2s)?   pifte
     begin   <28h>   "1"
        "000102030408090A0B0C101112131418191A1B1C30313435"
     end
     begin   <40h>   "8"
        "020306101112131415161 71C1D1E1F21222428"
     end
     swap2   -(2s)   2*(s)   1+(s)   dup;1+(s)   sub(st,2s)
     +(2st)!
  end
  <57h>   over;>(2s)?   ?:dq/sk
  begin   <51h>   -(2s)   HEXA   "81B"   swap;+(2st)!   end
  <56h>   -(2s)   { "09" "0A" "400" "420" "6300" "64000" }
  swap   get(cp,s)
end
```

◆ LE SOUS-PROGRAMME ASS5

Il est chargé de traiter les instructions dont la première partie est un mot de 5 lettres. Les différentes instructions possibles sont lues dans une chaîne, contenue dans la variable **'ST5'**. Vous devez donc d'abord stocker la chaîne ci-dessous (**sans les espaces et les sauts de ligne**) dans la variable **'ST5'**:

N.B: la commande **BYTES**, appliquée au contenu de cette chaîne, donne les résultats suivants: Taille 515 octets, CheckSum #66CDh.

```
"A=A+B B=B+C C=C+A D=D+C A=A+A B=B+B C=C+C D=D+D B=A+B C=B+C
 A=C+A C=D+C A=A-1 B=B-1 C=C-1 D=D-1 A=A-B B=B-C C=C-A D=D-C
 A=A+1 B=B+1 C=C+1 D=D+1 B=B-A C=C-B A=A-C C=C-D A=B-A B=C-B
 C=A-C D=C-D A=A&B B=B&C C=C&A D=D&C B=A&B C=B&C A=C&A C=D&C
 A=A!B B=B!C C=C!A D=D!C B=A!B C=B!C A=C!A C=D!C ?A>=B ?B>=C
 ?C>=A ?D>=C ?A<=B ?B<=C ?C<=A ?D<=C ?HS=0 ?ST=0 ?ST=1 RTNSC
 RTNCC CLRST CSTEX P=P+1 P=P-1 AR0EX AR1EX AR2EX AR3EX AR4EX
 CR0EX CR1EX CR2EX CR3EX CR4EX AD0EX AD1EX CD0EX CD1EX D0=AS
 D1=AS AD0XS AD1XS D0=CS D1=CS CD0XS CD1XS OUT=C C+P+1 RESET
 BUSCC SREQ? ?XM=0 ?SB=0 ?SR=0 ?MP=0 INTON BUSCB BUSCD APCEX
 CPCEX RTNNC"
```

Voici un résumé de l'action qui se déroule dans **ASS5**. Rappelons que les niveaux 1 et 2 contiennent deux chaînes représentant respectivement la première et la seconde partie de l'instruction. Appelons ces chaînes **Ch1** et **Ch2**.

- On commence par vérifier que la chaîne **Ch1** apparaît quelque part dans la chaîne **ST5**. Si ce n'est pas le cas, on appelle le programme **ASS56**, qui teste s'il s'agit d'une des instructions dont la première partie peut occuper 5 ou 6 caractères (voir plus loin les commentaires de **ASS56**).
- Si la chaîne **Ch1** est bien inscrite dans **ASS5**, on vérifie tout de même que la position qu'elle occupe en fait bien un des mots de l'assembleur du Saturn (il faut éviter, par exemple, que **Ch1** soit égale à **"A+BB="**, qui est lue dans **ST5** à la position 3, mais qui ne correspond pas à une instruction).

  Si ce n'est pas le cas, il y a une erreur de code 1 (***"Instruction inconnue"***). Sinon on place sur la pile l'entier **<p>** représentant le numéro de l'instruction lue dans **ST5** (la première, **A=A+B**, ayant le numéro **<0>**, et la dernière, **RTNNC**, ayant le numéro **<65h>**).

  On commence alors une succession de tests de manière à déterminer à quelle famille d'instructions appartient celle que nous étudions, et à appliquer le traitement approprié.
- Le premier test concerne le cas où l'entier **<p>** est inférieur à **<38h>** (instructions **A=A+B** à **?D<=C**). Dans ce cas le programme **HEXA**, se basant sur la valeur de l'entier **<p>**, permet de former le code de l'instruction, autant pour le registre **A** que pour un autre registre, et de préparer la pile pour un appel au programme **FIELD**, qui est chargé de contrôler que la chaîne **Ch2** représente bien un champ, et de laisser sur la pile le code de l'instruction qui correspond à ce champ.

- Le premier test concerne le cas où l'entier **<p>** est inférieur à **<3Bh>** (instructions **?HS=0**, **?ST=0**, **?ST=1**). Dans ce cas un appel au sous-programme **NIBBLE** permet de contrôler que la chaîne **Ch2** correspond bien à un entier **<n>** compris entre 0 et 15. On laisse alors sur la pile le code de l'instruction (où figure, en dernière position, le quartet **<n>**).
- Si ces deux tests ont été négatifs, l'instruction **drop(2)** élimine la chaîne **Ch2**, encore présente au niveau 2 de la pile. Toutes les instructions qui vont maintenant être traitées ne nécessitent en effet pas de deuxième partie.
- On entre ensuite dans une nouvelle série de tests qu'il serait trop long de détailler. Tous cependant conduisent à un traitement analogue (sauf pour l'instrucion **RTNNC**, de code "**500**", pour laquelle on devinera l'utilité de **DropRet**).

Voici le listing du programme **ASS5**:

```
begin
  ST5  over  spos(2st)  dup;=0(s)?:drdq/sk  ASS56  drop(2)
  1-(s)  <5h>  mdv(2s)  swap  <>0(s)?  pift  doerr001
  <38h>  over;>(2s)?  ?:dq/sk
  begin  <10h>  mdv(2s)  <3h>  over  4rolld  =(2s)?  pift
     8+(s)  HEXA  {"C" "Aa" "E" "Ba" "0EF" "0Ea" "8B" "9b"}
     rot  2*(s)  1+(s)  dup;1+(s)  sub(cp,2s)  break;drop
     3pick  +(2st)!  swap(1,3)  +(2st)!
     <0h>;swap  4roll  FIELD
  end
  <3Bh>  over;>(2s)?  ?:dq/sk
  begin  "8"  "367"  rot  <37h>  -(2s);dup  sub(st,2s)
     +(2st)!;swap  false  NIBBLE  +(2st)!
  end
  drop(2)  begin
     begin  <41h>  over;>(2s)?  ?:dq/sk
        begin  <3Ah>  "0"  "238BCD"  end
        <4Bh>  over;>(2s)?  ?:dq/sk
        begin  <40h>  "12"  "0123489ABC"  end
        <57h>  over;>(2s)?  ?:dq/sk
        begin  <4Ah>  "13"  "236789ABCDEF"  end
        <5Ch>  over;>(2s)?  ?:dq/sk
        begin  <56h>  "80"  "19ABE"  end
        <60h>  over;>(2s)?  ?:dq/sk
        begin  <5Bh>  "83"  "1248"  end
        <63h>  over;>(2s)?  ?:dq/sk
        begin  <5Fh>  "808"  "03D"  end
        <65h>  over;>(2s)?  ?:dq/sk
        begin  <62h>  "81B"  "67" end
        drop  "500"  DropRet
     end
     swap2  -(2s)  dup  sub(st,2s)  +(2st)!
  end
end
```

◆ LE SOUS-PROGRAMME ASS6

Il est chargé de traiter les instructions dont la première partie est un mot de 6 lettres. Les différentes instructions possibles sont lues dans une chaîne, contenue dans la variable **'ST6'**. Vous devez donc d'abord stocker la chaîne ci-dessous (**sans les espaces et les sauts de ligne**) dans la variable **'ST6'**:

N.B: la commande **BYTES**, appliquée au contenu de cette chaîne, donne les résultats suivants: Taille 383 octets, CheckSum #288Fh.

```
"D0=(2) D0=(4) D0=(5) D1=(2) D1=(4) D1=(5) A=-A-1 B=-B-1
 C=-C-1 D=-D-1 ASRB.F BSRB.F CSRB.F DSRB.F A=R0.F A=R1.F
 A=R2.F A=R3.F A=R4.F C=R0.F C=R1.F C=R2.F C=R3.F C=R4.F
 R0=A.F R1=A.F R2=A.F R3=A.F R4=A.F R0=C.F R1=C.F R2=C.F
 R3=C.F R4=C.F DAT0=A DAT1=A A=DAT0 A=DAT1 DAT0=C DAT1=C
 C=DAT0 C=DAT1 D0=D0+ D1=D1+ D0=D0- D1=D1- ABIT=0 ABIT=1
 CBIT=0 CBIT=1 RTNSXM SETHEX SETDEC RSTK=C C=RSTK OUT=CS
 UNCNFG CONFIG SHUTDN PC=(A) PC=(C) INTOFF CLRHST"
```

Voici un résumé du fonctionnement de **ASS6**. Au moment où débute ce programme les niveaux 1 et 2 sont occupés par deux chaînes (notons les **Ch1** et **Ch2**) représentant respectivement la première et la seconde partie de l'instruction à étudier.

- Comme pour le programme **ASS5**, on commence par vérifier que la chaîne **Ch1** est une sous-chaîne de **ST6**. Dans le cas contraire on appelle le sous-programme **ASS56**.

- On vérifie que la position où **Ch1** a été lue dans **ST6** est correcte (sinon erreur de code 1). On place ensuite sur la pile un entier-système **<p>** donnant le numéro (minimum <0> pour **D0=(2)**) de l'instruction lue dans **ST6**.
  On entame ensuite une succession de tests sur la valeur de **<p>**. Quand l'un des ces tests est OK, on applique le traitement correspondant, puis on quitte **ASS6**.

- Le premier test concerne le cas où **<p>** est inférieur à 6 (instructions **D0=(2)** à **D1=(5)**). On utilise d'abord **<p>** pour former la partie fixe du code de l'instruction.
  On appelle ensuite **GETARG** pour évaluer le contenu de la chaîne **Ch2**. Si le résultat est un nom local on le remplace par son contenu. Si le résultat est réel on le transforme en entier-binaire. Si le résultat final n'est pas un entier binaire, il y a une erreur de code 4 ("***Argument incorrect ou absent***").
  L'entier-binaire est alors transformé par **→STRI** en son image en mémoire, sous forme de chaîne. Cette chaîne est ensuite tronquée à sa bonne longueur (on utilise encore pour celà la valeur de l'entier **<p>**). La chaîne tronquée est alors ajoutée à la partie fixe du code de l'instruction.

- Le second test concerne le cas où **<p>** est inférieur à <22h> (instructions **A=−A−1** à **R4=C.F**). A l'intérieur de ce cas se dessinent plusieurs sous-cas qu'il serait trop long de détailler.
  Toujours est-il que ces différents sous-cas se rejoignent sur la fin, pour un appel à **HEXA** (de manière à former la partie fixe du code de l'instruction, tant pour le champ **A** que pour les autres champs) et un appel à **FIELD** (pour contrôler que la chaîne **Ch2** représente bien un champ de registre, et pour laisser sur la pile le code de l'instruction relatif à ce registre).

- Deux tests successifs sont alors nécessaires pour traiter les instructions restantes dont le numéro est inférieur à <2Ah> (instructions **DAT0=A** à **C=DAT1**).
  On commence par tester si la chaîne **Ch2** (la seconde partie de l'instruction) représente un nom global (on le vérifie par un appel à **GETARG**). Si c'est le cas, c'est sans doute que **Ch2** représente un nom de champ. On place alors sur la pile le code de l'instruction, tant pour un registre quelconque que pour le registre **B** que pour le registre **A**, et on appelle le sous-programme **FIELD** pour conclure et laisser sur la pile le bon code.
  Sinon, c'est que le deuxième champ de l'instruction doit être un entier **<n>** compris entre 1 et 16. Le sous-programme **HEXA** permet de former la partie fixe de l'instruction (on utilise l'entier **<p>**). On traite ensuite la deuxième partie de l'instruction par un appel à **NIBBLE** (et enfin on ajoute le caractère obtenu, de "**0**" à "**F**"), à la partie fixe déjà formée.

- Le test suivant traite le cas où **<p>** est inférieur à <2Eh> (instructions **D0=D0+** à **D1=D1−**). On crée d'abord, avec **<p>**, la partie fixe de l'instruction. Le traitement de la deuxième partie **Ch2**, qui doit représenter un entier de 1 à 16, est alors confié au programme **NIBBLE**.

- Le test suivant traite le cas où **<p>** est inférieur à <32h> (instructions **ABIT=0** à **CBIT=1**). Le traitement est analogue au précédent, excepté le fait que la deuxième partie **Ch2** doit représenter un entier compris entre 0 et 15. Le booléen transmis au sous-programme **NIBBLE** est alors **False** (voir les commentaires sur **NIBBLE**).

- A ce stade, l'instruction **drop(2)** (qui élimine la chaîne **Ch2**) montre que l'on va maintenant traiter des instructions qui ne nécessitent pas de deuxième partie.

- On entre alors dans une dernière série de tests, laissés à la curiosité du lecteur. Ces différents tests préparent la pile pour enfin se rejoindre sur les instructions "**swap2 -(2s);dup sub(st,2s) +(2st)!**", qui placent au niveau 1 de la pile le code qui correspond à l'instruction.

Voici enfin le listing du programme **ASS6**:

```
begin
  ST6   over   spos(2st)   dup;=0(s)?:drdq/sk   ASS56
  drop(2)   1-(s)   <6h>   mdv(2s)   swap
  <>0(s)?   pift   doerr001   <6h>   over;>(2s)?   ?:dq/sk
  begin
     1+(s)  "1"  "9ABDEF"  3pick  dup  sub(st,2s)  +(2st)!
     rot   GETARG   dup;ln?   pift   rcl(nm)
     dup;r?   pift   r=>b   dup;b?   ?:skip   doerr004
     ->STRI   <Bh>   { <Ch> <Eh> <Fh> <Ch> <Eh> <Fh> }
     5roll get(cp,s)   sub(st,2s)   +(2st)!
  end
  <22h>   over;>(2s)?   ?:dq/sk
  begin
     begin   <Ah>   over;>(2s)?   ?:dq/sk
        begin   6+(s)   "F"   "Bb"   rot   end
        <Eh>   over;>(2s)?   ?:dq/sk
        begin   <Ah>   -(2s)   "819F"  "819a"  rot  end
        begin   <18h>   over;>(2s)?   pifte
           begin   <Eh>   "81AF1"   "81Aa1"   end
           begin   <18h>   "81AF0"   "81Aa0"   end
           swap2  -(2s)  <5h>  mdv(2s)  8*(s)  +(2s)
        end
     end
     HEXA   swap;over   +(2st)!   3rolld   +(2st)!
     <0h>;swap   4roll   FIELD
  end
  <2Ah>  over;>(2s)?  3pick  GETARG  gn?  and(2?)  ?:dq/sk
  begin
     <22h>  -(2s)  dup  HEXA  "15"  over  +(2st)!  $a
     +(st,ch)   "14"   4roll   8+(s)   HEXA   +(2st)!
     "14"   4roll   +(2st)!   4roll   FIELD
  end
  <2Ah>   over;>(2s)?   ?:dq/sk
  begin   <1Ah>   -(2s)   HEXA   "15"   swap;+(2st)!
     swap;true   NIBBLE   +(2st)!
  end
  <2Eh>   over;>(2s)?   ?:dq/sk
  begin  <29h>  -(2s)  "1"  "678C"  rot;dup  sub(st,2s)
     +(2st)!   swap;true   NIBBLE   +(2st)!
  end
  <32h>   over;>(2s)?   ?:dq/sk
  begin  <2Dh>  -(2s)  "808"  "4589"  rot;dup  sub(st,2s)
     +(2st)!   swap;false   NIBBLE   +(2st)!
  end
  drop(2)
  begin   <37h>   over;>(2s)?   ?:dq/sk
     begin   <31h>   "0"   "04567"   end
     <3Bh>   over;>(2s)?   ?:dq/sk
     begin   <36h>   "80"   "0457"   end
     <3Eh>   over;>(2s)?   ?:dq/sk
     begin   <3Ah>   "808"   "CEF"   end
     drop   "82F"   DropRet
  end
  swap2   -(2s);dup   sub(st,2s)   +(2st)!
end
```

◆ LE SOUS-PROGRAMME ASS56

Il est chargé de contrôler des instructions de l'assembleur du Saturn dont la première partie a une longueur allant de 5 à 6 caractères. Ces instructions sont:

A=A+*(n+1)* *field*, **B=B+***(n+1)* *field*, **C=C+***(n+1)* *field*
A=A-*(n+1)* *field*, **B=B-***(n+1)* *field*, **C=C-***(n+1)* *field*

Dans les notations ci-dessus, *field* est un champ de registre, et *(n+1)* est un entier compris entre 1 et 16 (le fait que cet entier soit codé sur un ou deux caractères implique que la longueur totale de la première partie de l'instruction varie de 5 à 6 caractères.

Vous devez stocker, dans la variable '**ST56**', la chaîne suivante (**sans les espaces**): Taille 37 octets, CheckSum #E143h.

```
"A=A+ B=B+ C=C+ D=D+ A=A- B=B- C=C- D=D-"
```

Voici un résumé du fonctionnement de **ASS56**:

- On commence par contrôler que la chaîne formée par les 4 premiers caractères du premier argument de l'instruction fait bien partie de la liste **A=A+**,..,**D=D−**. Dans le cas contraire, il y a une erreur de code 1 ("***Instruction inconnue***").
- Soit **<p>** la position de l'instruction reconnue dans le chaîne **ST56** (de **<p>=<0>** pour "**A=A+**", à **<p>=<7>** pour "**D=D−**". Le sous-programme **HEXA** permet de créer, en utilisant **<p>**, la partie fixe de l'instruction, tant pour le champ **A** que pour les autres champs.

  Le sous-programme **NIBBLE** se charge de lire l'entier qui termine la première partie de l'instruction (par exemple l'entier 16 de l'instruction **B=B+16**, qui est transformé par **NIBBLE** en le caractère "**F**").
- Le code de l'instruction étant créé, autant pour le champ **A** que pour les autres champs, il reste à appeler le programme **FIELD**, qui se charge de contrôler que la deuxième partie de l'instruction est bien un spécification de champ, et qui laisse sur la pile le codage définitif de l'instruction.

Voici le listing du programme **ASS56**:

```
begin
  ST56   over   <5h>   sub(st,1,s-1)
  spos(2st)   dup;=0(s)?:dq/sk   doerr001
  1-(s)   <4h>   mdv(2s)   swap
  <>0(s)?   pift   doerr001
  <4h>   mdv(2s)   8*(s)   +(2s)   HEXA
  "818F"   over   +(2st)!   "818a"   rot   +(2st)!
  rot   <5h>   sub(st,s,∞)
  true   NIBBLE
  swap;over   +(2st)!   3rolld   +(2st)!
  <0h>;swap   4roll   FIELD
end
```

◆ LE SOUS-PROGRAMME ASS7

Il est chargé de traduire les instructions dont le premier argument est une chaîne longue de 7 caractères. Vous devez tout d'abord stocker la chaîne suivante (**sans espaces ni saut de ligne!**) dans la variable '**ST7**'.

N.B: la commande **BYTES**, appliquée au contenu de cette chaîne donne les résultats suivants: Taille 103 octets, CheckSum #9A9Eh.

```
"?ABIT=0 ?ABIT=1 ?CBIT=0 ?CBIT=1 AR0EX.F AR1EX.F AR2EX.F
 AR3EX.F AR4EX.F CR0EX.F CR1EX.F CR2EX.F CR3EX.F CR4EX.F"
```

Voici comment fonctionne le programme **ASS7**:

- On commence par vérifier que le premier argument de l'instruction est bien l'un des mots de la liste **ST7** (une séquence de 7 caractères, débutant à une position 7*p). Dans le cas contraire il y a une erreur de code 1.
- On a donc au niveau 1 l'entier **<p>** représentant la position de l'instuction dans **ST7** (**<p> = <0>** pour **?ABIT = 0** à **<p> = <Dh>** pour **CR4EX.F**).
- On fait 2 cas selon que **<p>** < 4 (instructions **?ABIT = 0** à **?CBIT = 1**) ou non (instruction **AR0EX.F** et suivantes).
- Dans le premier cas on ajoute à la chaîne "**808**" l'un des caractères "**6**", "**7**", "**A**", "**B**". On obtient ainsi la partie fixe du code. Il suffit alors d'évaluer le second argument de l'instruction grâce à un appel à **NIBBLE**. Le caractère héxa obtenu, de "**0**" à "**F**", est alors ajouté à la partie fixe de l'instruction.
- Dans le second cas, on prépare la pile pour un appel à l'nstruction **FIELD** (le sous-programme **HEXA** est appelé pour former le code de l'instruction, tant pour le champ **A** que pour les autres champs).

Voici donc le listing du programme **ASS7**:

```
begin
  ST7   swap   spos(2st)
  dup;=0(s)?:dq/sk   doerr001
  1-(s)   <7h>   mdv(2s)   swap
  <>0(s)?   pift   doerr001
  <4h>   over;>(2s)?   ?:dq/sk
  begin
     1+(s)   "808"   "67AB"   rot;dup   sub(st,2s)
     +(2st)!;swap   false   NIBBLE   +(2st)!
  end
  "81AF2"   "81Aa2"   rot   4-(s)   <5h>   mdv(2s)
  8*(s)   +(2s)   HEXA   swap;over   +(2st)!
  3rolld   +(2st)!   <0h>;swap   4roll   FIELD
end
```

◆ CONCLUSION

La pauvreté de l'éditeur de la HP48 rend l'utilisation du programme ASM48 assez incommode dès que votre texte dépasse quelques dizaines de lignes. La meilleure solution consiste alors à utiliser un éditeur 7 lignes, comme le String-Writer de J.Y.AVENARD et C.BOURGEOIS.

Le programme ASM48 rendra peut-être service aux lecteurs qui, faute d'ordinateur, doivent programmer en assembleur directement sur leur HP48 (encore que là aussi existe l'assembleur ASMFLASH de Phong NGUYEN, entièrement écrit en langage-machine). Ceux qui possèdent un compatible PC auront tout intérêt à se tourner vers les programmes EXTERNPC et ASMPC.

ASM48 doit surtout être considéré, à mon sens, comme un exemple de ce que peut apporter la programmation en Externals: une écriture à peu près aussi limpide que le RPL-utilisateur (pour peu que l'on programme directement avec les mnémoniques, comme le permet EXTERNPC), mais une syntaxe incomparablement plus riche.

UN ASSEMBLEUR SUR PC

Le programme EXTERNPC nous avait permis de porter l'écriture de programmes en Externals directement sur un ordinateur (une amélioration sensible par rapport à →**EXT**, sur les plans de la rapidité et de la convivialité).

De la même manière, pour ceux qui possèdent un ordinateur (de type compatible PC), l'assembleur ASMPC permet, dans de bien meilleures conditions de confort qu'ASM48, d'écrire des programmes dans le langage-machine du "Saturn".

◆ L'ASSEMBLEUR ASMPC

Ce chapitre est consacré à la présentation d'un programme, baptisé **ASMPC**, écrit en Turbo-Pascal (versions 4 et suivantes), qui vous offre un environnement intégré vous permettant, sans quitter le programme:

- D'éditer des fichiers-source en langage d'assemblage.
- D'assembler ces fichiers-source pour produire des fichiers-objet.
- De transmettre ces fichiers-objet à la HP48, via **KERMIT**. Les fichiers ainsi transmis seront du type "***objet-code***".
- De produire un listing (facultatif) de cet assemblage.

Il existe de très nombreux points communs entre **ASM48** et **ASMPC**, notamment en ce qui concerne l'algorithme utilisé (ce qui m'autorisera à ne pas trop insister, et ce qui facilitera votre compréhension...).

Il y a cependant quelques différences, et notamment celle-ci: Le programme **ASMPC** ne permet que de générer du code-machine.

Il n'est donc pas possible ici de créer, par exemple, des structures Rpl contenant des objets-code. La solution, pour celà, consiste à compiler au préalable des objets-code avec **ASMPC**, puis d'inclure ces objets-code dans une structure Rpl écrite avec le programme **EXTERNPC** (utilisation du préfixe [[).

Les fichiers-source utilisés doivent être des fichiers-texte au format ASCII pur (utilisez votre traitement de texte préféré, en dehors ou à partir du programme **ASMPC**). Voici quelques caractéristiques que doivent posséder ces fichiers-texte.

- Comme avec le programme **ASM48**, chaque ligne doit être consacrée à au plus une instruction. Une ligne peut fort bien être vide. Les espaces (ou tabulations) qui débutent la ligne ou qui séparent les deux premiers mots sont ignorés.
- Dans le cas d'une instruction en un seul mot (**RTN** par exemple), la suite éventuelle de la ligne n'est pas prise en compte (on peut y placer des commentaires). Il en va de même pour tout ce qui suit le deuxième mot dans une instruction qui doit être suivie d'un argument.
- Le deuxième mot d'une ligne, s'il existe, est coupé à la première tabulation ou au premier espace rencontrés (ou bien sûr à la fin de ligne). Celà peut être gênant si le deuxième argument de la ligne est une chaîne de caractères qui doit contenir un espace (exemple des instructions **LA(*n*)**, **LC(*n*)**, ou **ASC**).

  Dans ce cas, entourez cet argument par deux caractères ", qui seront enlevés lors du traitement par **ASMPC**.
- Contrairement au programme **ASM48**, vous pouvez fort bien faire débuter une ligne par un label (étiquette), et continuer sur cette ligne par une instruction en une ou deux parties. La syntaxe d'une ligne pourrait donc être:

  <Label> <Instruction> <Argument> <Commentaire>,

  aucun de ces quatre champs n'étant obligatoire, ni ne devant débuter à une colonne précise (les champs doivent être séparés par au moins un espace ou une tabulation).

◆ PARTICULARITÉS DE SYNTAXE

Le programme **ASMPC** connaît à peu près les mêmes particularités de syntaxe que **ASM48**. Rappelons que les espaces ou tabulations de tête sont toujours ignorés.

▸ **COMMENTAIRES**: pour qu'une ligne entière soit considérée comme un commentaire, faites la débuter par le signe * (***astérisque***). Celà vous permet éventuellement de soustraire une ou plusieurs lignes à l'assemblage. Rappelons également que vous pouvez terminer une ligne par des commentaires libres dès lors que l'instruction qui débute cette ligne est complète.

▸ **LABELS**: pour déclarer un label, faites débuter la ligne par le caractère : (***deux points***), suivi immédiatement par le nom du label. Il n'y a aucune contrainte sur les caractères que vous pouvez utiliser pour le nom du label. A vous d'éviter les ambiguités.

Une ligne peut commencer par un label et continuer par une instruction (ce n'était pas le cas avec ASM48).

▸ Bien entendu, vous ne pouvez pas utiliser un même label pour désigner deux points différents de votre fichier. De même il ne peut y avoir d'homonymie entre le nom d'un label et celui d'une variable (voir ci-après).

Il est bien sûr possible d'appeler un label (dans une instruction de saut) sans l'avoir encore déclaré. Le programme **ASMPC** se charge de tout. Il vous préviendra, à la fin de l'assemblage, si un label a été appelé sans être déclaré.

▸ **BRANCHEMENTS**: comme dans le programme **ASM48**, tous les branchements relatifs (GOTO, GOC, GONC, GOSUB, GOSUBL, GOYES, GOLONG) doivent s'effectuer vers un label (déjà déclaré, ou qui le sera plus tard dans le fichier-source).

Pour les branchements absolus, vous pouvez utiliser des adresses sous forme héxadécimale (pour l'adresse 0679B, par exemple, vous pouvez utiliser les 2 syntaxes #679B ou $679B). Vous pouvez aussi utiliser des variables locales.

▸ **VARIABLES LOCALES**: Vous pouvez créer des variables locales, dont la durée de vie va de la ligne où elles sont déclarées à la fin du fichier. Deux variables ne peuvent porter le même nom.

Pour déclarer une variable locale, faites débuter la ligne par le symbole = (***égal***) suivi immédiatement du nom de cette variable (qui doit lui aussi commencer par une lettre). Le contenu de cette variable est le deuxième mot de la ligne. ce doit être un entier (sous forme décimale, ou héxa: par exemple, les trois formes 123, #7B, ou $7B sont équivalentes).

Vous pouvez par exemple écrire les deux lignes suivantes:

```
=SAVEREGS#679B
.....
=VALEUR          123
..... (quelques lignes plus tard...)
GOSBVL           SAVEREGS
D0=(2)           VALEUR
```

▸ LIER UN FICHIER TEXTE:

Vous pouvez "***linker***" des fichiers-texte secondaires à votre fichier-source principal. Pour celà faites débuter la ligne par le symbole ***[*** (***crochet ouvrant***) suivi immédiatement du nom du fichier.

Le fichier-secondaire doit contenir des instructions en assembleur, tout comme le fichier principal (**N.B**: aucune vérification n'est faite sur la nature de ce fichier-secondaire). Tout fichier-secondaire peut lui aussi faire appel à d'autres fichiers.

C'est le fichier-principal qui est chargé d'effectuer le retour au Rpl ambiant. C'est donc lui qui contiendra les instructions habituelles effectuant ce retour (seule exception possible: la dernière instruction du fichier-source principal est une instruction de liaison avec un fichier-source secondaire, qui effectuera donc lui-même le retour au Rpl).

N.B: les variables locales déclarées dans un fichier secondaire peuvent ensuite être appelées après le retour dans le fichier principal.

De la même manière, les déclarations de labels (et les branchements vers des labels) peuvent se faire d'un fichier à l'autre (même si celà n'est pas très recommandé).

▸ LIER UN FICHIER BINAIRE:

Vous pouvez liez un fichier binaire au fichier objet en cours de création. Contrairement au cas précédent, l'inclusion de fichier se fait dans le code objet, et non pas dans le texte source.

Le fichier inclus doit avoir été créé par KERMIT (et transmis de la HP48 à l'ordinateur **en mode binary**). Ce point est vérifié par le programme **ASMPC**. Par exemple, le fichier inclus peut contenir un objet-code (déjà créé par **ASMPC**) ou un objet-graphique, etc....

Pour effectuer cette liaison, vous débutez la ligne par un double crochet ouvrant **[[**, suivi immédiatement (sans espace) du nom du fichier à inclure.

Cette instruction admet un argument optionnel qui, s'il est présent, doit être un nombre entier. Ce nombre représente le nombre de quartets qu'il faut "sauter" en tête de l'objet à inclure, avant de l'insérer dans le code-objet. Si ce nombre est absent, il est considéré égal à 0 (tout l'objet est inclus).

Imaginons par exemple que le fichier **IMAGE** contienne un objet-graphique, sauvé en mode binary.

L'instruction: **[[IMAGE** inclut la totalité du Grob dans le code-objet.

L'instruction: **[[IMAGE 20** inclut cet objet-graphique, à l'exception de ses 20 premiers quartets (adresse-préfixe, taille, nombre de lignes et nombre de colonnes).

◆ INSTRUCTIONS PARTICULIERES

Le programme **ASMPC** reconnait les mêmes instructions particulières que le programme **ASM48**, à l'exception de **EXE**, **INC**, et **inc**.

Il reconnait de plus les instructions **LAREL**, **LCREL** (chargement dans **A**:*A* ou **C**:*A* d'un offset relativement à un label), ou **LAABS**, **LCABS** (chargement dans **A**:*A* ou **C**:*A* de l'adresse absolue relative à un label).

▸ **INSTRUCTION HEX:**

Elle permet d'incorporer au code-objet un certain nombre (au plus 255) de caractères héxadécimaux. Une erreur est évidemment détectée si l'un au moins de ces caractères n'est pas dans l'ensemble '0'..'9','A'..'F'.

Exemple: HEX 4F2AA80C

▸ **INSTRUCTION ASC:**

Elle permet d'incorporer une chaîne de caractères au code-objet (au plus 127 caractères). Chaque caractère sera codé selon son code ascii, sur deux quartets.

Si cette chaîne doit contenir des espaces ou des tabulations, entourez la de deux caractères ". Ces deux caractères ne seront pas pris en compte.

Exemple: ASC ESSAI
ASC "NOUVEL ESSAI"

▸ **INSTRUCTION NUL:**

Elle permet d'incorporer un certain nombre de quartets nuls dans le code objet. Le nombre de ces quartets est précisé par le deuxième argument de l'expression.

Exemple: NUL #100 (256 quartets nuls).

▸ **INSTRUCTIONS REL2, REL3, REL4, REL5:**

Elles doivent être suivies d'un label (déjà déclaré ou non), et génèrent le décalage (négatif ou positif) entre l'adresse en cours et l'adresse du label (adresses relatives au code-objet).

▸ **INSTRUCTIONS LC(*n*) et LA(*n*):**

L'entier *n* peut être compris entre 1 et 16. L'argument de **LA(*n*)** et de **LC(*n*)** peut être un entier (décimal, ou héxadécimal débutant par **$** ou **#**) une variable (déjà déclarée, dont le contenu doit être un entier) ou une chaîne de caractères (encadrée de deux caractères ", qui seront ensuite ignorés).

- **INSTRUCTIONS LAREL et LCREL:**

  Elles doivent être suivies d'un label désignant un autre point du programme. Leur rôle est de charger, dans le registre **A** (sur 5 quartets) pour **LAREL**, et dans le registre **C** (sur 5 quartets) pour **LCREL**, l'offset (déplacement) calculé entre l'adresse où débute l'instruction et l'adresse correspondant au label.

  Considérons la séquence suivante:

```
adr1        :Label   A=DAT1   A
                     DAT0=A   A
                     D0=D+    5
            ......................
adr2                 LCREL    Label
```

  Ici le registre **C** est chargé (sur 5 quartets) avec la valeur (adr1-adr2). En fait tout se passe comme si on avait l'instruction **LC(5) adr1-adr2** (le contenu du registre **P** est important, car il indique la position à partir de laquelle les 5 quartets sont écrits dans le registre).

- **INSTRUCTIONS LAABS et LCABS:**

  Elles doivent être suivies d'un label désignant un autre point du programme. Leur rôle est de charger, dans le registre **A** (champ *A*) pour **LAABS**, et dans le registre **C** (champ *A*) pour **LCABS**, l'adresse absolue correspondant au label.

  Le code objet en cours d'exécution n'a aucune raison de se trouver à une adresse fixée (si ce code est contenu dans une variable, vous pouvez par exemple déplacer cette variable dans un menu VARS). Il est cependant parfois utile de connaître l'adresse réelle où, au moment présent, se trouve telle ou telle partie du code (il s'agit souvent de pointer sur des données à l'intérieur de ce code).

  Les instructions **LAABS** et **LCABS** sont en fait des macro-instructions, puisqu'elles génèrent un code correspondant à plusieurs instructions successives.

  Plus précisément, l'instruction "***LAABS Loop***" (où "Loop" serait le nom d'un label) génère le code qui correspondrait à la séquence:

```
P=0                 Met le registre P à 0 (pour le LAREL)
RSTK=C              Sauvegarde C:A
C=PC                L'adr. de l'instruct. LAREL.. va dans C:A
LAREL    Loop       A:A= adresse de "Loop" - adresse de "LAREL"
A=C+A    A          A:A= adresse de "Loop"
C=RSTK              Récupère C:A
```

  Vous remarquez donc que l'adresse absolue de "Loop" est calculée au moment de l'exécution (et pas au moment de la compilation), et que le contenu de **C** est inchangé dans cette opération. Par contre le registre **P** est mis à zéro, pour assurer le fonctionnement correct de l'ensemble.

  Il en va de même avec l'instruction **LCABS** (qui utilise mais ne modifie pas le registre **A**, et qui annule le registre **P**). En fait **LCABS** produit le même code que **LAABS**, mais encadré par deux instructions **CAEX A** (échange des champs *A* des registres **A** et **C**).

◆ LE LISTING DE ASMPC

Les pages qui suivent contiennent le listing du programme **ASMPC**. Elles appellent quelques remarques:

- Au début du programme, la directive de compilation **{$M 6000,0,4000}** permet de répartir l'espace mémoire disponible entre le segment de pile et le tas.
  Le tas doit contenir les variables dynamiques (les pointeurs: c'est sous la forme d'une liste chaînée de pointeurs que le programme **ASMPC** gère le problème des variables locales et des labels).
  Le segment de pile doit être assez important pour supporter les paramètres empilés à chaque appel de procédure.
  Enfin ce qui reste de mémoire vive doit être assez important pour vous permettre de charger, sans sortir de **ASMPC**, votre traitement de textes ou le programme **KERMIT**.
  Les valeurs que j'utilise pour cette directive me conviennent (ou plutôt conviennent à mon ordinateur). Il n'est pas impossible que vous ayez vous même à effectuer quelques règlages.

- Les déclarations de constantes du programme **ASMPC** font apparaître des tableaux de chaînes de caractères (**ST3**, **ST4**, etc...) où vous retrouverez toutes les instructions du SATURN (faites bien attention que dans ces chaînes, **chaque instruction est suivie d'un espace**).

- La variable **tab** est un tableau de caractères indicé de 0 à 50000. C'est dans ce tableau que sont placés les quartets du code-objet, au fur et à mesure de l'assemblage. Comme il faut deux quartets pour faire un octet du programme-objet final, on se rend compte que **ASMPC** permet de créer un objet-code d'au plus 25000 octets.

- Le contenu du tableau **tab** est converti à la fin de l'assemblage, et est seulement alors écrit dans le fichier-objet. Il n'y a aucune écriture dans le fichier-objet si une erreur est apparue au moment de l'assemblage.
  Une erreur éventuelle n'interromp pas le processus. Les différents messages d'erreur apparaissent à l'écran (avec une pause tous les 10 messages, si vous voulez faire une copie d'écran par exemple).

- **Le menu initial:**
  Le programme **ASMPC** s'ouvre sur un menu initial qui rappelle celui du programme **EXTERNPC**. Il propose les options suivantes:
  0) quitter le programme **ASMPC**.
  1) éditer un fichier-source.
  2) assembler un fichier-source en un fichier-objet.
  3) transmettre un fichier-objet à la HP48.
  4) lancer le programme **KERMIT**.
  5) lancer votre éditeur de textes ASCII.
  6) lister sur disque le résultat de l'assemblage.

- **Listing de l'assemblage:**
  L'option 6 (qui fonctionne en bascule) permet de produire le listing de l'assemblage du fichier source. Le fichier utilisé pour ce listing porte le même nom que le fichier-objet, mais avec l'extension **LST**.
  Ce listing fait apparaître les adresses relatives dans votre fichier-objet (en commençant à l'adresse 0), les codes machines résultant de l'assemblage, et les mnémoniques utilisés dans le fichier-source initial.
  L'intérêt de ce listing est qu'il peut vous "*rassurer*" sur la façon dont les choses se sont passées, et que vous pouvez tout à loisir examiner les erreurs qui ont pu se produire lors de l'assemblage.
  Ce listing possède un défaut, dû au fait qu'il est produit au fur et à mesure de l'assemblage: il ne sait pas compléter les instructions de sauts lorsqu'elles s'effectuent à destination d'un label qui n'a pas encore été déclaré. Les quartets sur lesquels sont codés ces sauts sont remplacés par des caractères ? dans le listing (évidemment il n'en va pas de même dans le fichier-objet produit par ASMPC, et qui n'est écrit que lorsque l'assemblage est terminé).
  Peut-être un lecteur motivé pourra-t-il corriger ce défaut (sans doute faudrait-il traiter les lignes du fichier-listing comme des variables dynamiques...).

- **Extensions aux noms de fichiers:**
  Le programmes **ASMPC** attribue par défaut les extensions suivantes à vos fichiers.
  Fichier-source: extension **ASM**.
  Fichier-objet: extension **COD**.
  Fichier-listing: extension **LST**.
  Dans les deux premiers cas, vous pouvez bien entendu indiquer d'autres extensions. Notez également que le programme **ASMPC** retient le nom du dernier fichier évoqué, ce qui accélère les mises au point répétées d'un même programme.

- **Majuscules et minuscules:**
  Le programme **ASMPC** exige que les mnémoniques soient écrites en majuscules. Il est néanmoins plus tolérant en ce qui concerne les noms de variables ou de labels. Dans ce cas en effet, il ne fait pas la différence entre majuscules et minuscules (tant lors de la déclaration de ces variables ou de ces labels que lors de leur utilisation).

- Comme le programme **ASM48**, le programme **ASMPC** ne reconnait qu'un seul mnémonique pour une instruction donnée (le problème pouvant se poser pour des instructions synonymes du type, par exemple, **A = A + B** et **A = B + A**). On se reportera aux constantes **ST3**, **ST4**, **ST56**, **ST5**, **ST6**, et **ST7** (dans le listing de **ASMPC**) pour vérifier quels sont les mnémoniques reconnus.
  Dans les cas possibles d'homonymie, le programme **ASMPC** choisit toujours d'ordonner les registres de travail suivant les couples **(A,B)**, **(B,C)**, **(C,A)** et **(D,C)**.

- Les listings des deux programmes **EXTERNPC** et **ASMPC** utilisent en commun un certain nombre de procédures et de fonctions. Celles-ci sont groupées en tête du programme **ASMPC** (depuis la fonction "***Valeur***" jusqu'à la procédure "***Edition***").
  Les principes de la bonne programmation commanderaient de faire une "***unité***" de tous ces outils communs. Je n'ai pas cru utile de le faire ici pour donner aux deux programmes une pleine autonomie (mais ceux-ci ne demandent qu'à être améliorés par vos soins).

- Pour ce qui est du "***scanning***" du fichier-source et de la traduction des instructions lues en les codes-machine qui leur correspondent, il y a beaucoup de points communs entre les programmes **ASM48** et **ASMPC**. J'ai d'ailleurs utilisé les mêmes noms (**ASS3**, **FIELD**, etc...) pour désigner des procédures qui ont la même signification dans les deux programmes.
  Disons simplement que les choses se passent de la façon suivante:
  La procédure "***Assemblage***" vous demande de choisir le fichier-source et le fichier-objet, en même temps qu'elle procède à quelques initialisations.
  Elle appelle la procédure "***AssFile***" qui lit le fichier-source ligne par ligne. Chaque ligne est traitée par la procédure "***Assembler_ligne***", dont le rôle est d'isoler le premier et le second champ (baptisés ***str1*** et ***str2***) de la ligne.
  La procédure "***assembler***" (qui attend les paramètres ***str1*** et ***str2***) effectue le *dispatching* suivant le premier caractère de ***str1***, ou sinon suivant sa longueur.
  Les codes-machine qui résultent de la traduction par la procédure appelée sont placées dans la variable "***result***" et ajoutés au tableau "***tab***" (procédure "***ajouter***"), et ainsi de suite...

- Pour terminer, sachez que sur mon PC (386SX, 16 Mhz, pas très rapide donc) l'assemblage d'un fichier de **600 lignes** (tous les mnémoniques possibles étant représentés) s'effectue en **2 secondes** (tout compris: lecture du source, assemblage, écriture de l'objet).

Il est grand temps maintenant de prendre connaissance du programme **ASMPC**. Je ne prétends pas qu'il soit le meilleur possible. Les programmeurs avertis y verront sans doute ça et là des maladresses. J'assume. Tout ce que je peux dire est que j'ai écrit ce programme pour mon usage personnel, et que, dans son actuelle rusticité, il me convient parfaitement.

```
ASMPC (Listing)
(******************* DÉBUT DU LISTING DE ASMPC ******************)
Program ASMPC;

{$N-}
{$M $6000,0,$4000}
Uses Crt, Dos;

Const
      Version = 'M'; (* M = version E de la HP48 *)
      ST3 : Array[0..2] of string[48] =
            ('RSI RTN RTI P== ?P# ?P= P=C C=P HEX ASC NUL ',
             'A=0 B=0 C=0 D=0 A=B B=C C=A D=C B=A C=B A=C ',
             'C=D ASL BSL CSL DSL ASR BSR CSR DSR ');
      COD3: Array[0..7] of string[5] =
            ('80810','01','0F','2','88','89','80D','80C');
      ST4 : Array[0..8] of string[55] =
         ('REL2 REL3 REL4 REL5 ?A=B ?B=C ?C=A ?D=C ?A#B ?B#C ?C#A ',
          '?D#C ?A=0 ?B=0 ?C=0 ?D=0 ?A#0 ?B#0 ?C#0 ?D#0 ?A>B ?B>C ',
          '?C>A ?D>C ?A<B ?B<C ?C<A ?D<C ABEX BCEX CAEX DCEX A=-A ',
          'B=-B C=-C D=-D CPEX HS=0 ST=0 ST=1 R0=A R1=A R2=A R3=A ',
          'R4=A R0=C R1=C R2=C R3=C R4=C A=R0 A=R1 A=R2 A=R3 A=R4 ',
          'C=R0 C=R1 C=R2 C=R3 C=R4 D0=A D1=A D0=C D1=C A=IN C=IN ',
          'C=ID ASLC BSLC CSLC DSLC ASRC BSRC CSRC DSRC ASRB BSRB ',
          'CSRB DSRB XM=0 SB=0 SR=0 MP=0 PC=A PC=C A=PC C=PC C=ST ',
          'ST=C RTNC NOP3 NOP4 NOP5 ');
      ST4bis = '000102030408090A0B0C101112131418191A1B1C30313435';
      ST4ter = '020306101112131415161 71C1D1E1F21222428';
      COD4: Array[0..9] of string[5] =
        ('80F','82','84','85','09','0A','400','420','6300','64000');
      ST5 : Array[0..11] of string[54] =
          ('A=A+B B=B+C C=C+A D=D+C A=A+A B=B+B C=C+C D=D+D B=A+B ',
           'C=B+C A=C+A C=D+C A=A-1 B=B-1 C=C-1 D=D-1 A=A-B B=B-C ',
           'C=C-A D=D-C A=A+1 B=B+1 C=C+1 D=D+1 B=B-A C=C-B A=A-C ',
           'C=C-D A=B-A B=C-B C=A-C D=C-D A=A&B B=B&C C=C&A D=D&C ',
           'B=A&B C=B&C A=C&A C=D&C A=A!B B=B!C C=C!A D=D!C B=A!B ',
           'C=B!C A=C!A C=D!C ?A>=B ?B>=C ?C>=A ?D>=C ?A<=B ?B<=C ',
           '?C<=A ?D<=C ?HS=0 ?ST=0 ?ST=1 RTNSC RTNCC CLRST CSTEX ',
           'P=P+1 P=P-1 AR0EX AR1EX AR2EX AR3EX AR4EX CR0EX CR1EX ',
           'CR2EX CR3EX CR4EX AD0EX AD1EX CD0EX CD1EX D0=AS D1=AS ',
           'AD0XS AD1XS D0=CS D1=CS CD0XS CD1XS OUT=C C+P+1 RESET ',
           'BUSCC SREQ? ?XM=0 ?SB=0 ?SR=0 ?MP=0 INTON BUSCB BUSCD ',
           'APCEX CPCEX RTNNC ');
      ST5bis: String[45]=
           '367238BCD0123489ABC236789ABCDEF19ABE124803D67';
```

```
      ST6 : Array[0..7] of string[56] =
         ('D0=(2) D0=(4) D0=(5) D1=(2) D1=(4) D1=(5) A=-A-1 B=-B-1 ',
          'C=-C-1 D=-D-1 ASRB.F BSRB.F CSRB.F DSRB.F A=R0.F A=R1.F ',
          'A=R2.F A=R3.F A=R4.F C=R0.F C=R1.F C=R2.F C=R3.F C=R4.F ',
          'R0=A.F R1=A.F R2=A.F R3=A.F R4=A.F R0=C.F R1=C.F R2=C.F ',
          'R3=C.F R4=C.F DAT0=A DAT1=A A=DAT0 A=DAT1 DAT0=C DAT1=C ',
          'C=DAT0 C=DAT1 D0=D0+ D1=D1+ D0=D0- D1=D1- ABIT=0 ABIT=1 ',
          'CBIT=0 CBIT=1 RTNSXM SETHEX SETDEC RSTK=C C=RSTK OUT=CS ',
          'UNCNFG CONFIG SHUTDN PC=(A) PC=(C) INTOFF CLRHST ');
      ST6bis: String[26] = '9ABDEF678C4589045670457CEF';
      ST56 : String[40] = 'A=A+ B=B+ C=C+ D=D+ A=A- B=B- C=C- D=D- ';
      ST7 : Array[0..1] of string[56] =
         ('?ABIT=0 ?ABIT=1 ?CBIT=0 ?CBIT=1 AR0EX.F AR1EX.F AR2EX.F ',
          'AR3EX.F AR4EX.F CR0EX.F CR1EX.F CR2EX.F CR3EX.F CR4EX.F ');
      LSTR=  'LAREL LCREL LAABS LCABS ';
      CHAMP: Array[0..8] of string[2]
             = ('A','P','WP','XS','X','S','M','B','W');
      JP1: Array[0..9] of string[6]
             = ('GOC','GONC','GOTO','GOLONG',
                'GOSUB','GOSUBL','GOVLNG','GOSBVL','GOINA','GOINC');
      JP2: Array[0..9] of string[5]
             = ('4','5','6','8C','7','8E','8D','8F','80824','34');

Type string6 = string[6]; string3 = string[3];
     variable = ^data;
     data = record
              name: string[20];
              typ: byte;
              cont, adr1, adr2: longint;
              next: variable;
            end;

Var ok,test,error,listing, LineNul: boolean;
    source,sortie: text; objet: file;
    numline,count,numerror: word;
    line, linebis: string; choix: char;
    filename, oldsrc, oldobj: string[64]; (* noms de fichiers *)
    pathK: string[64]; (* Chemin pour trouver KERMIT *)
    pathTT: string[64]; (* Chemin pour trouver un éditeur *)
    str1, str2, result: string;
    first: variable; pdebut: pointer;
    tab: array[0..50000] of char;
    buffer: array[0..127] of byte;
```

```
Function Valeur(var k:char):byte;
  (* Transforme un caractère de '0' à 'F', en un entier de 0 à 15 *)
  begin
    case k of
      '0'..'9': valeur:=ord(k)-48;
      'A'..'F': valeur:=ord(k)-55
    end;
  end;

Function Car(k:byte): char;
  (* Transforme un entier de 0 à 15, en un caractère de '0' à 'F' *)
  begin
    case k of
      0..9: car:=chr(48+k);
      10..15: car:=chr(55+k);
    end;
  end;

Function Adr(n:longint):string;
  (* transforme un entier long en une adresse,
       sous la forme d'une chaîne de 5 caractères *)
  var st:string; i:integer;
  begin
    st:='00000'; if n<0 then n:=n+1+MaxLongInt;
    for i:=1 to 5 do
    begin st[i]:=car(n mod 16); n:=n div 16; end;
    adr:=st;
  end;

Procedure Pause(mess:string);
  (* Emet un bip, puis attend un appui sur ENTRÉE, pour continuer *)
  begin
     write(^G+mess+' (ENTRÉE pour continuer)'); readln; clrscr
  end;

Procedure Choix_Fichier_Source(suff: string3);
  (* Choix d'un fichier-source, pour lancer son édition dans le
     traitement de texte, ou pour le compiler en un fichier-objet *)
  begin
    ok:=true; if oldsrc<>'' then write('('+oldsrc+') ');
    readln(filename);
    if filename<>'' then  oldsrc:=filename
                    else  ok:=(oldsrc<>'');
    if pos('.',oldsrc)=0 then oldsrc:=oldsrc+'.'+suff;
  end;
```

```
Procedure Ouvrir_Fichier_Source(suff: string3);
  (* Ouverture du fichier source, contenant les instructions sous
       forme de mnémoniques en assembleur *)
  begin
     repeat
       write('Nom du fichier source: '); Choix_fichier_source(suff);
       if ok then begin
         if pos('.',oldsrc)=0 then oldsrc:=oldsrc+'.'+suff;
         assign(source,oldsrc)
         {$I-};reset(source);{$I+};
         ok:=(IoResult=0);
       end;
       if not ok then  begin  write(^G); oldsrc:='' end
     until ok;
  end;

Procedure Choix_Fichier_Objet(suff: string3);
  (* Choix d'un fichier-objet, pour recevoir la traduction
       d'un fichier-source, ou pour transmission à la HP48 *)
  begin
    ok:=true;
    if oldsrc<>'' then  oldobj:=copy(oldsrc,1,pos('.',oldsrc))+suff;
    if oldobj<>'' then write('('+oldobj+') ');
    readln(filename);
    if filename<>'' then  oldobj:=filename
                    else  ok:=(oldobj<>'');
    if pos('.',oldobj)=0 then oldobj:=oldobj+'.'+suff;
  end;

Procedure Ouvrir_Fichier_Objet(suff:string3);
  (* Ouverture du fichier-objet pour recevoir la
            traduction d'un fichier-source *)
  begin
     repeat
       write('Nom du fichier objet: '); Choix_fichier_objet(suff);
       if ok then
       begin
         assign(objet,oldobj);
         {$I-};rewrite(objet,1);{$I+};
         ok:=(IoResult=0);
       end;
       if not ok then begin write(^G); oldobj:='' end
     until ok;
  end;
```

```
Procedure Transmission(suff: string3);
  (* Transmet un fichier-objet à la HP48, via Kermit *)
  begin
    repeat
      Write('Nom du fichier à transmettre: ');
      Choix_Fichier_Objet(suff);
      if not ok then begin write(^G); oldobj:='' end
    until ok;
    Exec(PathK,'S '+oldobj); ClrScr;
    if (DosError+DosExitCode=0) then pause('Transmission terminée')
                               else pause('Transmission non effectuée');
  end;

Procedure Edition(suff: string3);
  (* Lance la saisie d'un fichier-source, sur l'éditeur *)
  begin
    repeat
      Write('Nom du fichier source: '); Choix_Fichier_Source(suff);
      if not ok then begin write(^G); oldsrc:='' end
    until ok;
    Exec(PathTT,oldsrc); ClrScr;
    if (DosError+DosExitCode=0) then pause('Edition terminée')
                                else pause('Edition non effectuée');
  end;

Procedure EcrireLn(st: string);
  begin
    writeln(st);
    if listing then writeln(sortie,st);
  end;

Procedure Init;
  (* Initialise le nom, avec le chemin d'accès, du fichier KERMIT
       et de votre éditeur ASCII préféré. Vous remplacerez bien
       sûr les déclarations ci-dessous, qui correspondent à mon
       installation, par les vôtres... *)
  begin
    ClrScr; oldsrc:=''; oldobj:=''; listing:=false;
    new(first); mark(pdebut);
    PathK:= '\HP\KERMIT\KERMIT.EXE';  (*Chemin Kermit *)
    PathTT:= '\TP6\TURBO.EXE';  (*Chemin Editeur Ascii*)
  end;
```

```
Procedure Erreur(n:integer);
  (* Une erreur a été rencontrée, dans la lecture du fichier-source *)
  var mess:string;
  begin
    case n of
      1: mess:='Instruction inconnue, en ligne ';
      2: mess:='Erreur de syntaxe, en ligne ';
      3: mess:='Nom déjà utilisé, en ligne ';
      4: mess:='Argument incorrect ou absent, en ligne ';
      5: mess:='Saut trop long, en ligne ';
      6: mess:='Champ incorrect ou absent, en ligne ';
      7: mess:='Nom incorrect, en ligne ';
      8: mess:='Link incorrect, en ligne ';
      9: mess:='RTN/GOYES attendu, en ligne ';
      10: mess:='Argument trop long, en ligne ';
    end;
    error:=true; inc(numerror);
    Ecrireln('==> '+linebis);
    Writeln('    ',mess,numline);
    if listing then writeln(sortie,'     ',mess,numline);
    if (numerror mod 10 = 0) and (not listing) then pause('');
  end;

Procedure CloreAssemblage;
  (* Fin de l'assemblage du fichier-source en le fichier-objet *)
  begin
    close(source); close(objet);
    if listing then close(sortie);
    if error then pause('Assemblage interrompu')
             else pause('Assemblage terminé');
  end;

Procedure Ouvrir_Fichier_listing;
  var k:integer;
  begin
    k:=pos('.',oldobj); filename:=copy(oldobj,1,k)+'LST';
    assign(sortie,filename);
    {$I-}; rewrite(sortie); {$I+}; Ok:=(IoResult=0);
    if not ok then begin
       Pause('Listing impossible dans '+filename);
       listing:=false;
    end;
  end;
```

```
Function Item(var line:string):string;
  (* Cette fonction isole un nouvel identificateur du fichier source.
     Elle modifie le contenu de la variable "line", qui représente
     la ligne en cours de lecture dans ce fichier source *)
  var st:string;
  begin
    st:='';
    while (line<>'') and (line[1] in [#9,#32]) do delete(line,1,1);
    if line<>'' then
      if line[1]=#34 then
         begin
          repeat
            st:=st+line[1]; delete(line,1,1);
          until (line='') or (line[1]=#34);
          delete(line,1,1); st:=st+#34;
        end
        else repeat
          st:=st+line[1]; delete(line,1,1);
        until (line='') or (line[1] in [#9,#32]);
    item:=st;
  end;

Function duplic(var st:string; var ptr: variable): boolean;
  begin
    ptr:=first; duplic:=false;
    while ptr^.next <> nil do begin
      ptr:=ptr^.next;
      if ptr^.name=st then begin duplic:=true; exit; end;
    end;
  end;

Function OkName(var st:string): boolean;
  begin
    OkName:= (st<>'') and (upcase(st[1]) in ['A'..'Z']);
  end;

Procedure Majuscule(var st:string);
  var i:integer;
  begin
    for i:=1 to length(st) do st[i]:=upcase(st[i]);
  end;
```

```
Function Offset(n,a1,a2: longint): string;
  var ptr: variable; max,ofs,dest: longint;
  begin
    majuscule(str2);
    if duplic(str2,ptr) and (ptr^.typ <> 0) then
      if ptr^.typ=2 then erreur(3) else begin
        dest:=ptr^.cont; ofs:=dest-a2;
        max:= 1 shl (4*n-1);
        if abs(ofs) >= max then erreur(5)
                          else offset:=copy(adr(ofs),1,n);
      end
    else
      if not OkName(str2) then erreur(7)
      else begin new(ptr);
        with ptr^ do  begin
          name:=str2; typ:=0;
          cont:=n; adr1:=a1; adr2:=a2;
          next:=first^.next;
        end;
        first^.next:=ptr;
        offset:=copy('?????',1,n);
      end;
  end;

Procedure IfTest;
  begin
    test:=false;
    if str1='GOYES' then result:=offset(2,count,count)
    else if str1='RTNYES' then result:='00'
                          else erreur(9);
  end;

Function Argument: longint;
  var ptr: variable; n:longint; err:integer;
  begin
    majuscule(str2);
    case duplic(str2,ptr) of
      true:  if ptr^.typ<2 then erreur(4)
                           else argument:=ptr^.cont;
      false: begin
               if (str2<>'') and (str2[1]='#') then str2[1]:='$';
               val(str2,n,err);
               if err>0 then erreur(2) else argument:=n;
             end;
    end;
  end;
```

```
Procedure AjouterBack(ad,n:longint; st:string6);
  var i:longint;
  begin
    for i:=0 to n-1 do tab[ad+i]:=st[i+1];
  end;

Procedure ReplEtiq(var ptr:variable);
  var max,offs:longint; ptr2:variable;
  begin
    while ptr^.next <> nil do begin
      ptr2:=ptr^.next;
      if ptr2^.name=str1 then with ptr2^  do
      begin max := 1 shl (4*cont-1);
        offs:=count-adr2;
        if offs>=max then erreur(5) else
                    AjouterBack(adr1,cont,adr(offs));
        ptr^.next:=next; dispose(ptr2);
      end
      else ptr:=ptr2;
    end;
  end;

Function OkEtiq(var ptr: variable): boolean;
  begin
    ptr:=first;
    while ptr^.next <> nil do begin ptr:=ptr^.next;
      if ptr^.typ=0 then begin OkEtiq:=false; exit; end
    end; OkEtiq:=true;
  end;

Procedure Etiq;
  var ptr: variable;
  begin
    str2:=str1; delete(str1,1,1); majuscule(str1);
    if not OkName(str1) then erreur(7) else begin
      if duplic(str1,ptr) and (ptr^.typ>0) then erreur(3)
      else begin
        new(ptr); with ptr^ do begin
          name:=str1; typ:=1; cont:=count; next:=first^.next;
        end;
        first^.next:=ptr; ReplEtiq(ptr);
      end;
    end;
    str1:=item(line);
    if (str1='') and listing then  writeln(sortie,str2);
  end;
```

```
Procedure LLink;
  var ObjFile:file; k,err:integer; n: longint;
      st:string[6]; b:byte; flag:boolean;
  begin
    delete(str1,1,1); assign(ObjFile,str1);
    {$I-};reset(ObjFile,1);{$I+};
    if IoResult<>0 then begin erreur(8); exit; end;
    BlockRead(ObjFile,buffer,8,k);
    If k<8 then begin erreur(8); exit; end;
    st:='';
    for k:=0 to 6 do st:=st+chr(buffer[k]);
    if st<>'HPHP48' then begin erreur(8); exit; end;
    val(str2,n,err);
    if err>0 then begin erreur(4); exit; end;
    {$I-};seek(ObjFile,8+(n div 2));{$I+}
    if IoResult<>0 then begin erreur(8); exit; end;
    flag := not odd(n);
    While not eof(ObjFile) do
    begin
        BlockRead(ObjFile,b,1);
        if flag then begin tab[count]:=car(b mod 16); inc(count); end;
        flag:=true;
        tab[count]:=car(b div 16); inc(count);
    end;
    result:=''; close(ObjFile);
  end;

Procedure AssFile(name: string; var source: text); forward;

Procedure Link;
  var source2: text;
  begin
    delete(str1,1,1);
    if (str1<>'') and (str1[1]='[') then LLink else
    begin
      if listing then writeln(sortie,'['+str1);
      if pos('.',str1)=0 then str1:=str1+'.'+'asm';
      assign(source2,str1);
      {$I-};reset(source2);{$I+};
      error:= (IoResult<>0);
      if error then erreur(8)
         else begin AssFile(str1,source2); close(source2); end;
    end;
  end;
```

```
Procedure Equate;
  var ptr: variable; n: longint; err:integer;
  begin
    delete(str1,1,1); majuscule(str1);
    if not OkName(str1) then begin erreur(7); exit end;
    if duplic(str1,ptr) then begin erreur(3); exit end;
    if (str2<>'') and (str2[1]='#') then str2[1]:='$';
    val(str2,n,err);
    if err<>0 then erreur(2) else begin
      new(ptr);
      with ptr^ do begin
        name:=str1; typ:=2; cont:= n; next:=first^.next;
      end;
      first^.next:=ptr;
    end;
  end;

Procedure Saut;
  var p:integer; ad:longint;
  begin
    p:=0;
    while (p<8) and (str1<>JP1[p]) do inc(p);
    if p=8 then erreur(1) else begin result:=JP2[p];
      if p<8 then ad:=count+length(result);
      if p=0 then p:=1;
      case p of
        1..3: result:=result+offset(p+1,ad,ad);
        4..5: result:=result+offset(p-1,ad,ad+p-1);
        6..7: result:=result+adr(argument);
      end;
    end;
  end;

Function Nibble(var st: string; flag:boolean): char;
  var n,err: integer;
  begin
    if (st<>'') and (st[1]='#') then st[1]:='$';
    val(st,n,err);
    if err<>0 then erreur(2) else begin
      if flag then
        begin  flag:=(n in [1..16]); n:=n-1;  end
        else flag:=(n in [0..15]);
      if not flag then erreur(4) else nibble:=car(n);
    end;
  end;
```

```
Procedure ConvertAscii(var st:string);
  var st2:string; i,n:integer;
  begin
    n:=length(st); if n>127 then begin erreur(10); exit; end;
    st2[0]:=chr(2*n);
    for i:=1 to n do begin
      n:=ord(st[i]); st2[2*i-1]:=car(n mod 16);
      st2[2*i]:=car(n div 16);
    end; st:=st2;
  end;

Function OkHexa(var st:string): boolean;
  var i:integer; test:boolean;
  begin
    if (st<>'') and (st[1] in ['#','$']) then delete(st,1,1);
    test:=(st<>''); for i:=1 to length(st) do
      test := test and (st[i] in ['0'..'9','A'..'F']);
    OkHexa:=test;
  end;

Procedure LoadCst;
  var st:string[3]; i,n,k,err:integer;
  begin
    st:=copy(str1,1,3); if st='LA(' then result:='8082'
       else if st='LC(' then result:='3'
        else begin erreur(1); exit; end;
    n:=length(str1); if (n<5) or (str1[n]<>')')
    then erreur(1) else begin
      str1:=copy(str1,4,n-4);
      result:=result+nibble(str1,true);
      val(str1,k,err); if str2[1]=#34 then
      begin
        n:=length(str2); n:=n-2; if n>8 then n:=8;
        str2:=copy(str2,2,n);
        for i:=n to 7 do str2:=str2+#32;
        ConvertAscii(str2); result:=result+copy(str2,1,k);
      end
      else if OkHexa(str2) then begin
        n:=length(str2);
        for i:=n to 15 do str2:='0'+str2;
        for i:=1 to k do result:=result+str2[17-i];
      end
      else erreur(2);
    end;
  end;
```

```
Procedure LoadAC;
var p:integer;
begin
  p:=pos(str1,LSTR);
  case p of
   1: result:='80824'+offset(5,count+5,count);
   7: result:='34'+offset(5,count+2,count);
   13: result:='200681B580824'+offset(5,count+13,count+8)+'CA07';
   19: result:='20DE0681B580824'+offset(5,count+15,count+10)+'CA07DE';
   else LoadCst;
  end;
end;

Procedure field(stA, stB, st : string6);
  var i,j:integer;
  begin
    i:=0; while (i<9) and (str2<>champ[i]) do inc(i);
    if (i=7) and (stB<>'') then result:=stB else
    case i of
      0: result:=stA;   9: erreur(6);
      else begin
        j:=pos('b',st); if j=0 then j:=pos('a',st) else i:=i+8;
        st[j]:=car(i-1); result:=st;
      end;
    end;
  end;

Procedure div3(p:integer);
  var i:integer; n:longint;
  begin
    case p of
      8: if OkHexa(str2) then result:=str2 else erreur(2);
      9: begin
           if str2[1]=#34 then str2:=copy(str2,2,length(str2)-2);
           ConvertAscii(str2); result:=str2;
         end;
      10: begin
            result:=''; n:=argument; if n>255 then erreur(2)
            else for i:=1 to n do result:=result+'0';
          end;
    end;
  end;
```

```
Procedure ass3;
  var i,p:integer; c:char;
  begin
    i:=0; p:=0; while (i<3) and (p=0) do begin
      p:=pos(str1,ST3[i]); inc(i);
    end;
    if p=0 then erreur(1) else begin  p:=(p div 4)+11*(i-1);
      case p of
        0..2: result:=cod3[p];
        3..7: result:=cod3[p]+nibble(str2,false);
        8..10: div3(p);
        11..22: begin c:=car(p-11); field('D'+c,'','Ab'+c); end;
        23..30: begin c:=car(p-23); field('F'+c,'','Bb'+c); end;
      end;
    end;
  end;


Procedure ass4;
  var p,i: integer; c:char;
  begin
    i:=0; p:=0;
    while (i<9) and (p=0) do
    begin
      p:=pos(str1,ST4[i]); inc(i);
    end;
    if p=0 then erreur(1) else
    begin
      p:=(p div 5)+11*(i-1);
      case p of
        0..3  : result:=offset(p+2,count,count);
        4..19 : begin c:=car(p-4); field('8A'+c,'','9a'+c); end;
        20..27: begin c:=car(p-20); field('8B'+c,'','9b'+c); end;
        28..31: begin c:=car(p-16); field('D'+c,'','Ab'+c); end;
        32..35: begin c:=car(p-24); field('F'+c,'','Bb'+c); end;
        36..39: result:=cod4[p-36]+nibble(str2,false);
        40..63: result:='1'+copy(ST4bis,2*p-79,2);
        64..82: result:='8'+copy(ST4ter,2*p-127,2);
        83..86: result:='81B'+car(p-81);
        87..92: result:=cod4[p-83];
      end;
    end;
  end;
```

```
Procedure ass56;
  var p: integer; c,d: char;
  begin
    p:=pos(copy(str1,1,4),ST56);
    if p=0 then erreur(1) else
    begin
      p:=(p div 5); c:=car((p mod 4)+8*(p div 4));
      str1:=copy(str1,5,255); d:=nibble(str1,true);
      field('818F'+c+d,'','818a'+c+d);
    end;
  end;

Procedure ass5;
  var p,i: integer; c:char;
  begin
    i:=0; p:=0;
    while (i<12) and (p=0) do begin
      p:=pos(str1,ST5[i]); inc(i);
    end;
    if p=0 then ass56 else begin
      p:=(p div 6)+9*(i-1);
      case p of
        0..15 : begin
                  c:=car(p); field('C'+c,'','Aa'+c);
                end;
        16..31: begin
                  c:=car(p-16); field('E'+c,'','Ba'+c);
                end;
        32..47: begin
                  c:=car(p-32); field('0EF'+c,'','0Ea'+c);
                end;
        48..55: begin
                  c:=car(p-40); field('8B'+c,'','9b'+c);
                end;
        56..58: result:= '8'+ST5bis[p-55]+nibble(str2,false);
        59..64: result:='0'+ST5bis[p-55];
        65..74: result:='12'+ST5bis[p-55];
        75..86: result:='13'+ST5bis[p-55];
        87..91: result:='80'+ST5bis[p-55];
        92..95: result:='83'+ST5bis[p-55];
        96..98: result:='808'+ST5bis[p-55];
        99..100: result:='81B'+ST5bis[p-55];
        101: result:='500';
      end;
    end;
  end;
```

```
Procedure LOADD(p:integer);
  var n: byte;
  begin
    case p of
      0,3: n:=2;  1,4: n:=4;  2,5: n:=5;
    end;
    result:='1'+ST6bis[p+1]+copy(adr(argument),1,n);
  end;

Procedure ass6;
  var p,i: integer; c,d:char;
  begin
    i:=0; p:=0;
    while (i<8) and (p=0) do
    begin p:=pos(str1,ST6[i]); inc(i); end;
    if p=0 then ass56 else begin
      p:=(p div 7)+8*(i-1);
      case p of
        0..5  : loadD(p);
        6..9  : begin  c:=car(p+6); field('F'+c,'','Bb'+c); end;
        10..13: begin  c:=car(p-10); field('819F'+c,'','819a'+c);
                end;
        14..23: begin
                  p:=p-14; c:=car(8*(p div 5)+(p mod 5));
                  field('81AF1'+c,'','81Aa1'+c);
                end;
        24..33: begin
                  p:=p-24; c:=car(8*(p div 5)+(p mod 5));
                  field('81AF0'+c,'','81Aa0'+c);
                end;
        34..41: begin
                  p:=p-34; i:=0; c:=car(p); d:=car(p+8);
                  while (i<9) and (str2<>champ[i]) do inc(i);
                  if i<9 then field('14'+c,'14'+d,'15'+c+'a')
                         else result:='15'+d+nibble(str2,true);
                end;
        42..45: result:='1'+ST6bis[p-35]+nibble(str2,true);
        46..49: result:='808'+ST6bis[p-35]+nibble(str2,false);
        50..54: result:='0'+ST6bis[p-35];
        55..58: result:='80'+ST6bis[p-35];
        59..61: result:='808'+ST6bis[p-35];
        62: result:='82F';
      end;
    end;
  end;
```

```
Procedure ass7;
  var p,i:integer; c:char;
  begin
    i:=0; p:=0;
    while (i<2) and (p=0) do begin
      p:=pos(str1,ST7[i]); inc(i);
    end;
    if p=0 then erreur(1) else begin
      p:=(p div 8)+7*(i-1);
      case p of
        0..3 : result:='808'+copy('67AB',p+1,1)+nibble(str2,false);
        4..13: begin
                 p:=p-4; c:=car(8*(p div 5)+(p mod 5));
                 field('81AF2'+c,'','81Aa2'+c);
               end;
      end;
    end;
  end;

Procedure Assembler_Ligne; forward;

Procedure assembler(var str1,str2: string);
  begin
    result:=''; LineNul:= (str1='') or (str1[1]='*');
    if not LineNul then
      if test then iftest else
      case str1[1] of
        '[' : link;
        '=' : equate;
        'G' : saut;
        'L' : loadac;
        else
        begin
          if str1[1]='?' then test:=true;
          case length(str1) of
             2,3: ass3;
             4:   ass4;
             5:   ass5;
             6:   ass6;
             7:   ass7;
             else erreur(1);
          end;
        end;
     end;
  end;
```

```
Procedure ajouter(var st:string);
  var i:integer;
  begin
    for i:=1 to length(st) do begin
       tab[count]:=st[i]; inc(count);
    end;
  end;

Procedure Sortir(st:string; n:integer);
  var i,k:integer;
  begin
    k:=length(st);
    if k>=n then st:=copy(st,1,n-3)+'...'
            else for i:=k to  n-1 do st:=st+' ';
    write(sortie,st);
  end;

Procedure Assembler_Ligne;
  (* Traitement d'une des lignes du fichier source *)
  var i,n:word; st:string[5];
  begin
    inc(numline); ok:=true; linebis:=line;
    str1:=item(line);
    if str1[1]=':' then etiq;
    str2:=item(line);
    assembler(str1,str2);
    if not error then begin
       ajouter(result);
       if listing and (not LineNul) then begin
          if result<>'' then
          begin
             st:='00000';
             n:=count-length(result)-$1A;
             for i:=5 downto 1 do
             begin st[i]:=car(n mod 16); n:=n div 16; end;
             write(sortie,st,': ');
             sortir(result,15);
          end;
          writeln(sortie,linebis);
       end;
    end;
  end;
```

```
Procedure EcrireTab;
  var i,j:longint;
  begin
    tab[count]:=chr(0); inc(count); j:=0;
    for i:=0 to (count div 2)-1 do
    begin
      buffer[j]:=valeur(tab[2*i])+16*valeur(tab[2*i+1]);
      inc(j); if j=128 then begin
        BlockWrite(objet,buffer,128); j:=0;
      end;
    end;
    if j>0 then BlockWrite(objet,buffer,j);
  end;

Procedure AssFile(name: string; var source: text);
  begin
    Ecrireln('Début de l''assemblage du fichier '+name);
    while not eof(source) do begin
      readln(source,line); assembler_ligne;
    end;
    Ecrireln('Fin de l''assemblage du fichier '+name);
    LineNul:=True; Result:='';
  end;

Procedure Assemblage(suff1,suff2: string3);
  var ptr: variable;
  begin
    numline:=0; count:=0; numerror:=0; test:=false;
    error:=false; first^.next:=nil; result:='HPHP48-'+version;
    ConvertAscii(result); ajouter(result);
    result:='CCD2000000'; ajouter(result);
    ouvrir_fichier_source(suff1); ouvrir_fichier_objet(suff2);
    if listing then ouvrir_fichier_listing;
    AssFile(oldsrc,source);
    if not OkEtiq(ptr) then begin write(^G);
       EcrireLn('Label '+ptr^.name+' appelé et non déclaré!');
       error:=true; inc(numerror);
    end;
    if not error then begin
      AjouterBack(21,5,adr(count-21)); EcrireTab;
    end;
    release(pdebut);
    write(numerror); if listing then write(sortie,numerror);
    Ecrireln(' erreur(s) détectée(s)');
    CloreAssemblage;
  end;
```

```
Function Menu:char;  (*** Menu Initial ***)
  var c:string[1];
  begin
     repeat
       clrscr; writeln('0: quitter le programme ASMPC');
       writeln('1: éditer un fichier-source');
       writeln('2: assembler un fichier-source en fichier-objet');
       writeln('3: transmettre un fichier-objet à la HP48');
       writeln('4: lancer le programme KERMIT');
       writeln('5: lancer votre éditeur de textes ASCII');
       write('6: '); if listing then write('ne pas ');
       writeln('lister sur disque le résultat de l''assemblage');
       writeln;
       write('Faites votre choix: '); readln(c);
     until (c<>'') and (c[1] in ['0'..'6']);
     menu:=c[1]; ClrScr;
  end;

begin
  init;
    repeat choix:=menu; case choix of
      '1': edition('asm');
      '2': assemblage('asm','cod');
      '3': transmission('cod');
      '4': Exec(PathK,'');
      '5': Exec(PathTT,'');
      '6': Listing:=Not(listing);
    end;
  until choix='0';
end.

(****************** FIN DU LISTING DE ASMPC ******************)
```

LA RAM-SYSTEME

La RAM-Système est la zone de la mémoire vive qu'utilise la HP48 pour contrôler le fonctionnement du système.

On y trouve des pointeurs (permettant de localiser tel ou tel objet important, comme par exemple l'objet-graphique PICT ou le début du répertoire HOME), des buffers (pour recueillir et accumuler des données avant que de les traiter, comme par exemple le buffer du clavier ou celui de la sortie série), et bien d'autres choses encore.

Ce chapitre contient la description de la totalité de cette RAM-Système, presque sans aucun point d'ombre.

C'est-dire l'importance des renseignements que le programmeur en langage machine va trouver ici.

◆ PLAN DE LA RAM SYSTEME

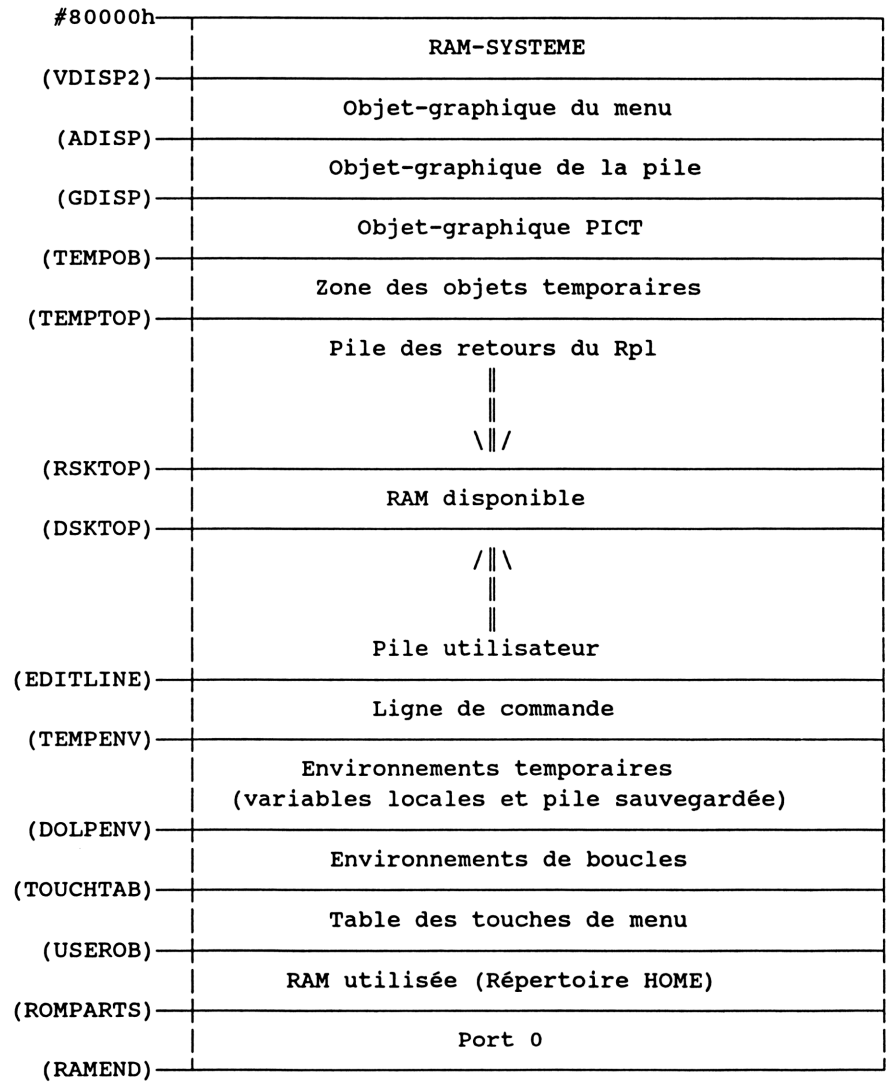
La RAM-Système s'étend de l'adresse #80000h à l'adresse #80CF6h (environ car pour cette dernière adresse celà dépend du nombre de librairies).

Voici en gros comment les choses sont organisées. Les explications sont bien sûr ici beaucoup trop parcellaires, mais c'est un bon moyen de retrouver rapidement l'endroit approximatif où se trouve une information précise.

#80000h	CMOS (test rapide de configuration)	5 quartets
#80005h	Configuration de la RAM	11 quartets
#80010h	Pile des Resets	72 quartets
#80058h	Extinction auto, TimeOut, Batteries AutoTest, StopWatch.	137 quartets
#800E1h	Statuts des ports, de la mémoire.	20 quartets
#800F5h	Boîte aux lettres pour les lectures en mémoire "couverte"	256 quartets
#8030Eh	Un pointeur...?	5 quartets
#80319h	Buffer sortie série, et paramètres.	522 quartets
#80523h	Données de configuration.	184 quartets
#805DBh	Sauvegardes pour les interruptions.	142 quartets
#80669h	Buffer du clavier.	36 quartets
#8068Dh	Sauvegardes de l'écran.	67 quartets
#806D0h	Pointeurs vers les GROBs.	25 quartets
#806E9h	Pointeurs vers mémoire (zones des objets-temporaires, de la pile, de la ligne de commande, etc...)	80 quartets
#80739h	Pile utilisée par LASTARG.	25 quartets
#80752h	Pointeurs sur 2 entiers binaires.	10 quartets
#8075Ch	Pile utilisée pour LAST CMD.	20 quartets
#80770h	Pointeurs divers.	145 quartets
#80801h	Flags internes au système.	20 quartets
#80815h	Données diverses.	44 quartets
#80841h	Flags indicateurs en ligne d'état.	2 quartets
#80843h	Flags-Système.	16 quartets
#80853h	Flags-Utilisateur.	16 quartets
#80863h	Données sur ligne de commande.	96 quartets
#808C3h	Données sur menus (actuel et last).	9 quartets
#808CCh	Sauvegardes diverses.	133 quartets
#80951h	Noms XLIBs affectés aux touches.	82 quartets
#809A3h	Table des numéros et adresses des librairies connues.	

◆ PLAN DE LA RAM

Voici un plan général de la RAM (de haut en bas, les adresses vont en augmentant).



Sur la gauche de ce tableau, et entre parenthèses, sont indiqués les pointeurs de la RAM-Système qui permettent de désigner tel ou tel point de la RAM.

La description de ces pointeurs apparaîtra dans les pages suivantes. Par exemple le pointeur **USEROB**, dont l'adresse est #80711h dans la RAM-Système, pointe sur le début du répertoire HOME.

◆ CONTENU DE LA RAM SYSTEME

Nous allons maintenant procéder à un état des lieux détaillé de la RAM-Système. Chaque "champ" est défini par un mnémonique, son adresse dans la RAM-Système, et sa taille. Les Mnémoniques donnés ici sont ceux-là même qu'utilisent les programmeurs de Hewlett-Packard.

C'est parti....

Nom: **CMOS**	Adresse: **#80000h**	Taille: 5 quartets

Le début de la RAM. On y trouve 5 quartets (toujours à F3C5A), pour tester rapidement la RAM-système. Si ces quartets sont modifiés, il y a un redémarrage à chaud (reset).

Nom: **IRAMMASK**	Adresse: **#80005h**	Taille: 5 quartets

Taille de la RAM intégrée, en complément à #100000h. Par exemple:
#C0000h = > RAM de 128K
#F0000h = > RAM de 32K

Nom: **HOMEMASK**	Adresse: **#8000Ah**	Taille: 5 quartets

Comme le précédent, apparemment. Peut-être ont-ils un rôle différent pendant les opérations de reconfiguration.

Nom: **HRAMEND**	Adresse: **#8000Fh**	Taille: 1 quartet

Un quartet indiquant la taille de la RAM, de la manière suivante:
#9 = > RAM de 32K
#C = > RAM de 128K

Nom: **FAILSTK1**	Adresse: **#80010h**	Taille: 18 quartets
Nom: **FAILSTK2**	Adresse: **#80022h**	Taille: 18 quartets
Nom: **FAILSTK3**	Adresse: **#80034h**	Taille: 18 quartets
Nom: **FAILSTK4**	Adresse: **#80046h**	Taille: 18 quartets

Ces quatre champs décrivent les 4 derniers redémarrages à chaud de la calculatrice, selon le format suivant:
<Cause (1 quartet)> <Date et Heure (13 quartets)> <CRC>
Le CRC (code de redondance cyclique) est calculé sur les 14 premiers quartets.
Le champ **FAILSTK1** décrit le redémarrage le plus récent.
Pour plus de détail sur la signification des "causes", voyez le manuel de référence du programmeur (référence 00048-90136).

Nom: **NEXTIRQ**	Adresse: **#80058h**	Taille: 13 quartets

Date et heure où la HP48 s'éteindra d'elle même si vous n'y touchez plus...

Nom: **TIMECRC**	Adresse: **#80065h**	Taille: 4 quartets

CRC du champ précédent.

Nom: **TIMEOUT**	Adresse: **#80069h**	Taille: 13 quartets

Date et heure de la prochaine erreur "Time Out" (utilisée dans les entrées/sorties).

Nom: **TIMEOUTCLK**	Adresse: **#80076h**	Taille: 1 quartet

Normalement à #Fh. Si zéro, c'est qu'un "TimeOut" vient d'être détecté.

Nom: **LoBatTime**	Adresse: **#80077h**	Taille: 1 quartet

Flag indiquant une situation "Low Bat(s)"

Nom: **StartTime**	Adresse: **#80078h**	Taille: 13 quartets

Date et heure de départ dans l'auto-test.

Nom: **FailTime**	Adresse: **#80085h**	Taille: 13 quartets

Date et heure d'échec dans l'auto-test

Nom: **TESTMSG**	Adresse: **#80092h**	Taille: 44 quartets

Buffer pour les messages durant l'auto-test.

Nom: **SW_Image**	Adresse: **#800BEh**	Taille: 22 quartets

Buffer pour l'application HP "StopWatch".

Nom: **SW_ETime**	Adresse: **#800D4h**	Taille: 13 quartets

Temps écoulé (en ticks) depuis le top dans "StopWatch".

Nom: **PortStat**	Adresse: **#800E1h**	Taille: 1 quartet

Copie du quartet lu en #0010Fh. Si on le note <b1t3>..<bit0>:
Le bit 0 est armé s'il y a une carte dans le slot 1 (et dans ce cas le bit 2 est armé si cette carte est ouverte en écriture).
Le bit 1 est armé s'il y a une carte dans le slot 2 (et dans ce cas le bit 3 est armé si cette carte est ouverte en écriture).

Nom: **Port1CRC**	Adresse: **#800E2h**	Taille: 4 quartets

Code de redondance cyclique pour la carte présente dans le slot 1.

Nom: **AccessInit**	Adresse: **#800E6h**	Taille: 2 quartets

Sauvegarde du quartet INITEN (en #8065Ah) et du quartet sALLOWINTR.

Nom: **COVERstate**	Adresse: **#800E8h**	Taille: 3 quartets

Etat de la RAM avant dé-couvrement d'une zone cachée.

Nom: **COVERsave**	Adresse: **#800EBh**	Taille: 10 quartets

Zone de sauvegarde pendant les opérations de dé-couverture.

Nom: **IRAMBUFF**	Adresse: **#800F5h**	Taille: 256 quartets

Buffer (Boîte aux lettres, MailBox) utilisée pour les transferts en une zone cachée de la mémoire et la RAM des objets temporaires.

Nom: **GraphPrtHook**	Adresse: **#8030Eh**	Taille: 11 quartets

Peut contenir un nom XLIB (voir routine en #32CB6), mais pourquoi?

Nom: **uart_buffer**	Adresse: **#80319h**	Taille: 512 quartets

Buffer circulaire de la liaison série.

Nom: **uart_buf_end**	Adresse: **#80519h**	Taille: 2 quartets

Nombre de caractères présents dans le buffer série (1 caractère = 2 quartets).

Nom: **uart_error**	Adresse: **#8051Bh**	Taille: 1 quartet

Quartet indiquant (si non nul) qu'un débordement a eu lieu dans le buffer série.

Nom: **uart_buf_st**	Adresse: **#8051Ch**	Taille: 2 quartets

Position du premier caractère (le plus ancien), exprimée en caractères (2 quartets), à partir du début physique du buffer (#80319h). (position 0 = adresse #80319h).

Nom: **uart_handshk**	Adresse: **#8051Eh**	Taille: 1 quartet

8 bits utilisés (mais comment?) pour paramétrer la liaison série.

Nom: **uart_modes**	Adresse: **#8051Fh**	Taille: 1 quartet

Idem....

Nom: **uart_parity**	Adresse: **#80520h**	Taille: 1 quartet

Contrôle la parité du nombre de quartets transmis.

Nom: **uart_timeout**	Adresse: **#80521h**	Taille: 2 quartets

Contient le nombre de secondes que peut attendre **SRECV** ou **XMIT** avant une erreur "Time Out". Voir la commande **STIME**.

Nom: **IOCNIB**	Adresse: **#80523h**	Taille: 1 quartet

Copie de l'octet lu en #110h.

Nom: **CONFRAM**	Adresse: **#80524h**	Taille: 7 quartets

Table de configuration des slots 1 et 2. <Slot1><Slot2><Bancs><ID>.
Les champs <Slot1> et <Slot2> font deux quartets. Le premier définit le statut et le second définit la taille.
Les quartets "statut" ont la forme:
bit3: ouvert en lecture? bit2: ouvert en écriture?
bit1: fusionné? bit0: inutilisé.
Les quartets "taille" signifient: 0:rien, 1:32K, 2: 128K
<Bancs> = nombre de bancs de 128K. <ID> indique une configuration (?)

Nom: **CONFTAB**	Adresse: **#8052Bh**	Taille: 11 quartets

Comme ci-dessus, précédés d'un CRC de 4 quartets.

Nom: **PORT0EOS**	Adresse: **#80536h**	Taille: 5 quartets

Contient l'adresse suivant immédiatement la fin du port 0.

Nom: **PORT1EOS**	Adresse: **#8053Bh**	Taille: 5 quartets
Nom: **PORT2EOS**	Adresse: **#80540h**	Taille: 5 quartets
.....		
Nom: **PORT33EOS**	Adresse: **#805DBh**	Taille: 5 quartets

Comme ci-dessus, mais pour les ports n°1 à n°33.

Voici maintenant la RAM qui est réservée à la sauvegarde du contexte par le gestionnaire d'interruptions.

Nom: **INTRAM**	Adresse: **#805DBh**	Taille: 16 quartets
Nom: **SAVE_MODES**	Adresse: **#805EBh**	Taille: 5 quartets
Nom: **SAVE_C[A]**	Adresse: **#805F0h**	Taille: 5 quartets
Nom: **SAVE_A**	Adresse: **#805F5h**	Taille: 16 quartets
Nom: **SAVE_ST**	Adresse: **#80605h**	Taille: 3 quartets
Nom: **SAVE_B**	Adresse: **#80608h**	Taille: 16 quartets
Nom: **SAVE_D**	Adresse: **#80618h**	Taille: 16 quartets
Nom: **SAVE_R0**	Adresse: **#80628h**	Taille: 16 quartets
Nom: **SAVE_PC**	Adresse: **#80638h**	Taille: 5 quartets
Nom: **SAVE_D0**	Adresse: **#8063Dh**	Taille: 5 quartets
Nom: **SAVE_OR**	Adresse: **#80642h**	Taille: 5 quartets

Voici le rôle attribué à ces registres:

INTRAM sauvegarde D1[4321], C[15,..,5]
SAVE_MODES sauvegarde P,Carry,SB
SAVE_C[A] sauvegarde C[A]
SAVE_A sauvegarde A[W]
SAVE_ST sauvegarde S0-S11
SAVE_B sauvegarde B
SAVE_D sauvegarde D
SAVE_R0 sauvegarde R0
SAVE_PC sauvegarde PC
SAVE_D0 sauvegarde D0
SAVE_OR sauvegarde "OR register"

Nom: **DRSTART**	Adresse: **#80645h**	Taille: 5 quartets

Contient l'adresse de début pour le rafraichissement d'écran (pointe sur le début de la bitmap des menus, ou sur le début de la bitmap de PICT).

Nom: **DREND**	Adresse: **#8064Ah**	Taille: 5 quartets

Contient l'adresse de fin pour le rafraichissement d'écran. (pointe sur le dernier quartet du Grob de la pile, ou sur le dernier quartet du Grob PICT).

Nom: **IREG**	Adresse: **#8064Fh**	Taille: 3 quartets

Sert à sauvegarder l'historique du gestionnaire d'interruptions.

Nom: **SEMAPH**	Adresse: **#80652h**	Taille: 2 quartets

Octets de contrôle pour le champ IREG précédent.

Nom: **IOSAVE**	Adresse: **#80654h**	Taille: 1 quartet

Sauvegarde du quartet haut de ANNCTRL.

Nom: **CSPEED**	Adresse: **#80655h**	Taille: 5 quartets

Vitesse du processeur (en unités de 16hz).

Nom: **INITEN**	Adresse: **#8065Ah**	Taille: 1 quartet

Si ce quartet est à zéro, alors les arrêts système par ON-C, ON-D, ON-E, ON-SPC sont impossibles. Ce quartet est fréquemment remis à une valeur non nulle par le système. A n'utiliser qu'en assembleur...

Nom: **HANDSHK**	Adresse: **#8065Bh**	Taille: 1 quartet

Si ce quartet n'est pas à zéro, les touches appuyées ne sont pas placées dans le buffer. Vous pouvez cependant continuer à traiter les appuis par OUT=C et C=IN.

Nom: **KEYSTATE**	Adresse: **#8065Ch**	Taille: 13 quartets

Table utilisée pour les appuis sur plusieurs touches simultanément.

Nom: **INPUTSTREAM**	Adresse: **#80669h**	Taille: 34 quartets

Buffer circulaire du clavier (15 touches maxi). Chaque touche est représentée par le plan (shifté ou non) puis par son code de position sur le clavier (au total deux quartets). Les deux premiers quartets sont réservés:

Le premier (en #80669h) indique la position de la touche la plus ancienne.

Le second (en #8066Ah) indique la première position libre.

Nom: **POPPEDKEY**	Adresse: **#8068Bh**	Taille: 2 quartets

Indique le code de la dernière touche retirée du buffer (utile pour tester les répétitions de touche).

Nom: **DISP1CTLg**	Adresse: **#8068Dh**	Taille: 5 quartets

Sauvegarde du registre DISP1CTL (en #00120h) c'est-à-dire du début de la bitmap de l'écran.

Nom: **LINENIBSg**	Adresse: **#80692h**	Taille: 3 quartets

Sauvegarde du registre LINENIBS (en #00125h) c'est-à-dire de la marge à droite.

Nom: **DISP2CTLg**	Adresse: **#80695h**	Taille: 5 quartets

Sauvegarde du registre DISP2CTL (en #00130h) c'est-à-dire du début de la bitmap des menus.

Nom: **LINECOUNTg**	Adresse: **#8069Ah**	Taille: 2 quartets

Sauvegarde de ce qu'on écrit dans le registre LINECOUNT (en #00128h). L'octet écrit dans ce registre est transmis à une mémoire interne qui est utilisée toutes les 15 milli-secondes pour initialiser un compteur; ce dernier est décrémenté après chaque rafraichissement d'un pixel à l'écran.

Une lecture dans LINECOUNT ne renvoie pas la valeur dans le registre interne (c-à-d ce qu'on avait mis dans LINECOUNT), mais la valeur courante du compteur. Celà permet aux programmes de "suivre" et d'utiliser le rafraichissemnt de l'écran.

Nom: **Stk0save**	Adresse: **#8069Ch**	Taille: 5 quartets
Nom: **Stk1save**	Adresse: **#806A1h**	Taille: 5 quartets
Nom: **Stk2save**	Adresse: **#806A6h**	Taille: 5 quartets
Nom: **Stk3save**	Adresse: **#806ABh**	Taille: 5 quartets
Nom: **Stk4save**	Adresse: **#806B0h**	Taille: 5 quartets
Nom: **Stk5save**	Adresse: **#806B5h**	Taille: 5 quartets

Ces registres servent à sauvegarder les cinq niveaux de pile interne, altérés par l'interruption provoquée par la touche ON-IO (impression de l'écran par infra-rouge).

Nom: **R2[A]save**	Adresse: **#806BAh**	Taille: 5 quartets
Nom: **R2[S]save**	Adresse: **#806BFh**	Taille: 1 quartet
Nom: **R1[A]save**	Adresse: **#806C0h**	Taille: 5 quartets

Comme précédemment, avec les registres R2 (champs A et S) et R1 (champ A).

Nom: **SAVE_BO**	Adresse: **#806C5h**	Taille: 1 quartet
Nom: **SAVE_LC**	Adresse: **#806C6h**	Taille: 2 quartets
Nom: **SAVE_LN**	Adresse: **#806C8h**	Taille: 3 quartets
Nom: **SAVE_OFFSET**	Adresse: **#806CBh**	Taille: 5 quartets

Ces quatre registres servent de sauvegarde lors de l'opération consistant à changer d'objet-graphique courant.

SAVE_BO sauvegarde le registre BitOffset (#00100h, bit3)
SAVE_LC sauvegarde le registre LineCount (#00128h)
SAVE_LN sauvegarde le registre LineNibs (#00125h)
SAVE_OFFSET sauvegarde le registre WindowOffset (#00100h, bit2,1,0)

Nom: **VDISP2**	Adresse: **#806D0h**	Taille: 5 quartets

Contient l'adresse du Grob des menus.

Nom: **ADISP**	Adresse: **#806D5h**	Taille: 5 quartets

Contient l'adresse du Grob de la pile.

Nom: **VDISP1**	Adresse: **#806DAh**	Taille: 5 quartets

Contient l'adresse du Grob en cours (stack ou Pict).

Nom: **VDISP3**	Adresse: **#806DFh**	Taille: 5 quartets

Contient l'adresse du Grob qui n'est pas affiché (contraire du précédent).

Nom: **GDISP**	Adresse: **#806E4h**	Taille: 5 quartets

Contient l'adresse du Grob PICT.

Nom: **TEMPOB**	Adresse: **#806E9h**	Taille: 5 quartets

Contient l'adresse du début de la mémoire des objets temporaires.

Nom: **TEMPTOP**	Adresse: **#806EEh**	Taille: 5 quartets

Contient l'adresse de la fin de la mémoire des objets temporaires. C'est aussi l'adresse du début de la pile des retours du System-Rpl.

Nom: **RSKTOP**	Adresse: **#806F3h**	Taille: 5 quartets

Contient l'adresse de la fin de la pile des retours (juste après les 5 quartets de la dernière adresse de retour empilée). c'est aussi le début de la mémoire vive. Cette adresse est en général contenue dans le registre **B** (la routine **SaveRegs**, en #679Bh, sauve **B** en #806F3, et la routine **RclRegs**, en #67D2h, y récupère le contenu de **B**).

Nom: **DSKTOP**	Adresse: **#806F8h**	Taille: 5 quartets

Contient l'adresse du dernier niveau empilé sur la pile utilisateur (à cette adresse on lit l'adresse de l'objet qui figure donc, pour l'utilisateur, au niveau 1).

L'adresse placée en #806F8h marque également la fin de la mémoire vive disponible. Cette adresse est en général placée dans le registre **D1** (les routines **SaveRegs** et **RclRegs** écrivent et lisent le contenu de **D1** en #806F8).

Nom: **EDITLINE**	Adresse: **#806FDh**	Taille: 5 quartets

Contient l'adresse où débute la ligne de commande.

Nom: **TEMPENV**	Adresse: **#80702h**	Taille: 5 quartets

Contient l'adresse qui suit immédiatement la zone de la ligne de commande. C'est également le début de la zone de l'environnement temporaire (variables locales et dernière pile sauvegardée et récupérable par UNDO).

Nom: **DOLPENV**	Adresse: **#80707h**	Taille: 5 quartets

Contient l'adresse de la zone où on trouve l'environnement temporaire relatif à une boucle (indice courant, valeur de stop, etc...). Utilisé aussi pour les environnements spécifiques à une séquence IFERROR...THEN....

Nom: **TOUCTAB**	Adresse: **#8070Ch**	Taille: 5 quartets

Contient l'adresse d'une table où apparaissent les adresses des objets relatifs aux différents touches de menu (labels et objets à évaluer).

Nom: **USEROB**	Adresse: **#80711h**	Taille: 5 quartets

Contient l'adresse où débute le répertoire HOME.

Nom: **ROMPARTS**	Adresse: **#80716h**	Taille: 5 quartets

Contient l'adresse où débute le port 0. C'est l'adresse où débute le dernier Backup Object (ou la dernière librairie) qu'on a stocké dans ce port.

Nom: **CONTEXT**	Adresse: **#8071Bh**	Taille: 5 quartets

Contient l'adresse où débute le répertoire en cours.

Nom: **STOPSIGN**	Adresse: **#80720h**	Taille: 5 quartets

Contient l'adresse où débute le répertoire "maître". En général il s'agit du répertoire **HOME**, mais l'instruction **isol(di)** (voir le chapitre consacré aux répertoires dans la deuxième partie du tome 1 des "Secrets") montre qu'il est possible d'isoler un répertoire (et toute sa descendance, pour laquelle il devient le répertoire "maître").

Nom: **UserKeys**	Adresse: **#80725h**	Taille: 5 quartets

Contient l'adresse de la liste des redéfinitions de touches, dans le répertoire caché (voir le chapitre consacré à ce répertoire, dans le tome 1 des "Secrets").

Nom: **ALARMS**	Adresse: **#8072Ah**	Taille: 5 quartets

Contient l'adresse de la liste des alarmes (au format interne, voir, dans le tome 1 des "Secrets", le chapitre consacré au répertoire caché).

Nom: **INTRPPTR**	Adresse: **#8072Fh**	Taille: 5 quartets

Contient l'adresse du prochain objet à évaluer, dans le cadre du fonctionnement habituel de la calculatrice. Cette adresse est en temps normal contenue dans le registre **D0**. D'ailleurs, la routine **SaveRegs** (adresse #679B) stocke à cette adresse le contenu de **D0**, alors que la routine **RclRegs** (adresse #67D2) y récupère une adresse qu'elle place dans ce registre.

Nom: **OSAVE**	Adresse: **#80734h**	Taille: 5 quartets

La HP48 utilise ces cinq quartets pour sauvegarder l'adresse de différents objets.

Nom: **LASTARG1**	Adresse: **#80739h**	Taille: 5 quartets
Nom: **LASTARG2**	Adresse: **#8073Eh**	Taille: 5 quartets
Nom: **LASTARG3**	Adresse: **#80743h**	Taille: 5 quartets
Nom: **LASTARG4**	Adresse: **#80748h**	Taille: 5 quartets
Nom: **LASTARG5**	Adresse: **#8074Dh**	Taille: 5 quartets

Ces différents champs contiennent les adresses des arguments de la dernière commande évaluée (en général une commande intégrée). Les numéros correspondent ici aux niveaux de la pile où se trouvaient ces arguments. Quand une commande utilise moins de cinq arguments (ce qui est en général le cas), les champs "LASTARG*n*" correspondants sont mis à zéro.

Nom: **leeway**	Adresse: **#80752h**	Taille: 5 quartets

Contient l'adresse d'un entier héxadécimal long (de #B8h=184 chiffres). Cet entier binaire est créé (avec la valeur zéro) par l'External situé à l'adresse #17F40. Il est éliminé dans les situations de très faible mémoire disponible (par un External situé à l'adresse #17C55, que je ne vous recommande pas trop d'utiliser, car il interdit toute entrée en ligne de commande).

Ce grand entier est peut-être utilisé dans le processus de compilation de la ligne de commande.

Nom: **ITEM1STATE**	Adresse: **#80757h**	Taille: 5 quartets

Contient l'adresse d'un entier binaire variable, utilisé dans l'environnement Equation-Writer.

Nom: **HISTORY1**	Adresse: **#8075Ch**	Taille: 5 quartets
Nom: **HISTORY2**	Adresse: **#80761h**	Taille: 5 quartets
Nom: **HISTORY3**	Adresse: **#80766h**	Taille: 5 quartets
Nom: **HISTORY4**	Adresse: **#8076Bh**	Taille: 5 quartets

Ces quatre champs contiennent les adresses des quatre dernières lignes de commandes évaluées (HISTORY1 désigne la ligne de commande la plus récente).

Nom: **PDCHXS**	Adresse: **#80770h**	Taille: 5 quartets

Utilisé dans l'environnement Equation-Writer. Contient peut-être la taille de l'objet-graphique affiché.

Nom: **KERMERRM**	Adresse: **#80775h**	Taille: 5 quartets

Contient l'adresse du dernier message d'erreur de Kermit.

Nom: **PAINTTREE**	Adresse: **#8077Ah**	Taille: 5 quartets

Contient une adresse utilisée dans l'environnement Equation-Writer.

Nom: **EXITMSG**	Adresse: **#8077Fh**	Taille: 5 quartets

Contient l'adresse du dernier message d'erreur personnalisé (le champ est remis à zéro par chaque RESET).

Nom: **AppDisplay**	Adresse: **#80784h**	Taille: 5 quartets
Nom: **AppKeys**	Adresse: **#80789h**	Taille: 5 quartets
Nom: **AppExitCond**	Adresse: **#8078Eh**	Taille: 5 quartets
Nom: **AppError**	Adresse: **#80793h**	Taille: 5 quartets

Les quatre champs précédents contiennent les adresses d'objets à utiliser dans certains environnements personnalisés (comme le sont les environnements Equation-Writer ou Matrix-Writer):

- "***AppDisplay***" contient l'adresse de l'objet à évaluer avant d'exécuter l'action associée à une touche quelqu'elle soit.
- "***AppKeys***" contient l'adresse d'une structure Rpl gérant l'action des touches redéfinies dans cet environnement.
- "***AppExitCond***" contient l'adresse d'une structure Rpl dont l'évaluation conduit aux Externals True ou False, indiquant si on doit sortir ou non de l'environnement.
- "***AppError***" contient l'adresse d'une structure Rpl devant être évaluée si une erreur se produit dans l'environnement.

Nom: **CtlAlarm**	Adresse: **#80798h**	Taille: 5 quartets

Contient une adresse utilisée dans la gestion des alarmes.

Nom: **MenuDef**	Adresse: **#8079Dh**	Taille: 5 quartets

Contient l'adresse où se trouve l'objet représentant le menu en cours.

Nom: **LastMenuDef**	Adresse: **#807A2h**	Taille: 5 quartets

Comme ci-dessus, mais pour le menu précédant le menu en cours.

Nom: **MenuData**	Adresse: **#807A7h**	Taille: 5 quartets

Contient l'adresse d'un objet décrivant les labels des menus.

Nom: **MenuRowAct**	Adresse: **#807ACh**	Taille: 5 quartets

Contient l'adresse d'un objet à évaluer lors de chaque appui sur **NXT** ou **PREV**.

Nom: **LabelDef**	Adresse: **#807B1h**	Taille: 5 quartets

Contient l'adresse d'une routine affichant un label de menu à une colonne déterminée (exprimée en pixels). Cette routine attend le label (sous forme de chaîne) au niveau 1 et le numéro de colonne (sous forme d'un entier-système) au niveau 2.

Nom: **MenuKeyNS**	Adresse: **#807B6h**	Taille: 5 quartets

Contient l'adresse d'une routine à évaluer lorsqu'une touche de menu est actionnée "non shiftée". Cette routine attend au niveau 1 l'objet associé à cette toucche.

Nom: **MenuKeyLS**	Adresse: **#807BBh**	Taille: 5 quartets

Comme ci-dessus, pour une touche de menu actionnée "shiftée à gauche".

Nom: **MenuKeyRS**	Adresse: **#807C0h**	Taille: 5 quartets

Comme ci-dessus, pour une touche de menu actionnée "shiftée à droite".

Nom: **ReviewKey**	Adresse: **#807C5h**	Taille: 5 quartets

Contient l'adresse d'une routine à évaluer lorsque l'on actionne la touche ┌▸**VIEW**.

Nom: **LastContext**	Adresse: **#807CAh**	Taille: 5 quartets

Contient une sauvegarde de l'adresse du dernier répertoire-maître.

Nom: **TrackAct**	Adresse: **#807CFh**	Taille: 5 quartets

Contient l'adresse d'une routine à évaluer lorsque l'on change de menu. Ce champ est modifié par la HP48 à chaque fois qu'on change de menu.

Nom: **MenuExitAct**	Adresse: **#807D4h**	Taille: 5 quartets

Contient l'adresse d'une routine à évaluer lorsque l'on sort du menu en cours. Ce champ est modifié par la HP48 à chaque fois qu'on change de menu.

Nom: **LASTROMWDOB**	Adresse: **#807D9h**	Taille: 5 quartets

Contient l'adresse de la dernière commande de librairie évaluée. Cette sauvegarde est utile à la HP48 pour attribuer une erreur à la commande qui l'a provoquée.

Nom: **KeyOb**	Adresse: **#807DEh**	Taille: 5 quartets

Contient l'adresse du label de la dernière touche actionnée.

Nom: **RAMEND**	Adresse: **#807E8h**	Taille: 5 quartets

Contient l'adresse du dernier quartet de la RAM.

Nom: **AVMEM**	Adresse: **#807EDh**	Taille: 5 quartets

Contient la taille de la mémoire disponible, l'unité étant l'adresse (bloc de cinq quartets). Cette valeur est le plus souvent stockée dans le registre **D** du microprocesseur. D'ailleurs c'est ici que la routine **SaveRegs** (adresse #679B) stocke le registre **D**, et que la routine **RclRegs** (adresse #67D2) y récupère le contenu de ce registre.

Nom: **ERROR**	Adresse: **#807F2h**	Taille: 5 quartets

Contient le numéro de la prochaine erreur à afficher. Dans le cas d'une erreur personnalisée, le numéro est <70000h>.

Nom: **ATTNFLG**	Adresse: **#807F7h**	Taille: 5 quartets

Contient le nombre de fois que la touche **ON** a été actionnée. Ce champ est évidemment remis à zéro lors de tout retour au mode interactif.

Nom: **FIRSTPROC**	Adresse: **#807FCh**	Taille: 5 quartets

Contient l'adresse de la routine constituant la "Main Loop" du fonctionnement Rpl de la HP48 (La boucle qui est exécutée en permanence pour attendre vos instructions et y répondre). En principe il s'agit de l'adresse #3858Eh.

Les champs qui vont suivre (de **SysNib1** à **SysNib20**) sont utilisés par la HP48 pour gérer des flags internes. Ils ne faut pas les confondre avec les flags-système et les flags-utilisateurs. Les indicateurs dont il s'agit ici sont régulièrement modifiés par la HP48.

Ces champs occupent tous un quartet. Nous noterons bit3, bit2, bitl et bit0 les quatre bits de ce quartet (du bit de poids fort au bit de poids faible). Les indications qui sont données ici concernent uniquement l'état armé de ces bits.

Nom: **SysNib1**	Adresse: **#80801h**	Taille: 1 quartet

- Bit0: indique que la zone 2 de l'écran est gelée.
- Bit1: indique que les objets de la pile doivent être désignés seulement par leur type (situation de très faible mémoire disponible).
- Bit2: indique que la zone 2b de l'écran montre la ligne de commande.
- Bit3: pendant l'attente d'une touche au clavier, ignore les alarmes éventuelles.

Nom: **SysNib2**	Adresse: **#80802h**	Taille: 1 quartet

- Bit0: Utilisé pour les interruptions si l'horloge est affichée.
- Bit1: Indique que l'on est en mode serveur.
- Bit2: Indique que le menu en cours doit être comparé avec le menu précédent.
- Bit3: Indique que le menu en cours est incorrect.

Nom: **SysNib3**	Adresse: **#80803h**	Taille: 1 quartet

- Bit0: Sauvegarde automatique de la pile.
- Bit1: Sélectionne le mode insertion (recouvrement si désarmé).
- Bit2: Sélectionne le mode ALG.
- Bit3: Utilisé dans les opérations d'impression ?

Nom: **SysNib4**	Adresse: **#80804h**	Taille: 1 quartet

- Bit0: indique que la zone 2a de l'écran est gelée.
- Bit1: indique que la zone 2b de l'écran est gelée.
- Bit2: indique que la zone 3 de l'écran est gelée.
- Bit3: indique que les labels du menu doivent être "recalculés" à chaque changement de menu.

Nom: **SysNib5**	Adresse: **#80805h**	Taille: 1 quartet

- Bit0: Sélectionne la sauvegarde des lignes de commande.
- Bit1: Active l'interruption Timer1.
- Bit2: Sélectionne le mode "Minuscules" pour la ligne de commande.
- Bit3: La décompilation ne s'effectue que sur la pile, mais pas en ligne de commande.

Nom: **SysNib6**	Adresse: **#80806h**	Taille: 1 quartet

- Bit0: Sélectionne le mode 1USER.
- Bit1: Autorise une commande à suspendre l'environnement en cours.
- Bit2: Indique que l'environnement temporaire a été corrompu.
- Bit3: Indique que la zone 1 de l'écran est gelée.

Nom: **SysNib7**	Adresse: **#80807h**	Taille: 1 quartet

- Bit0: Indique que la zone 1 de l'écran est valide.
- Bit1: Indique que la zone 2a de l'écran est valide.
- Bit2: Indique que la zone 2b de l'écran est valide.
- Bit3: Indique que la zone 3 de l'écran est valide.

Nom: **SysNib8**	Adresse: **#80808h**	Taille: 1 quartet

- Bit0: Indique que la zone 1 de l'écran n'a pas changé.
- Bit1: Indique que la zone 2a de l'écran n'a pas changé.
- Bit2: Indique que la zone 2b de l'écran n'a pas changé.
- Bit3: Indique que la zone 3 de l'écran n'a pas changé.

Nom: **SysNib9**	Adresse: **#80809h**	Taille: 1 quartet

- Bit0: Indique que la zone 1 de l'écran est invalide.
- Bit1: Indique que la zone 2a de l'écran est invalide.
- Bit2: Indique que la zone 2b de l'écran est invalide.
- Bit3: Indique que la zone 3 de l'écran est invalide.

Nom: **SysNib10**	Adresse: **#8080Ah**	Taille: 1 quartet

- Bit0: Indique qu'une ligne de commande existe.
- Bit1: Indique que les touches non réassignées par un environnement personnalisé sont tout de même acceptées dans cet environnement.
- Bit2: Indique que les touches non réassignées sont acceptées dans le mode USR.
- Bit3: Indique qu'un environnement personnalisé est actif.

Nom: **SysNib11**	Adresse: **#8080Bh**	Taille: 1 quartet

- Bit0: Indique que les parenthèses sont implicites pour /,^,SQRT dans l'environnement EQUATION-WRITER.
- Bit1: Utilisé dans les opérations de décompilation en ligne de commande.
- Bit2: Indique qu'une toute nouvelle ligne de commande vient d'être créée.
- Bit3: Indique que seules les définitions standard des touches sont acceptées (même si on est en mode USR).

Nom: **SysNib12**	Adresse: **#8080Ch**	Taille: 1 quartet

- Bit0: Indique si l'horloge doit être affichée, indépendamment des choix faits par l'utilisateur (entrée CLK du menu MODES/MISC). Il faut en effet parfois effacer l'horloge, ne serait-ce que lorsqu'un message d'erreur est affiché.
- Bit1, Bit2, Bit3: Inutilisés.

Nom: **SysNib13**	Adresse: **#8080Dh**	Taille: 1 quartet
Nom: **SysNib14**	Adresse: **#8080Eh**	Taille: 1 quartet
Nom: **SysNib15**	Adresse: **#8080Fh**	Taille: 1 quartet
Nom: **SysNib16**	Adresse: **#80810h**	Taille: 1 quartet
Nom: **SysNib17**	Adresse: **#80811h**	Taille: 1 quartet
Nom: **SysNib18**	Adresse: **#80812h**	Taille: 1 quartet
Nom: **SysNib19**	Adresse: **#80813h**	Taille: 1 quartet
Nom: **SysNib20**	Adresse: **#80814h**	Taille: 1 quartet

Les bits correspondant à ces quartets sont inutilisés.

Nom: **AppCount**	Adresse: **#80815h**	Taille: 2 quartets

Donne le nombre d'environnements personnalisés emboîtés.

Nom: **ITEM1LINES**	Adresse: **#80817h**	Taille: 1 quartets

Nombre de lignes utilisées par l'objet situé au niveau 1 de la pile.

Nom: **VIEWLEVEL**	Adresse: **#80818h**	Taille: 5 quartets

Niveau de la pile en cours de visualisation en ligne de commande.

Nom: **DEPTHSAVE**	Adresse: **#8081Dh**	Taille: 5 quartets

Taille de la pile-utilisateur, exprimée en quartets: cette valeur est égale à 5*(DEPTH+1). C'est la différence entre l'adresse où débute la ligne de commande et l'adresse où est placé l'objet du niveau 1. c'est donc le nombre de quartets que nécessite la pile. La valeur initiale est ici 5 (les 5 quartets nuls de fin de pile).

Nom: **RNSEED**	Adresse: **#80822h**	Taille: 15 quartets

Corps du nombre réel utilisé comme racine du générateur de nombres aléatoires.

Nom: **SAVCLK**	Adresse: **#80831h**	Taille: 1 quartet

Sauvegarde de l'état affiché ou non de l'horloge.

Nom: **ALARMSDUE**	Adresse: **#80832h**	Taille: 1 quartet

Mis à #F pour signaler qu'une alarme est dûe, et à zéro dans le cas contraire.

Nom: **PASTDUE**	Adresse: **#80833h**	Taille: 1 quartet

Mis à #F pour signaler qu'une alarme est échue et non réceptionnée, et à #0 dans le cas contraire.

Nom: **DOUSEALARM**	Adresse: **#80834h**	Taille: 1 quartet

Mis à #F par la séquence ON-4. Mis à zéro par l'éxécution d'une alarme répétitive.

Nom: **NOALARMSRV**	Adresse: **#80835h**	Taille: 1 quartet

Si contient #F, alors les alarmes sont ignorées.

Nom: **ALARM#**	Adresse: **#80836h**	Taille: 5 quartets

Numéro de l'alarme dûe.

Nom: **PASTDUE#**	Adresse: **#8083Bh**	Taille: 5 quartets

Numéro de la dernière alarme échue et non réceptionnée.

Nom: **LPD_HIST**	Adresse: **#80840h**	Taille: 1 quartet

Utilisé quand une situation de "batterie insuffisante" est rencontrée.

Nom: **ANNUNCIATORS**	Adresse: **#80841h**	Taille: 2 quartets

On trouve ici 8 bits dont le rôle est de rappeler quels sont les indicateurs allumés en ligne de commande. Les modifier permet d'allumer ou d'éteindre ces indicateurs (sauf bit0, bit2, bi7).

bit0: indicateur "radians".
bit1: indicateur d'entrée-sortie.
bit2: indicateur "halt".
bit3: indicateur d'alarme.
bit4: indicateur shift gauche.
bit5: indicateur shift droit.
bit6: indicateur "alpha".
bit7: indicateur "sablier".

Nom: **SystemFlags** Nom: **UserFlags**	Adresse: **#80843h** Adresse: **#80853h**	Taille: 16 quartets Taille: 16 quartets

Ces deux tables contiennent respectivement les 64 bits des flags-système (n°-1 à -64) et les 64 bits des flags-utilisateur (n°1 à 64).

Le quartet de l'adresse #80843h+N ($0 \leq N \leq 15$) s'écrit *xyzt*, où *x*,*y*,*z*,*t* sont les bits correspondants aux flags de numéro (-4N-4), (-4N-3),(-4N-2), (-4N-1).

Le quartet de l'adresse #80853h+N ($0 \leq N \leq 15$) s'écrit *xyzt*, où *x*,*y*,*z*,*t* sont les bits correspondants aux flags de numéro (4N+4), (4N+3),(4N+2), (4N+1).

L'accès aux flags n'est pas facilité par le fait qu'ils soient rangés par bits successifs sur les zones décrites ci-dessus.

Les morceaux de programmes qui suivent permettent de lire ou d'écrire un de ces flags. On utilise avant tout le point d'entrée #537AD, qu'il faut commenter:

Label: **ReadFlag**	Adresse: **#537ADh**	Appel: GOSBVL

En entrée (A:A) doit contenir un numéro d'indicateur (de 1 à 64) et le bit 0 du registre ST doit contenir 0 pour accéder à la table des flags-utilisateur et 1 pour celle des flags-système. Si la valeur de A est incorrecte il y a une erreur "Bad Arg. Value", (erreur non précédée d'une récupération de B,D,D0,D1, qui à ce stade n'ont pas été utilisés).
Si A est correct, alors D0 pointe sur le quartet où se situe le bit qui correspond au flag. Le champ (C:XS) contient ce quartet, et le champ (A:XS) contient 8,4,2 ou 1, selon que le bit à lire dans ce quartet est le bit 3,2,1 ou 0.
Le registre D0 est sauvegardé dans R0. Les registres B,D,D1 ne sont pas affectés. P est remis à 0. ST n'est pas modifié

Exemples:

Pour les séquences ci-dessous voir les conditions d'entrée et de sortie indiquées dans le tableau précédent.

```
GOSBVL   #537AD   *Cette séquence arme le bit Carry
C=C&A    XS       *  si et seulement si le flag de
?C#0     XS       *  numéro (A:A) est armé.
```

```
GOSBVL   #537AD   *Cette séquence désarme le flag
A=-A-1   XS       *  dont le numéro est dans (A:A)
C=C&A    XS
DAT0=C   XS
```

```
GOSBVL   #537AD   *Cette séquence arme le flag dont
C=C!A    XS       *  le numéro est dans (A:A)
DAT0=C   XS
```

Nom: **ELEMENT**	Adresse: **#80863h**	Taille: 2 quartets

Compteur de profondeur, dans l'opération de décompilation d'un objet.

Nom: **FIRSTCHAR**	Adresse: **#80865h**	Taille: 5 quartets

Offset vers le premier caractère visible dans la ligne de commande.

Nom: **CR_COUNT**	Adresse: **#8086Ah**	Taille: 5 quartets

Nombre de passages à la ligne dans la ligne de commande.

Nom: **STACKNUM**	Adresse: **#8086Fh**	Taille: 5 quartets

Numéro du premier item visible dans la ligne de commande.

Nom: **TOPLINE**	Adresse: **#80874h**	Taille: 5 quartets

Numéro de la première ligne visible dans la ligne de commande.

Nom: **LASTLAST?**	Adresse: **#80879h**	Taille: 1 quartet

Contient une valeur non nulle si la dernièe touche actionnée est UNDO.

Nom: **HISTORYLEVEL**	Adresse: **#8087Ah**	Taille: 1 quartet

Indique le premier niveau libre sur la pile des sauvegardes de lignes de commande (voir HISTORY1 en #8075C).

Nom: **LASTARGCOUNT**	Adresse: **#8087Bh**	Taille: 1 quartet

Nombre d'arguments sauvegardés par la dernière commande.

Nom: **LASTARGf**	Adresse: **#8087Ch**	Taille: 1 quartet

Mis à une valeur non nulle si la dernière commande a provoqué la sauvegarde de plus de trois arguments.

Nom: **LASTERROR**	Adresse: **#8087Dh**	Taille: 5 quartets

Sauvegarde du numéro de la dernière erreur.

Nom: **CURSOREPOSN**	Adresse: **#80882h**	Taille: 5 quartets

Position du curseur sur la ligne de commande (valeur mini: 1).

Nom: **CURSORROW**	Adresse: **#80887h**	Taille: 5 quartets

Numéro de ligne où se trouve le curseur, sur la ligne de commande (valeur mini 1).

Nom: **CURSORPOSN**	Adresse: **#8088Ch**	Taille: 2 quartets

Numéro de colonne, sur l'écran, où se trouve le curseur dans la ligne de commande (minimum 0, maximum 21).

Nom: **CURSORSTATE**	Adresse: **#8088Eh**	Taille: 1 quartet

Si ce quartet est à zéro, le curseur est visible en ligne de commande. Sinon c'est le caractère qui est visible.

Nom: **CURSORCHR**	Adresse: **#8088Fh**	Taille: 2 quartets

Code du caractère présent sous le curseur.

Nom: **CURSORGROB**	Adresse: **#80891h**	Taille: 40 quartets

Objet-Graphique représentant le caractère sous le curseur.

Nom: **CURSORX**	Adresse: **#808B9h**	Taille: 5 quartets

Coordonnée en X du curseur (exprimée en pixels).

Nom: **CURSORY**	Adresse: **#808BEh**	Taille: 5 quartets

Coordonnée en Y du curseur (exprimée en pixels).

Nom: **CURRENTMENU**	Adresse: **#808C3h**	Taille: 2 quartets

Numéro du menu en cours.

Nom: **MENULEVEL**	Adresse: **#808C5h**	Taille: 5 quartets

Première entrée visible du menu en cours (varie de 6 en 6).

Nom: **OLDMENU**	Adresse: **#808CAh**	Taille: 2 quartets

Numéro du menu précédant le menu en cours.

Nom: **T1COUNT**	Adresse: **#808CCh**	Taille: 1 quartet

Compteur utilisé par le gestionnaire d'interruptions.

Nom: **PADCOUNT**	Adresse: **#808CDh**	Taille: 1 quartet

Compteur d'espaces dans la ligne de commande.

Nom: **GARBSCRATCH1**	Adresse: **#808CEh**	Taille: 5 quartets

Sauvegarde un niveau de la pile des retours du RPL, dans l'opréation de "Garbage Collection" (qui est gourmande en Gosub).

Nom: **GARBSCRATCH2**	Adresse: **#808D3h**	Taille: 5 quartets

Sauvegarde le contenu du registre TEMPENV (#80702), c'est-à-dire l'adresse de début de la zone des objets temporaires, dans un "Garbage Collection".

Nom: **SAVECROSS**	Adresse: **#808D8h**	Taille: 10 quartets

Champ utilisé pour sauvegarder la position du curseur graphique (lors d'un tracé).

Nom: **PADJSAVE1**	Adresse: **#808E2h**	Taille: 1 quartet

Utilisé pour la sauvegarde du registre ST.

Nom: **PADJSAVE2**	Adresse: **#808E3h**	Taille: 10 quartets

Utilisé pour sauvegarder un niveau de la pile RSTK.

Nom: **KERMMODE**	Adresse: **#808EDh**	Taille: 1 quartet

Information sur le mode Kermit.

Nom: **DcompWidth**	Adresse: **#808EEh**	Taille: 2 quartets

Taille du buffer de décompilation.

Nom: **FONTHEIGHT**	Adresse: **#808F0h**	Taille: 1 quartet
Nom: **FONTWIDTH**	Adresse: **#808F1h**	Taille: 1 quartet
Nom: **FONTCOUNT**	Adresse: **#808F2h**	Taille: 3 quartets
Nom: **NODECOUNT**	Adresse: **#808F5h**	Taille: 3 quartets
Nom: **OBTREELEN**	Adresse: **#808F8h**	Taille: 3 quartets
Nom: **LASTOP**	Adresse: **#808FBh**	Taille: 1 quartet

Les champs ci-dessus sont utilisés dans l'environnement Equation-Writer:

FONTHEIGHT: Sélecteur de la hauteur de fonte.
FONTWIDTH: Sélecteur de la largeur de fonte.
FONTCOUNT: Compteur.
NODECOUNT: Compteur de profondeur logique, dans une expression.
OBTREELEN: Longueur d'objet.
LASTOP: utilisé pour désigner le type du dernier objet (opérateur unaire, opérateur binaire, ou opérande).

Nom: **LEFTTREE**	Adresse: **#808FCh**	Taille: 3 quartets
Nom: **RIGHTTREE**	Adresse: **#808FFh**	Taille: 3 quartets
Nom: **PARENTTREE**	Adresse: **#80902h**	Taille: 3 quartets
Nom: **PRECSTACK**	Adresse: **#80905h**	Taille: 7 quartets
Nom: **KEYLIST**	Adresse: **#8090Ch**	Taille: 5 quartets
Nom: **KEYLOCK**	Adresse: **#80911h**	Taille: 1 quartet
Nom: **ACCUM**	Adresse: **#80912h**	Taille: 1 quartet
Nom: **OB/EXP?**	Adresse: **#80913h**	Taille: 1 quartet
Nom: **CLOWIDTH**	Adresse: **#80914h**	Taille: 2 quartets
Nom: **ENTRWISE**	Adresse: **#80916h**	Taille: 1 quartet
Nom: **PARENCOUNT**	Adresse: **#80917h**	Taille: 2 quartets
Nom: **STRETCHCOUNT**	Adresse: **#80919h**	Taille: 2 quartets

Les champs ci-dessus sont également utilisés dans Equation-Writer.

Nom: **ClkOnNlb**	Adresse: **#8091Bh**	Taille: 1 quartet

Si ce quartet n'est pas à zéro, c'est que l'horloge est affichée.

Nom: **XmitSrcvTOut**	Adresse: **#8091Ch**	Taille: 2 quartets

Durée du "Time Out" pour l'instruction SRECV ou pour XMIT.

Nom: **DelayCt**	Adresse: **#8091Eh**	Taille: 2 quartets
Nom: **GCOLCOUNT**	Adresse: **#80920h**	Taille: 2 quartets
Nom: **COLCOUNT**	Adresse: **#80922h**	Taille: 2 quartets

Les champs précédents sont utilisés dans les opérations d'impression.
DelayCt: Delai entre deux impressions de ligne.
GCOLCOUNT: Nombre de colonnes, pour impression graphique.
COLCOUNT: Nombre de colonnes sur chaque ligne de texte (en pixels).

Nom: **PrtStatus**	Adresse: **#80924h**	Taille: 3 quartets

Ce champ de 12 bits contient les caractéristiques de l'impression en cours:
bit0: Le denier caractère de la ligne était du texte.
bit1: Le dernier caractère de la ligne était "Escape".
bit2: On imprime en double largeur.
bit3: On est dans une séquence graphique.
bit4: Le dernier caractère de la ligne était un Line Feed.
bit5: ?
bit6: Attente d'une ligne à imprimer.
bit7: Erreur de batterie faible.
bit8: Double espacement après le prochain Line Feed.
bit9: Revient à l'espacement simple.
bit10: ?
bit11: Désactive PRLCD.

Nom: **IOCsave**	Adresse: **#80927h**	Taille: 1 quartet
Nom: **LineByteCt**	Adresse: **#80928h**	Taille: 2 quartets
Nom: **FifoByteCt**	Adresse: **#8092Ah**	Taille: 2 quartets
Nom: **LastPrntTime**	Adresse: **#8092Ch**	Taille: 11 quartets
Nom: **PFIFO**	Adresse: **#80937h**	Taille: 16 quartets

Ces trois champs sont encore utilisés dans les opérations d'impression.

Nom: **MenuRow**	Adresse: **#80947h**	Taille: 5 quartets

Contient le numéro de la première entrée affichée dans le menu en cours. Il suffit de modifier ce champ (de +1, par ex.), pour décaler le menu d'une entrée vers la droite.

Nom: **EqPtr**	Adresse: **#8094Ch**	Taille: 5 quartets

Pointe sur l'équation courante, dans le cas d'une liste d'équations.

Nom: **KeyRomPtr0**	Adresse: **#80951h**	Taille: 11 quartets
Nom: **KeyRomPtr1**	Adresse: **#8095Ch**	Taille: 11 quartets
Nom: **KeyRomPtr2**	Adresse: **#80967h**	Taille: 11 quartets
Nom: **KeyRomPtr3**	Adresse: **#80972h**	Taille: 11 quartets
Nom: **KeyRomPtr4**	Adresse: **#8097Dh**	Taille: 11 quartets
Nom: **KeyRomPtr5**	Adresse: **#80988h**	Taille: 11 quartets
Nom: **KeyRomPtr6**	Adresse: **#80993h**	Taille: 11 quartets

Le premier champ est un nom XLIB désignant la routine de traitement de la touche pressée. Les 6 champs suivants contiennent des noms XLIB désignant les routines associées aux 6 différentes touches de menu, quand celui-ci est lui-même défini par un tableau de Noms XLIB.

Nom: **LastMenuRow**	Adresse: **#8099Eh**	Taille: 5 quartets

Contient le numéro de la première entrée affichée dans le menu qui précède le menu en cours (voir le champ **MenuRow**, ci-dessus).

Nom: **ROMPTAB**	Adresse: **#809A3h**	Taille: .....

La zone qui commence à cette adresse a été décrite dans le chapitre du tome 1 qui est consacré à la mémoire cachée. Rappelons que la structure de cette table est:

```
<n>             nombre de librairies, sur 3 quartets
<lib0>          champ de la librairie n°0 (16 quartets)
<lib1>          champ de la librairie n°1 (16 quartets)
<lib2>          champ de la librairie n°2 (16 quartets)
....
<libn>          champ de la librairie de numéro maxi.
etc...
```

Le champ descriptif d'une librairie a le format suivant (16 quartets):

<id>	identificateur (numéro) sur 3 quartets
<AdrId>	adresse du champ "numéro" de la librairie
<AdrAcc>	adresse de la routine d'accès
<???>	trois quartets réservés

Le champ **<AdrId>** évoqué ci-dessus contient l'adresse où se trouve, dans le corps de la librairie, le numéro de celle-ci (juste après le champ réservé au nom, juste avant le champ réservé à l'offset vers la Hash-Table: voir le chapitre sur les librairies, dans la première partie du tome 1 des "Secrets").

A la suite de la table décrivant où on trouve toutes les librairies (attachées ou non), on lit éventuellement un ou plusieurs objets de type "Access-Pointer".

Il y a autant d'"Access-Pointers" qu'il y a de librairies attachées dont la Hash-Table se trouve dans une zone couverte de la mémoire. Chacun d'eux décrit l'endroit où se trouve la Hash-Table.

Dans le cas d'une HP48G, Il n'y a aucun "Access-Pointer" de ce type.

Dans le cas d'une HP48GX, il n'y en a pas non plus si les slots 1 et 2 sont libres (car la librairie n°#AB, qui contient les nouvelles fonctions de la 48G par rapport à la 48S, n'est pas couverte).

Quand une librairie est attachée à HOME et que sa Hash-Table est dans une zone couverte de la mémoire, le début de HOME contient une référence vers l'Access-Pointer dont nous venons de parler (voir, dans la première partie du tome 1 des "Secrets", le chapitre consacré aux répertoires).

A la suite de cette séquence (éventuellement vide) d'"Access-Pointers", on trouve l'objet graphique du menu, puis celui de la pile, puis celui de PICT. On a vu dans ce chapitre quels champs de la RAM-Système permettaient de pointer sur ces Grobs (**VDISP2** en #806D0 pointe sur le Grob des menus, **ADISP** en #806D5 pointe sur le Grob de la pile, et **GDISP** en #806E4 pointe sur PICT).

Voici maintenant la table de tous les "champs" de la RAM-Système, rangée par ordre alphabétique des mnémoniques. De cette manière il est plus facile de retrouver l'adresse d'un champ particulier.

AccessInit	#800E6h	FONTCOUNT	#808F2h
ACCUM	#80912h	FONTHEIGHT	#808F0h
ADISP	#806D5h	FONTWIDTH	#808F1h
ALARM#	#80836h	GARBSCRATCH1	#808CEh
ALARMS	#8072Ah	GARBSCRATCH2	#808D3h
ALARMSDUE	#80832h	GCOLCOUNT	#80920h
ANNUNCIATORS	#80841h	GDISP	#806E4h
AppCount	#80815h	GraphPrtHook	#8030Eh
AppDisplay	#80789h	HANDSHK	#8065Bh
AppError	#80798h	HISTORY1	#8075Ch
AppExitCond	#80793h	HISTORY2	#80761h
AppKeys	#8078Eh	HISTORY3	#80766h
ATTNFLG	#807F7h	HISTORY4	#8076Bh
AVMEM	#807EDh	HISTORYLEVEL	#8087Ah
ClkOnNIb	#8091Bh	HOMEMASK	#8000Ah
CLOWIDTH	#80914h	HRAMEND	#8000Fh
CMOS	#80000h	INITEN	#8065Ah
COLCOUNT	#80922h	INPUTSTREAM	#80669h
CONFRAM	#80524h	INTRAM	#805DBh
CONFTAB	#8052Bh	INTRPPTR	#8072Fh
CONTEXT	#8071Bh	IOCNIB	#80523h
COVERsave	#800EBh	IOCsave	#80927h
COVERstate	#800E8h	IOSAVE	#80654h
CR_COUNT	#8086Ah	IRAMBUFF	#800F5h
CRCxmit	#80076h	IRAMMASK	#80005h
CSPEED	#80655h	IREG	#8064Fh
CtlAlarm	#8079Dh	ITEM1LINES	#80817h
CURRENTMENU	#808C3h	ITEM1STATE	#80757h
CURSORCHR	#8088Fh	KERMERRM	#8077Ah
CURSOREPOSN	#80882h	KERMMODE	#808EDh
CURSORGROB	#80891h	KEYLIST	#8090Ch
CURSORPOSN	#8088Ch	KEYLOCK	#80911h
CURSORROW	#80887h	KeyOb	#807E3h
CURSORSTATE	#8088Eh	KeyRomPtr0	#80951h
CURSORX	#808B9h	KeyRomPtr1	#8095Ch
CURSORY	#808BEh	KeyRomPtr2	#80967h
DcompWidth	#808EEh	KeyRomPtr3	#80972h
DelayCt	#8091Eh	KeyRomPtr4	#8097Dh
DEPTHSAVE	#8081Dh	KeyRomPtr5	#80988h
DISP1CTLg	#8068Dh	KeyRomPtr6	#80993h
DISP2CTLg	#80695h	KEYSTATE	#8065Ch
DOLPENV	#80707h	LabelDef	#807B6h
DOUSEALARM	#80834h	LASTARG1	#80739h
DREND	#8064Ah	LASTARG2	#8073Eh
DRSTART	#80645h	LASTARG3	#80743h
DSKTOP	#806F8h	LASTARG4	#80748h
EDITLINE	#806FDh	LASTARG5	#8074Dh
ELEMENT	#80863h	LASTARGCOUNT	#8087Bh
ENTRWISE	#80916h	LASTARGf	#8087Ch
EqPtr	#8094Ch	LastContext	#807CFh
ERROR	#807F2h	LASTERROR	#8087Dh
EXITMSG	#80784h	LASTLAST?	#80879h
FAILSTK1	#80010h	LastMenuDef	#807A7h
FAILSTK2	#80022h	LastMenuRow	#8099Eh
FAILSTK3	#80034h	LASTOP	#808FBh
FAILSTK4	#80046h	LastPrntTime	#8092Ch
FailTime	#80085h	LASTROMWDOB	#807DEh
FifoByteCt	#8092Ah	leeway	#80752h
FIRSTCHAR	#80865h	LEFTTREE	#808FCh
FIRSTPROC	#807FCh	LineByteCt	#80928h

LINECOUNTg	#8069Ah
LINENIBSg	#80692h
LoBatTime	#80077h
LPD_HIST	#80840h
MenuData	#807ACh
MenuDef	#807A2h
MenuExitAct	#807D9h
MenuKeyLS	#807C0h
MenuKeyNS	#807BBh
MenuKeyRS	#807C5h
MENULEVEL	#808C5h
MenuRow	#80947h
MenuRowAct	#807B1h
NEXTIRQ	#80058h
NOALARMSRV	#80835h
NODECOUNT	#808F5h
OB/EXP?	#80913h
OBTREELEN	#808F8h
OLDMENU	#808CAh
OSAVE	#80734h
PADCOUNT	#808CDh
PADJSAVE1	#808E2h
PADJSAVE2	#808E3h
PAINTTREE	#8077Fh
PARENCOUNT	#80917h
PARENTTREE	#80902h
PASTDUE	#80833h
PASTDUE#	#8083Bh
PDCHXS	#80770h
PDCSYMB	#80775h
PFIFO	#80937h
POPPEDKEY	#8068Bh
PORT0EOS	#80536h
Port1CRC	#800E2h
PORT1EOS	#8053Bh
PORT2EOS	#80540h
PortStat	#800E1h
PRECSTACK	#80905h
PrtStatus	#80924h
R1[A]save	#806C0h
R2[A]save	#806BAh
R2[S]save	#806BFh
RAMEND	#807E8h
ReviewKey	#807CAh
RIGHTTREE	#808FFh
RNSEED	#80822h
ROMPARTS	#80716h
ROMPTAB	#809A3h
RSKTOP	#806F3h
SAVCLK	#80831h
SAVE_A	#805F5h
SAVE_B	#80608h
SAVE_BO	#806C5h
SAVE_C[A]	#805F0h
SAVE_D	#80618h
SAVE_DO	#8063Dh
SAVE_LC	#806C6h
SAVE_LN	#806C8h
SAVE_MODES	#805EBh
SAVE_OFFSET	#806CBh
SAVE_OR	#80642h
SAVE_PC	#80638h
SAVE_R0	#80628h
SAVE_ST	#80605h
SAVECROSS	#808D8h
SEMAPH	#80652h
STACKNUM	#8086Fh
StartTime	#80078h
Stk0save	#8069Ch
Stk1save	#806A1h
Stk2save	#806A6h
Stk3save	#806ABh
Stk4save	#806B0h
Stk5save	#806B5h
STOPSIGN	#80720h
STRETCHCOUNT	#80919h
SW_ETime	#800D4h
SW_Image	#800BEh
SysNib1	#80801h
SysNib10	#8080Ah
SysNib11	#8080Bh
SysNib12	#8080Ch
SysNib13	#8080Dh
SysNib14	#8080Eh
SysNib15	#8080Fh
SysNib16	#80810h
SysNib17	#80811h
SysNib18	#80812h
SysNib19	#80813h
SysNib2	#80802h
SysNib20	#80814h
SysNib3	#80803h
SysNib4	#80804h
SysNib5	#80805h
SysNib6	#80806h
SysNib7	#80807h
SysNib8	#80808h
SysNib9	#80809h
SystemFlags	#80843h
T1COUNT	#808CCh
TEMPENV	#80702h
TEMPOB	#806E9h
TEMPTOP	#806EEh
TESTMSG	#80092h
TIMECRC	#80065h
TIMEOUT	#80069h
TIMEOUTCLK	#80076h
TIMExmit	#80069h
TOPLINE	#80874h
TOUCTAB	#8070Ch
TrackAct	#807D4h
uart_buf_end	#80519h
uart_buf_st	#8051Ch
uart_buffer	#80319h
uart_error	#8051Bh
uart_handshk	#8051Eh
uart_modes	#8051Fh
uart_parity	#80520h
uart_timeout	#80521h
UserFlags	#80853h
UserKeys	#80725h
USEROB	#80711h
VDISP1	#806DAh
VDISP2	#806D0h
VDISP3	#806DFh
VIEWLEVEL	#80818h
XmitSrcvTOut	#8091Ch

QUELQUES POINTS D'ENTREE

La programmation en assembleur est très facilitée par la connaissance de certains points d'entrée de la HP48.

Ces points d'entrée, contrairement aux Externals, sont accessibles en assembleur, par une instruction GOSBVL ou GOVLNG. La liste que je vais vous présenter est évidemment très partielle. Elle résulte d'une sélection rapide, opérée à la lecture de mes listings personnels. Il ne tient qu'à vous de la compléter.

NB: Quand une adresse #*****h est précédée du caractère ▪, c'est qu'elle est "garantie" par HP (elle est à la même place depuis la HP48 version A, et elle ne changera pas sur les futures versions de la G/GX).

◆ SELECTION DE POINTS D'ENTREE:

Les adresses qui suivent pourront vous aider à écrire des programmes en assembleur. Elles réalisent en effet des tâches qui reviennent fréquemment.

Vous pouvez les appeler depuis vos programmes assembleur, au moyen d'un branchement absolu (**GOSBVL** ou **GOVLNG**). J'ai précisé d'ailleurs quel type de branchement correspondait à chaque adresse (en fait, on appelle par **GOVLNG** tous les points d'entrée qui effectuent un retour au Rpl).

J'ai essayé de décrire l'action de ces sous-programmes, notamment pour ce qui est des registres qu'ils utilisent. Une connaissance parfaite de leur fonctionnement ne pourra être acquise que si vous désassemblez entièrement le code qu'ils désignent.

J'attire en particulier votre attention sur l'importance des routines **SavRegs** et **RclRegs**. Notez également que vos programmes en langage-machine doivent se terminer en mode **HEX**, et que le contenu de **P** doit être nul (instruction **P= 0**). J'ai utilisé des labels pour chaque point d'entrée: ils n'ont qu'un rôle descriptif...

Clavier (points d'entrée)

Label: **ClrKeybd**	Adresse:▪**#00D57h**	Appel: GOSBVL
Vide le buffer du clavier, en stockant 00000 à l'adresse #80669h. Registres affectés: Le champ A de **C**, le bit **CARRY** (désarmé) et le registre **D1** (qui pointe sur #80669h)		

Label: **ClrAttn**	Adresse:▪**#00D8Eh**	Appel: GOSBVL
Vide le compteur de "Touche ON", en stockant 00000 en #807F7h. Registre(s) affecté(s): C champ A; D1 (pointe sur #807F7h)		

Label: **ClrAttn?**	Adresse: **#04988h**	Appel: GOSBVL
Lit C:A à l'adresse #807F7h (compteur de touche ON). Arme le bit Carry si le résultat est non nul, le désarme sinon. Modifie D1.		

Label: **AttnKey?**	Adresse:▪**#00C74h**	Appel: GOSBVL
Charge le contenu de **IN** dans le le champ **C** (champ 0-3) Arme le bit CARRY si et seulement si le bit 15 de (C) est à 1, c'est-à-dire si la touche **Attn** (ON) est actionnée		

Label: **KeyInBuf?**	Adresse:▪**#04999h**	Appel: GOSBVL
Charge A:S avec le contenu de (#80669) (début buffer clavier) et C:S avec celui de (#8066A) (fin du buffer). Arme Carry si ces valeurs sont distinctes (cad s'il y a une touche dans le buffer) et désarme Carry sinon. A la sortie (D1)=#8066A.		

Label: **GetKeyC**	Adresse:▪**#04840h**	Appel: GOSBVL
Si le buffer clavier est non vide, charge C:B avec le caractère le plus ancien (le sauvegarde en #8068B) et désarme Carry. Si le buffer est vide, le bit Carry est armé. Affecte le reg D1. Registres affectés: A,C,D1,P (qui est mis à zéro)		

Voici quelques points d'entrée qui sont plus particulièrement destinés aux applications graphiques.

Ecran (points d'entrée)

Label: **ScrOff**	Adresse:▪**#01BBDh**	Appel: GOSBVL
Eteint l'écran (désarme le bit 3 du quartet (#100h)). Affecte D0,C.		
Label: **ScrOn**	Adresse:▪**#01B8Fh**	Appel: GOSBVL
Allume l'écran (arme le bit 3 du quartet (#100h)). Affecte D0 et C.		
Label: **ScrOff+**	Adresse: **#01BD3h**	Appel: GOSBVL
Eteint l'écran par ScrOff, puis désarme le bit 3 du quartet (#10Ch) (éteint donc l'indicateur "busy")		
Label: **ScrOn+**	Adresse: **#01BA5h**	Appel: GOSBVL
Allume l'écran par ScrOn, puis arme le bit 3 du quartet (#10Ch) (rallume donc l'indicateur "busy").		
Label: **D0=BitScr**	Adresse:▪**#01C31h**	Appel: GOSBVL
A la sortie D0 pointe sur le début de la BitMap écran (20 quartets après le début de l'objet-graphique). A:A est égal à D0.		
Label: **D0=BitMen**	Adresse:▪**#01C58h**	Appel: GOSBVL
A la sortie D0 pointe sur le début de la BitMap menu (20 quartets après le début de l'objet-graphique). A:A est égal à D0.		
Label: **ClrScrMn**	Adresse: **#01BFFh**	Appel: GOSBVL
Efface le contenu du Grob écran, et du Grob des menus. Modifie A:W, C:A, et D0. Exécute SETHEX. Retour par RNTCC.		
Label: **ClrMenu**	Adresse: **#01C15h**	Appel: GOSBVL
Efface le contenu du Grob des menus. Modifie A:W, C:A, D0.		
Label: **ResBlank**	Adresse:▪**#115B3h**	Appel: GOSBVL
Réserve en mémoire la place pour un objet graphique blanc de taille R0:A (pour la largeur) et R1:A (pour la hauteur). A la sortie, D0 pointe sur la bitmap du Grob (cad 20 quartets après le premier quartet de l'adresse-préfixe).		
Label: **ReplGrob**	Adresse: **#116B5h**	Appel: GOSBVL
Applique le Grob pointé par D0 sur le Grob pointé par D1, à partir du point dont les coordonnées dans D1 sont R1:A (abscisse) et R0:A (ordonnée). Attention: il ne doit pas y avoir débordement.		

SÉLECTION DE POINTS D'ENTRÉE

Label: **IntOff**	Adresse:▪**#01115h**	Appel: GOSBVL
Interdit les interruptions (ST=0 15, met à 1 puis à zéro le bit 0 du quartet (#10Eh), exécute INTOFF, puis retour par RNTCC)		
Label: **IntOn**	Adresse:▪**#010E5h**	Appel: GOSBVL
Autorise les interruptions: ST=1 15, INTON, la suite dépendant de l'état du bit Carry ou du bit 14 du registre ST.		
Label: **A=in**	Adresse:▪**#0115Ah**	Appel: GOSBVL
Sous-programme ne contenant que l'instruction A=IN		
Label: **C=in**	Adresse:▪**#01160h**	Appel: GOSBVL
Sous-programme ne contenant que l'instruction C=IN		
Label: **Out=CC=in**	Adresse: **#01EECh**	Appel: GOSBVL
Exécute C=IN après avoir effectué OUT=C.		

Label: **ResetONC**	Adresse:▪**#01FBDh**	Appel: GOVLNG
Effectue un redémarrage à chaud, comme celui réalisé par le double appui sur ON et C. Mêmes conséquences sur la pile, PICT, etc...		
Label: **TryToRM?**	Adresse: **#015F0h**	Appel: GOVLNG
Lance la séquence "Try To Recover Memory?"		
Label: **Coma+**	Adresse:▪**#04292h**	Appel: GOVLNG
Plonge la HP48 dans un profond sommeil. Le réveil s'effectue par un appui sur ON. A ce réveil, il y un arrêt-système.		
Label: **BeepDC**	Adresse:▪**#017A6h**	Appel: GOSBVL
Active le buzzer de la HP48, pour une durée de C:A millisecondes, et sur une fréquence de D:A hertz. Pas de son si (flag -56)=1.		
Label: **SpeedA**	Adresse:▪**#018E2h**	Appel: GOSBVL
Place dans A:A la vitesse instantanée de l'horloge. Le nombre qui est obtenu varie aux alentours de #30000h		
Label: **GetTimeC**	Adresse:▪**#012EEh**	Appel: GOSBVL
Charge le contenu de C:W avec l'horloge du système (en ticks)		

L'opération de ***Garbage Collection*** (littéralement "ramassage des ordures") est destinée à nettoyer la mémoire des objets qui ne sont plus référencés (dont l'adresse ne figure plus sur la pile, etc...). Ce coup de balai salutaire peut être utile pour dégager la place que vos données utiliseront en mémoire. Notez que certaines instructions de réservation de mémoire procèdent d'elles mêmes à un Garbage Collection si nécessaire.

Garbage Collection

Label: **GarbColl**	Adresse:▪**#0613Eh**	Appel: GOSBVL
Effectue un Garbage Collection (suppression des objets en mémoire qui ne sont pas référencés, optimisation de l'espace disponible) Après un Garb. Coll., vous pouvez récupérer B,D,D0,D1 par RclRegs.		

Label: **SafeGarb**	Adresse: **#0341Dh**	Appel: GOSBVL
Effectue un Garbage Collection en actualisant les reg. B,D,D0,et D1 puis retranche C:A au registre D:A. (erreur n°1 si carry=1)		

Label: **SafGarbD-1**	Adresse: **#05F0Fh**	Appel: GOSBVL
Effectue un Garbage Collection en actualisant les reg. B,D,D0,et D1 puis retranche 1 au registre D:A. (erreur n°1 si carry=1)		

Label: **SafGarb«**	Adresse: **#05F47h**	Appel: GOVLNG
Effectue un Garbage Collection en actualisant les reg. B,D,D0,et D1 puis retour au Rpl.		

Label: **MoveC0<1**	Adresse:▪**#066B9h**	Appel: GOSBVL
Copie C:A quartets, de la zone se terminant à l'adresse pointée par D0-1, vers la zone se terminant à l'adresse pointée par D1-1. Il faut ici avoir D0<D1. A la fin D0 pointe sur le début de la zone originale et D1 sur le début de la zone copiée. Retour par RTNCC.		

Label: **MoveC0>1**	Adresse:▪**#0670Ch**	Appel: GOSBVL
Copie C:A quartets de la zone commençant à l'adresse pointée par D0 vers la zone commençant à l'adresse pointée par D1. Il faut ici avoir D1<D0. A la fin D0-1 pointe sur le fin de la zone originale et D1-1 sur la fin de la zone copiée. Retour par RTNCC.		

Label: **ClrD1C**	Adresse: **#0675Ch**	Appel: GOSBVL
Met à zéro C:A quartets, à partir de l'adresse pointée par D1. A la sortie D1 pointe juste après la zone balayée. Annule A:W N.B: Si on appelle à l'adresse #0675F, la zone est recouverte par des exemplaires de A:W.		

SÉLECTION DE POINTS D'ENTRÉE

Dans un programme en langage-machine, il est fréquent d'avoir à opérer la réservation d'une certaine quantité de mémoire, sans que cette zone n'empiète sur des données déjà utilisées, et de telle manière que cette zone soit "à l'abri". Les points d'entrées qui suivent vous seront donc utiles (notamment celui de l'adresse #39BEh).

Label: **GetMemC**	Adresse: **#06806h**	Appel: GOSBVL

Place dans C:A la taille mémoire disponible (avant tout Garbage Collection éventuel). Pour celà, fait la différence entre le contenu de #806F8 (sauvegarde de D1) et celui de #806F3 (sauve--garde de B). Le résultat est en quartets. Modifie D0 et A:A.
Le retour de sous-programme se fait par RTNCC.

Label: **AdjMem**	Adresse: **#069F7h**	Appel: GOSBVL

Sauve en #807ED la taille de la mémoire disponible, exprimée en quartets, déduite du résultat ci-dessus par une division par 5.
A la sortie cette valeur est également dans C:A.
Retour par RTNCC. Registres modifiés: A,C,D,D0.

Label: **MakCTmp**	Adresse: **#06AD8h**	Appel: GOSBVL

Réserve une zone de longueur C:A en mémoire temporaire.
A la sortie D0 pointe sur le début de cette zone.
Le bit Carry est armé si la réservation de mémoire a échoué.
Il n'y a de Garbage Collection. Pas d'erreur non plus.

Label: **CResD0**	Adresse: **#039BEh**	Appel: GOSBVL

Crée une zone de C:A quartets en mémoire. A la sortie cette zone est pointée par D0. Si la réservation initiale par MakCtmp échoue il y a un Garbage Collection. Si ce dernier échoue aussi alors il y a une erreur "Insufficient Memory".

Label: **MakC$**	Adresse: **#05B7Dh**	Appel: GOSBVL

Crée une chaîne de caractères, formée de C:A quartets. A la sortie D0 pointe sur le premier caractère, et R0 sur le premier quartet de la zone (donc le 1er quartet de l'adresse-préfixe).
De même, C et D1 pointent en sortie sur le niveau 1. Retour RTNCC.

Label: **Mak2C$**	Adresse: **#05B79h**	Appel: GOSBVL

Comme ci-dessus, mais forme une chaîne de C:A caractères.

Les adresses suivantes opèrent des calculs de CheckSum (codes de redondance cyclique). Ils permettent de vérifier si une zone de données a subi une altération.

Les calculs de CRC sont extrêmement rapides puisqu'ils résultent d'une simple lecture des quartets de la zone à contrôler, le CheckSum étant parallèment actualisé à l'adresse #104h, sur quatre quartets (cette zone doit préalablement être annulée, et il convient de suspendre les interruptions clavier par ST=0 15).

Label: **Crc0A**	Adresse:▪**#05981h**	Appel: GOSBVL

Calcule le CRC (code de redondance cyclique de la zone débutant à l'adresse pointée par D0, et longue de A:A quartets. Le résultat va dans A:A. Modifie C,D1,D0 (qui pointe à la fin sur le premier quartet qui suit la zone lue).
Supprime dès le départ les interruptions-claviers (ST=0 15), et termine par une autorisation de ces interruptions.

Label: **CrcCA**	Adresse:▪**#0597Eh**	Appel: GOSBVL

Exécute D0=C, puis le sous-programme Crc0A ci-dessus.

Label: **ChkCrcLib**	Adresse:▪**#0A00Eh**	Appel: GOSBVL

D0 doit pointer sur le premier quartet de l'adresse-préfixe d'une librairie. Le sous-programme ChkCrcLib vérifie que le CRC de la librairie est correct (c'est-à-dire égal au contenu des quatre derniers quartets).
Le bit Carry est armé si résultat incorrect, et désarmé sinon.
A la sortie C:A pointe sur la taille de la librairie.
Les interruptions-clavier sont interdites pendant le calcul, par ST=0 15, et elles sont permises à la fin.

Label: **Nibbles**	Adresse: **#05912h**	Appel: GOSBVL

Dépile l'objet du niveau 1, sauve les pointeurs par SaveRegs, et place dans A:A (et R0:A) le nombre de quartets occupés par cet objet. Appelle le sous-programme NxtObjD0.
A la sortie, R1 et C:A pointent le début de l'objet initial.

Label: **NxtObjD0**	Adresse:▪**#03019h**	Appel: GOSBVL

Au départ, D0 doit pointer sur un objet quelconque. A la fin, D0 pointe sur l'objet suivant. Modifie D1,D:A,B:A,D1,A:A,C:A.
A condition de sauvegarder la valeur initiale de D0, on peut ainsi calculer le nombre de quartets occupés par l'objet. Ce sous-prg fait la différence entre un objet désigné par une adresse (5 qts) et un objet physiquement présent (adresse préfixe + corps objet).

SÉLECTION DE POINTS D'ENTRÉE

Les sous-programmes **SavRegs** et **RclRegs** sont d'une importance capitale car ils permettent de sauvegarder et de restaurer le contenu des registres B,D,D0,D1, qui sont indispensables au fonctionnement des programmes du Rpl-Système.

Une bonne habitude consiste à sauver les registres B,D,D0,D1 au début de votre code, puis de les restaurer lors du retour au Rpl. Notez aussi que de nombreux points d'entrée décrits dans ce chapitre procèdent eux-mêmes à ces opérations.

Notez enfin que la sauvegarde des registres B,D,D0,D1 s'effectuant à adresses fixes, seule la dernière sauvegarde compte.

Label: **Exit**	Adresse: **#2D564h**	Appel: GOVLNG
Retour au Rpl (A=DAT0 A, D0=D0+ 5, PC=(A))		
Label: **PC=((C))**	Adresse: **#08D75h**	Appel: GOSBVL
Lit A:A à l'adresse pointée par C, puis exécute PC=(A).		
Label: **SavRegs**	Adresse: **#0679Bh**	Appel: GOSBVL
Sauvegarde D0 en #8072F, D1 en #806F8, B en #806F3, et D en #807ED. Le registre C est utilisé pour ces sauvegardes. Retour par RTNCC.		
Label: **DropSr**	Adresse: **#3251Fh**	Appel: GOSBVL
Elimine Obj1 (actualise donc D1 et D) avant d'appeler SavRegs.		
Label: **PopASr**	Adresse: **#3251Ch**	Appel: GOSBVL
Place adresse objet du niveau 1 dans A:A puis appelle DropSr.		
Label: **RclRegs**	Adresse: **#067D2h**	Appel: GOSBVL
Récupère D en #807ED, B en #806F3, D1 en #806F8 et D0 en #8072F. Le registre C est utilisé pour ces rappels. Retour par RTNCC.		
Label: **RclRegs«**	Adresse: **#05143h**	Appel: GOVLNG
Restaure les registres D,B,D1,D0, puis retour au Rpl.		
Label: **RrT«**	Adresse: **#25CE1h**	Appel: GOVLNG
Restaure les registres, place True sur la pile, puis retour Rpl.		
Label: **RrF«**	Adresse: **#26FAEh**	Appel: GOVLNG
Restaure les registres, place False sur la pile, puis retour Rpl.		

Voici les différentes façons de provoquer une erreur (et un retour au Rpl) depuis votre programme en assembleur (je vous conseille l'adresse #10F40h).

Label: **DoErrAC**	Adresse:▪**#05023h**	Appel: GOSBVL
Exécute l'erreur de code C:A. En entrée, le registre A:A doit lui contenir le code d'erreur, car il est écrit en #807F2		

Label: **SafErrC**	Adresse: **#10F40h**	Appel: GOSBVL
Exécute SETHEX, P=0, place C:A dans A:A, rappelle les registres par RclRegs, puis appelle DoErrAC. Pour l'utilisateur, c'est le moyen le plus sûr que le retour au mode d'exécution directe s'effectue dans de bonnes conditions.		

Label: **DoErrCp**	Adresse: **#0501Eh**	Appel: GOVLNG
Annule A:A, place C:P dans A:P puis exécute DoErrAC. A utiliser quand le code d'erreur est compris entre 0 et #F=15.		

Label: **DoErrC**	Adresse:▪**#10F80h**	Appel: GOVLNG
Annule C:A dans A:A, puis exécute DoErrAC.		

Label: **Kill**	Adresse: **#10FDBh**	Appel: GOVLNG
Charge C:A avec #123h puis exécute DoErrC. Interromp le programme en cours d'exécution.		

Label: **Cont**	Adresse: **#1107Bh**	Appel: GOVLNG
Charge C:A avec #13Eh puis exécute DoErrC. Poursuit un programme suspendu par Halt.		

Label: **St10GbRr**	Adresse:▪**#0554Ch**	Appel: GOSBVL
Si le bit 10 de ST est armé, alors erreur "Insufficient Memory". Sinon, Effectue un Garbage Collection, arme le bit 10 de ST, avant de restaurer les registres par RclRegs. Sous-programme souvent appelé par la HP48, de la manière suivante: a) Désarmer le bit 10 de ST. b) Exécuter une procédure. Si la mémoire est insuffisante alors on appelle St10GbRr, puis on essaie de nouveau au point b). Si la mémoire est de nouveau insuffisante, il y a donc une erreur, sinon la procédure s'achève normalement. N.B: on trouve exactement la même routine en #12D68h.		

Label: **St10Gb**	Adresse: **#1633Dh**	Appel: GOSBVL
Comme ci-dessus, à la différence que le Garbage Collection n'est pas suivi par une restauration des registres B,D,D0,D1		

Quelques exemples d'erreurs toutes prêtes...

Label: **RrErrIM**	Adresse: **#065AAh**	Appel: GOVLNG
Restaure les registres par RclRegs, puis "Insufficient Memory"		
Label: **D+CErrIM**	Adresse: **#0343Ah**	Appel: GOVLNG
Rajoute C:A à D:A puis erreur "Insufficient Memory"		
Label: **D=0ErrIM**	Adresse: **#05F26h**	Appel: GOVLNG
Annule D:A puis erreur "Insufficient Memory"		
Label: **RrErrIM**	Adresse: **#039E3h**	Appel: GOSBVL
Restaure les registres par RclRegs, puis erreur "Insufficient Mem"		

```
DoErrPU ("Positive Underflow") en #29DD1h (erreur n° #301h)
DoerrNU ("Negative Underflow") en #29DE1h (erreur n° #302h)
DoerrO  ("Overflow") en #29DF1h (erreur n° #303h)
DoerrUR ("Undefined Result") en #29E01h (erreur n° #304h)
DoerrIR ("Infinite Result") en #29E11h (erreur n° #305h)
DoerrTFA ("Two Few Arguements") en #18CC7h (erreur n° #201h)
DoerrBAT ("Bad Argument Type") en #18CB7h (erreur n° #202h)
DoerrBAV ("Bad Argument Value") en #18CA7h (erreur n° #203h)
DoerrUN ("Undefined Name") en #18C97h (erreur n° #204h)
DoErrIM ("Insufficient Memory") en #4FBBh. (erreur n°1)
DoErrULN ("Undefined Local Name") en #4FD3h. (erreur n°3)
DoErrUXN ("Undefined XLIB Name") en #501Bh. (erreur n°4)
DoErrPL ("Power Lost") en #4FAFh. (erreur n°6)
DoErrICD ("Invalid Card Data") en #4FDFh. (erreur n°8)
DoErrOIU ("Object In Use") en #4FEBh. (erreur n°9)
DoErrPNA ("Port Not Avalaible") en #4FF7h. (erreur n°#A)
DoErrNRP ("No Room in Port") en #5003h. (erreur n°#B)
DoErrONP ("Object Not in Port") en #500Fh. (erreur n°#C)
DoErrCENC ("Can't Edit Null Char.") en #10F59h (erreur n° #102h)
DoErrNCE ("No Current Equation") en #10F79h  (erreur n° #104h)
DoErrIS ("Invalid Syntax") en #10F8Bh  (erreur n° #106h)
DoErrIP ("Invalid PPAR") en #10F9Bh  (erreur n° #12Eh)
DoErrNRR ("Non-Real Result") en #10FABh  (erreur n° #12Fh)
DoErrHNA ("Halt Not Allowed") en #10FCBh  (erreur n° #126h)
DoErrCR ("Circular Reference") en #1101Bh  (erreur n° #129h)
DoErrDNA ("Directory Not Allowed") en #1102Bh  (erreur n° #12Ah)
DoErrNED ("Non-Empty Directory") en #1103Bh  (erreur n° #12Bh)
DoErrID ("Invalid Definition") en #1104Bh  (erreur n° #12Ch)
DoErrML ("Missing Library") en #1105Bh  (erreur n° #12Dh)
DoErrNC ("Name Conflict") en #1106Bh  (erreur n° #13Ch)
```

Les points d'entrée suivants permettent de lire un entier-système sur la pile, ou au contraire d'y écrire un entier-système. A la page suivante, c'est la même chose pour des nombres réels.

Label: **Pop<>A**	Adresse:▪**#06641h**	Appel: GOSBVL
Place dans A:A les 5 quartets q5-q9 de Obj1, et élimine Obj1 Utile si Obj1 est un entier-système (alors A:A contient la valeur de cet entier) ou si q5-q9 désignent la longueur de Obj1.		
Label: **Pop<>AC**	Adresse:▪**#03F5Dh**	Appel: GOSBVL
Place dans A:A les 5 quartets q5-q9 de Obj2, et dans C:A les 5 quartets q5-q9 de Obj1. Obj2 et Obj1 sont éliminés. Utile si Obj2 et Obj1 sont des entiers-système.		
Label: **Push0<>**	Adresse:▪**#06537h**	Appel: GOSBVL
Place sur la pile l'entier-système **n** dont la valeur est le contenu du registre R0 (champ A). Si **n**<#2C, **n** est basé en ROM.		
Label: **Push0<>«**	Adresse:▪**#0357Fh**	Appel: GOVLNG
Appel de Push0<>, puis retour au Rpl.		
Label: **Push0<>¿«**	Adresse:▪**#2E31Fh**	Appel: GOVLNG
Appelle Push0<>, place False sur la pile, puis retour au Rpl.		
Label: **PushA<>«**	Adresse:▪**#0357Ch**	Appel: GOVLNG
Place A dans R0, puis appel de Push<>«.		
Label: **Push0<>?«**	Adresse:▪**#036F7h**	Appel: GOVLNG
Appel de Push0<>, place True au niveau 1, puis retour au Rpl.		
Label: **Push01<>**	Adresse:▪**#06529h**	Appel: GOSBVL
Place sur la pile l'entier-système n dont la valeur est le contenu de R0:A (au niveau 2) et l'entier-système m dont la valeur est le contenu de R1:A (au niveau 1). Mêmes remarques que ci-dessus si n<#2C ou si m<#2C.		
Label: **Push01<>«**	Adresse:▪**#03F14h**	Appel: GOVLNG
Appel de Push01<>, puis retour au Rpl, sans appel de RclRegs.		
Label: **PushAC<>«**	Adresse: **#03F0Eh**	Appel: GOVLNG
Place A dans R0, C dans R1, puis exécute Push01<>«.		

Label: **Pop%ASr**	Adresse:▪**#29FDAh**	Appel: GOSBVL

Charge A:W avec les 16 quartets formant le réel du niveau 1:
(A)=<s><m0..m11><e1..e3> (signe, 12 qts de mantisse, 3 d'expos.)
Ce réel est éliminé. Le registre C:A est modifié. Exécute SETDEC.
Termine en sauvant les registres B,D,D0,D1 par un appel à SavRegs.

Label: **Pop%ACSr**	Adresse:▪**#2A002h**	Appel: GOSBVL

Charge A:W avec les 16 quartets du réel niveau 2, et C:W avec les 16 quartets du réel niveau 1 (voir ci-dessus). Ces deux réels sont enlevés de la pile. Sauve la valeur finale des registres B,D,D0,D1. Exécute SETDEC. Retour par RTNCC. Modifie D:A.

Label: **SepA%**	Adresse:▪**#2BC4Ah**	Appel: GOSBVL

Au départ A contient les 16 quartets d'un réel (voir Pop%Asr).
A la fin, (A)=<s><...><e1..e5>, et (B)=<0><m0..m11><000>, où <e1..e5> est une conversion de l'exposant sur 5 quartets.
N.B: Le mode DEC est sélectionné.

Label: **ReunAB%**	Adresse:▪**#29E46h**	Appel: GOSBVL

C'est le contraire de SepA%. Réunit donc dans A:W les 16 quartets formant un réel, à partir du contenu de A et de B.
Pensez à faire suivre par l'instruction SETHEX.

Label: **Pop%SrSep**	Adresse:▪**#29FD0h**	Appel: GOSBVL

Appelle Pop%ASr, puis exécute SepA%

Voici des séquences effectuant une opération sur le réel r pris sur la pile, et maintenant séparé sur A,B.
Dans toutes ces séquences, le bit XM est désarmé au départ. Il sert de flag pour tracer les erreurs. S'il est armé, c'est qu'une erreur s'est produite. Les séquences ci-dessous contiennent le traitement de ces erreurs.
Le résultat est le réel contenu dans A:W (16 quartets)
GOSBVL #2AEBEh, GOSBVL #2AEC5h, GOSBVL #29E21h: inverse de r.
GOSBVL #2B9F3h, GOSBVL #2AEC5h, GOSBVL #29E21: racine carrée de r≥0
GOSBVL #2B6AAh, GOSBVL #29E21h: exponentielle de r.
GOSBVL #2B698h, GOSBVL #2AEC5h, GOSBVL #29E21h: Log népérien de r≥0
GOSBVL #2B6A1h, GOSBVL #2AEC5h, GOSBVL #29E21h: Log décimal de r≥0
GOSBVL #2AEF6h, GOSBVL #2B6E0h, GOSBVL #29E21h: calcule sin(r)
GOSBVL #2AEF6h, GOSBVL #2B6E9h, GOSBVL #29E21h: calcule cos(r)
GOSBVL #2AEF6h, GOSBVL #2B6F2h, GOSBVL #2AEC5h, GOSBVL #29E21h: calcule tan(r)
GOSBVL #2AEF6h, GOSBVL #2B704h, GOSBVL #2AEC5h, GOSBVL #29E21h: calcule arcsin(r), -1≤r≤1
GOSBVL #2AEF6h, GOSBVL #2B70Dh, GOSBVL #2AEC5h, GOSBVL #29E21h: calcule arccos(r), -1≤r≤1
GOSBVL #2AEF6h, GOSBVL #2B6FBh, GOSBVL #29E21h: calcule atan(r)
HS=0 3, GOSBVL #2B716, GOSBVL #29E21h: calcule fact(r) (r≥0)

Voici les adresses qu'utilise la HP48 pour lire deux réels sur la pile, ou pour les y écrire, ou encore pour effectuer telle ou telle opération sur ces deux réels.

Label: **SepAC%**	Adresse: **#2BCA0h**	Appel: GOSBVL

Au départ A et C contiennent les 16 quartets de deux réels *rA,rB*.
rA séparé sur A et B, et *rC* est séparé sur C et D (voir ci-dessus)
Le mode DEC est sélectionné.

Label: **Pop%ACSrSep**	Adresse: **#29FF8h**	Appel: GOSBVL

Appelle Pop%ACSr puis exécute SepAC% .

Label: **Comp2%**	Adresse: **#2BD76h**	Appel: GOSBVL

Compare le réel r1 "séparé" sur A et B, et le réel r2 "séparé" sur C et D. En entrée, le registre P doit contenir 1 ou 4. Si P=1 le test est (r1<r2?). Si P=4 c'est le test (r1>r2?). Ne modifie pas les registres D0,D1, et R0 à R4.
Le bit CARRY est armé si le test renvoie la valeur VRAI.
N.B: remettre P=0 et le mode HEX à la sortie.

Voici des séquences effectuant une opération sur les 2 réels r1, r2 pris sur la pile, maintenant séparés sur A,B (r1) et C,D (r2):
Dans toutes ces séquences, le bit XM est désarmé au départ. Il sert de flag pour tracer les erreurs. S'il est armé, c'est qu'une erreur s'est produite. Les séquences ci-dessous contiennent le traitement de ces erreurs (GOSBVL #29E21h)
Le résultat est un réel placé dans A:W (16 quartets)
La séquence HS=0 3, GOSBVL #2B91Eh, GOSBVL #29E21h effectue le le produit de r1 et r2.
La séquence HS=0 3, GOSBVL #2B977h, GOSBVL #2AEC5h, GOSBVL #29E21h divise le réel r1 par le réel r2.
La séquence GOSBVL #2B67Dh, GOSBVL #29E21h calcule r1 mod r2.
La séquence HS=0 3, GOSBVL #2B7DCh, GOSBVL #29E21h effectue la somme des deux réels r1 et r2.
La séquence C=-C-1 S, HS=0 3, GOSBVL #2B7DCh, GOSBVL #29E21h calcule la différence r1-r2.
La séquence GOSBVL #2B728h, GOSBVL #2AEC5h, GOSUBL #29E21h calcule le réel comb(r1,r2).
La séquence GOSBVL #2B71Fh, GOSBVL #2AEC5h, GOSUBL #29E21h calcule le réel perm(r1,r2).
La séquence HS=0 3, GOSBVL #2AAC0h, GOSBVL #2AEC5h, GOSBVL #29E21h calcule r1^r2.

Label: **PushA%Rr**	Adresse: **#2A188h**	Appel: GOSBVL

Place sur la pile le réel contenu dans A (voir notations Pop%ASr)
Termine en restaurant les registres par un appel à RclRegs.
(Entre temps, ces registres ont été actualisés, sauf D0).
N.B: si le réel est compris entre -9 et 9, on le pointe en ROM.

Label: **PushA%Rr«**	Adresse: **#2A23Dh**	Appel: GOVLNG

Comme ci-dessus, puis retour au Rpl.

Label: **Get1%AB**	Adresse: ▪**#2BFE3h**	Appel: GOSBVL

On suppose que D1 pointe sur le premier quartet d'un réel.
Si ce réel s'écrit <p1..p5><q1..q15><s>, (5 quartets-préfixe, 15 quartets d'exposant et de mantisse, 1 quartet de signe), alors à la sortie: A=<s><....><p5..p1>, B=<0><q15..q1>, et le registre D1 pointe sur le premier quartet qui suit le réel.

Label: **Get0%AB**	Adresse: ▪**#2BFFDh**	Appel: GOSBVL

Comme ci-dessus, en remplçant D1 par D0.

Label: **Get1%CD**	Adresse: **#2C017h**	Appel: GOSBVL

Comme Get1%AB, en remplaçant A par C et B par D.

Label: **Get0%CD**	Adresse: ▪**#2C031h**	Appel: GOSBVL

Comme Get1%CD, en remplaçant A par C et B par D.

Label: **PutAB%1**	Adresse: **#2C066h**	Appel: GOSBVL

Si A=<s><....><p5..p1> et B=<0><q15..q1>, alors écrit, à l'adresse pointée par D1, les 21 quartets <p1..p5><q1..q15><s>.
Cette opération est donc le contraire de Get1%AB.
A la sortie le registre D1 pointe sur le premier quartet qui suit cette zone de 21 quartets. A et B restent inchangés.

Label: **PutAB%0**	Adresse: ▪**#2C04Bh**	Appel: GOSBVL

Comme PutAB%1, mais en remplaçant D1 par D0.

Label: **PopCarry**	Adresse: ▪**#61A02h**	Appel: GOSBVL

Arme le bit CARRY si Obj1=True, et élimine Obj1.

Label: **SzStrC**	Adresse: ▪**#2FFBAh**	Appel: GOSBVL

D1 pointe sur le début d'une chaîne de caractère. Place dans C:A le nombre de caractères de cette chaîne. A la sortie D1 pointe sur le 1er caractère de cette chaîne.

Label: **SzStkStrC**	Adresse: ▪**#2FFB4h**	Appel: GOSBVL

Comme ci-dessus, avec la chaîne placée au niveau 1 de la pile.
Cette chaîne n'est pas retirée de la pile. NB la modification de D1

Label: **Pop#ASr**	Adresse: **#53F8Dh**	Appel: GOSBVL
Charge A avec les quartets de l'entier binaire du level 1, dans la limite du nombre de bits défini par le dernier STWS. Sauve les registres après le dépilage de cet entier.		
Label: **Pop#ACSr**	Adresse: **#53F77h**	Appel: GOSBVL
Charge A avec l'entier binaire du niveau 2, et C avec celui du niveau 1 (voir ci-dessus). sauve registres après double dépilage.		
Label: **PushA#**	Adresse: **#5422Ch**	Appel: GOSBVL
Place sur la pile un entier binaire dont le contenu est A:WP. Le nombre de chiffres de cet entier est donc P+1. Les registres sont rétablis par RclRegs avant le placement sur la pile. Le contenu de P est remis à 0 avant le retour du sous-prog.		
Label: **PushA#«**	Adresse:▪**#0596Dh**	Appel: GOVLNG
Comme ci-dessus, puis retour au Rpl.		
Label: **MulAC#**	Adresse:▪**#53EE4h**	Appel: GOSBVL
Produit des contenus des registres A:W et C:W. Résultat dans A:W		

Label: **Ad1=T«**	Adresse:▪**#62080h**	Appel: GOVLNG
P=0, puis copie l'adresse de True sur Adr1, puis retour Rpl.		
Label: **RrAd1=T«**	Adresse:▪**#62076h**	Appel: GOVLNG
Restaure registres D1,D0,D,C puis exécute Ad1=T«		
Label: **Ad1=F«**	Adresse:▪**#620A0h**	Appel: GOVLNG
P=0, puis copie l'adresse de False sur Adr1, puis retour Rpl.		
Label: **RrAd1=F«**	Adresse:▪**#62096h**	Appel: GOVLNG
Restaure registres D1,D0,B,D puis appelle Ad1=F«		
Label: **Ad1#Cy«**	Adresse:▪**#6207Dh**	Appel: GOVLNG
Si Carry=0 alors appelle Ad1=T«, sinon appelle Ad1=F«		
Label: **Ad1=Cy«**	Adresse:▪**#6209Dh**	Appel: GOVLNG
Si Carry=1 alors appelle Ad1=T«, sinon appelle Ad1=F«		
Label: **RrAd1=Cy«**	Adresse:▪**#62073h**	Appel: GOVLNG
Restaure les registres, puis appelle Ad1=Cy«.		

Label: **PushT«**	Adresse:▪**#620C3h**	Appel: GOVLNG
P=0, place True sur la pile, et retour au Rpl.		
Label: **RrPushT«**	Adresse:▪**#620B9h**	Appel: GOVLNG
Restaure registres D0,D1,B,D, puis appel de PushT«.		
Label: **PushF«**	Adresse:▪**#620DCh**	Appel: GOVLNG
P=0, place False sur la pile, et retour au Rpl.		
Label: **RrPushF«**	Adresse:▪**#620D2h**	Appel: GOVLNG
Restaure registres D0,D1,B,D, puis appel de PushF«.		
Label: **PushCy«**	Adresse:▪**#620D9h**	Appel: GOVLNG
D'abord P=0; Si Carry=1 place True sur la pile, sinon place False, et retour au Rpl.		
Label: **Push#Cy«**	Adresse:▪**#620C0h**	Appel: GOVLNG
D'abord P=0; Si Carry=0 place True sur la pile, sinon place False, et retour au Rpl.		
Label: **RrPushCy«**	Adresse:▪**#620B6h**	Appel: GOVLNG
Restaure les registres, puis appelle PushCy« (avec la valeur du bit Carry antérieure à la restauration des registres)		

Label: **PushA«**	Adresse:▪**#03A86h**	Appel: GOVLNG
Place le contenu de A:A comme adresse de Obj1, puis retour au Rpl. On trouve exactement la même routine en #03AC5h.		
Label: **RrPushA**	Adresse:▪**#54266h**	Appel: GOSBVL
P=0, puis place le contenu de A:A comme adresse de Obj1. Utilise le registre #80734h comme sauvegarde de A en cas de Garbage Collect.		
Label: **Push(C)«**	Adresse:▪**#08D66h**	Appel: GOVLNG
Lit A:A à l'adresse pointée par C, puis exécute PushA«.		
Label: **RrCopyA«**	Adresse:▪**#03672h**	Appel: GOVLNG
Restaure D0,D1,C,D par RclRegs, puis copie A:A sur Adr1, en remplacement de l'adresse précédente. Enfin, retour au Rpl.		
Label: **RrCopy0«**	Adresse:▪**#0366Fh**	Appel: GOVLNG
Place le contenu de R0 dans A, puis appel de RrCopyA«.		

Label: **A+5P4**	Adresse: **#2BE2Dh**	Appel: GOSBVL
P=4, puis ajoute 5 au champ P de A. Ne remet pas P à 0.		

Label: **A+5P**	Adresse: **#2BE2Fh**	Appel: GOSBVL
Ajoute 5 au champ P de A. Rappel: l'instruction "A=A+*n field*" ne reconnait que les champs X,S,M,B,W,A. N.B: #2BE32h: ajoute 4, #2BE35h: ajoute 3, #2BE38h: ajoute 2		

Label: **C+5P4**	Adresse: **#2BE40h**	Appel: GOSBVL
P=4, puis ajoute 5 au champ P de C. Ne remet pas P à 0.		

Label: **C+5P**	Adresse: **#2BE42h**	Appel: GOSBVL
Ajoute 5 au champ P de C. Rappel: l'instruction "C=C+*n field*" ne reconnait que les champs X,S,M,B,W,A. N.B: #2BE45h: ajoute 4, #2BE48h: ajoute 3, #2BE4Bh: ajoute 2		

Label: **AsrW5**	Adresse:▪**#0D5E5h**	Appel: GOSBVL
Divise A:W 5 fois de suite par 16 (5 décal. à droite d'un quartet) NB: #0D5E8h => AsrW4, #0D5EBh => AsrW3		

Label: **AslW5**	Adresse:▪**#0D5F6h**	Appel: GOSBVL
Multiplie A:W 5 fois de suite par 16 (5 décal. à gche d'un quartet) NB: #0D5F9h => AslW4, #0D5FCh => AslW3		

Label: **CsrW5**	Adresse:▪**#0D607h**	Appel: GOSBVL
Divise C:W 5 fois de suite par 16 (5 décal. à droite d'un quartet) NB: #0D60Ah => CsrW4, #0D60Dh => CsrW3		

Label: **CslW5**	Adresse:▪**#0D618h**	Appel: GOSBVL
Multiplie C:W 5 fois de suite par 16 (5 décal. à gche d'un quartet) NB: #0D61Bh => CslW4, #0D61Eh => CslW3		

Label: **Cdiv5**	Adresse: **#06A8Eh**	Appel: GOSBVL
Divise le contenu de C:A par 5. A,C,D sont modifié sur q0-q10. N.B: pour que l'opération soit exacte C:6-15 doit être nul.		

Label: **ACmultB**	Adresse:▪**#03991h**	Appel: GOSBVL
Effectue le produit de A:A et C:A. Résultat dans B:A.		

Label: **ACdiv**	Adresse:▪**#03F24h**	Appel: GOSBVL
Effectue la division de A:A par C:A. Le quotient entier est dans C:A et le reste dans la division est dans A:A. Les registres B:A et D:A sont utilisés.		

Label: **ExABCD**	Adresse:▪**#2BE53h**	Appel: GOSBVL
Echange les registres A et C, et les registres B et D (champs W)		
Label: **SavA0B1**	Adresse:▪**#2BE61h**	Appel: GOSBVL
Sauvegarde A dans R0 et B dans R1 (champs W)		
Label: **SavA2B3**	Adresse:▪**#2BE6Fh**	Appel: GOSBVL
Sauvegarde A dans R2 et B dans R3 (champs W)		
Label: **SavC0D1**	Adresse:▪**#2BE7Dh**	Appel: GOSBVL
Sauvegarde C dans R0 et D dans R1 (champs W)		
Label: **SavC2D3**	Adresse:▪**#2BE8Bh**	Appel: GOSBVL
Sauvegarde C dans R2 et D dans R3 (champs W)		
Label: **ExA0B1**	Adresse:▪**#2BE99h**	Appel: GOSBVL
Echange les registres A et R0, et les registres B et R1 (champs W)		
Label: **ExA2B3**	Adresse:▪**#2BEA7h**	Appel: GOSBVL
Echange les registres A et R2, et les registres B et R3 (champs W)		
Label: **RclA0B1**	Adresse:▪**#2BEB5h**	Appel: GOSBVL
Rappelle dans A le contenu de R0, et dans B celui de R1 (champs W)		
Label: **RclA2B3**	Adresse:▪**#2BEC0h**	Appel: GOSBVL
Rappelle dans A le contenu de R2, et dans B celui de R3 (champs W)		
Label: **RclC0D1**	Adresse:▪**#2BECBh**	Appel: GOSBVL
Rappelle dans C le contenu de R0, et dans D celui de R1 (champs W)		
Label: **RclC2D3**	Adresse:▪**#2BED6h**	Appel: GOSBVL
Rappelle dans C le contenu de R2, et dans D celui de R3 (champs W)		

Label: **ChkXmit**	Adresse: **#311B3h**	Appel: GOSBVL

P=0, puis vérifie que le bit 4 du quartet (#110) est à 1 (émission autorisée sur RS232). A la sortie Carry est désarmé si tout est ok, et armé si émission interdite (avec (C:A)=#C09h, code de l'erreur "Port Closed"). Modifie D0 et C:A.

Label: **LenArD1**	Adresse: **#03968h**	Appel: GOSBVL

Calcule le nombre d'éléments du tableau pointé par D1. Le résultat est dans A:A. Modifie D1, C:A, D:A, B:A.

Label: **D01<>C**	Adresse: **#07831h**	Appel: GOSBVL

Vérifie si les C:X quartets situés à l'adresse pointée par D0 sont égaux à ceux pointés par D1. Arme Carry si différence trouvée.
Modifie D0, D1, C:W, A:W, B:B.
A la fin D0 pointe juste après sa zone de vérification.

Label: **D01<>WP**	Adresse: **#0786Fh**	Appel: GOSBVL

Vérifie si les P quartets situés à l'adresse pointée par D0 sont égaux à ceux pointés par D1. Arme Carry si différence trouvée.
A la sortie D0 a été incrémenté de P quartets, et D1 est inchangé.
A:WP, C:WP sont modifiés. P=0 est rétabli à la fin.

Label: **ChkLibs**	Adresse: **#077B2h**	Appel: GOSBVL

Dépile l'entier-système N du au niveau 1 (appel de PopA<> en #6641) sauve registres par SaveRegs, et vérifie le nombre de librairies attachées à HOME.
A la sortie, D1 pointe sur HOME, D0 sur le n° de la 1ère librairie attachée à HOME, B:A contient le nombre L de libs attachées, et A:X contient le ET logique entre #7FF et N.

Label: **ChkLibsA**	Adresse: **#077B8h**	Appel: GOSBVL

Comme ci-dessus, l'entier N étant pris dans A:A.

Label: **PushXLIB**	Adresse: **#07ED8h**	Appel: GOSBVL

Place sur la pile le nom XLIB correspondant à la librairie R2:A, et au numéro d'objet R1:A. Retour par RclRegs.

Pour terminer, deux points d'entrée pour rechercher le contenu d'une variable locale ou globale donnée par son nom.

Label: **LrclR0A**	Adresse: **#07DA2h**	Appel: GOSBVL

```
Le registre R0:A doit pointer sur le début du corps d'un nom local,
  juste après l'adresse-préfixe. Ce sous programme cherche si ce
  nom local existe actuellement.
Si la recherche est infructueuse, alors le bit Carry est armé. Si
  le nom est trouvé, Carry est désarmé et le registre D0 pointe
  sur le début du contenu de cette variable.
```

Label: **GrclR01A**	Adresse: **#0788Eh**	Appel: GOSBVL

```
Le registre R0:A doit pointer sur le début du corps d'un nom global
  juste après l'adresse-préfixe. Ce sous programme cherche si ce
  nom global existe dans le répertoire pointé par R1:A.
Si la recherche est infructueuse, alors le bit Carry est armé. Si
  le nom est trouvé, Carry est désarmé et le registre D0 pointe
  sur le début du contenu de cette variable.
Pour effectuer la recherche dans le répertoire courant, effectuez
  préalablement: D0=(5) #8071Bh, A=DAT0 A, R1=A.
```

EXEMPLES DE PROGRAMMES EN ASSEMBLEUR

Les pages qui suivent, et qui vont terminer ce tome 2, contiennent quelques programmes simples en langage d'assemblage.

Ces exemples sont inspirés de programmes écrits en Externals dans la première partie de ce livre. C'est l'occasion de comparer les deux langages, tant sur le plan de la commodité que la rapidité. Pour ce dernier point, il n'y a pas photo: l'assembleur est évidemment ultra rapide. Néanmoins les temps de développement, s'ils sont pris en compte, équilibrent les mérites des deux langages.

Les listings en assembleur présentés ici ont tous été obtenus avec le programme **ASMPC**.

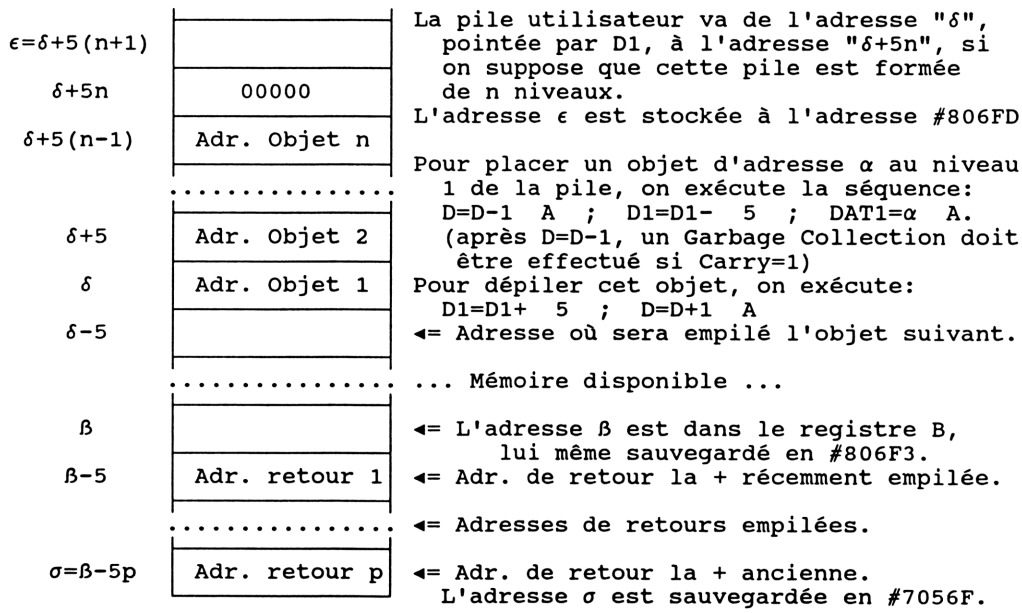
EXEMPLE 1: Inverser les n premiers niveaux de la pile.

Nous allons voir ici une version assembleur du programme **REVST** qui vous a été présenté (en RPL-utilisateur et en Externals) à la page 17 de ce livre.

Cet exemple nous permettra de "réviser" un certain nombre de points importants concernant la pile-utilisateur, et la position qu'elle occupe en mémoire. En particulier:

- La pile-utilisateur est formée des adresses de chacun des objets auxquels elle permet d'accéder. Elle se développe en direction des adresses basses.
- L'adresse de l'objet du niveau 1 est stockée dans le registre **D1** (lui même sauvegardé à l'adresse **#806F8**).
- La mémoire disponible s'étend entre le haut de la pile des retours du Rpl-système (adresse stockée dans le registre **B**, lui-même sauvegardé en **#806F3**), et le bas de la pile-utilisateur (adresse du niveau 1).
- La taille de la mémoire disponible, exprimée en adresses (blocs de 5 quartets) est stockée dans le registre **D**, lui-même sauvegardé à l'adresse **#807ED**.

On peut représenter les choses de la manière suivante:



On voit donc comment les adresses lues en **#806FD** et **#806F8** permettent d'encadrer la pile-utilisateur, et de connaître en particulier le nombre de quartets (puis le nombre de niveaux) qu'elle utilise. Quand la pile est vide, **D1** pointe sur l'adresse **ε-5** (notations ci-dessus) qui contient 5 quartets nuls.

Dans cet exemple, nous voulons inverser les **n** premiers objets de la pile, où **n** est un entier placé au niveau 1 (et qui ne doit pas être comptabilisé parmi les **n** objets à inverser). Ici **n** sera attendu sous la forme d'un entier-système.

Si **<n>** est supérieur au nombre d'objets réellement présents sur la pile, il doit y avoir une erreur "*Two Few Arguments*". Au contraire, si **<n>** est égal à 0 ou 1, il ne doit rien se passer.

Voici une solution, au format de **ASMPC**:

```
   GOSBVL   #6641     *Dépile entier-système <n>, qui va dans A:A
   LC(5)    #FFFF
   ?C<A     A         *<n> est-il supérieur ou égal à #10000h ?
   GOYES    TFA       *Si oui, erreur "Two Few Arguments"
   GOSBVL   #679B     *Sauvegarde B,D,D0,D1
   C=A      A         *(C:A)=n
   A=A+A    A         *(A:A)=2n
   A=A+A    A         *(A:A)=4n
   A=C+A    A         *(A:A)=5n (n niveaux => 5n quartets)
   CD1EX              *(C:A)= adresse du niveau 1 de la pile.
   D1=C               *D1 garde sa valeur
   C=C+A    A         *(C:A)= adresse du niveau n+1 de la pile.
   GOC      TFA       *Si retenue, erreur "Two Few Arguments"
   D0=(5)   #806FD
   A=DAT0   A         *(A:A)= adresse après zone de la pile.
   A=A-5    A         *Pointe les 5 qts nuls après dernier objet
   ?C>A     A         *La pile a moins de <n> niveaux occupés ?
   GOYES    TFA       *Si oui, "Two Few Arguments"
   D0=C
   D0=D0-   5         *D0 pointe sur le dernier niveau utile
   AD1EX              *A pointe sur le niveau 1 de la pile
   D1=A               *D1 garde sa valeur
   C=C-A    A
   B=C      A         *(B:A)= 5x(nombre k de niveaux utiles)
:LOOP
   B=B-10   A         *Décrémente B de 10 (10 qts = 2 niveaux)
   GOC      FIN       *Si retenue, c'est fini.
   A=DAT1   A         *Lit adresse niveau j
   C=DAT0   A         *Lit adresse niveau k+1-j
   DAT1=C   A
   DAT0=A   A         *On vient de procéder à l'échange.
   D1=D1+   5
   D0=D0-   5         *D0 et D1 pointent sur niveaux suivants
   GOTO     LOOP      *On recommence...
:FIN
   GOVLNG   #5143     *Restaure registres B,D,D0,D1 et retour Rpl
:TFA
   LC(5)    #201      *Code de l'erreur "Two Few Arguments"
   GOVLNG   #10F40    *RclRegs et erreur de code (C:A)
```

La solution précédente est suffisamment commentée pour qu'il soit superflu d'en rajouter.

On peut maintenant comparer les performances, en termes de vitesse (sur une pile de 500 objets) et de taille, pour les trois versions de **RVST**:

- Version Rpl "normal": 31.5 octets; temps: 5.997 secondes
- Version en Externals: 12.5 octets; temps: 3.368 secondes
- Version en Assembleur: 72.0 octets; temps: 0.019 secondes

Evidemment la comparaison des temps d'exécution est impressionnante. On constate que celà se paye un peu par la taille du programme. De la même manière, le temps de développement, plus long en Assembleur qu'en Externals et surtout qu'en RPL-utilisateur doit être pris en compte. Chaque langage aura donc toujours ses partisans, tous pour d'excellentes raisons.

A titre indicatif, voici le fichier-listing de ce programme, si vous choisissez cette possibilité dans **ASMPC** (remarquez les quartets de sauts remplacés par des ? quand le label n'est pas déclaré. Rassurez vous, le fichier-objet généré par **ASMPC** est correct).

```
Début de l'assemblage du fichier revst.asm
00000: 8F14660      GOSBVL      #6641     *Dépile entier-système <n>, qui va ...
00007: 34FFFF0      LC(5)       #FFFF
0000E: 8B6          ?C<A        A         *<n> est-il supérieur ou égal à #1000h
00011: ??           GOYES       TFA       *Si oui, erreur "Two Few Arguments"
00013: 8FB9760      GOSBVL      #679B     *Sauvegarde B,D,D0,D1
0001A: D6           C=A         A         *(C:A)=n
0001C: C4           A=A+A       A         *(A:A)=2n
0001E: C4           A=A+A       A         *(A:A)=4n
00020: CA           A=C+A       A         *(A:A)=5n (n niveaux => 5n quartets)
00022: 137          CD1EX                 *(C:A)= adresse du niveau 1 de la pile
00025: 135          D1=C                  *D1 garde sa valeur
00028: C2           C=C+A       A         *(C:A)= adresse du niveau n+1 de la...
0002A: 4??          GOC         TFA       *Si retenue, erreur "Two Few Arguments
0002D: 1BE7507      D0=(5)      #806FD
00034: 142          A=DAT0      A         *(A:A)= adresse après zone de la pile.
00037: 818F84       A=A-5       A         *Pointe sur les 5 quartets nuls apr...
0003D: 8B2          ?C>A        A         *La pile a-t-elle moins de <n> nive...
00040: ??           GOYES       TFA       *Si oui, "Two Few Arguments"
00042: 134          D0=C
00045: 184          D0=D0-      5         *D0 pointe sur le dernier niveau utile
00048: 133          AD1EX                 *A pointe sur le niveau 1 de la pile
0004B: 131          D1=A                  *D1 garde sa valeur
0004E: E2           C=C-A       A
00050: D5           B=C         A         *(B:A)= 5x(nombre K de niveaux utiles)
:LOOP
00052: 818F99       B=B-10      A         *Décrémente B de 10 (10 qts = 2 niv...
00058: 4??          GOC         FIN       *Si retenue, c'est fini.
0005B: 143          A=DAT1      A         *Lit adresse niveau j (j a...
0005E: 146          C=DAT0      A         *Lit adresse niveau k+1-j
00061: 145          DAT1=C      A
00064: 140          DAT0=A      A         *On vient de procéder à l'échange.
00067: 174          D1=D1+      5
0006A: 184          D0=D0-      5         *D0 et D1 pointent sur niveaux suiv...
0006D: 64EF         GOTO        LOOP      *On recommence...
:FIN
00071: 8D34150      GOVLNG      #5143     *Restaure registres B,D,D0,D1 et re...
:TFA
00078: 3410200      LC(5)       #201      *Code de l'erreur "Two Few Arguments"
0007F: 8D04F01      GOVLNG      #10F40    *RclRegs et erreur de code (C:A)
Fin de l'assemblage du fichier revst.asm
0 erreur(s) détectée(s)
```

EXEMPLE 2: Ne conserver que les niveaux 1 à n de la pile.

C'est une version assembleur du programme **KEEP** (cf page 19). La syntaxe est **<n> KEEP**, où **<n>** est un entier-système. A la sortie, la pile ne doit contenir que les niveaux **1** à **n** initiaux (les autres sont éliminés). Si **<n>**=0, la pile doit être vidée. Si **<n>** est supérieur à la taille de la pile, il y a une erreur "***Two Few Arguments***".

Le schéma utilisé précédemment pour représenter la pile en mémoire devrait vous être utile pour comprendre le programme ci-dessous. Dans la solution adoptée, on utilise deux points d'entrée, d'adresses respectives **#66B9h** et **#69F7h**.

- Le premier de ces points d'entrée (cf page 349) effectue la copie de (**C:A**) quartets. Au départ **D0** doit pointer juste après la zone source, et **D1** juste après la zone cible.
  On doit avoir **D0**≤**D1**. La copie se fait "à reculons". A la fin, **D1** pointe sur le premier quartet de la zone cible et **D0** sur celui de la zone source.
  Le registre **D** n'est pas affecté dans cette opération (utile pour ce qui suit).
- Le second point d'entrée (voir page 350) actualise la taille mémoire disponible exprimée en adresses (le contenu habituel de **D**) et sauvegardée à l'adresse **#807EDh**. Ce calcul se fait à partir du contenu des adresses **#806F8h** (sauvegarde de **D1**) et **#806F3h** (sauvegarde de **B**).

```
   GOSBVL   #6641    *Dépile entier-système <n>, qui va dans A:A
   LC(5)    #FFFF
   ?C<A     A        *<n> est-il supérieur ou égal à #10000h ?
   GOYES    TFA      *Si oui, erreur "Two Few Arguments"
   GOSBVL   #679B    *Sauvegarde B,D,D0,D1
   C=A      A        *(C:A)=n
   A=A+A    A        *(A:A)=2n
   A=A+A    A        *(A:A)=4n
   C=C+A    A        *(C:A)=5n (n niveaux => 5n quartets)
   RSTK=C            *Sauvegarde C:A dans le registre RSTK
   AD1EX             *(A:A)= adresse du niveau 1 de la pile
   A=C+A A           *(A:A)= adresse du niveau n+1 de la pile
   GOC   TFA         *Si retenue, erreur "Two Few Arguments"
   D0=A              *Pointe donc sur le niveau n+1
   D1=(5)   #806FD
   C=DAT1   A        *(C:A)= adresse après zone de la pile
   C=C-5    A
   D1=C              *Pointe donc sur les 5 qts nuls après pile
   ?C<A     A        *La pile a-t-elle moins de <n> niveaux ?
   GOYES    TFA      *Si oui, erreur "Two Few Arguments"
   C=RSTK            *Récupère (C:A)=5n
   GOSBVL   #66B9    *Déplace le contenu de la pile
   CD1EX             *C pointe sur le nouveau niveau 1
   D0=(5)   #806F8   *Registre où est stockée l'adresse du niveau 1
   DAT0=C   A        *Actualise ce registre
   GOSBVL   #69F7    *Actualise mémoire disponible en #807ED
   GOVLNG   #5143    *Restaure registres B,D,D0,D1 et retour Rpl
:TFA
   LC(5)    #201     *Code de l'erreur Two Few Arguments
   GOVLNG   #10F40   *Restaure registres et erreur de code (C:A)
```

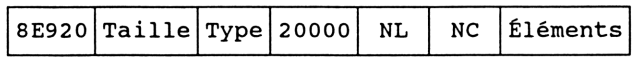
EXEMPLE 3: Inverser deux lignes d'une matrice.

On a vu, page 20 et suivantes, un programme baptisé **ECHL** et permettant d'échanger le contenu de deux lignes d'une matrice.

Dans la version en assembleur que nous allons voir ici, la syntaxe sera: **T <m> <n> ECHL**, où **T** est une matrice, à coefficients réels ou complexes, et où **<m>** et **<n>** sont deux entiers-système désignant les numéros de ligne à échanger. A la sortie, seule doit subsister la matrice sur la pile.

On a vu, dans la version "Externals" de **ECHL**, qu'il existe une procédure réalisant ce travail, directement implantée en langage-machine. Il n'est évidemment pas question de la recopier. La solution adoptée ici sera différente (sans l'être à 100%, bien sûr).

Avant toute chose, il faut se remémorer la façon dont est codée une matrice de réels ou complexes, en mémoire. Un tel objet se présente de la manière suivante:



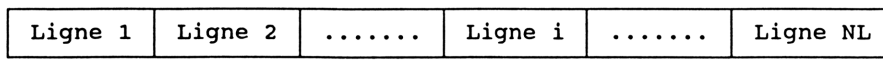
- Tous les champs, jusqu'à **NC**, sont longs de 5 quartets.
- **#029E8** est l'adresse-préfixe des tableaux.
- "**Taille**" est la taille totale du tableau, hors préfixe.
- "**Type**" contient l'adresse **#02933h** ou **#02977h** selon qu'il s'agit d'un tableau réel ou d'un tableau complexe.
- Le champ suivant doit contenir **#2**, (matrice = tableau à deux indices).
- Les deux champs suivants "**NL**" et "**NC**", contiennent respectivement le nombre de lignes et le nombre de colonnes du tableau.

- "**Eléments**" contient la représentation des nombres du tableau, dans l'ordre naturel (ceux de la première ligne, puis ceux de la seconde, etc..., jusqu'à ceux de la dernière ligne).

  Ces nombres (tous réels ou tous complexes) sont codés sans leur adresse-préfixe (qui apparaît une fois pour toutes dans le champ "**Type**" évoqué ci-dessus).

  Dans ces conditions un réel est codé sur 16 quartets et un complexe sur 32.

On peut donc considérer que le champ "Eléments" se décompose de la manière suivante (rappelons les notations, le tableau a **NL** lignes et NC colonnes):



Chaque ligne est longue de **16*NC** quartets dans le cas d'un tableau réel, et de **32*NC** quartets dans le cas d'un tableau complexe. L'opération d'échange des lignes **<m>** et **<n>** consistera donc à positionner les registres **D0** et **D1** au début de chacune des deux lignes, puis a effectuer des échanges sur des champs de 16 quartets (**NC** échanges pour un tableau réel, **2*NC** échanges pour un tableau complexe).

Les explications qui précèdent permettront sans doute de comprendre facilement la solution ci-dessous, elle même largement commentée.

Notez l'appel au point d'entrée **#3991** (cf page 361) qui effectue le produit des champs (**A:A**) et (**C:A**), le résultat étant placé dans le champ (**B:A**). Seuls ces trois registres sont modifiés (sur leur seul champ A) pendant cet appel.

```
   GOSBVL   #3F5D     *(A:A)=ligne n°<m>; (C:A)=ligne n°<n>
   A=A-1    A
   R0=A.F   A         *Sauvegarde (A:A)=m-1
   C=C-1    A
   R1=C.F   A         *Sauvegarde (C:A)=n-1
   GOSBVL   #679B     *Sauvegarde registres B,D,D0,D1
   A=DAT1   A
   D1=A               *Pointe sur le début du tableau
   D1=D1+   15        *Pointe sur le nombre d'indices
   A=DAT1   A         *(A:A)= nombre d'indices du tableau
   LC(5)    #2
   ?C#A     A         *Le tableau n'a pas deux dimensions?
   GOYES    INVD      *Alors erreur "Invalid Dimension"
   D1=D1+   5         *Pointe sur le nombre NL de lignes
   A=DAT1   A         *(A:A)= le nombre NL de lignes
   C=R0.F   A         *Récupère la valeur <m-1>
   ?C>=A    A         *n° de ligne trop élevé ?
   GOYES    BAV       *Dans ce cas, erreur "Bad Argument Value"
   C=R1.F   A         *Récupère la valeur <n-1>
   ?C>=A    A         *n° de ligne trop élevé ?
   GOYES    BAV       *Dans ce cas, erreur "Bad Argument Value"
   D1=D1+   5         *Pointe sur le nombre NC de colonnes
   C=DAT1   A         *(C:A)= nombre NC de colonnes
   D=C      A         *Place ce nombre dans (D:A)
   D1=D1-   15        *Recule pour pointer sur le type du tableau
   A=DAT1   A         *(A:A) = type du tableau
   LC(5)    #02933    *Adresse préfixe des réels
   ?C=A     A         *Est-ce un tableau réel ?
   GOYES    SUIT2
   LC(5)    #02977    *Adresse préfixe des complexes
   ?C=A     A         *C'est un tableau complexe ?
   GOYES    SUIT1
   LC(5)    #202      *Code de l'erreur "Bad Argument Type"
:ERR
   GOVLNG   #10F40    *Restaure registres et erreur de code (C:A)
:INVD
   LC(5)    #501      *Code de l'erreur "Invalid Dimension"
   GOTO     ERR
:BAV
   LC(5)    #203      *Code de l'erreur "Bad Argument Value"
   GOTO     ERR
********************* Continue page suivante *******************
```

```
********************** Suite du listing ************************
:SUIT1
   D=D+D     A          *(D:A)=2*NC (cas d'un tableau complexe)
:SUIT2
   DSL       A          *Multip. par 16. (D:A)=longueur d'une ligne
   D1=D1+    10
   D1=D1+    10         *Pointe sur le premier quartet du premier élt
   C=D       A          *(C:A)= longueur L d'une ligne.
   A=R0.F    A          *(A:A)= m-1
   GOSBVL    #3991      *(B:A)= (m-1)L
   CD1EX                *(C:A) Pointe sur le 1er quartet du 1er élt
   D1=C                 *D1 garde sa valeur
   C=B+C     A
   D0=C                 *Pointe sur le début de la ligne n°<m>
   C=D       A          *(C:A)= longueur L d'une ligne.
   A=R1.F    A          *(A:A)= n-1
   GOSBVL    #3991      *(B:A)= (n-1)L
   CD1EX                *(C:A) Pointe sur le 1er quartet du 1er élt
   C=B+C     A
   D1=C                 *Pointe sur le début de la ligne n°<n>
   D=D-1     A          *(D:A)= longueur d'une ligne, diminuée de 1
:LOOP
   A=DAT0    W          *Lit 16 quartets sur ligne <m>
   C=DAT1    W          *Lit 16 quartets sur ligne <n>
   DAT0=C    W
   DAT1=A    W          *L'échange de ces 16 quartets est terminé
   D0=D0+    16
   D1=D1+    16         *D0 et D1 ont avancé de 16 quartets.
   D=D-16    A          *Décrémente D de 16 quartets
   GONC      LOOP       *On continue si D n'est pas repassé par 0.
   GOVLNG    #5143      *Restaure registres et retour RPL
```

Vous pouvez noter, dans le listing précédent, la position centrale qu'occupent les sauts vers erreur. Ceci est dû au fait que ces erreurs sont appelées par des instructions **GOYES**, ce qui limite l'amplitude de l'appel à l'intervalle [-128,127]. La longueur du listing ne permet pas de placer les "sauts vers erreur" à la fin.

J'ai mesuré de façon précise (moyenne prise sur 20 essais successifs) les temps d'exécution des différentes versions de **ECHL** (l'opération consistant à intervertir deux lignes d'une matrice carrée réelle de 20 lignes).

- Instruction intégrée **RSWP**: 0.048 secondes.
- Version Externals: 0.0341 secondes.
- Version Assembleur: 0.00365 secondes.

Comme on dit, les chiffres sont éloquents (rappelons que le très bon score de la version Externals est dû à l'appel d'une routine de la ROM écrite elle-même en langage machine).

On peut objecter que la version "Assembleur" de **ECHL** n'effectue pas de test sur la présence et la nature des arguments sur la pile.

Pour contourner ce reproche, on peut ajouter quelques instructions en assembleur au début du code de **ECHL**.

Le plus simple reste toutefois d'utiliser un peu d'Externals, ce qui va nous permettre d'utiliser conjointement les deux programmes **EXTERNPC** et **ASMPC**.

Première étape:

Lancer le programme **ASMPC**.
Editer le programme **ECHL** (suffixe **ASM** par défaut).
L'assembler en un fichier-code (le nom est **ECHL.COD** par défaut).
Sortir du programme **ASMPC**.

Deuxième étape:

Lancer le programme **EXTERNPC**.
Charger un fichier de mnémoniques (il en faut peu ici !)
Editer le programme **ECHL** (suffixe **SRC** par défaut) suivant:

```
begin   need3   ChkArg   <41F1Fh>   [[ECHL   end
```

(il est inutile de préciser le suffixe **COD**, utilisé par défaut).
Le compiler en un fichier-objet (le nom est **ECHL.OBJ** par défaut).
Transmettre ce fichier-objet à la HP48, par **KERMIT**.
Sortir du programme **EXTERNPC**.

Le résultat est une structure Rpl de 147.5 octets, dont l'image sur la pile est:

```
    External   External   <41F1Fh>   Code
```

Soumise au même test de rapidité, cette dernière version de **ECHL** (qui est, elle, parfaitement sûre) tourne en 0.0043 secondes (ce qui est encore mieux que la version en Externals, utilisant la routine en langage-machine de la ROM).

La preuve est donc faite (mais quelques codeurs avisés le savent déjà) qu'il est possible d'améliorer en rapidité certaines routines internes de la HP48 (pourtant déjà écrites en langage-machine).

EXEMPLE 4: Trier les éléments d'un tableau

Nous reprenons le programme présenté page 80, triant un tableau réel par la méthode de la bulle (on ne rit pas! c'est la pire de toutes les méthodes de tri, d'accord, mais nous allons voir qu'en assembleur, les bulles remontent beaucoup plus vite...).

Le principe de la méthode de la bulle a été rappelé page 80. Il est inutile d'insister. Voici donc une solution.

```
   GOSBVL   #679B
   C=DAT1   A          *Pointe sur le début du tableau
   C=C+5    A          *Pointe sur le champ "taille" du tableau
   D1=C
   D1=D1+   10         *Pointe sur le nombre de dimensions
   A=DAT1   A          *(A:A) nombre de dimensions du tableau
:LOOP
   D1=D1+   5          *A la sortie de cette boucle, le registre
   A=A-1    A          *  D1 va pointer sur le premier quartet
   GONC     LOOP       *  du premier réel du tableau.
   AD1EX
   R3=A.F   A          *Sauvegarde cette adresse dans (R3:A)
   D0=C                *Pointe sur le champ "taille"
   A=DAT0   A          *Taille du tableau, hors préfixe.
   C=C+A    A          *Pointe sur le 1er quartet après le tableau
:BCL1
   C=C-16   A          *Pointe sur le dernier réel de la zone
   R4=C.F   A          *Sauvegarde cette adresse dans (R4:A)
   A=R3.F   A          *Pointe le premier réel du tableau
   ?C>A     A          *Il reste au moins deux réels?
   GOYES    BCL2       *Dans ce cas, on doit parcourir cette zone
   GOVLNG   #5143      *Fin: rétablit les registres et retour au Rpl
:BCL2
   D0=A                *Pointe un premier réel r1
   D1=A
   D1=D1+   16         *Pointe le réel suivant r2
   A=DAT0   W          *Le contenu du réel r1
   R1=A                *Le sauvegarde dans R1
   C=DAT1   W          *Le contenu du réel r2
   R2=C                *Le sauvegarde dans R2
   GOSBVL   #2BCA0     *"Sépare" r1 sur A et B, et r2 sur C et D
   P= 4                *Utile pour l'appel suivant
   GOSBVL   #2BD76     *Teste si r1 est supérieur à r2
   P= 0                *C'est plus prudent
   SETHEX              *idem
   GONC     SUIT       *Pas d'échange si r1≤r2
   A=R1                *Retrouve le réel r1
   DAT1=A   W          *L'écrit à la place de r2
   C=R2                *Retrouve le réel r2
   DAT0=C   W          *L'écrit à la place de r1
:SUIT
   AD1EX               *(A:A) pointe sur la position initiale de r2
   C=R4.F   A          *(C:A) pointe sur dernier réel de zone
   ?C>A     A          *On n'est pas arrivé en fin de zone?
   GOYES    BCL2       *Alors on continue dans cette zone
   GOTO     BCL1       *C'est reparti pour un nouveau parcours
```

Le listing précédent appelle plusieurs remarques.

- On utilise les points d'entrée **#2BCA0h** et **#2BD76h** qui ont été décrits dans le chapitre précédent.

- Le programme traite tous les tableaux réels, c'est-à-dire autant les vecteurs que les matrices. Remarquez la toute première boucle du programme qui permet cette propriété.

- Le programme n'effectue aucun contrôle de la nature de l'objet placé au niveau 1 de la pile (et même de la présence de cet objet). Appeler ce code avec une pile incorrecte a forcément des conséquences fâcheuses.
  Si on veut un programme qui vérifie que la pile contient bien un tableau réel au niveau 1, il faut l'écrire en Externals. Si on note **TRIT** notre programme, on crée le fichier-code **TRIT.COD** dans **ASMPC** (à partir du listing précédent), on passe dans **EXTERNPC** où on crée le texte **TRIT.SRC** suivant (que l'on compile en le fichier objet **TRIT.COD**, lui même transmis à la HP48).

```
begin   need1   dup;ra?#:BAT!   [[TRIT   end
```

(Rappelons que l'instruction **dup;ra?#:BAT!** génère une erreur "*Bad Argument Type*" si l'objet du niveau 1 n'est pas un tableau réel).
La structure Rpl obtenue (**TRIT.OBJ**) a une taille de 94 octets (ce qui est assez comparable à la version initiale en Externals).
On peut alors parler vitesse d'exécution. Cette version du programme **TRIT** ordonne un vecteur de 25 réels en 0.168 seconde (rappelons qu'il fallait 1.6 secondes pour la version en Externals et 15.3 secondes pour la version RPL-utilisateur).
Sur un vecteur de 100 réels, le temps de calcul de la version assembleur passe à 2.578 secondes (normal, ça augmente proportionnellement au carré du nombre d'éléments).

- <u>Pour accélérer</u>:
  La meilleure solution pour augmenter la vitesse d'exécution serait évidemment de changer de méthode de tri!
  Bullons quand même jusqu'au bout. On peut accélérer les choses de plusieurs manières:
  - Eteindre l'écran par **GOSBVL #1BBD** (juste après **GOSBVL #679B**), puis le rallumer à la fin par **GOSBVL #1B8F**. Le gain de temps est d'environ 10%
  - Supprimer les interruptions du clavier par **ST=0 15** (au début) avant de les rétablir à la fin (par **ST=1 15**). Le programme ne tourne pas plus vite, mais il ne peut plus être interrompu par ON-C ou même ON-A-F.
  - Supprimer toutes les interruptions par **GOSBVL #1115h** (au début) avant de les rétablir par **GOSBVL #10E5h** (à la fin). Le programme tourne quelques 3% plus vite.

EXEMPLE 5: L'adresse du niveau 1

Nous retrouvons ici le programme →**ADR**, présenté page 116. Rappelons qu'il s'agit de remplacer l'objet présent au niveau 1 de la pile par son adresse, sous la forme d'un entier binaire.

C'est un exemple intéressant car ici la solution en Assembleur est plus simple et plus courte (bien sûr plus rapide) que la solution en Externals.

Voici donc une solution minimum, utilisant deux points d'entrée.

```
GOSBVL   #3251C      *Dépile Adr1 dans A:A, puis SaveRegs
P=       4           *Pour obtenir binaire à 5 chiffres
GOVLNG   #596D       *Empile A:A en un entier binaire et retour
```

La taille de ce programme est simplement de...13 octets (à comparer aux 47 octets de la version en Externals). Ici "Adr1" désigne l'adresse de l'objet du niveau 1.

Les deux points d'entrée ont été décrits aux pages 352 et 359. Disons simplement que le premier GOSBVL pourrait être remplacé par:

```
A=DAT1   A           *Charge (A:A) avec Adr1
D1=D1+   5           *La pile remonte d'un niveau
D=D+1    A           *5 quartets de plus en mémoire libre
GOSBVL   #679B       *Sauve les registres D et D1 (et B,D0)
```

De même, l'instruction GOVLNG #596D pourrait être remplacée par:

```
GOSBVL   #5422C      *Empile A:A en un entier binaire (cf p.327)
GOVLNG   #2D564      *Retour au Rpl-Système
```

Ce tout dernier GOVLNG pouvant être lui-même remplacé par:

```
A=DAT0   A           *(A:A)= adresse prochain objet (obj0)
D0=D0+   5           *pointe sur l'objet qui suit (obj1)
PC=(A)               *Evalue l'objet obj0
```

N.B: Il est intéressant de constater les variations de l'adresse du niveau 1, quand on exécute systématiquement ce programme →**ADR**. On voit en effet l'adresse augmenter (au fur et à mesure que le programme →**ADR** crée de nouveaux entiers binaires) avant de diminuer brutalement lors de chaque *Garbage Collection*.

EXEMPLE 6: Conversion Objet = ▸ Chaîne

Il s'agit ici de la version Assembleur du programme →**STRI** qui a été présenté page 118. On se propose donc de remplacer l'objet du niveau 1 par la chaîne de caractères qui représente cet objet en mémoire.

Exemples:

L'entier 1 doit être remplacé par "33920000000000000010"
Le mot intégré DUP par "D9D205AA8188130B2130"
La chaîne "ABC" par "C2A20B0000142434"

Ce programme nous permettra de voir comment réserver "proprement" une zone non encore occupée dans la mémoire vive.

Commençons par prendre connaissance d'une solution. Les explications suivent.

```
   GOSBVL   #679B     *Sauvegarde B,D,D0,D1
   C=DAT1   A         *L'adresse de Obj1 va dans (C:A)
   R1=C.F   A         *Elle est sauvegardée dans (R1:A)
   D0=C               *Pointe sur l'objet du niveau 1
   GOSBVL   #3019     *Pour passer sur l'objet suivant
   CD0EX              *(C:A) pointe sur le 1er quartet après Obj1
   AR1EX              *(A:A) pointe sur le 1er quartet de Obj1
   C=C-A    A         *Taille n de Obj1, en quartets
   RSTK=C             *La sauvegarde dans le registre RSTK
   GOSBVL   #5B79     *Crée chaîne de n caractères, pointée par R0
                      *D0 pointe sur le premier caractère
   C=DAT1   A
   D1=C               *Pointe sur le début de l'objet initial
   C=RSTK             *Récupère nombre quartets objet initial
   C=C-1    A
   B=C      A         *Nombre de quartets dans (B:A), diminué de 1
:LOOP
   A=DAT1   1         *Lit un quartet de l'objet initial Obj1
   D1=D1+   1         *Pointe sur quartet suivant
   LC(2)    #39       *Octet code ascii de 9
   ?C>=A    P         *Le quartet lu est inférieur ou égal à 9?
   GOYES    SUIT
   LC(2)    #47
   A=C+A    P         *Positionne le quartet 0 de A
:SUIT
   C=A      P         *Positionne le quartet 0 de C
   DAT0=C   B         *Ecrit un caractère dans la chaîne
   D0=D0+   2         *Pointe sur le caractère suivant
   B=B-1    A         *C'est le nombre de quartets restants
   GONC     LOOP      *Si pas fini, on continue...
   A=R0.F   A         *Récupère dans (A:A) l'adresse de la chaîne
   GOVLNG   #3672     *Restaure registres, niveau 1 = la chaîne
                      *  et retour au Rpl
```

Le programme précédent est l'occasion d'utiliser trois nouveaux points d'entrée:

- **GOSBVL #3019**:
  Il s'agit du programme permettant de "*skiper*" l'objet pointé par **D0**. Il a été présenté page 351. A la sortie **D0** pointe sur le premier quartet après cet objet. La sauvegarde du **D0** initial permet alors, par une simple soustraction, de calculer le nombre de quartets occupés par l'objet.

- **GOSBVL #5B79**:
  Ce sous-programme crée une chaîne de ***n*** caractères, où ***n*** est le contenu du champ ***A*** du registre **C** (voir page 350). Cette création s'accompagne éventuellement d'un *Garbage Collection* (et si celui-ci échoue, il y a une erreur "*Insufficient Memory*").
  A la sortie de ce sous-programme, le registre **R0** pointe sur le début de la chaîne (ce qui sera utile à la fin pour faire de cette chaîne l'objet du niveau 1 de la pile), **D0** pointe sur le premier caractère (utile pour la copie qui va suivre), et **D1** pointe bien sur le niveau 1 de la pile (après le *Garbage Collection* éventuel, c'est plutôt intéressant).

- **GOVLNG #3672**:
  Il s'agit ici d'effectuer un retour au Rpl dans les conditions suivantes: restauration des registres, copie de l'adresse contenue dans (**A:A**) à l'adresse pointée par **D1** (ce qui permet au niveau 1 de la pile de ne plus désigner l'objet initial, mais son image finale sous forme de chaîne de caractères), puis retour effectif au Rpl (A=DAT0 A, D0=D0+ 5, PC=(A)).

La copie des quartets de l'objet initial sur les caractères de la chaîne s'effectue dans une boucle débutant au label **LOOP**. Le compteur de cette boucle est le nombre initial de quartets (diminué de 1, la décrémentation s'effectuant en fin de boucle).

On remarquera les instructions permettant de traduire un quartet (dont la valeur va de 0 à F, et stocké dans le quartet 0 du registre **A**), en un octet contenant le code Ascii (et placé dans le champ *B* de **C**). Il suffit de noter que les quartets 0 à 9 sont codés de #30h à #39h, et que les quartets A à F sont codés de #41h à #46h.

On peut améliorer la rapidité d'exécution de cette version de **→STRI** en limitant les opérations de copie à des blocs de 8 quartets (lus dans l'objet initial) et de 16 quartets (écrits dans la chaîne finale). Evidemment, celà se paye par une augmentation de la longueur du programme (du fait par exemple que le nombre de quartets de l'objet initial n'est pas forcément un multiple de 8).

Telle qu'elle est, la version Assembleur de **→STRI** occupe 57.5 octets (50 octets pour la version en Externals). La matrice identité d'ordre 20 est transformée en chaîne (6430 caractères) en 0.275 seconde (0.390 seconde pour la version Externals, qui s'appuye essentiellement sur une routine interne en langage-machine).

EXEMPLE 7: Tests sur le clavier

Il y a deux problèmes distincts qui se rapportent aux tests du clavier dans un programme en assembleur. Le premier est de savoir si telle touche, en particulier, est actuellement enfoncée. Le second est de savoir si une touche quelconque du clavier est enfoncée et laquelle.

▪ Les couples out/in:

Pour résoudre la première question, il faut avoir recours aux couples **out/in** de chacune des touches du clavier. Voyons rapidement de quoi il s'agit:

Le registre **OUT** (des sorties) est long de 12 bits (3 quartets).
Le registre **IN** (des entrées) est long de 16 bits (4 quartets).
Le fait de charger **OUT** avec le contenu du champ ***X*** du registre **C** (au moyen de l'instruction **OUT=C**) "interroge" la carte électronique du clavier. On prend connaissance de la réponse à cette sollicitation au moyen de l'instruction **A=IN** (ou **C=IN**) qui copie le contenu du registre **IN** sur les quartets 0 à 3 du registre **A** (ou **C**).

Evidemment, l'interrogation du clavier par **OUT=C** et la lecture de la réponse par **A=IN** ou **C=IN** ne se font pas au hasard: chaque touche se voit attribuer un couple de valeurs particulières que nous appelerons ***out*** et ***in*** (ce sont des champs de trois quartets). Vous remarquerez que dans chacun des champs ***out*** et ***in*** d'une touche particulière, seul un bit est non nul (la position des ces bits est fonction de la position de la touche sur le clavier).
Pour savoir si la touche en question est actionnée, on effectue la séquence:

```
P=        0         *Le plus souvent, P contient déjà 0
LC(3)     out       *Place dans (C:X) le champ "out" de la touche
LA(3)     in        *Place dans (A:X) le champ "in" de la touche
OUT=C               *Interroge le clavier pour cette touche
GOSBVL    #1160     *Appelle sous-programme effectuant C=IN
                    *  (réponse dans les quartets 0 à 3 de C)
C=C&A     X         *"Et logique" entre cette réponse et "in"
```

A la sortie, le contenu du champ (**C:X**) est non nul si et seulement si la touche considérée est enfoncée: celà revient en fait à savoir si le seul bit qui n'est pas nul dans le champ "***in***" est également armé dans la réponse chargée dans (**C:X**).

Si on veut faire de cette séquence un sous-programme armant le bit CARRY si la touche considérée est enfoncée (le désarmant sinon), alors on ajoutera les instructions:

```
?C#0      X         *La touche est-elle enfoncée ?
RTNYES              *Retour de sous-prog, avec bit CARRY armé
RTN                 *Retour avec bit CARRY désarmé.
```

Voyons les couples ***out/in*** des différentes touches du clavier (pour **ON**, pour laquelle il y a une procédure spéciale). La table des couples ***out/in*** figure en ROM, à l'adresse **#AEBh**.

La quatrième colonne contient le code de chaque touche dans le buffer du clavier (voir plus loin). Ces codes apparaissent en ROM, à partir de l'adresse **#C11h** (c'est la numérotation classique des touches, de gauche à droite et de haut en bas, avec des exceptions pour les trois touches shift).

Key	out	in	Code
A	002	010	01
B	100	010	02
C	100	008	03
D	100	004	04
E	100	002	05
F	100	001	06
MTH	004	010	07
PRG	080	010	08
CST	080	008	09
VAR	080	004	0A
↑	080	002	0B
NXT	080	001	0C
'	001	010	0D
STO	040	010	0E
EVAL	040	008	0F
<-	040	004	10
↓	040	002	11
→	040	001	12
SIN	008	010	13
COS	020	010	14
TAN	020	008	15
√x	020	004	16
y^x	020	002	17
1/x	020	001	18

Key	out	in	Code
ENTER	010	010	19
+/-	010	008	1A
EEX	010	004	1B
DEL	010	002	1C
◂=	010	001	1D
α	008	020	80
7	008	008	1F
8	008	004	20
9	008	002	21
÷	008	001	22
◂=╗	004	020	40
4	004	008	24
5	004	004	25
6	004	002	26
*	004	001	27
╔▸	002	020	C0
1	002	008	29
2	002	004	2A
3	002	002	2B
-	002	001	2C
0	001	008	2E
.	001	004	2F
SPC	001	002	30
+	001	001	31

Exemple: le programme suivant boucle sur lui-même jusqu'à ce que la touche **MTH** soit actionnée. Il y a alors retour au RPL.

```
   GOSBVL   #679B     *Sauve Regs. Le faire même si inutile.
   LA(3)    #010      *Place dans (A:X) le champ "in" de MTH
:LOOP
   LC(3)    #004      *Place dans (C:X) le champ "out" de MTH
   OUT=C              *Interroge le clavier pour cette touche
   GOSBVL   #1160     *Appelle sous-programme effectuant C=IN
                      *  (réponse dans les quartets 0 à 3 de C)
   C=C&A    X         *"Et logique" entre cette réponse et "in"
   ?C=0     X         *Touche MTH non actionnée?
   GOYES    LOOP      *Dans ce cas on continue
   GOVLNG   #5143     *Restaure Regs et retour au Rpl
```

N.B: Dans cet exemple, l'action de la touche **MTH** est évaluée lors du retour au RPL. Celà vient du fait que chaque appui sur une touche génère une interruption qui stocke cette touche dans le buffer du clavier. Pour l'éviter, on peut insérer un **GOSBVL #D57** (vide le buffer du clavier, voir page 346) juste avant le retour au RPL.

▪ Une particularité de A=IN et C=IN:

Vous pouvez vous demander pourquoi l'exemple précédent exécute l'instruction **C=IN** au moyen d'un **GOSBVL #1160** (appel d'un sous-programme, évoqué page 348, dont la seule instruction est précisément **C=IN**). La réponse est que les instructions **A=IN** et **C=IN** ne font leur travail que si elles débutent à une adresse paire.

Voici une preuve de ce comportement bizarre, en reprenant pratiquement l'exemple précédent.

```
   GOSBVL   #679B     *Sauve Regs. Le faire même si inutile.
   LA(3)    #010      *Place dans (A:X) le champ "in" de MTH
:LOOP
   LC(3)    #004      *Place dans (C:X) le champ "out" de MTH
   OUT=C              *Interroge le clavier pour cette touche
   NOP3               *Instruction vide sur 3 quartets
   C=IN               *Réponse dans les quartets 0 à 3 de C)
   C=C&A    X         *"Et logique" entre cette réponse et "in"
   ?C=0     X         *Touche MTH non actionnée?
   GOYES    LOOP      *Dans ce cas on continue
   GOVLNG   #5143     *Restaure Regs et retour au Rpl
```

Placez l'objet-code résultant de l'assemblage de ce fichier-source dans une variable et évaluez cette variable. Il y a une chance sur deux pour que le programme fonctionne normalement (s'il ne fonctionne pas, vous l'interrompez par un **ON-C**).

Modifiez alors le source ci-dessus en intervertissant les deux instructions **NOP3** (3 quartets) et **OUT=C**. Assemblez le résultat et placez le code-objet dans la même variable: la taille n'ayant pas changé, le code reste à la même adresse, mais l'adresse de l'instruction **OUT=C** n'a plus la même parité.

Vous verrez alors que la deuxième version marche si la première ne marche pas, et réciproquement.

Conclusion: on doit toujours évaluer **C=IN** et **A=IN** à des adresses paires. Le mieux est donc d'appeler les sous-programmes des adresses **#115A** (seule instruction **A=IN**) ou **#1160** (seule instruction **C=IN**). Il est bien sûr très correct d'utiliser le sous-programme de l'adresse **#1EEC** (qui effectue d'abord **OUT=C** puis **GOVLNG #1160**: voir page 348).

▪ Tester un groupe de plusieurs touches:

Il est possible de tester simultanément un groupe de plusieurs touches, pour savoir si l'une d'elles au moins est actionnée. Il suffit pour celà d'utiliser comme champs "***out***" et "***in***" le OU logique des différents champs "***out***" d'une part, et des différents champs "***in***" d'autre part.

Par exemple, pour tester si l'une des 6 touches de menu est actionnée, on utilise le champ **out=#102h** (c'est le OU logique de #002h pour la touche A et de #100h pour les 5 autres touches) et le champ **in=#01Fh** (c'est le OU logique des champs #10 pour A et B, #8,#4,#2,#1 pour C,D,E,F).

```
   GOSBVL   #679B    *Sauve Regs. Le faire même si inutile.
   LA(3)    #01F     *Le champ "in" des 6 touches de menu
:LOOP
   GOSBVL   #D57     *Vide le buffer du clavier
   LC(3)    #102     *Le champ "out" des 6 touches de menu
   GOSBVL   #1EEC    *OUT=C, puis C=IN
   C=C&A    X        *"Et logique" entre la réponse et le "in"
   ?C=0     X        *Aucune touche de menu actionnée?
   GOYES    LOOP     *Dans ce cas on continue
   GOSBVL   #D57     *Vide le buffer du clavier
   GOVLNG   #5143    *Restaure Regs et retour au Rpl
```

Dans l'exemple précédent, le test **?C=0 X** renvoie la valeur logique VRAI si aucun des bits armés du champ "***in***" des 6 touches de menus n'est armé dans la réponse lue dans le registre **C** par **C=IN** (c'est-à-dire si aucune des 6 touches de menu n'est actionnée à ce moment là).

On peut penser à utiliser le test **?C=A X** (ou son contraire). Ce test renvoie la valeur VRAI si **tous** les bits armés du champ "***in***" des 6 touches de clavier sont également armés dans la réponse lue dans **C**, et donc en particulier si les 6 touches du clavier sont **simultanément** actionnées.

En fait, compte tenu que le bit armé des champs "***in***" des deux touches A et B est le même, il est dans ce cas impossible de distinguer si les 6 touches sont actionnées ou si 5 de ces touches le sont (l'une des deux touches A ou B étant relachée).

Il est tout de même utile de tester l'appui simultané sur deux touches, notamment quand l'une d'elles est une touche shift (α, ◄┐ , ou ┌►).

On remarquera que le champ "***in***" des 3 touches shift est égal à #020 (seul le bit numéro 5 est armé, ce qui caractérise ces champs "***in***" par rapport à ceux des autres touches, empêchant ainsi toute ambiguité).

Voici une nouvelle mouture de l'exemple précédent: pour sortir de la boucle, il faut appuyer simultanément sur les touches ┌► et NXT.

```
   GOSBVL   #679B    *Sauve Regs. Le faire même si inutile.
   LA(3)    #021     *Le "in" de ┌► et NXT (#020h OR #001h)
:LOOP
   GOSBVL   #D57     *Vide le buffer du clavier
   LC(3)    #082     *Le "out" de ┌► et NXT (#002h OR #080h)
   GOSBVL   #1EEC    *OUT=C, puis C=IN
   C=C&A    X        *"Et logique" entre la réponse et le "in"
   ?C#A     X        *Pas de ┌► et NXT simultanés ?
   GOYES    LOOP     *Dans ce cas on continue
   GOVLNG   #5143    *Restaure Regs et retour au Rpl
```

▪ Quelques tests "tout prêts":

Il est possible de tester l'appui sur la touche **ON**, ou sur l'une des trois touches shift, par un appel d'un sous-programme particulier situé en ROM. Chacun des ces sous-programmes arme le bit CARRY si et seulement si la touche considérée est actuellement actionnée.

- **GOSBVL #C74**: arme le bit CARRY si et seulement si la touche **ON** est actionnée (pour celà exécute **C=IN** puis teste si le bit 15 du registre **C** est armé).
- **GOSBVL #D14**: arme le bit CARRY si et seulement si la touche ◂┐ est actionnée.
- **GOSBVL #D1D**: arme le bit CARRY si et seulement si la touche ┌▸ est actionnée.
- **GOSBVL #D26**: arme le bit CARRY si et seulement si la touche α est actionnée.

▪ Le buffer du clavier:

On nomme ainsi la zone de la RAM-Système dans laquelle la HP48 stocke les codes des touches qui ont été actionnées, mais qui n'ont pas encore été traitées. C'est donc une réserve, qui permet de stocker jusqu'à 15 touches. Ces touches sont évaluées (et en même temps retirées du buffer, dans l'ordre où elles y sont rentré) lors d'un retour au mode d'exécution directe de la HP48.

Dans le buffer, chaque touche est représentée par un code. Ces codes ont été donnés page 380 (quatrième colonne) et suivent la numérotation de gauche à droite, ligne par ligne, sauf pour les touches *shift*.

Les deux points d'entrée les plus utiles ici ont été décrits page 346. Il s'agit des sous programmes situés à l'adresse **#4999h** (Label **Keypd?**) qui arme le bit CARRY si le buffer est non vide (entre autres) et à l'adresse **#4840h** (Label **GetKeyC**) qui arme CARRY si le buffer est vide, le désarme CARRY s'il est non vide, mais qui en outre charge (**C:B**) avec le code de la touche la plus ancienne (qui en même temps est retirée du buffer).

Exemple:

le programme suivant boucle sur lui-même (après avoir vidé le buffer du clavier) jusqu'à ce qu'une touche soit actionnée (on utilise le point d'entrée **#4999h**). On voit que ce point d'entrée ne sort pas la touche du buffer, puisque celle-ci est évaluée normalement lors du retour au Rpl.

```
   GOSBVL   #679B     *Sauve Registres B,D,D0 et D1.
   GOSBVL   #D57      *Vide le buffer du clavier
:LOOP
   GOSBVL   #4999     *Il y a une touche dans le buffer?
   GONC     LOOP      *Sinon on recommence à scruter
   GOVLNG   #5143     *Restaure Regs et retour au Rpl
```

Exemple:

Le programme qui suit exécute également une boucle jusqu'à ce qu'une touche soit actionnée. Il place alors le code de cette touche sur le niveau 1 de la pile (sous la forme d'un entier-système) et retourne au Rpl. On voit bien que la touche en question est retirée du buffer, car elle n'est pas évaluée lors du retour au mode d'exécution directe.

```
   GOSBVL   #679B    *Sauve Registres B,D,D0 et D1.
   GOSBVL   #D57     *Vide le buffer du clavier
:LOOP
   GOSBVL   #4840    *Le buffer est vide ?
   GOC      LOOP     *Alors on recommence à scruter
   A=0      A        *Nettoie le champ (A:A)
   A=C      B        *Place dans (A:B) le code de la touche
   GOVLNG   #357C    *Restaure Regs, Empile <(A)>, retour au Rpl
```

Remarque:

l'appui sur la touche **ON** n'est pas contrôlé par les deux programmes précédents. Cette touche, lorsqu'elle est actionnée, n'est pas stockée dans le buffer. On sait que le test d'un appui sur **ON** s'effectue avec **C=IN** et en vérifiant si le bit 15 de C est armé (ou en utilisant le point d'entrée **#C74h**, voir page 346).

▪ Désactiver les interruptions du clavier:

Rappelons que les interruptions liées au clavier peuvent être inhibées avec l'instruction **ST=0 15** (on désarme le bit 15 du registre **ST**), et de nouveau autorisées par **ST=1 15**. L'instruction **ST=0 15** va même assez loin puisqu'elle supprime les interruptions liées par exemple au retrait ou à l'ajout d'une carte d'extension (le RESET aura tout de même lieu lors du retour au mode d'exécution directe, qui réhabilite toutes les interruptions).

Il est par exemple assez masochiste de lancer le programme suivant, qui ne peut peut être interrompu que par un RESET sur la carte interne (voir tome 1, page 23):

```
   ST=0     15       *Inhibe interruptions clavier (et autres)
:LOOP
   GOTO     LOOP     *Boucle infernale !
   GOVLNG   #2D564   *Retour au Rpl (illusoire)
```

On peut également utiliser le couple **INTOFF/INTON** (à ceci près que dans ce cas, les combinaisons de touches avec **ON** sont encore reconnues).

Il est intéressant de remarquer que la suppression des interruptions par **ST=0 15** n'empêche pas le courant de passer dans le clavier. Il est donc toujours possible de tester des touches par les couples **OUT/IN** (simplement, les touches actionnées ne sont pas placées dans le buffer).

Le programme qui suit ne peut se terminer que par un appui sur la touche **MTH** (vous pouvez toujours essayer **ON-C**, retirer une carte, etc...).

```
   GOSBVL   #679B     *Sauve registres B,D,D0 et D1
   ST=0     15        *Inhibe interruptions clavier (et autres)
   LA(3)    #010      *Place dans (A:X) le champ "in" de MTH
:LOOP
   LC(3)    #004      *Place dans (C:X) le champ "out" de MTH
   GOSBVL   #1EEC     *OUT=C, puis C=IN
   C=C&A    X         *"Et logique" entre cette réponse et "in"
   ?C=0     X         *Touche MTH non actionnée?
   GOYES    LOOP      *Dans ce cas on continue
   GOVLNG   #5143     *Restaure registres et retour au Rpl
```

Enfin, nous terminons avec le programme suivant, qui boucle sans fin jusqu'à ce que vous ayez actionné, dans cet ordre, les 4 touches 1 7 8 9. Si vous actionnez une touche incorrecte en milieu de séquence, vous devez tout recommencer (et ne comptez pas sur **ON-C** pour vous tirer d'affaire). On peut s'inspirer de cet exemple pour imaginer une protection efficace de l'accès à votre calculatrice.

```
   GOSBVL   #679B     *Sauve registres B,D,D0 et D1
   ST=0     15        *Inhibe interruptions clavier (et autres)
   A=0      W         *Nettoie les 16 quartets de A
:L1
   LA(12)   #002004008008  *Les "in" de 9,8,7,1
   LC(12)   #008008008002  *Les "out" de 9,8,7,1
:L2
   R0=C               *Sauvegarde le contenu de C dans R0
:L3
   LC(3)    #1FF      *Place dans (C:X) le champ "out" universel
   GOSBVL   #1EEC     *OUT=C, puis C=IN
   ?C=0     X         *Aucune touche actionnée?
   GOYES    L3        *Dans ce cas, on recommence
   C=R0               *Récupère le contenu de C
   GOSBVL   #1EEC     *OUT=C, puis C=IN
   C=C&A    X         *"Et logique" avec le "in" de (A:X)
   ?C=0     X         *Ce n'était pas la touche attendue?
   GOYES    L1        *Alors on reprend à zéro!
:L4
   LC(3)    #1FF      *Le champ "out" universel
   GOSBVL   #1EEC     *OUT=C, puis C=IN
   ?C#0     X         *Une touche actionnée?
   GOYES    L4        *Dans ce cas, attendre qu'elle soit relâchée
   C=R0               *Récupère valeur de C
   CSR      W         *Décale C
   CSR      W         *  de 3 quartets
   CSR      W         *  à droite
   ASR      W         *Idem
   ASR      W         *  avec le
   ASR      W         *  registre A
   ?A#0     A         *On n'a pas fini?
   GOYES    L2        *Alors testons la prochaine touche
   GOVLNG   #5143     *Restaure registres et retour au Rpl
```

EXEMPLE 8: Scrollings verticaux

Les deux exemples qui suivent permettent d'effectuer un scrolling continuel de votre écran, d'une ligne vers le haut (programme **UP.ASM**) ou d'une ligne vers le bas (programme **DOWN.ASM**). Le scrolling se limite aux lignes 1 à 56 de l'écran (la totalité du Grob de la pile si celle-ci est affichée, ou les 56 premières des 64 lignes de l'écran si vous êtes devant **PICT**).

Dans ce scrolling, et en prenant l'exemple de **UP.ASM**, la ligne qui sort vers le haut rentre comme nouvelle 56ème ligne (comportement analogue pour **DOWN.ASM**).

Ces deux scrollings sont interrompus par l'appui sur une touche quelconque (sauf **ON**). A l'intérieur des deux listings, une boucle d'attente permet de ralentir le défilement (sinon c'est vraiment trop rapide). Vous pouvez facilement modifier la durée de cette boucle.

Les deux points d'entrée essentiels ici sont **#66B9h** et **#670Ch** (voir page 349) qui permettent d'effectuer une copie d'une zone pointée par **D0** vers une zone pointée par **D1** (en amont ou en aval). Quand les deux zones ne se chevauchent pas, l'ordre dans lequel elles sont en mémoire est sans importance (sinon voyez page 349).

Voici tout d'abord le listing du programme "**UP**". Comme dans celui du programme "**DOWN**", la ligne qui sort de l'écran est sauvegardée dans la zone du buffer de la sortie RS232, avant d'être restituée dans sa nouvelle position.

```
   GOSBVL   #679B     *Sauvegarde B,D,D0 et D1
:LOOP
   D0=(5)   #8068D    *Pointe l'adresse de la BitMap écran
   C=DAT0   A
   RSTK=C             *Sauvegarde cette adresse
   D0=C               *Pointe sur le début de la BitMap écran
   D1=(5)   #80319    *Pointe début buffer RS232
   LC(5)    #22       *Les 34 quartets de la première ligne
   GOSBVL   #670C     *Copie cette ligne dans le buffer RS232
   C=RSTK
   D1=C               *Pointe début BitMap écran
   LC(5)    #74E      *Encore 55 lignes de 34 quartets
   GOSBVL   #670C     *Les remonte d'une ligne
   D0=(5)   #80319    *Pointe sur sauvegarde première ligne
   LC(5)    #22       *Elle est toujours formée de 34 quartets
   GOSBVL   #670C     *La recopie sur la 56ème ligne
   LC(5)    #400      *C'est
:RPT                  *  la
   C=C-1    A         *  boucle
   GONC     RPT       *  d'attente
   GOSBVL   #4840     *Aucune touche appuyée?
   GOC      LOOP      *Alors on continue...
   GOVLNG   #5143     *Récupère B,D,D0,D1 et retour Rpl
```

Voici maintenant le listing du programme "**DOWN**":

```
   GOSBVL   #679B     *Sauvegarde B,D,D0 et D1
:LOOP
   D0=(5)   #8068D    *Sauvegarde de la BitMap écran
   C=DAT0   A         *Pointe sur début BitMap écran
   LA(5)    #770      *56 lignes de 34 quartets = #770h quartets
   C=C+A    A
   D0=C               *Pointe juste après les 56 lignes
   RSTK=C             *Sauvegarde cette adresse
   D1=(5)   #8033B    *34 quartets après début buffer RS232
   LC(5)    #22       *Le nombre de quartets de la 56ème ligne
   GOSBVL   #66B9     *Copie cette ligne dans le buffer RS232
   C=RSTK
   D1=C               *Pointe juste après les 56 lignes
   LC(5)    #74E      *Encore 55 lignes de 34 quartets
   GOSBVL   #66B9     *Les descend d'une ligne
   D0=(5)   #8033B    *Pointe juste après sauvegarde 56ème ligne
   LC(5)    #22       *Elle est toujours formée de 34 quartets
   GOSBVL   #66B9     *La recopie sur la première ligne écran
   LC(5)    #400      *C'est
:RPT                  *  la
   C=C-1    A         *  boucle
   GONC     RPT       *  d'attente
   GOSBVL   #4840     *Aucune touche appuyée?
   GOC      LOOP      *Alors on continue...
   GOVLNG   #5143     *Récupère B,D,D0,D1 et retour Rpl
```

Remarques:

L'ordre dans lequel s'effectuent les copies de zone (selon les adresses croissantes ou décroissantes) et la position qu'occupent les registres **D0** et **D1** par rapport à ces zones (avant et après la copie) ont une grande importance. Voyez les explications des deux points d'entrée utilisés, page 349.

Chaque ligne de l'écran occupe 34 quartets. Ces 34 quartets correspondent à 34*4=136 colonnes. L'écran ne contenant matériellement que 131 colonnes, il y a 5 colonnes de pixels cachées (ce détail est sans importance pour nos scrollings verticaux. il en aurait bien sûr pour des scrollings latéraux).

On peut très bien effectuer les scrollings sur les 64 lignes de l'écran. Il n'y a aucun problème si le Grob courant est PICT (dans la mesure ou celui-ci a au moins les dimensions 131*64). Par contre, si le Grob courant est celui de la pile, il faut modifier le programme ci-dessus pour incorporer la bitmap des menus au défilement (sans celà on scrolle dans la mémoire vive et ça n'est pas trop recommandé...).

INDEX

→ADR (Programme) 116, 376
→ADR (Programme) 238
→EXT 13, 25
→LIB (Programme) 154
→MES (Programme) 147
→OBJ (Programme) 120
→OBJ (Programme) 238
→ST4 (Programme) 129
→ST7 (Programme) 129
→STRI (Programme) 118, 377
→STRI (Programme) 238
→VEC (Programme) 137
→VMS (Programme) 145
$CONFIG 155
$HIDDEN 155
$MESSAGE 155
$ROMID 155
$TITLE 155
αENTER 128
A (champ) 181, 188
A (registre) 179
Addition de registres 195
Adressage 178
Again 85
Alpha-ENTER 128
Annulation d'un registre 191
Arithmétique BCD 187
Arithmétique non signée 185
Arithmétique signée 186
Arithmétique signée 212
ASC (ASM48) 242
ASC (dans ASMPC) 285
ASM48 (Programme) 238, 247
ASMPC (Listing) 290
ASMPC (Programme) 282
ASS3 (Sous-programme) 266
ASS4 (Sous-programme) 270
ASS5 (Sous-programme) 273
ASS56 (Sous-programme) 278
ASS6 (Sous-programme) 275
ASS7 (Sous-programme) 279
ASSLINE (Sous-programme) 250
B (champ) 181
B (registre) 179
B (Registre) 62
begin 38
Bêta-ENTER 128
Binaire Codé Décimal 178
Bits de A ou C 199
Boucles en Externals 77
Branch 112
BRANCHEMENTS 211
BUS (instructions sur le) 226
C (registre) 179
CARRY 180, 185
CARRY 190, 216
CHAMPS DES REGISTRES 181
Chargement de A ou C 199
Chargement de D0 et D1 206
ChkArg et ChkArg! 70
CHKT (Programme) 76
Clavier (points d'entrée) 346
Clavier (tests de touches) 379
Clavier (buffer) 383
ClrLastName 72
Comparer (large) 2 registres 222
Comparer (strict) 2 registres 221
Complément à 1 d'un registre 193
Complément à 2 186
Complément à 2 212
Complément à 2 d'un registre 192
Contrôle des erreurs 113
Copie dans R0 à R4 203
Copie de A,C dans R0..R4 204
Copie de A,C vers D0,D1 207
Copie de PC dans A ou C 215
Copie de registres 194
D (registre) 179
D (Registre) 62
D0 (registre) 179
D0 (Registre) 61
D1 (Registre) 116
D1 (registre) 179
D1 (Registre) 62
Décalage d'un bit à droite 201
Décalages 212
Décrémentation d'un registre 191, 198
Décrémentation de D0 et D1 207
Dictionnaire des code-machine 232
Dictionnaire des mnémoniques 227
DIV3 (Sous-programme) 268
Division par 16 d'un registre 200
Division par 2 d'un champ 202
Division par 2 d'un registre 201
Do 81
Do....until 81
Do...again 85
Dq ("do quit") 92
Dq/sk (alternative) 95
DropRet 106
DupB 139
DupRet 106

Echange de A,C avec PC 214
Echange de A,C et D0,D1 208
Echange de A,C et R0..R4 205
Echange de deux registres 193
ECHL (Programme) 20, 370
Ecran (points d'entrée) 347
Ecran (scrollings) 386
Ecriture en mémoire 210
end 38
Endwhile 82
ENTER redirigé 128
EQUATE (Sous-programme) 260
Erreurs dans ASM48 248
ET logique 197
ETIQ (Sous-programme) 256
Evaluation différée (≈) 99
EXE (ASM48) 242
EXT→ (Programme) 132
EXTERNALs 59
EXTERNALs 8
EXTERNPC 31
EXTERNPC (Listing) 43
fichier de mnémoniques 33, 37
fichier objet 33
fichier source 33
fichier-code 38
fichier-source secondaire 38
FIELD (Sous-programme) 265
FIND (Programme) 127
Garbage Collection 349
GETARG (Sous-programme) 253
GOC 213
GOLONG 213
GONC 213
GOSBVL 215
GOSUB 215
GOSUBL 215
GOVLNG 213
GOYES 217
Hardware Status Bits 180
Hash-Table 161
HEX (ASM48) 242
HEX (dans ASMPC) 285
HEX (mode) 188
HEXA (Sous-programme) 262
Héxadécimal 178
HS (opérations sur) 225
HS (registre) 180
Identificateurs de types 68
Identifications multiples 69
If...then...else 86
IfError 113
Ift 86
Ifte 86
IFTEST (Sous-programme) 252
IN (registre) 180
IN et OUT (opérations sur) 224, 379
INC ET inc (ASM48) 242
Incrémentation d'un registre 192, 198
Incrémentation de D0 et D1 207
Index 79
Indices de boucle 79
Interruptions 384
INTOFF 226, 384
INTON 226, 384
Inversion des quartets 178
KEEP (Programme) 19, 369
KERMIT 32
LA(n) 199
LA(n) (dans ASMPC) 285
LAABS (dans ASMPC) 286
Labels 212
Langage-machine 176
LAREL (dans ASMPC) 286
LC(n) 199
LC(n) (dans ASMPC) 285
LCABS (dans ASMPC) 286
LCREL (dans ASMPC) 286
Lecture en mémoire 209
LIB→ (Programme) 170
LibNeed&Chk(1 à 5) 71
LibNeed(r) 67
LibNeed0 à LibNeed5 67
LINK (Sous-programme) 258
Link-table 163
Listables (Commandes) 73
LOADAC (Sous-programme) 263
LVAR (Programme) 74
M (champ) 181
Makeb(s) 121
MapBin(st,b,2s) 120
MapStr(b) 118
Marquage de la pile 66
Méta-objets 139
mnémoniques (Fichier de) 33
Mnémoniques de ASM48 245
Mnémoniques du Saturn 189
MP (bit) 180
Need(r) 67
Need0 à need5 67
NETLINE (Sous-programme) 249
NewIndex(s) 79
NEWOB 118
NewStop(s) 79
Next 78
NextInArray 78
NIBBLE (Sous-programme) 262
NOP3 à NOP5 225
NUL (ASM48) 241
NUL (dans ASMPC) 285
OFFSET (Sous-programme) 254
OU logique 197
OUT (registre) 180
P (champ) 181
P (opérations sur) 223
P (registre) 179, 188

PC (registre)....179
PC (registre)....211, 216
PC = (A)....214
PC = (C)....214
PC = A....214
PC = C....214
PEEK (Programme)....123
Pift....86
Pifte....86
Pile des retours....106
Points d'entrée....346
POKE (Programme)....125
Produit par 16 d'un registre....200
Program Counter....179
Program Counter....211
Put16Nib(2b)....125
Quartets....177
Quit (variantes)....90
R0 à R4 (registres)....179
RAM-Système....311
Rcl16Nib(2b)....123
RclRegs....346, 352
Registres....177, 179
Registres d'entrée-sortie....180
Registres D0 et D1....206
Registres de sauvegarde....179
Registres de sauvegarde....202
Registres de travail....179
REL2 à REL5 (ASM48)....243
REL2 à REL5 (dans ASMPC)....285
Repeat....82
REPLA (Programme)....150
Retours de sous-programme....216
REVST (Programme)....17, 366
Rotation d'un registre....201
RPL SYSTEME....59
RPL-système....176
RPL-utilisateur....176
RPL utilisateur....60
RSI....226
RSTK (copie avec C)....223
RSTK (registre)....180, 188
RSTK (registre)....211, 216
RTI....216
RTN....216
RTNC....216
RTNCC....216
RTNNC....216
RTNSC....216
RTNSXM....216
RTNYES....211, 217
S (champ)....181
Saturn....176
SAUT (Sous-programme)....261
SavRegs....346, 352
SB (bit)....180
Sélecteurs de champ....182
SETDEC....187
SETDEC....226
SETHEX....187
SETHEX....226
Skip (et variantes)....89
Soustraction de registres....196
SR (bit)....180
ST (opérations sur)....224
ST (registre)....180, 188
Start (Variantes)....77
Status Bits....180
STDM (Programme)....149
Step(s)....78
StopIndex....79
StopUpIndex....79
suffixes (EXTERNPC)....42
SwapRet....106
SYSEVAL....8
ßENTER....128
TABLES DE SÉLECTION DE CHAMP....182
Tester registre non nul....219
Tester registre nul....218
Tester registres égaux....220
Tester registres inégaux....220
Tester un bit de A ou C....219
Tests de type....71
Tests divers (en Externals)....88
ThenError....113
TRIT (Programme)....374
TRSTK (Programme)....84
Until....81
UpIndex....79
VEC→ (Programme)....142
Vérifications d'arguments....65
VMS→ (Programme)....144
W (champ)....181
While....82
WP (champ)....181
X (champ)....181
XM (bit)....180
XS (champ)....181
≈ (évaluation différée)....99
≈ PostEnd....101

TABLE DES MATIERES DU TOME 1

INTRODUCTION 7
RAPPELS? 15

PREMIERE PARTIE: LES OBJETS

La notion d'objet 26
Les différents types 25
Les structures Rpl 29
Les "Primitive Code Objects" 32
"RPL Système" et "Rpl Utilisateur" 33
Nombres réels 35
Nombres complexes 36
Réels longs 37
Complexes longs 39
Entiers-système 42
Entiers binaires 46
Chaînes de caractères 48
Caractères 52
Tableaux 53
Noms globaux et noms locaux 59
Le répertoire caché 64
Répertoires 69
Expressions algébriques 76
Objets unités 79
Listes 81
Structures RPL 91
Objets taggués 97
Objets graphiques 98
Objets code 100
Noms XLIB 102
Librairies 110
Objets "Library Data" 121
Objets sauvegardes 122
Objets "Linked Array" 123
La Mémoire cachée 126

DEUXIEME PARTIE: LES SYSEVALS

Opérations sur la pile 147
Les booléens 151
Entiers-système 157
Entiers binaires 161
Nombres réels 165
Nombres complexes 168
Réels longs 169
Complexes longs 171
Les tableaux 172
Listes et objets-composés 181
Chaînes de caractères 191
Conversions de type 195
Expressions algébriques 202
Les "Méta-Objets" 208
Les "Blocs-Expressions" 215
Les alarmes 228
Le menu STAT 231
Objets-taggués 234
Heure et date 235
Objets-unités 238
Opérations d'impression 240
La liaison série 242
Menus SOLVE et PLOT 246
Les menus 251
La gestion du clavier 261
La ligne de commande 267
Les flags de la HP48 275
Le graphisme 282
Variables globales 303
Variables locales 310
Les répertoires 321
Les noms XLIB 325
Les librairies 328
Les ports 335
La mémoire 343
Sysevals de contrôle 352
La gestion des erreurs 356
Instructions standards 362
D'une édition à l'autre 380
La Big-Liste 390
Index 417

NOTES

NOTES

BON DE COMMANDE

OFFRE EXCLUSIVE (*)	
Les programmes des secrets de la HP 48 G/GX Tome 2 sur disquettes	
(1) ☐ **Version 3"1/2**	**75 Frs TTC**
☐ **Version 5"1/4**	**75 Frs TTC**

(1) Cochez la case correspondante

(*) Bon de commande réservé exclusivement au possesseur du Tome 2, Les secrets de la HP 48 G/GX, sur présentation d'une photocopie de la facture d'achat.

Si cette offre vous intéresse, veuillez nous retourner le bon de commande ci-dessus, avec votre paiement par chèque libellé à l'ordre de D3I, à :

D3I Diffusion - 20 rue Hermès - 31520 Ramonville-St-Agne

☎ 62 24 66 20

BON DE COMMANDE

DANS LA MEME COLLECTION...	Qté	Prix unitaire TTC	Prix Total TTC
"MATHEZ" LA HP 48 G/GX en 380 programmes		**180 Frs**	
Les secrets de la HP 48 G/GX Tome 1 6000 bonnes adresses		**200 Frs**	
Frais d'expédition			25 Frs
TOTAL TTC			

Si cette offre vous intéresse, veuillez nous retourner le bon de commande ci-dessus, avec votre paiement par chèque libellé à l'ordre de D3I, à :

D3I Diffusion - 20 rue Hermès - 31520 Ramonville-St-Agne

☎ 62 24 66 20

NOTES

Achevé d'imprimer par

PARA GRAPHIC

31240 L'UNION (Toulouse)
Tél. (16) 61.74.27.67
Dépôt légal : Septembre 1993

LES *SECRETS* DE LA HP48G/GX

TOME 2
Externals et Assembleur

Avec le tome 1 des ***"Secrets"***, vous avez découvert une très grande bibliothèque de points d'entrée pour votre HP48. Avec le tome 2, vous allez passer la vitesse supérieure en programmant directement en Rpl-Système! Imaginez un **langage plus riche** en instructions, **plus varié** sur le plan des structures conditionnelles et répétitives, et tellement **plus rapide**. Toutes ces nouvelles possibilités de programmation sont décrites ici, accompagnées de nombreux exemples.

Deux outils vous permettront d'écrire vos propres applications en utilisant directement ces points d'entrées:

- ➡ Le programme **EXTERN48**, utilisable sur votre HP48: vous écrivez en "Rpl-Système" en indiquant les adresses des points d'entrée utilisés.
- ➡ Le programme **EXTERNPC**, écrit en Turbo-Pascal pour compatibles PC vous permet de mettre le turbo: vous écrivez en utilisant directement les mnémoniques décrits dans les tomes 1 et 2, et, sans sortir de EXTERNPC, vous compilez ces listings en fichiers-objet et les chargez sur votre HP48.

Pour ceux que les "Externals" ne combleraient pas, la deuxième partie de ce livre est consacrée à la **programmation en Assembleur**. Le microprocesseur de la HP48 est décrit (registres, adressage), ainsi que toutes les instructions du langage-machine. Pour utiliser ces nouvelles possibilités, deux outils sont présentés:

- ➡ Le programme **ASM48** est un assembleur pour HP48, écrit en "Rpl-Système", ce qui vous permet d'apprécier les possibilités de ce langage.
- ➡ Le programme **ASMPC** est un assembleur en Turbo-Pascal pour compatibles PC. Sans sortir de ASMPC, vous éditez vos programmes en assembleur, les convertissez en fichiers-objet, et les transmettez à votre HP48.

Pour vous éviter la saisie des programmes de ce tome 2, vous pouvez vous procurer une **disquette** contenant tous les listings (fichiers-source), les "exécutables", et un fichier de plus de 2500 mnémoniques à utiliser avec le programme EXTERNPC.

ISBN 2-908791-11-0