JPC
NOVEMBRE 1992
NUMERO 81
Le numéro 40 F

A PROPOS DU CLUB

Le Bureau	Editorial	*1*
J. Belin	Bientôt, l'anniversaire de PPC Paris	*2*
J. Belin	Mettez un *Virus* dans votre HP	*2*

HP-48

G. Toublanc	RPN ou Alébrique : le pour et le contre	*6*
L. Grand	Un nouvel utilitaire de développement	*8*
L. Grand	Calcul rapide de factorielles	*9*
G. Toublanc	Assembleur, les débuts (Acte III)	*12*

HP-95

J. Belin	Du Saturn au 8086 (Acte II)	*16*
G. Toublanc / J. Belin	Le nouveau Coin des Codes	*27*

EDITORIAL

Comme vous le savez, PPC Paris va bientôt assumer son vénérable âge de dix ans. Dans des précédents numéros, nous vous avons déjà parlé de notre souhait d'organiser une grande fête pour célébrer l'évenement. Ce sera avant tout l'occasion révée pour rencontrer de nombreuses personalités du monde HP, ainsi que de nombreux membres de PPC Paris ou des clubs étrangers. Un des grands évenements de ces deux jours sera notre tournoi de programmes, basé sur un jeu inédit mais prenant ses sources dans l'informatique. Bien sûr, ce concours sera primé, plusieurs sociétés nous ayant déjà promis d'offrir de nombreux lots de grande valeur pour les vainqueurs. Il y aura d'autres événements, mais nous en reparlerons plus précisement le mois prochain.

Nos machines évoluant, il est logique que le journal évolue lui aussi. A partir de ce mois-ci, vous trouverez à la fin du journal une nouvelle formule de notre *Coin des Lex*. Ouvert à toutes les machines, et non plus uniquement au HP-71, ces pages vous permettront d'entrer vos programmes, sans utilitaire de développement, de la manière la plus fiable possible. Nous attendons vos commentaires sur cette nouvelle formule (particulièrement de la part des utilisateurs de HP-48).

Terminons cet éditorial par une note un peu pessimiste. Pratiquement tous les clubs éditant régulièrement un journal sont confrontés à un même problème : le manque quasi total de réception d'articles de la part de leurs membres. Certains journaux ne paraissent plus que de façon épisodique, ou sont contraints d'afficher une page blanche avec pour seule mention "mettez votre article ici", comme l'a fait dernièrement le club Hollandais Prompt HP-GC. Si, aujourdh'hui PPC Paris est le seul grand club réussissant à faire paraître son journal tous les mois, cela est dû au travail acharné d'un petit groupe de personnes, commençant à être à court d'idées pour les articles. Si vous voulez que le journal continue à sortir régulièrement, il n'y a qu'une seule méthode : Envoyez nous des articles !

Le bureau

L'ANNIVERSAIRE

Comme nous vous l'avions annoncé il y a quelque temps, nous allons organiser une grande fête pour célébrer les 10 ans de PPC Paris.

Elle aura lieu les samedi 30 et dimanche 31 janvier 1993, à notre lieu de réunion habituel, le centre Jean Verdier (voir l'adresse dans les pages centrales) qui ouvrira exceptionnellement ses portes ce dimanche pour l'occasion.

Au programme, outre le tournoi de machines dont vous trouverez le sujet plus loin et notre assemblée générale, il y aura des expositions, des présentations de matériels ou de logiciels, ainsi que quelques surprises dont nous garderons le secret jusqu'au jour J. Le programme n'est pas encore totalement fixé, alors si vous avez une réalisation totalement délirante à présenter, ne vous faites pas prier...

Un autre évenement, qui aura lieu le samedi soir, sera le banquet, organisé dans un très bon (sinon un "grand") restaurant parisien. Attention, le nombre de places risquant d'être limité, nous vous conseillons de vous inscrire dès à présent.

Bien sûr, que nos amis provinciaux ou étrangers ne s'inquiètent pas, nous avons déjà pris des contacts auprès des hotels proches du lieu de la réunion et pouvons réserver le nombre de chambres nécessaire. Comptez un prix de base de 300 F environ par nuit.

Evidement, organiser une telle manifestation coûte cher. A titre indicatif, la location de la salle nous coûte l'équivalent de deux ans de location de notre salle habituelle ! Nous aurions voulu l'éviter, mais nous pensons que vous comprendrez que nous devrons vous demander une participation aux frais. Nous avons décidé de la fixer à 150 F (100 F pour les étudiants). Pour ce prix, vous aurez droit, entre autres, aux diverses boissons et nouritures vous permettant de tenir le coup pendant ces deux jours.

Vous trouverez avec ce journal une fiche d'inscription à nous retourner avant le 1er Janvier (15 décembre pour ceux qui doivent se faire héberger). Mais plus vite vous nous la retournerez, plus vite nous saurons combien il y aura de personnes présentes, et plus belle sera la fête...

Si vous hésitez pour venir à cette fête, je ne ferai qu'une seule remarque : il vous faudra attendre 2003 pour la prochaine !

Jacques Belin (123)

METTEZ UN VIRUS DANS VOTRE HP

Non, ne vous inquiétez pas ! Nous ne vous proposons pas mettre un de ces infâmes programmes destructeurs de mémoire dans votre machine, mais *Virus* est le nom du jeu qui servira de pretexte à notre tournoi de programmes et qui a été spécialement créé pour l'occasion...

Règle du jeu

Présentation

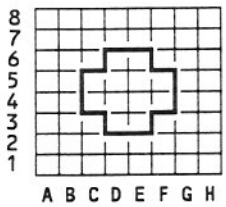
Ce jeu est très inspiré du *Jeu de la Vie*, un exercice informatique consistant à simuler la vie et la mort d'une colonie de bactéries.

Virus reprend le principe de ce jeu, en le simplifiant et en permettant de jouer contre un adversaire.

Le but du jeu est de posséder, en fin de partie, plus de pions que son adversaire sur le plateau.

Le Matériel

Le jeu se déroule sur un espace de 8×8 = 64 cases. Une zone centrale de 12 cases est définie pour le placement des premiers pions en début de partie :



Les joueurs disposent de 58 pions, blancs d'un coté et noirs de l'autre. Le joueur blanc dipose en plus de 3 pions blancs marqués d'une étoile (les Virus), le joueur noir dispose aussi de 3 Virus de couleur noire.

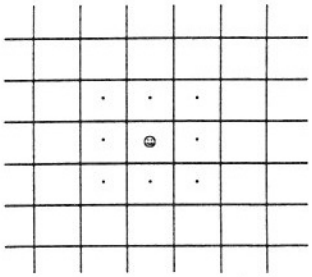
Début de la partie

Le premier joueur dispose 3 bactéries noires dans la zone centrale du plateau de jeu, puis le second joueur place 3 bactéries dans cette même zone. Les deux joueurs doivent respecter les *règles de vie* pour ces placements.

Après ce stade, chaque joueur placera un bactérie à tour de rôle.

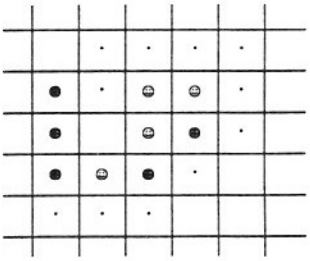
Règles de vie des bactéries

Une bactérie en cotoie une autre si cette derniere est sur une des 8 cases voisines :



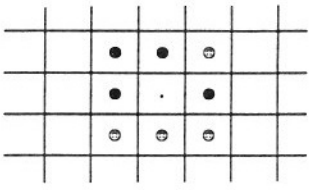
Une bactérie ne peut vivre que si :

- Elle cotoie au moins une bactérie (ou un Virus) de sa couleur :
- Elle cotoie moins de quatre bactéries de la couleur oposée.



"." : Cases possibles pour Blanc

Certaines cases ne peuvent être occupées par aucune bactérie, car elles peuvent être entourées par 4 bactéries de chaque couleur :



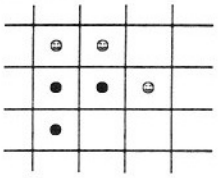
La case centrale n'est viable pour aucun des deux joueurs

Chaque joueur dispose de 3 bactéries particulières : les *Virus*. Ils sont immortels. Cela implique qu'on peut les placer sur n'importe quelle case libre, même si il n'y a pas de bactérie de sa couleur dans ce secteur, et qu'ils ne peuvent pas être tués par les pions de la couleur opposée. Ils peuvent donc, par exemple, occuper la case décrite précédement et participer à l'élimination des bactéries adverses.

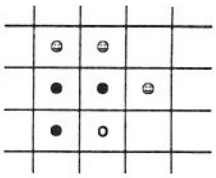
Prise des bactéries adverses

Un joueur peut "tuer" une bactérie adverse en l'entourant de quatre bactéries de sa couleur. La bactérie est immédiatement retirée du jeu. La case qu'elle occupait pourra être occupée ultérieurement par une autre bactérie de n'importe quelle couleur (à condition de satisfaire aux exigences des règles de vie).

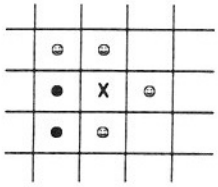
Exemple :



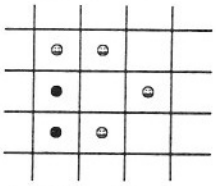
1) Position initiale



2) Blanc place la bactérie 'o'



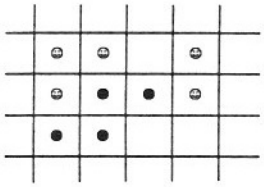
3) La bactérie 'X' est tuée



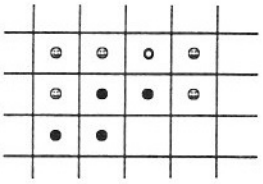
4) Configuration Finale

Un joueur peut tuer plusieurs bactéries adverses si, par le placement d'une seule bactérie, il faire en sorte que plusieurs bactéries adverses soient entourées de 4 pions de sa couleur.

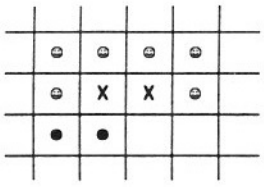
Exemple :



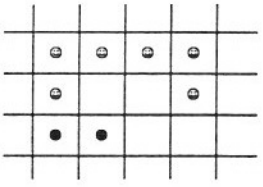
1) Position initiale



2) Blanc place la bactérie 'o'



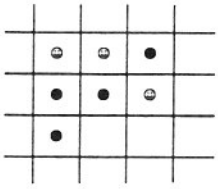
3) Les bactérie 'X' sont tuées



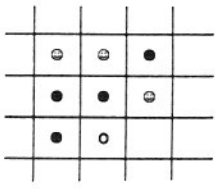
4) Configuration finale

Les règles de vie indiquent qu'une bactérie ne peut vivre si elle est isolée. Ceci implique que si une bactérie tuée rend une autre bactérie isolée, celle-ci est aussi tuée immédiatement.

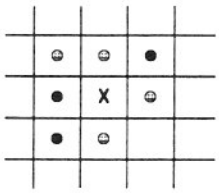
Exemple :



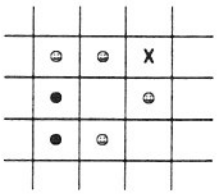
1) Position initiale



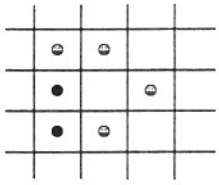
2) Blanc place la bactérie 'o'



3) La bactérie 'X' est tuée

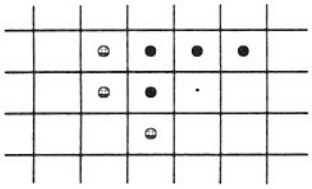


4) La bactérie 'X' est aussi tuée



5) Configuration finale

Un joueur ne peut pas placer un pion de sa couleur dans une case entourée d'au moins quatre bactéries adverses, même si, par cette pose de pion, il tuerait une de ces bactéries adverses :



Case marquée par '.' interdite pour blanc

Passage de tour

Les joueurs sont obligés de jouer à chaque tour. Toutefois, lorsqu'un joueur ne dispose plus de case libre où une de ses bactéries serait viable, et qu'il ne dispose plus de Virus en réserve, il passe son tour.

Fin de la partie

Lorsque tous les Virus out été joués et qu'il n'y a plus de possibilité de placement ou de prise de pions adverse dans certaines zones contrôlées par une seule, couleur, les colonies peuvent se développer librement dans ces zones. Les joueurs, d'un commun accord, peuvent donc ajouter ensemble les dernières bactéries de façon à remplir le tableau de jeu (toujours en respectant les règles de vie) et compter le nombre de pions de chaque couleur.

Le gagnant est celui qui dispose du plus grand nombre de pions sur le plateau.

Extraits du règlement du tournoi

Le concours est ouvert à tous les utilisateurs de calculateurs et ordinateurs de poche Hewlett-Packard.

- Chaque case est repérée en suivant les conventions du jeu d'echecs : lettres (de A à H) en abscisse et chiffres (de 1 à 8) en ordonnée. Les machines doivent communiquer leurs coups en donnant ces coordonnées.

- La position de départ est librement choisie par le programme, tout en restant limité à la zone fixée par les règles de *Virus*.

- Afin de rendre les parties reproductibles en cas de litige, les programmes utilisant des algorithmes possédant une possibilité de choix aléatoire sont interdits.

- On ne peut changer aucun paramètre interne du programme en cours de partie. Si un programme demande l'entrée d'un ou plusieurs éléments exterieurs (cartes mémoire, disquette, tables de paramètres...), ceus-ci doivent être clairement référencés de façon identique sur par le programme *et* sur l'élément exterieur.

- Chaque machine dispose d'un temps de reflexion limité pour jouer chaque partie (indépendant du temps de reflexion de l'autre concurrent). Ce temps est fixé à 30 minutes.

- Chaque participant dispute 5 parties.

- Une victoire vaut deux points, un match nul un point et une défaite zéro. Le vainqueur est le concurent possédant le plus de points.

La version complète du règlement sera envoyée à toutes les personnes inscrites au tournoi, ainsi qu'à toutes celles qui en feront la demande.

Jacques Belin (123)

Note (simple précaution, mais on ne sait jamais !) : Les droits de ce jeu sont réservés par PPC Paris.

HP-48

G. Toublanc	RPN ou Alébrique : le pour et le contre	*6*
L. Grand	Un nouvel utilitaire de développement	*8*
L. Grand	Calcul rapide de factorielles	*9*
G. Toublanc	Assembleur, les débuts (Acte III)	*12*

LES DIFERENTS MODES DE PROGRAMMATION

Le pour et le contre

Nos HP-48 et HP-28 nous offrent deux langages de programmation que l'on peut utiliser naturellement en les mélangeant: *RPN* et *Algebraic*. D'autre part il est possible d'utiliser des variables locales et aussi de faire des programmes récursifs.

Chaque mode a ses avantages et ses inconvénients :

RPN

Pour : occupation mémoire moindre et exécution plus rapide que l'Algebraic si le programme est conçu correctement.

Contre : demande au programmeur une petite période d'adaptation puis une attention très soutenue pour suivre l'évolution de la pile. Listings difficiles à comprendre sans commentaires.

Algebraic

Pour : traduction naturelle des formules. Facilite la tâche du programmeur. Listings plus faciles à comprendre que ceux réalisés en RPN.

Contre : occupation mémoire plus importante et exécution plus lente que la RPN.

Utilisation des variales locales

Pour : traduction naturelle des formules. Facilite la tâche du programmeur. Listings assez faciles à comprendre.

Contre : occupation mémoire plus importante surtout si l'on utilise des noms très explicites et longs. Exécution plus lente.

Programmes récursifs

Pour : valeur esthétique. Pour certains problèmes méthode presqu'incontournable.

Contre : à l'exécution l'occupation mémoire peut être importante comme le résultat peut se faire attendre.

Pour illustrer ce qui a été dit ci-dessus je vais prendre comme exemple un ensemble de programmes publiés dans *JPC 76* sous la signature de notre ami Asdin Aoufi. Il s'agit des programmes contenus dans les deux articles *Nombres de Bernoulli* et *Intégrales eulériennes complexes* et où sont utilisés les différents types de programmation énumérés ci-dessus ce qui constitue un très bon exemple.

Le type de programmation (Algebraic) choisi par Asdin a l'avantage de permettre un suivi assez facile entre les formules et le programme sans que celui-ci soit commenté.

Pour mettre en évidence les inconvénients j'ai traduit en RPN pure les différents programmes, listant alternativement les deux types. A la lecture on constatera que la RPN est moins gourmande en octets mais faute de commentaires est aussi moins compréhensible.

Lorsque qu'une programmation sera un mixage de RPN et d'Algébrique elle conservera cette dernière dénomination par opposition à la RPN pure.

BETA (algébrique, par Asdin, 115 octets)

```
« → x y
  « '(GAMMA(x)+GAMMA(y))/GAMMA(x+y)'
    →NUM
  »
»
```

BETA (RPN, par Guy, 53 octets)

```
« DUP2 + GAMMA  SWAP GAMMA ROT GAMMA + SWAP /
»
```

GAMMA (algébrique, par Asdin, 224.5 octets)

```
« → z
  « 'LN(2*π)/2+(z-.5)*LN(z)-z' →NUM
    1 13
    FOR m
      'B(m+ 2,1)/(B(m+2,2)*2*m*(2*m-1)*z^(2*m-1))'
      →NUM +
    NEXT EXP
  »
»
```

GAMMA (RPN, par Guy, 167 octets)

```
« .5 OVER - OVER LN * OVER + π →NUM 2 * LN
  2 / SWAP - B 5 1 14
  FOR m
    GETI 3 ROLLD GETI 4 ROLL SWAP 2 m * DUP
    1 - 8 PICK OVER ^ * * * / 4 ROLL + 3 ROLLD
```

```
  NEXT
  DROP2 EXP SWAP DROP
»
```

BER (algébrique, par Asdin, 513.5 octets)

```
« → indice
  « { } indice 2 + + 2 + 0 CON → ber
    « 1 'ber(1, 1)' STO 1 'ber(1,2)' STO -1
      'ber(2,1)' STO 2 'ber(2,2)' STO 1 indice
      FOR p 1 2 p * - 2 1 p 1 -
        FOR l
          IF p 1 >
          THEN 2 p * 1 + 2 l * COMB 'ber(l+2,1)' →NUM
          * 'ber(l+2,2)' →NUM SIMPLIFIE SOMME
        END
      NEXT 2 p * 1 + * NEG SIMPLIFIE 'ber(p+2, 2)'
      STO 'ber(p+2,1)' STO
      NEXT ber
    »
  »
»
```

BER (RPN, par Guy, 286 octets)

```
« DUP 2 + 2 DUP →LIST 0 CON 1 DUP PUTI 1 PUTI
  -1 PUTI 2 PUT 1 ROT
  FOR p 1 2 p * - 2 1 p 1 -
    FOR l
      IF p 1 >
      THEN 2 p * 1 + 2 l * COMB 4 PICK 2 l OVER + *
        1 - GETI 3 ROLLD GET ROT ROT * SWAP
        SIMPLIFIE SOMME
      END
    NEXT 2 p * 1 + * NEG SIMPLIFIE SWAP ROT 2 p
    OVER + * 1 - ROT PUTI ROT PUT
  NEXT
»
```

SOMME (Variables locales, par Asdin, 142.5 octets)

```
« → n1 d1 n2 d2
  « d1 d2 PPCM → ppcm
    « n1 ppcm * d1
      / n2 ppcm * d2 / + ppcm SIMPLIFIE
    »
  »
»
```

SOMME (Sans variables locales, par Guy, 75 octets)

```
« 3 PICK OVER PPCM ROT OVER * ROT / 4 ROLLD ROT
  OVER * ROT / ROT + SWAP SIMPLIFIE
»
```

SIMPLIFIE (Variables locales, par Asdin, 181.5 octets)

```
« → n d
  « n d PGCD → pgcd
    « n pgcd / d pgcd / → nn dd
      «
        IF dd 0 <
        THEN nn NEG 'nn' STO dd NEG 'dd' STO
        END nn dd
      »
    »
  »
»
```

SIMPLIFIE (Sans variables locales, par Guy, 35 octets)

```
« DUP2 PGCD ROT OVER / 3 ROLLD / »
```

PPCM (Variables locales, par Asdin, 57 octets)

```
« → a b
  « a b * a b PGCD /
  »
»
```

PPCM (Sans variables locales, par Guy, 25 octets)

```
« DUP2 PGCD / * »
```

PGCD (Programmation récursive, par Asdin, 50 octets)

```
« DUP 0 ==
  IF
  THEN DROP
  ELSE SWAP OVER MOD PGCD
  END
»
```

PGCD (Prog. non récursive, par Guy, 35 octets)

```
« WHILE SWAP OVER MOD DUP
  REPEAT
  END DROP
»
```

L'ensemble des programmes mis dans un répertoire occupe 1356 octets pour *Asdin* contre 748 octets pour *Guy*. Le gain en faveur de la RPN pure est donc très appréciable et peut même être très fort dans le cas du programme SIMPLIFIE :

101.5 octets pour la version *avec* variables locales.
35 octets pour la version *sans* variables locales.

Voyons quelques résultats côté rapidité :

Pour trouver le PGCD de 13! et 14! il faut :
.075 seconde avec la méthode récursive.
.024 seconde avec la méthode non récursive.

Pour simplifier 13!/14! il faut :
0.137 seconde avec les variabes locales plus la récursivité (appel de PPCM et PGCD).
0.0427 seconde avec la RPN pure.

Pour 13 BER le résultat est trouvé en :
63.3 secondes pour l'algébrique.
26.5 secondes pour la RPN.

Pour 5 GAMMA le résultat est trouvé en :
3.3 secondes pour l'algébrique.
1.8 seconde pour la RPN.

Pour 2 2 BETA le résultat est trouvé en :
9.8 secondes pour l'algébrique.
5 seconde pour la RPN.

Pour (5,5) GAMMA le résultat est trouvé en :
5.5 secondes pour l'algébrique.
4.2 seconde pour la RPN.

L'avantage est toujours pour la RPN pure, le gain de rapidité allant parfois dans le rapport de 1 à 3. La rapidité est, souvent, inversement proportionnelle à l'encombrement mémoire.

Conclusion :

Aucune méthode n'est à écartée à priori. Si on doit programmer une formule qui ne sera utilisée que temporairement et ne comportant pas de très nombreuses itérations alors la programmation de type algébrique est tout à fait indiquée. Nos HP48 nous aident dans ce sens avec la possibilité de réaliser des fonctions utilisateur. Si on envisage la constitution d'une librairie de programmes à usage fréquent, d'encombrement mémoire important et d'exécution assez longue alors je pense qu'il faudra faire l'effort de penser RPN. Ceci peut faciliter la tâche pour le passage de la *User Rpn* à la RPN.

Tous les types de programmation sont les bienvenus dans *JPC*. Certains programmeurs peuvent avoir déjà utilisé d'autres langages avec d'autres machines et donc se sentent plus à l'aise, sur HP48, avec un type donné de programmation. Il n'est pas rare qu'un même programme soit repris par plusieurs dans différents modes et avec des améliorations. Cette pratique c'est maintes fois réalisée dans JPC et en particulier lors du passage du langage Basic à l'assembleur sur HP-71, le même mérite revenant au créateur du programme en Basic et au programmeur en assembleur.

Alors faites feu de tout bois.

Guy Toublanc (276)

UN COMPILATEUR "MAISON"

Dans son article *Débuter en RPL* du *JPC 80*, Monsieur Guy Toublanc a fait une brève référence au compilateur que j'ai écrit. Ce compilateur, que j'ai simplement appelé HP-48, tourne sur tout système compatible MS-DOS. Il presque totalement compatible avec les *HP TOOLS* (à traduire : les outils de HP). Pour les initiés, les principales différences résident dans les déclarations de constantes, notamment des adresses que l'on utilise dans les parties écrites en assembleur.

Toutefois, HP-48 a l'avantage, par rapport aux *HP TOOLS*, de permettre des parties en langage RPL conventionnel, c'est-à-dire des parties conformes aux règles établies par les manuels d'utilisation de la machine. Un simple exemple de cet avantage est le problème suivant : on aimerait insérer dans un programme une expression symbolique (par exemple 'SIN(X)'). Je ne sais toujours pas comment on le fait avec les HP TOOLS, mais ceci devient évident avec HP-48 : il suffit d'insérer les lignes :

```
USRRPL
'SIN(X)'
```

La première ligne n'étant pas obligatoire si l'on est déjà en mode RPL conventionnel. Un exemple de programme est donné dans l'article *Fast-orielles* de ce numéro.

Si vous êtes intéressé par ce compilateur, vous pouvez en obtenir une copie "non-enrgistrée" auprès de PPC Paris (pour la France) ou de moi-même (pour la Suisse), moyennant une enveloppe affranchie à votre nom et une disquette 3"1/2 formatée DOS. Si vous désirez une copie à votre nom, il vous suffit de me faire parvenir votre adresse ainsi que l'équivalent de 30 francs suisses (c'est-à-dire soit 100 FF, soit 20 dollars américains). L'attrait de la version non-enregistrée réside sans aucun doute dans son prix. Celui de la version personnelle est le fait que vous recevrez automattiquement et gratuitement la prochaine version du compilateur.

Voici mon adresse personnelle :

Laurent Grand
Chemin du Borgeaux
1817 Brent
SUISSE

Pour ceux d'entre vous qui ont accès aux réseaux informatiques mondiaux, vous trouverez la version non-enregistrée de HP-48 sur le serveur

```
seq.uncwil.edu
username: anonymous
password: <votre adresse E-mail>
```

et dans le répertoire hp48/hp48-compiler.

Une dernière précision : la documentation que je fournis est en anglais.

Laurent Grand (516)

FAST-ORIELLES

Il y a des sujets qui, comme les calculs sur les polynômes, réapparaissent périodiquement dans les journaux spécialisés. Le calcul de factorielles avec tous le chiffres significatifs en fait partie. A une certaine époque (cf *JPC 65* et *JPC 67*), les meilleurs résultats étaient aux alentours de deux minutes et demie pour le calcul de 100! sur une HP-28S en double vitesse.

Dès lors, jugeant qu'une telle performance n'était pas digne de cette machine, je décidais d'écrire mon propre programme de factorielle, mais, contrairement aux programmes des *JPC* susmentionnés, le plus gros des calculs serait écrit en assembleur. Il y eut plusieurs versions de mon programme, dont je vous présente la dernière, adaptée à la HP48.

Le fichier source étant compilé avec mon compilateur HP-48, je vous donne le fichier listing produit par ce compilateur. La première colonne de chiffres est le code de l'instruction. Cette forme de mise en page a le net avantage de permettre aux utilisateurs d'autres calculateurs HP (je pense aux HP-28C et HP-28S notamment) d'adapter assez facilement le programme présenté à leur machine.

Passons maintenant au programme proprement dit : il s'agit de FFACT. Il est constitué d'une partie en RPL conventionnel et d'une seconde partie en assembleur.

La partie en RPL calcule directement n! si n≤14, ou calcule le nombre de chiffres de n! selon la formule de Stirling :

$$n! = 124\sqrt{(2\pi n)}\times n^n\times e^{-n}\times(1+1/12n)$$

C'est alors que la partie en assembleur intervient : elle récupère la valeur calculée par la partie en RPL afin de réserver la mémoire nécessaire, puis calcule n! selon un algorithme simple et naïf, mais efficace. Pour preuve, sur mon ancienne HP-28S, j'obtenais 100! en une seconde en mode double vitesse, et, si ma mémoire est bonne, 1000! en trois minutes. Avec la HP-48, j'obtiens 1000! en 203 secondes.

FFACT a besoin d'un entier réel au niveau 1 et renvoie un chaîne de caractères au niveau 1.

Exemple d'utilisation :

```
20 FFACT   -->   "2432902008176640000"
```

Pour terminer, voici le listing : (les huit colonnes de gauche représentent les codes à taper sur votre machine) :

```
                ; Partie en RPL conventionnel

D9D20E16    «
32
78BF1       DUP
33920100    14
00000000
00410
CFCE1       ≤
3CE22       IF
AFE22D9D    THEN
20
14BB1       FACT
B0BC1       →STR
B21305BF    ELSE
22D9D20
78BF1       DUP
ED2A2       2
EEDA1       *
DBAA1       \pi
EEDA1       *
6C9B1       LOG
ED2A2       2
50FA1       /
92CF1       OVER
78BF1       DUP
6C9B1       LOG
```

```
EEDA1      *
76BA1      +
92CF1      OVER
32BA1      e
6C9B1      LOG
EEDA1      *
90DA1      -
92CF1      OVER
33920100   12
00000000
00210
EEDA1      *
872B1      INV
9C2A2      1
76BA1      +
6C9B1      LOG
76BA1      +
F0CB1      CEIL
B9691      R→B
DBBF1      SWAP
B9691      R→B

        BEGINCODE

        ; Partie en assembleur

        ; Convention d'écriture :
        ; ========================

        ;    grp[k] = (k+1)ème groupe de
        ;             5 chiffres
        ;    R0.A = champ A du registre R0

        ; Utilisation des registres :
        ; ============================

        ; A : registre de calcul
        ; B : idem
        ; C : idem
        ; D : N° du groupe de 5 chiffres courant

        ; D0 : ^grp[k]
        ; D1 :

        ; R0 : champ A : nombre total de groupes
        ;      reste : nombre de chiffres de n!
        ; R1 ; champ A : adresse de n!
        ;      reste :
        ; R2 : n, n-1, n-2, ... , 1, 0
        ; R3 : retenue temporaire
        ; R4 :

        ; On commence par initialiser les
        ; divers registres et réserver
        ; la mémoire nécessaire

         include addr-48.as
```

```
CCD20          nibhex  begin_code
BD100          rel(5)  fin_code
8FB9760        gosbvl  save_reg
8FE3160        gosbvl  garb_collect
8F2D760        gosbvl  load_reg
143            a=dat1  a
130            d0=a
169            d0=d0+10
AF0            a=0     w
142            a=dat0  a
102            r2=a           ; R2 = n
174            d1=d1+5
143            a=dat1  a
130            d0=a
169            d0=d0+10
142            a=dat0  a      ; A = # de chiffres
D6             c=a     a      ;     de n!
BF2            csl     w
BF2            csl     w
BF2            csl     w
BF2            csl     w
BF2            csl     w
108            r0=c
AF2            c=0     w
3450000        lchex   #00005
C2             c=c+a   a
C6             c=c+c   a
8F8DA60        gosbvl  malloc
531            gonc    ok     ; si assez de mémoire
8F2D760        gosbvl  load_reg
142            a=dat0  a
164            d0=d0+5
808C           pc=(a)
132       ok   ad0ex           ; A = ^n!
101            r1=a
8F2D760        gosbvl  load_reg
111            a=r1
141            dat1=a  a
131            d1=a
34C2A20        lchex   prolog_string
145            dat1=c  a
174            d1=d1+5
3450000        lchex   #00005
DA             a=c     a
118            c=r0
BF6            csr     w
BF6            csr     w
BF6            csr     w
BF6            csr     w
BF6            csr     w
C6             c=c+c   a
C2             c=c+a   a
145            dat1=c  a
E2             c=c-a   a
174            d1=d1+5
8FC5760        gosbvl  write_zeroes
119            c=r1
135            d1=c
```

```
179             d1=d1+10
301             lchex   #1
15D0            dat1=c  1       ; grp[0] := 1

        ; On calcule n! comme suit :

        ; 0. R0.A := 0; grp[0] := 1;
        ; 1. retenue := 0;
        ; 2. pour k := 0, k <= R0.A faire
        ;         c := grp[k] * n
        ;         c := c + retenue
        ;         grp[k] := c mod 100000
        ;         retenue := c div 100000
        ;    fin_pour
        ; 3. si retenue # 0 alors
        ;         R0.A := R0.A + 1;
        ;         grp[R0.A] := retenue
        ;    fin_si
        ; 4. n := n - 1;
        ; 5. si n # 0 alors aller à 1.
        ;            sinon stop
        ;    fin_si

AF2     loop    c=0     w ; boucle principale
10B             r3=c      ; ajuste la retenue
119             c=r1
134             d0=c
169             d0=d0+10  ; D0 = ^grp[k=0]
118             c=r0      ; Nombre total de
D7              d=c     a ; groupes utilisés
AF0     l1      a=0     w
142             a=dat0  a
11A             c=r2
05              setdec    ; petite astuce ...
8FB9BD0         gosbvl  multiply
113             a=r3
A72             c=c+a   w ; ajoute la retenue
04              sethex    ; retour en hexa
144             dat0=c  a ; grp[k] = C.A
BF6             csr     w
BF6             csr     w
BF6             csr     w
BF6             csr     w
BF6             csr     w
10B             r3=c      ; nouvelle retenue
164             d0=d0+5   ; k := k+1
CF              d=d-1   a ; encore un groupe ?
5BC             gonc    l1 ; oui
113             a=r3      ; dernière retenue
8A8             ?a=0    a ; retenue nulle ?
D0              goyes   l2 ; oui ==> l2
140             dat0=a  a ; grp[k] := retenue
110             a=r0      ; on incrémente le
E4              a=a+1   a ; nombre total de
100             r0=a      ; groupes utilisés
11A     l2      c=r2
CE              c=c-1   a
10A             r2=c      ; on décrémente R2
8AE             ?c!=0   a ; R2 # 0 ?
79              goyes   loop ; oui

        ; Maintenant, il faut transformer notre
        ; nombre en une chaine de caractères.
        ; On commence par recopier le nombre
        ; à la fin de l'espace mémoire réservé.

118             c=r0
BF6             csr     w
BF6             csr     w
BF6             csr     w
BF6             csr     w
BF6             csr     w
D5              b=c     a
D7              d=c     a
111             a=r1
130             d0=a
169             d0=d0+10
CA              a=a+c   a
131             d1=a
179             d1=d1+10
8FC0760         gosbvl  copy_nibs

        ; Puis on transforme chaque chiffre
        ; en code hexadécimal correspondant

111             a=r1
130             d0=a
169             d0=d0+10
C0              a=a+b   a
131             d1=a
179             d1=d1+10
CF              d=d-1   a
3103    l3      lchex   #30
15F0            c=dat1  1
14C             dat0=c  b
161             d0=d0+2
170             d1=d1+1
CF              d=d-1   a
5CE             gonc    l3
181             d0=d0-2
AF3             d=0     w
D9              c=b     a
D7              d=c     a
81F             dsrb
CF              d=d-1   a
111             a=r1
131             d1=a
179             d1=d1+10
14B     l4      a=dat1  b
14E             c=dat0  b
148             dat0=a  b
14D             dat1=c  b
171             d1=d1+2
181             d0=d0-2
CF              d=d-1   a
5BE             gonc    l4
```

```
8F2D760          gosbvl  load_reg
142              a=dat0  a
164              d0=d0+5
808C             pc=(a)
        fin_code         ENDCODE

        ; Retour en mode RPL conventionnel

DBBF1    SWAP
8DBF1    DROP
B21305DF END
22
93632B21 »
30
```

Vous devriez obtenir une chaîne de 757 caractères. Une fois assemblé, la fonction BYTES du menu MEMORY doit vous retourner la valeur # 5139h au niveau 2 de la pile. Il y a certainement des améliorations à apporter au programme ... à vous de jouer !

Vous trouverez les chaînes de codes à assembler dans le *coin de codes*, à la fin du journal.

Laurent Grand (516)

Remarque :

Si vous avez l'occasion de désassembler une partie de le ROM de votre machine, essayez les diverses adresses utilisées par le programme (je pense en particulier à multiply (#0DB9Bh) ou à write_zeroes (#0675Ch)). C'est très instructif.

ASSEMBLEUR

les outils du début - Acte III

Pour terminer cette rubrique voici les programmes complémentaires de PEEK : POKE et ADDR.

Les fichiers sources sont présentés de manière à être compilables avec le compilateur HP (voir *Les débuts en rpl*, paru dans le numéro précédent).

POKE

syntaxe :
niveau 2 : 1 entier binaire l'adresse où poker
                             en Ram ou Iram
niveau 1 : la chaîne de codes à poker

Exemple :

si l'adresse en Ram de la chaîne "AA" est # 78A10h

```
# 78A10 10 + "2424" POKE
```

En rappelant la chaîne sur la pile on lira : "BB" (le code en mémoire d'un "B" étant 24).

Listing du programme POKE :

```
RPL                          * le compilateur passe
                             * en mode rpl
::                           * début programme
  CK2&Dispatch               * vérifie s'il y a 2
                             * objets de type à
                             * à préciser
  #B3                        * pour générer le
                             * system binary <B3h>
                             * niveau 2 type 11 :
                             * entier binaire
                             * niveau 1 type 3
                             * chaîne
  ::                         * marqueur bloc de pro-
                             * gramme à exécuter si
                             * arguments acceptés
    SWAP                     * niveau 2:      chaîne
                             * niveau 1: entier bin.
    HXS>#                    * conversion en system
                             * binary
CODE                         * indique au compila-
                             * teur le mode assem-
                             * bleur
        gosbvl  =POP#        * a(a) := adresse où
                             * poker et drope le
                             * niveau 1
        gosbvl  =SAVPTR      * sauve les pointeurs
        d0=a                 * pointe l'adresse du
                             * poke
        a=dat1  a            * a(a) =: adr. chaîne
        d1=a                 * pointe cette adresse
        d1=d1+  5            * pointe (longueur+5)
        a=0     m            * pour asrb après
        a=dat1  a            * (a(a) =: longueur+5)
        a=a-con a,5          * a=a-5 a   longueur de
                             * la chaîne en quartets
        asrb                 * longueur du code à
                             * poker
        b=a     a            * compteur de digits
        d1=d1+  5            * début des codes à
                             * poker
        lchex   39           * pour la conversion
        gonc    decr         * saut pour le départ
loop                         * conversion
                             * ASCII -> HEXA
        a=dat1  2            * lit 1 caractère
        ?a<=c   b            * digit < 10 ?
```

```
        goyes   inf10       * alors on le garde
        a=a-con   b,7       * a=a-7 b   pour d > 9
inf10   dat0=a  1           * poke 1 digit
        d1=d1+  2           * actualise les
        d0=d0+  1           * pointeurs
decr    b=b-1   a           * actualise le compteur
        gonc    loop        * si encore 1 digit
        govlng =GETPTRLOOP  * restaure les
                            * poineurs et retour au
                            * rpl
ENDCODE                     * fin du mode
                            * assembleur
DROP                        * élimine l'adresse
  ;                         * marqueur de fin
                            * correspondant à ::
                            * après vérification
                            * des arguments
;                           * marqueur de fin de
                            * programme
```

ADDR

syntaxe :
niveau 1 : l'objet rappelé sur la pile

résultat :
niveau 1 : l'adresse de l'objet → 1 entier binaire

Listing du programme ADDR :

```
RPL

::
CK1                         * vérifie s'il y a 1
                            * niveau de pile
ZERO                        * system binary <0>
TOTEMPSWAP                  * crée cet objet en Ram
                            * puis SWAP
                            * niveau 2 : <0>
                            * niveau 1 : l'objet
CODE                        * mode assembleur
      a=dat1  a             * adresse de l'objet
      d=d+1   a             * drope celui-ci
      d1=d1+  5             *
      c=dat1  a             * adresse de <0>
      cd1ex                 * pointe <0>
                            * c(a) =:adresse de <0>
      d1=d1+  5             * passe le prologue
      dat1=a  a             * remplace <0> par
                            * l'adresse de l'objet
      d1=c                  * restaure d1
      govlng  #05149        * actualise pc et
                            *  retour au rpl
ENDCODE                     * fin boc assembly code

#>HXS                       * conversion syst. bin.
                            *   -> entier binaire
;                           * fin programme
```

Vous trouverez les chaînes de codes à assembler dans le *coin de codes*, à la fin du journal.

Le but de cette rubrique était donc de fournir des outils pour les curieux et de donner des exemples d'utilisation de la *rpl*, ce que n'ont pas fait P.Courbis et S.Lalande. On remarquera que je n'ai pas proposé de programme CHK pour vérifier le nombre et la nature des objets. En effets toute une famille de routines existe dans la Rom de nos machines pour faire cela beaucoup mieux que CHK. Au travers des différents programmes que je vous ai proposés, une utilisation de quelques unes de ces routines a toujours été faite.

Lorsque le nombre de programmes augmentera, nous pourrons envisager de constituer une librairie qui permettra d'exécuter chaque programme stocké en Rom ce que ne permet pas un répertoire normal.

J'espère qu'il va se constituer un bon noyau de programmeurs dans le domaine de l'assembleur comme comme au bon temps de la HP-71. Malgré tout ce qui existe déjà dans le monde il y a encore des beaux jours pour les découvertes de toutes sortes.

Guy Toublanc (276)

CHOC EN RETOUR

Le programme EDIV en assembleur et publié dans *JPC 79* page 10, peut poser quelques problèmes si on taquine la pile avec CMD aussi je vous propose une version corrigée et améliorée. En effet il m'a semblé qu'il serait plus pratique d'avoir pour les entrées et les sorties les mêmes types de données :

- Si l'argument est un réel (entier positif ou négatif) le résultat sera un réel

- Si l'argument est un entier binaire le résultat sera un entier binaire alors que pour la version précédente le résultat était toujours un réel.

D'autre part j'ai ajouté, pour préserver l'avenir, la possibilité de préciser la limite des tests de divisibilité.

```
EDIV
```

syntaxe :

niveau 1: 1 réel ou 1 entier binaire ou 1 liste

n	#n	{ N l }
résultat: 1 réel positif r	1 entier binaire #r	nombre de même type que N

n : entier de valeur absolue < 10^{12}

#n : entier binaire < 2^{64}

N : n (résultat → r)
#n (résultat → #r)
l : limite des tests de divibilité
1 réel entier ou 1 entier binaire
si aucun facteur premier < = à l
alors résultat → 0

résultat : r ou #r
si argument premier → 0
sinon → plus petit facteur premier

Extrait du fichier source

Le partie *Rpl* du programme a seulement été modifiée aussi ce qui ne change pas ne sera pas listé. La présentation sera celle justifiée dans l'article *Les débuts en Rpl*, paru le mois dernier.

```
RPL                                  les modifications
 ::                                  ---------------]
  CK&DISPATCH1                                      ]
  ELEVEN                                            ]
   ::                                               ]
     DUP #>% %SQRT %># SWAP                         ]
CODE                                                ]
        GOTO    start                               ]
ENDCODE                                             ]
   ;                                                ]
  list                                              ]
   ::                                               ]
     DUPLENCOMP #2= ?SKIP SETTYPEERR                ]
     INCOMPDROP                                     ]
     ::                                             ]
       DUPTYPEREAL? case :: %ABS %># SWAP ;         ]
       DUPTYPEHSTR? case :: HXS>% %># SWAP ;        ]
       SETTYPEERR                                   ]
     ;                                              ]
     ::                                             ]
      DUPTYPEREAL? case :: %ABS %>#                 ]
CODE                                                ]
       GOTO      start                              ]
ENDCODE                                             ]
                          #>% ;                     ]
       DUPTYPEHSTR? case ::                         ]
CODE                                                ]
       GOTO      start                              ]
ENDCODE                          ;                  ]
       SETTYPEERR                                   ]
     ;                                              ]
   ;                                                ]
  real                                              ]
   ::                                               ]
    %ABS  DUP %SQRT %># SWAP %>#  ------------------]
CODE                             ******************
start   GOSBVL =PopASavptr                |      |
        ........                          | J    |
        ........                          |  P   |
        ........                          |   C  |
        ........                          |      |
        ........                          | 7 9  |
        C=DAT1 16                         |      |
        RTN                               | p10  |
ENDCODE                                   |      |
   #>%                                    |      |
   ;                                      ******************
 ;
```

Vous trouverez la liste des codes dans le *coin des codes*, à la fin du journal.

Les programmes d'applications de EDIV ne sont pas modifiés sauf GDEC qui nécessite un argument de type binaire (le programme R?B n'est plus utile) et qui renvoie une liste d'entiers binaires. Voici ce programme modifié.

```
GDEC

« { } SWAP
  WHILE DUP EDIV
    # 0d OVER ≠ OVER
    # 1d ≠ AND
  REPEAT
    DO ROT OVER 1
      →LIST + SWAP ROT
      OVER / SWAP
    UNTIL OVER B→R
      OVER B→R MOD
    END DROP
  END DROP DUP # 1d
  ≠
  « 1 →LIST + »
  « DROP »
  IFTE
»
```

Pour aujourd'hui il s'agissait seulement de donner une version déboguée et améliorée de EDIV, vous réservant des explications plus détaillée pour la partie *Rpl* qui me servira d'exemple pour la suite des *Débuts en Rpl*.

Guy Toublanc (276)

HP-95

J. Belin — Du Saturn au 8086 (Acte II) — *16*

DU SATURN AU 8086
ACTE II

Cet article constitue la deuxième partie d'une introduction à l'assembleur 8086 sur IBM PC, destiné aux programmeurs du microprocesseur Saturn sur HP-71, désirant migrer sur le HP-95. Je vous rappelle que ces articles sont conçus comme étant une introduction à l'assembleur sur l'IBM PC, et ne se veulent pas une référence à ce domaine. Certains aspects ont été intentionnellement omis, et je vous conseille expressément de vous référer à un des nombreux livres afin d'obtenir des précisions supplémentaires sur l'assembleur, le fonctionnement de l'IBM PC et les fonctions internes (voir bibliographie en fin d'article).

Après avoir abordé, dans le numéro 79, les jeux d'instructions des deux microprocesseurs, nous allons maintenant voir la structure et la façon d'écrire les programmes MS-DOS.

Structures des programmes HP-71 et MS-DOS

Il existe deux types de programmes binaires sur le HP-71 : les programmes BIN et LEX. Le premier type ne peut être appelé que sous forme de sous-programme, à l'aide de l'instruction CALL. Les programmes LEX possèdent la particularité de gérer ces sous programmes de façon plus complète, comme si ils étaient des mots clefs, au même niveau que les fonctions de la Rom. Ceci est effectué grace à la présence, au début de chaque Lex, d'une table contenant des données attribuées à chaque mot clef, grace au couple ID/Token. Ils possèdent en plus d'autres possibilités, comme la gestion des "Poll Handlers" et l'ajout de tables de messages.

Sur l'IBM PC, il existe aussi deux sortes de programmes, portant l'extension COM ou EXE. Cependant, les différences entre ces deux types de programmes sont plus subtiles mais moins importantes que pour les programmes HP-71. En effet, un programme portant l'extension EXE possèdent quelques différences internes, que nous verrons plus tard, mais ne possèdent aucune différence quant à leur utilisation.

Lorsque nous lançons un programme sur un IBM PC, MS-DOS charge le programme en mémoire vive (à partir du disque) en allouant la place nécessaire, lance l'exécution (le programme lui-même peut alors utiliser le reste de la mémoire vive pour ses données), puis libère la mémoire à la fin de l'exécution afin de garder la place libre pour un autre programme. En dehors du système d'exploitation et des éventuels drivers de périphériques, il n'y a donc en théorie qu'un seul programme présent en mémoire. Il est cependant possible de conserver certains programmes spéciaux en mémoire, en les rendant résidents. Dans ce cas, le système d'expoitation ne libère pas la mémoire utilisée par le programme et place le programme suivant à la suite de la mémoire occupée par le résident. Nous reviendrons plus tard sur ce type de programme.

Le système d'exploitation du HP-71 procède d'une action un peu plus complexe. En effet, sur cette machine, les mémoires de masse (servant au stockage des programmes et des données) et de travail (servant au stockage des programmes en cours d'exécution et des données) cohabitent de façon plus étroite. En effet, les programmes n'ayant pas à être chargés à partir d'un périphérique externe sont en permanence accessibles par le système d'exploitation, qui peut les exécuter sans avoir à les déplacer dans une zone de mémoire réservée à cet effet. La mémoire libre est donc alors utilisée uniquement par les données créées par le programmes en cours d'exécution.

En fait, structurellement parlant, les programmes BIN, la plupart des mots clefs des LEX et les programmes MS-DOS pouraient être considérés comme étant équivalents, puisque dans tous les cas, on ne fait qu'appeler une fonction, avec les paramètres nécessaires. Le fait que le programme soit exécuté directement ou non à partir de la mémoire de masse est uniquement le problème du système d'expoitation, et n'entre pas en jeu dans la façon dont doivent être écrits les programmes BIN et MS-DOS, qui ont une structure relativement équivalente.

Par contre, la structure des fichiers LEX est beaucoup plus complexe, puis qu'elle inclut des mécanismes qui sont inconnus par MS-DOS. Le premier est bien sûr le fait que le système d'exploitation du HP-71 peut accéder directement, à partir d'un fichier LEX à un ou plusieurs mots clefs distincts contenus dans celui-ci, alors qu'il n'existe qu'un seul mot clef correspondant au nom du fichier dans le cas des programmes MS-DOS. Une autre chose, prise en compte sur le LEX seul, est la possibilité de gérer des tables de messages.

Plus intéressante est la comparaison entre la notion de *Poll Handler* et celle de programme résident de MS-DOS. En fait, un Poll Handler peut être assimilé, dans une certaine mesure, à un gestionnaire d'interruption tel que ceux qui sont utilisés pour l'activation des programmes résidents. La grande différence entre les deux machines est que ce mécanisme est inclus d'origine dans la strucure du fichier LEX, alors que sa mise en oeuvre sous

MS-DOS est totalement à la charge du programmeur (pour plus de détails, voir mon article sur TJR8 dans *JPC 78*). De plus, un poll handler est un mécanisme purement logiciel, résultant d'un appel décidé par un programme ou le système d'exploitation, alors qu'une interruption peut être activée à n'importe quel moment par un evénement interne ou externe à l'ordinateur. Mais ceci dit, rien ne nous empêche de créer pour nos programme résidents un système permettant d'émuler, par exemple, la fonction VER$ du HP-71. Celui-ci serait composé d'un gestionaire d'interruption implanté de telle façon que le contrôle de l'exécution serait chaîné vers tous les résidents concernés. Ce gestionnaire afficherait une chaine de caractère vers l'affichage, puis transfererait le contrôle vers le résident suivant. L'activation de ce mécanisme serait effectué par un simple programme COM (de moins de dix octets) qui ne ferait qu'activer une interruption que nous aurions choisie parmi celles qui ne sont pas utilisées par les programmes ou le système d'exploitation.

Une autre différence fondementale entre les programmes MS-DOS et les programmes binaires HP-71 concerne la mise en oeuvre des variables utilisées pour le fonctionnement du programme. Sur le HP-71, nous avons l'habitude de stocker ces variables dans certaines parties de la mémoire réservée, aux adresses STMTRO ou SNAPBF par exemple. Avec MS-DOS, par contre, de telles zones n'existent pas. Afin de simplifier la conception des programmes, il est donc usuel de placer les variables dans le programme lui même, avant la zone réservée au code. Quasiment tous les programmes MS-DOS utilisent cette méthode. Il existe cependant quelques problèmes avec cette approche. La première est que si le programme utilise de nombreuses variables ou des tableaux de grandes taille, le programme EXE ou COM résultant sera d'autant plus gros. Ce problème peut être cependant éliminé en demandant une zone de mémoire libre au système d'exploitation, au détriment d'un peu de code supplémentaire. Le second problème, qui peut apparaitre dans certains cas (dont le HP95 fait partie, avec les fonctions MS-DOS intégrées en Mémoire morte) est que cette méthode interdit l'exécution du programme directement à partir de la ROM. Il faut donc réécrire ces programmes de telle façon que les données soient situées en mémoire vive, en utilisant la méthode décrite précédement pour les programmes utilisant beaucoup de données.

A propos, ceci nous amène enfin à deux autres types de programmes spécifiques au HP95. Les programmes portants l'extension EXM sont destinés à être exécutés sous le *System Manager*. Ils ont à peu près la même structure que les programmes EXE, à ceci près que les zones de code et de données sont totalement séparées. Ceci permet, en cas de changement d'application (pour lancer Lotus ou le Memo par exemple) de conserver les données en mémoire, alors que la zone utilisée par le code est réutilisée immédiatement par le code du nouveau programme. Un autre type de programme, prévu mais non utilisé actuellement sur le HP95, porte l'extension XIP (eXecute In Place). Il est destiné à l'exécution des programmes directement à partir des cartes mémoire (Rom ou Ram), sans transfert du code dans la mémoire vive.

Maintenant que nous avons abordé la structure générale des programmes, nous pouvons maintenant aborder plus en détail leur programmation. Pour cela, nous allons utiliser le thème bien connu du programme affichant `Hello, World` sur l'écran de notre ordinateur, en comparant les sources pour les deux machines.

Programme HP-71

Pour cette machine, nous créerons un fichier BIN plutôt qu'un LEX, car ce type de fichier correspond plus à la structure d'un fichier MS-DOS. Nous ferons cependant de nombreuses références aux fichiers LEX, quand cela sera utile.

```
        BIN     'HELLO'
        CHAIN   -1
BF2DSP  EQU     #01C0E
ENDBIN  EQU     #0764B

        GOSUB   POP
        NIBASC  'Hello,'
        NIBASC  ' World'
        NIBHEX  D0A0FF
POP     C=RSTK
        D1=C
        GOSBVL  BF2DSP          affichage chaine
        GOVLNG  ENDBIN          fin programme
        END
```

Programme MS-DOS

```
        DOSSEG
        .MODEL  SMALL
        .STACK  100h
        .DATA
Chaine  DB      "Hello, World",13,10,"$"
        .CODE
start:  mov     bx,@data
        mov     ds,bx
        mov     dx,OFFSET Chaine
        mov     ah,09h
        int     21h
        mov     ah,4ch
```

```
int    21h
END    start
```

En-Têtes des programmes

La première ligne des fichiers HP-71, contenant le type d'exécutable (LEX, BIN ou FORTH) suivi du nom du fichier résultat n'a pas d'équivalent sous MS-DOS car le nom de ce fichier correspondra soit au nom du fichier source, soit à une option de la ligne de commande.

Toujours dans le fichier HP-71, la directive CHAIN est utilisée dans le cas où le fichier BIN contient plusieurs sous programmes, appelables séparément par l'instruction CALL. Comme mentionné préxédement pour les fichiers LEX, chaque fichier MS-DOS n'est appelable que par son nom de fichier. Il n'y a donc pas d'équivalent à CHAIN dans les programmes MS-DOS.

La directive EQU existe aussi sur l'assembleur Microsoft et peut être utilisée de façon identique. Nous ne l'utilisons pas cependant pour définir des adresses de la ROM, car nous utilisons une autre méthode pour accéder aux resources du système. Ceci sera précisé plus loin dans cet article.

La première ligne du fichier MS-DOS contient la directive DOSSEG. Celle-ci permet d'indiquer à l'assembleur d'ordonner le programme en suivant les conventions définies par Microsoft, à savoir placer d'abord le code, puis les variables dans le fichier executable, à l'inverse de ce qui est écrit dans le fichier source.

La directive .MODEL (ne pas oublier le point) permet de définir la façon dont seront traités la segmentation des programmes, les appels de fonctions et les réferences aux variables. Six modèles de gestion de la mémoire sont définis :

TINY	Code + Données = 64 Ko Maximum
SMALL	Code < 64 Ko, Données < 64 Ko
COMPACT	Code < 64 ko, Données > 64 ko
MEDIUM	Code > 64 ko, Données < 64 ko
LARGE	Code > 64 ko, Données > 64 ko (blocs< =64k)
HUGE	Code > 64 ko, Données > 64 ko (blocs>64k)

Si nous prenons pour exemple le modèle COMPACT, cela veut dire que la taille du code sera d'un maximum de 64 Ko. Les appels de fonctions et les sauts seront donc calculés sur 16 bits par des appels NEAR, en utilisant seulement la valeur de l'offset du segment de code utilisé (en fait, seul le registre IP sera mis à jour). Les données, par contre, pourront avoir une taille totale supérieure à 64 ko. Chaque bloc de donnée (un tableau par exemple) ne peut cependant pas avoir une taille supérieure à 64 ko.

Dans le modèle TINY, le code et les données partagent le même segment. Le programme complet ne peut donc pas excéder 64 ko.

Le modèle HUGE peut par contre utiliser des tableaux de données d'une taille supérieure à 64 ko, puisque toutes les données sont accessibles sur 32 bits.

En général, pour nos petits programmes, nous utiliserons de préférence les modèles SMALL et TINY.

D'autres options de .MODEL permettent de contrôler la façon dont sera assemblé le fichier, dans le cas où celui-ci sera interfacé avec un langage de haut niveau, par exemple.

La directive .STACK permet de définir une un segment de mémoire devant contenir la pile de données du programme. Cette pile sera utilisée, par exemple, pour sauvegarder le contenu courant des registres pendant les appels aux interruptions que nous verrons plus bas. Le paramètre correspond à la taille de la pile, en octets. Cette zone ne fait pas partie intégrante du programme et sa taille n'a pas d'influence sur celle du programme. Cependant, ne spécifiez pas une valeur trop grande, car cela sera au détriment de la mémoire vive disponible. Au contraire, une pile trop petite provoquera un écrasement des données avoisinantes ou du code.

La directive .DATA permet de définir un segment de mémoire devant contenir les variables utilisées par le programme. La déclaration de chaque variable est composée d'une étiquette indiquant son nom, du type de la variable définissant sa longueur et éventuellement de données initialisées.

Dans notre exemple, nous déclarons une variable contenant une chaine de caractères, que nous initialisons avec la chaine que nous désirons afficher.

Il y a 7 types de données, permettant de réserver la place nécessaire au stockage des différents types de variables :

DB	1 octet
DW	2 octets : Mot (Word)
DD	4 octets : Double mot
DF,DP	6 octets : Pointeur 386 Long
DQ	8 octets : Quadruple mot
DT	10 octets

Les variables peuvent être initialisées avec une ou plusieurs valeurs de la façon suivante :

```
Var1    DB      2Fh
```
Alloue un octet initialisé à 2F.
```
Var2    DB      2Fh,67,(16*7+2)
```
Alloue un tableau de 3 octets initialisés
```
Var3    DB      "ABC",13,10
```
Alloue une chaine de caractères terminée par CR/LF.
```
Var4    DW      3B4Fh, 2Ch
```
Alloue un tableau de deux mots.

Il est possible de déclarer une variable sans cependant l'initialiser. Pour cela, il faut remplacer les données d'initialisation de la façon suivante :

```
Var5    DW      (?)
```
Alloue un mot, sans l'initialiser.

Il est enfin possible d'allouer un tableau, en initialisant chaque élément avec une valeur déterminée, grace à la directive DUP :

```
Var6    DB      25  DUP ('Z')
```
Alloue un tableau de 25 octets, avec tous les octets initialisés avec le caractère 'Z'.

Il est bien sûr aussi possible d'allouer un tableau, sans vouloir l'initialiser :

```
Var7    DW      100h  DUP (?)
```
Alloue un tableau de 256 mots, non initialisés

Une fois que les variables sont déclarées, il est possible d'y accéder directement en donnant leur nom comme paramètre d'une instruction :

```
var1    DW      (?)
        .
        .
        mov     [var1],ax
```

Ceci placera le contenu du registre AX dans la variable var1.

Il est aussi possible de spécifier les variables de telle façon que l'on puisse accéder aux différents éléments les constituant :

```
Var7    LABEL WORD
Var7_l  DB    (?)
Var7_h  DB    (?)
```

La directive LABEL WORD n'alloue pas par elle-même de mémoire, mais indique que les données déclarées ensuite peuvent être accédées ensemble en utilisant le nom Var7, qui pointe sur un mot. Par exemple :

```
        mov [Var7_h],02h
        .
        .
        mov [Var7_l],08h
        .
        .
        mov bx,[Var_7]
```

Placera 0208h dans le registre BX.

Corps du programme

Dans le programme EXE, nous devons indiquer à l'assembleur le début du segment de code. Cela est fait en plaçant la directive .CODE juste avant la première instruction.

La première instruction du code doit être précédée d'un label, dont le nom peut être choisi librement par le programmeur. Vous pouvez constater (contrairement à ce qui se fait sur l'assembleur Saturn) que ce nom est suivi d'un caractère ":". Cela est obligatoire pour tous les labels indiquant une adresse de branchement (pas ceux référencant une variable ou un segment).

La grande différence dans le fonctionnement des deux programmes concerne la façon dont on accède à la chaine de caractères. Sur le HP-71, la fonction BF2DSP nécessitant la connaissance de l'adresse absolue de cette chaine. Or, le code assemblé ne peut pas donner directement cette adresse, car le programme n'est pas logé à une adresse fixe. On a recours alors a une astuce consistant à placer la chaîne juste après un GOSUB, et à dépiler ensuite l'adresse de retour pour obtenir l'adresse de la chaine.

Sur le PC, par contre, les programmes EXE ont la perticularité de posséder, dans leur en-tête, une table dite "de relocation" contenant l'emplacement de chaque instruction utilisant une valeur de segment. Au tout début de l'exécution du programme, le système d'exploitation connait l'emplacement réel du programme dans la mémoire. Il pourra donc lire cette table et stocker les valeurs réelles de segment dans le programme, dans chaque instruction le nécessitant. En cours d'exécution le programme traitera ces valeurs comme si elles avaient été entrées sous forme de constantes.

Dans notre programme EXE, nous devons utiliser cette possibilité, car nous utilisons un appel de fonction nécessitant de connaitre la valeur de segment de notre chaine de caractères. Cette valeur doit être stockée manuellement dans le registre DS, car le système d'exploitation ne se charge pas de la manoeuvre. Pour cela, nous devrons donc entrer les lignes suivantes :

```
mov     bx,@data
mov     ds,bx
```

Le terme `@data` permet de faire savoir à l'assembleur que le paramètre devra être pris en compte dans la table de relocation, afin que le programme exécute l'instruction avec la valeur du segment de données.

En plus de la valeur de segment, notre instruction d'affichage de chaîne nécessite aussi l'adresse de la chaine par rapport au début du segment. Dans ce cas, il n'y a pas de problème car cette valeur est indépendante de la position du programme dans la mémoire. Il suffira donc d'écrire la ligne :

```
mov     dx, OFFSET Chaine
```

Maintenant que nous avons décrit la façon d'accéder aux données, nous devons maintenant les exploiter. Les méthodes d'accès aux resources systèmes des deux machines sont en même temps très proches et trés différentes. En effet, si dans les deux cas nous devons charger des valeurs dans certains registres, puis éventuellement lire le résultat dans d'autres registres, le mécanisme d'appel à ces fonctions est totalement différent. Sur le HP-71, nous effectuons simplement un appel vers un sous programme, présent à une adresse normalisée de la ROM. Or, les concepteurs de la ROM de l'IBM PC ont prévu le fait que les différentes versions du BIOS ne pourraient pas assurer que toutes les fonctions soient situées à la même adresse. Il ont donc choisi d'accéder à ces fonctions par l'intermédiaire d'appels à des interruptions, qui elles pouraient être normalisées. Sous MS-DOS, nous considérons qu'il existe deux types d'interruptions : Les interruptions BIOS et les interruptions DOS. En simplifiant, les premières traitent de tous les accès au matériel, qui sont indépendants du système d'exploitation, comme par exemple les accès de bas niveau sur les disquettes ou le disque dur. Les secondes traitent, par exemple, de tout ce qui concerne la création et la façon dont sont stockés les fichiers. En général, les fonctions traitant d'un même sujet sont regroupées au sein de la même interruption. Par exemple, l'interruption 10 gère tous les accès de bas niveau au contrôleur vidéo. La plupart des interruptions MS-DOS sont regroupées dans l'interruption portant le numéro 21h. Il n'existe malheureusement pas, sur l'IBM PC, certaines fonctions qui ont fait notre bonheur sur le HP-71, telles les fonctions mathématiques ou les manipulations de chaînes. En fait, il y a moins de 300 fonctions DOS ou BIOS sur le PC, contre plus de 1000 sur le HP-71.

Au sein de chaque interruption, les différentes fonctions sont accessibles en plaçant un code normalisé dans le registre AH. En sortie, nous pouvons, comme sur le HP-71, récupérer des valeurs dans certains registres, ou tester le Carry.

Pour le programmeur, il n'y a pas de complication supplémentaire, puisque dans les deux cas il suffit de consulter sa documentation technique afin de trouver la fonction qui nous intéresse et les registres à prendre en compte.

La méthode des interruptions permet en outre d'avoir la possibilité de modifier le comportement des fonctions de la ROM, en détournant les vecteurs d'interruptions vers des fonctions que nous avons réécrites. C'est ce qui permet, par exemple, de créer des utilitaires de cache disque comme SMARTDRV, qui traitent différement les fonctions d'accès au disque. Sur le HP-71, le fait que nous accèdions à une adresse fixée à l'avance nous interdit de créer ce type de programmes. Seules certaines fonctions, gérées par des polls handlers nous permettent de modifier leur comportement.

En ce qui concerne notre programme, les deux fonctions d'affichage de chaîne, ont un fonctionnement très équivalent. Sur le HP-71, nous utilisons la fonction BF2DSP qui demande l'addresse du premier caractère dans le registre D1. Sur le PC, nous utiliserons la fonction 09h du DOS (interruption 21h) qui demande aussi l'adresse de la chaîne, mais avec ses deux composandes (offset et segment) dans le couple de registre DS et DX. La seule différence concerne le fait que le caractère indiquant la fin de la chaîne est le signe "$" sous MS-DOS, alors que c'est le caractère de code 255 (FF en hexa) sur le HP-71.

Vous pouvez constater qu'il n'y a pas de différences dans la démarche du programmeur, puisque dans les deux cas, il nous suffit de consulter une documentation technique afin de connaitre un nom ou un numéro de fonction, accompagné des registres utilisés par celle ci.

Si l'on veut être rigoureux, ou en cas d'assemblage de fichiers sources multiples, nous devons déclarer les sous programmes à l'aide de directives spécifiques, permettant à l'assembleur de savoir si il doit génerer des appels longs ou courts. Cette déclaration peut être de la forme suivante, pour la fonction `fct_1` :

```
        .
        .               ;programme
        call    fct_1
        .
        .

fct_1   LABEL   PROC NEAR  ;sous programme
        mov     bx,1
        .
```

```
        .
        ret
fct1    ENDP


fct_2   LABEL  PROC FAR
        .
        .
        .
```

La directive LABEL PROC NEAR indique que le call devra être généré avec un appel court (sur 16 bits, dans le même segment). Si le NEAR est remplacé par FAR, ce sera un appel long (sur 32 bits, de la forme segment:offset) qui sera généré.

La directive ENDP indique la fin du sous-programme.

Vous pouvez aussi noter, dans ce source, la présence de commentaires. Le caractère ";" indique que les caractères suivants appartiennent à des commentaires. Contrairement à l'assembleur Saturn, ce caractère est toujours obligatoire, même après une ligne de code.

Fin du programme

En fin d'exécution du programme, sur les deux machines, notre programme doit retourner sous le contrôle du système d'exploitation en exécutant une fonction prédéfinie.

Sur le HP-71, nous procédons à un GOVLNG vers des adresses prédefinies : ENDBIN, NXTSTM, FNRTN1...

Sous MS-DOS, le retour est effectué en appelant la fonction 4Ch de l'interruption 21h. D'une façon identique à FNRTN1, Il est possible de renvoyer une valeur, pouvant être testée dans un fichier BAT par la commande ERRORLEVEL. Ceci est fait en placant le code de retour dans le registre AL, avant l'appel de l'interruption.

Il existe une autre fonction permettant de retourner au DOS, portant le numéro 31h, qui permet au programme de rester résident. C'est celle ci qui m'a permis de créer les utilitaires TJR8 et TJPS parus dans des précédents JPC.

La dernière instruction de notre source est proche de ce qui se fait sur le HP-71, puisque il s'agit d'une directive END. Nous devons cependant y ajouter, en paramètre, le label correpondant à la première instruction exécutée par le programme.

Compilation du programme

Le source de ce programme peut aussi bien être compilé avec l'assembleur Microsoft (MASM) qu'avec celui de Borland (TASM). En effet, et en ce qui concerne ce que nous avons déjà vu, ils sont totalement compatibles entre eux. La seule différence concerne le nom des programmes.

Le programme que nous avons écrit ayant été, par exemple, sauvegardé sous le nom HELLO.ASM (ASM étant l'extention standard pour un fichier assembleur), nous désirons obtenir un programme exécutable portant le nom HELLO.EXE. Pour cela nous exécuterons, si nous utilisons l'assembleur Microsoft, les commandes suivantes :

```
masm hello
```

Avec le *Turbo Assembler* ce sera :

```
tasm hello
```

Ceci ne créera pas tout de suite le fichier EXE, mais un fichier "objet" intermédiaire portant l'extension OBJ. Pour transformer ce fichier en exécutable, nous devons ensuite utiliser la commande suivante :

```
link hello
```

ou, pour l'assembleur Borland :

```
tlink hello
```

Nous obtenons ainsi le fichier HELLO.EXE, qui peut être exécuté comme n'importe quel programme.

Cette procédure en deux étapes est peut être contraignante, mais cela est dû au fait que ces utilitaires ont été conçus pour avoir la possibilité de compiler des programmes ayant été écrits de façon modulaire, sur plusieurs fichiers sources indépendants. Il est ainsi possible de se créer des bibliothèques de fonctions, que nous stockerons dans des fichiers objets séparés, puis un programme utilisant ces fonctions, qui sera lui aussi stocké dans un fichier objet indépendant. Par exemple, si notre programme HELLO affichait son texte au milieu d'un cercle, il serait intéressant de créer un module indépendant gérant le graphisme, que nous appellerions peut-être GRAPH.ASM. Dans ce cas nous procèderions de la façon suivante :

```
masm hello
masm graph
link hello graph
```

La dernière opération, appelée *Edition de Liens* consiste à lier les appels des fonctions et les références aux variables entre les différents modules. Vous avez pu noter que je n'ai pas mentionné les différentes extensions OBJ ou EXE, car les utilitaires les utilisent par défaut. Dans le dernier exemple, le nom du fichier EXE résultant sera toujours celui correspondant au premier fichier objet mentionné sur la ligne de commande.

Conversion du programme EXE en COM

Le progamme que nous venons de compiler fonctionne parfaitement (enfin, je l'espère !). Cependant, sa taille (plus de 500 octets) est sans commune mesure à ce qu'il exécute réellement. Ceci est dû principalement à la présence de la table de relocation dont nous avons parlé précédement.

Il existe cependant d'autres différences entre les deux types de programmes. La plus importante est qu'un fichier EXE n'a pas de limitation de taille (si ce n'est celle de la mémoire vive) alors qu'un programme COM est limité à une taille maximale de 64 Ko, données comprises. De plus, le programme COM ne possède pas de segment de pile séparé et doit utiliser la zone de mémoire située entre la fin du programme et la limite de 64 ko (ce qui veut dire que si le programme approche de très près de 64ko, il risque de ne pas avoir suffisement de place dans la pile). En fait, le programme COM ne peut occuper et utiliser que le code et les données contenues dans un seul et même segment, alors qu'un programme EXE peut être composé d'un nombre indéterminé de segments stockant du code ou des données. Nous verrons cela un peu plus en détail plus loin dans cet article. En terme d'utilisation, il n'y a absolument aucune différence entre ces deux types de programmes.

Pratiquement, nous devrons effectuer quelques modifications sur notre programme afin de le rendre convertible en COM.

Mais, au point où nous en sommes, profitons en pour parler de l'autre manière d'écrire les programmes en assembleur.

Il existe deux façons d'indiquer à l'assembleur la façon dont nous voulons qu'il traite la segmentation. Les termes qui sont utilisés sont *Segmentation Standard* et *Segmentation Simplifiée*. Celle que nous avons vu précédement est la segmentation simplifiée, qui est nommée ainsi car nous n'avons besoin que de quelques instructions (.MODEL, .DATA...) placées au début du programme pour indiquer à l'assembleur comment attribuer et traiter les segments. Si nous voulons un contrôle plus précis de notre travail, nous pouvons utiliser la segmentation standard.

Pratiquement, nous devrons déclarer explicitement tous les segments qui seront utilisés par le programme. Ceci est fait en remplaçant les directives .STACK, .DATA et .CODE par d'autres directives, de la façon suivante :

```
        DOSSEG
Stack   SEGMENT PARA STACK 'STACK'
        DB      200h  DUP (0)
Stack   ENDS
Data    SEGMENT PARA 'DATA'
Chaine  DB      "Hello, World",13,10,"$"
Data    ENDS
Code    SEGMENT PARA 'CODE'
        ASSUME cs:Code,ss:Stack,ds:Data
start:  mov     bx,Data
          .
          .
          .
Code    ENDS
        END     start
```

La directive SEGMENT permet de définir le début d'un segment, de telle sorte que le label associé soit accessible à cette adresse avec un offset égal à 0. Le paramètre PARA indique permet de positionner ce segment au début du prochain paragraphe (groupe de 16 octets) physiquement adressable en mémoire vive. D'autres valeurs permettent par exemple d'aligner le segment sur le prochain octet (BYTE), le prochain mot (WORD) ou la prochaine page (groupe de 256 octets, PAGE).

Les paramètres 'STACK', 'DATA' et 'CODE' permetttent à l'editeur de liens de regrouper sous un même segment tous les modules de code ou de données ayant le même paramètre.

Le paramètre STACK placé entre PARA et 'STACK' permet à l'éditeur de liens de placer dans les registres SS:SP,, en début d'exécution du programme, les valeurs correspondant à la fin du segment. En effet, la pile se remplit à partir des adresses hautes vers les adresses basses de la mémoire.

La directive ASSUME indique à l'assembleur quels registres sont associés à ces segments. Ceci lui permettra de savoir, par exemple, à partir de quel segment effectuer le traitement de l'opérande [di] dans une instruction du type mov bx,[di]. Ceci permet aussi de controler la validité de ce type de références. Notez bien cependant que cette directive ne se substitue pas à l'obligation de charger un registre avec la valeur du segment, par une instruction MOV. Les paramètres de ASSUME sont composés du nom du registre et du label du segment séparés du caractère ":".

Enfin, la directive ENDS indique la fin du segment spécifié par le label placé juste avant.

Vous pouvez aussi remarquer que pour la première ligne de code, le terme @Data a été remplacé par Data, qui correspond au label. Ceci n'est pas une erreur, elle correspond à la notation en vigueur dans ce mode de segmentation.

Maintenant que nous avons abordé le principe de la segmentation standard, revenons à notre programme, et étudions les modifications à apporter afin de le transformer en COM.

Premièrement, nous avons dit que ce type de programme ne comportait pas de pile. Nous devons donc supprimer tout ce qui concerne le segment Stack, y compris dans la directive ASSUME.

Ensuite, nous intégrons le segment de données à l'intérieur du segment de code. Habituellement nous gardons ces données au début du programme. Or, en début d'exécution, le pointeur du programme est positionné sur le premier octet du segment. Il faut donc impérativement y placer une instruction de saut, qui permettra de passer au dessus des données et continuer l'exécution du programme après celles-ci.

Les programme COM ne possèdant pas de table de relocation, nous ne pouvons pas effectuer des chargements de registres à l'aide de références de segments. Nous devons donc remplacer, par exemple, les lignes :

```
strt:   mov     bx,Data
        mov     ds,bx
```

par :

```
strt:   mov     bx,cs
        mov     ds,bx
```

En effet, les données étant maintenant intégrées au segment de code, le plus simple pour connaitre celui-ci est de lire sa valeur directement dans le registre CS, qui est toujours valide.

Enfin, après leur chargement en mémoire, l'exécution d'un programme COM commence à l'adresse 100h (par rapport au segment de code) alors que les programmes EXE commencent à l'adresse 0h. Il faut donc ajouter une directive indiquant à l'assembleur d'en tenir compte. Ceci est fait à l'aide de la directive ORG 100h.

En définitive nous obtenons le programme suivant :

```
        DOSSEG
code    segment para 'CODE'
        assume cs:code
        ORG     100h
start:  jmp     strt


Chaine  DB      "Hello, World",13,10,"$"

strt:   mov     bx,cs
        mov     ds,bx
        mov     dx,OFFSET Chaine
        mov     ah,09h
        int     21h
        mov     ah,4ch
        int     21h
Code    ends
        END     start
```

Une fois le source sauvegardé, sous le nom HELLO2.ASM par exemple, nous pouvons l'assembler de la même façon que le programme précédent :

```
masm    hello2
link    hello2
```

Ne vous inquietez pas du message Warning : no Stack qui aparaitra, cela est tout à fait normal puisque nous n'en avons pas spécifié. Même sous cette forme, le programme fonctionnera sans problème.

Nous pouvons maintenant transformer notre programme EXE en COM. Pour cela, nous utiliserons un utilitaire nommé EXE2BIN, qui est livré en standard avec MS-DOS (vous devriez le trouver dans votre directory \DOS). Il peut être exécuté de la façon suivante :

```
EXE2BIN HELLO2.EXE HELLO2.COM
```

Ceci créera le fichier HELLO2.COM. Vous pourrez donc effacer le fichier HELLO2.EXE qui ne sert plus à rien. Notez à ce sujet que quand il existe dans un répertoire trois programmes ayant le même nom, mais avec des extensions différentes BAT, EXE ou COM, ce sera le programme COM qui sera exécuté. Prenez donc soin de l'effacer à chaque fois si vous prenez l'habitude de tester le programme EXE avant de le convertir en COM.

Nous obtenons en définitive un programme ayant une taille de 33 octets, contre 543 précédement. Il est à noter que le même programme écrit en C, puis converti en COM aurait fait plus de 5500 octets ! Ceci est dû au fait que ce type de programme est chargé, au cours de l'édition de liens, avec des bibliothèques de fonctions standard qui ne servent à rien ici.

Autres Notions...

Cette partie traite de certains éléments non traités précédement, mais qui peuvent être importants pour la rédaction des programmes.

Analyse des paramètres

MS-DOS ne contient pas, comme sur le HP-71, de fonction permettant le contrôle et l'analyse des paramètres passés par la ligne de commande. Il faudra donc tout faire soi-même.

Une copie de la ligne de commande est disponible dans la *PSP*, qui est une zone placée en mémoire juste avant le début du programme.

Il est possible de connaitre l'adresse de cette zone en recourant à la fonction 51h du DOS :

```
mov     ah,51h
int     21h
```

En retour, nous avons dans le registre BX le segment pointant sur cette zone (l'offset étant toujours égal à zéro).

Il est aussi possible de connaitre cette valeur plus simplement, en lisant le contenu du registre ES, qui est initialisé au début de l'exécution. Mais cela doit être fait dans les premiers pas de notre programme, car nous (ou une fonction DOS) sommes suceptibles d'utiliser ce registre.

Les champs qui nous intéressent dans cette zone sont le 128eme octet, qui contient le nombre de caractères de la ligne de commande, puis les 127 octets suivants qui contiennent le texte de cette ligne, y compris l'espace situé entre le nom du programe et le premier paramètre. La chaine est terminée par un caractère CR (code 13), non compté dans le nombre de caractères.

La méthode la plus simple pour accéder à cette zone est, par exemple, d'écrire une routine de ce type (en supposant que ES contienne déjà l'adresse de la PSP) :

```
mov     di,128          ;adresse ligne comm.
cmp     di,0
jmp     no_arg          ;pas d'argument
inc     di              ;accès premier caract
  .
  .                     ;traitement à définir
  .
  .
```

L'allocation dynamique de la mémoire

Nous avons vu précédement comment réserver de la place dans le programme pour stocker des variables. Cependant ceci n'est interessant que dans le cas d'un bloc de données de taille relativement faible, car une telle zone est partie intégrante du programme. De plus, sa taille étant fixée une fois pour toutes, le programmeur devra généralement prévoir une valeur dépassant souvent l'utilisation réelle.

Pour remédier à cela, il est possible de demander à MS-DOS de donner au programme un ou plusieurs blocs de données inutilisés, qu'il pourra utiliser à sa convenance sans crainte d'écrasement par d'autres programmes ou gestionnaires d'interruption. Une fois qu'on a plus besoin de ces blocs de mémoire, nous pouvons la rendre à MS-DOS.

Nous pouvons assimiler le principe à l'utilisation des buffers sur le HP-71. La grande différence est que sur le HP-71 chaque buffer est identifié par un ID unique, permettant au programme "propriétaire" de toujours en retrouver l'adresse alors que, sous MS-DOS, nous devrons toujours conserver cette adresse dans une variable car nous n'aurons plus de moyen de la retrouver ultérieurement. De plus c'est cette adresse qui permettra au DOS d'identifier ce bloc mémoire au cours des opérations de modification de taille et de libération du bloc.

Ces opérations sont accessibles par des fonctions DOS. L'unité de travail est le paragraphe de 16 octets, ce qui veut dire que nous devrons diviser la taille en octets par 16. Si, par exemple, nous désirons utiliser une zone de données de 18000 octets, puis plus tard l'agrandir à 32000 octets et enfin l'effacer, nous utiliserons les instructions suivantes :

```
buf_seg DW   (?)    ;stockage adresse du buffer
          .
          .
          .
        mov    ah,48h           ;fct d'allocation
        mov    bx,(18000/16)+1  ;au moins 18000 oct
        int    21h              ;appel fonction
        jc     error            ;si plus de place
        mov    [buff_seg],ax    ;sauvegarde
                                ;du segment de
                                ;départ du bloc
                                ;(offset = 0)
          .
          .
          .
        mov    es,[buff_seg]
        mov    di,45            ;accès au 45e
        mov    ax,[di]          ; octet du buffer
          .
```

```
        .
        .
    mov     ah,4Ah              ;fct modification
                                ; de taille
    mov     es,[buff_seg]       ;placer adresse
                                ; du buffer en ES
                                ; (ceci permet au
                                ; DOS de savoir de
                                ; quel buffer il
                                ; s'agit)
     mov    bx,(32000/16)+1     ;au moins 32000 oct
     int    21h                 ;appel fonction
     jc     error2              ; si Carry = 1
                                ; ->plus de place
                                ; ou segment non
                                ; alloué
        .
        .
        .
     mov    ah,49h              ;fct libération
                                ; du bloc
     mov    es,[buff_seg]       ;placer adresse
                                ; du buffer en ES
     int    21h                 ; appel fonction
     jc     error2              ;si Carry = 1
                                ; -> segment non
                                ; alloué
```

Il est à noter que nous devrons utiliser la fonction de modification de taille de bloc dans un cas précis. En effet, les programmes COM ont la particularité de se réserver la totalité de la mémoire, même si ils n'utilisent effectivement que 64 ko. Si désirons allouer un bloc de mémoire ou lancer un programme "fils", nous devrons réajuster la taille du bloc utilisé par le programme de façon à ce qu'il coincide à sa taille réelle. Ceci peut être fait de la façon suivante :

```
code    segment para 'CODE'
          .
          .
        assume cs:code
        .
        .
start:  mov     ah,4Ah
        mov     es,cs           ;es = segment de
                                ; départ du programme
        mov     bx,offset fin ;bx = taille prgm
                                ; en octets
        mov     cl,4            ;conversion taille
        shl     bx,cl           ; en paragraphes
        inc     bx
        int     21              ;on réajuste la
                                ; taille du bloc
        mov     sp,offset fin
          .
          .
          .
        mov     ah,4Ch          ;retour au dos
        int     21h

        DB      200h            ;on se crée un espace
                                ;espace pour la pile
fin     equ     this byte
code    ends
        end     start
```

L'expression `equ this byte`, indique que la valeur de la constante `fin` est égale à l'adresse pointée par celle-ci, telle qu'elle sera calculée lors de l'édition de liens. Nous nous servons de cette valeur pour obtenir la taille du programme et y positionner la nouvelle fin du Bloc, grâce à la fonction de réallocation.

Toutefois, le fait de réduire la taille de ce bloc provoque aussi la libération de la zone occupée par la pile. Il faut donc recréer un espace afin de l'y loger. Cela est fait en créant une zone de données (de 512 octets dans notre exemple) et en repositionnant le pointeur de pile (le registre SP) à la fin de cette zone.

Notons enfin que la fonction d'allocation, utilisée d'une façon particulière, nous permet d'obtenir de façon relativement simple la mémoire disponible. Il nous suffit d'écrire les lignes suivantes :

```
mov     ah,48h
mov     bx,FFFFh
int     21h
mov     [mem_dispo],bx
```

Ceci nous permet d'obtenir, dans la variable `mem_dispo`, la mémoire disponible en paragraphes. En réalité, il ne s'agit pas réellement de la taille de la mémoire disponible, mais plutôt celle du plus grand bloc non utilisé. En effet, si MS-DOS a effectué beaucoup de créations et d'effacements de blocs de données, il est possible que la mémoire vive soit fragmentée en plusieurs morceaux, entre lesquels nous pouvons trouver des blocs de données utilisés par d'autres programmes. Mais ceci est en fait plus significatif, car ce que nous avons besoin de savoir la plupart du temps est justement la taille du plus grand bloc possible de mémoire.

Bibliographie

Avant de vous laisser, je pense qu'il est temps de vous indiquer quelques un des nombreux livres traitant du fonctionnement du PC, et l'assembleur. Je ne vous citerai donc que ceux dont je me sers habituellement.
Si vous vous sentez d'attaque à étudier le fonctionnement en détail de l'IBM PC, allez tout droit voir "La bible PC" aux éditions Micro

Application. Vous y trouverez en plus de nombreux exemples de programmes en assembleur.
Si vous ne désirez que les références aux différentes fonctions du DOS et du BIOS, il existe un livre au format de poche (donc à bas prix) aux éditions Marabout : Aide mémoire des interruptions MS-DOS.
En ce qui concerne l'assembleur pur, je n'ai pas de référence précise, si ce n'est les notices du Turbo Assembleur qui sont très bien faites.

Ceci termine pour aujourd'hui cet article. N'hesitez pas à m'écrire (adressez votre courrier au club) si vous désirez avoir des précisions suplémentaires sur ce qui a été traité aujourd'hui.

Jacques Belin (123)

LE COIN DES CODES
nouvelle présentation

La compilation de certains programmes, tels ceux écrits en assembleur, nécessitent souvent un logiciel que ne possèdent pas tous nos lecteurs. C'est pour cela qu'a été créé, il y a quelques années, le *Coin des Lex*, spécialement destiné au HP-71.

Depuis, d'autre machines (telles que la HP-48 ou le HP95) sont apparues, nécessitant souvent d'entrer certains programmes de façon équivalente.

Nous avons donc décidé de créer un nouveau **Coin des codes**, commun à toutes les machines et bénéficiant d'une présentation homogène. Celle-ci est pratiquement équivalente à ce qui se faisait pour le HP-71.

Nous avons aussi profité de cette occasion pour modifier la méthode de calcul du checksum, qui n'était pas suffisament fiable.

Notes importantes :

Même si la présentation des listings est identique, le traitement de ceux-ci est différente suivant le programme d'entrée et la machine de destination. Chaque listing est prévu pour être entré sur sa machine de destination.

Par exemple, ne tentez pas d'entrer un programme HP48 dans un fichier MS-DOS à l'aide du programme MAKEDOS. Vous obtiendrez un fichier que vous ne pourrOez pas transférer dans la HP48.

D'autre part, vous ne trouverez dans cette section que les programmes faisant moins de 1200 octets, soit environ une page. En effet, nous pensons qu'au delà de cette taille, personne n'aura le courage d'entrer un programme sous cette forme. A titre indicatif, un programme compilé en langage C possède une taille minimum de 5000 octets... Pour obtenir ces programmes, il vous faudra donc vous adresser directement au club.

Programmes HP48

Par rapport aux méthodes habituelles sur cette machine, notre méthode effectuant un calcul local du checksum en fin de chaque ligne, permet de faciliter la recherche d'erreurs, par rapport à une même recherche dans une chaîne de plusieurs centaines d'octets.

Ceux qui n'ont pas encore de programme assembleur pourront procéder de la manière suivante:

- par mesure de sécurité sauvegardez vos programmes et fichiers, éventuellement verrouillez vos cartes RAM pour devenir ROMs.
- tapez le programme ASSCOD.

ASSCOD

7B74h
879.5 octets

```
« RCLF HEX 64 STWS -2 SF 1 CF "" 'tmpcod' STO
  "nombre d'octets"  "" INPUT OBJ→ 2 * 16
  DUP2 / IP 3 ROLLD  MOD DUP2
  IF
  THEN 1 +
  END 3 ROLLD 1 + 4 ROLL
  # 0h → n r f s
    « 1 SWAP
      FOR i
        DO
          "ligne        "
          "00" i R→B →STR 3 OVER SIZE 1 -
          SUB + DUP SIZE DUP 2 - SWAP SUB + DUP
          "                    @ chaîne commençant
          chaine" +            @ par ← (newline)
          "                    @ chaîne commençant
          ___ ___ ___ ___" @ par ← (newline)
                               @ et composée de 4
                               @ groupes de 4
                               @ CHR 95 obtenus par
                               @ α shift bleu ×
                               @ 1 espace séparant
                               @ 2 groupes
          n i <
          IF
          THEN 1 r SUB
          END + 1 FC?C
          IF
          THEN « "" α »
          ELSE ROT
          END INPUT 0 OVER SIZE 1 SWAP
          FOR j OVER j DUP SUB NUM j * +
          NEXT s + DUP # FFFh AND "#"
          "                  @ "somme de controle"
          somme de controle  @ précédée et suivie
          ___"               @ de ← (newline)
                             @ puis 3 CHR 95
                             @ (α shift bleu ×)
          6 ROLL SWAP + « "" α »
          INPUT + OBJ→ ==
          IF
          THEN 1
          ELSE DROP 1000 .5 BEEP 1 SF 0
          END
        UNTIL
```

```
      END 's' STO 'tmpcod' DUP RCL
      ROT + SWAP STO
    NEXT f STOF
  »
tmpcod
WHILE DUP " " POS DUP
REPEAT DUP2 1 SWAP 1 - SUB 3 ROLLD 1 + MAXR
   SUB +
END DROP
"GROB 8 "                  @ si vous avez ASC→
  OVER SIZE 2 / " " +      @ JPC 79 page 14
  + SWAP + STR→            @ vous pouvez remplacer
  # 4017h SYSEVAL          @ placer ces lignes
  # 56B6h SYSEVAL          @ par ASC→
  DROP NEWOB               @ (plus rapide)
»
```

Donc à partir de maintenant pour tout assemblage de chaîne de codes procédez de manière suivante :

1- lancez le programme LISTCOD.
2- donnez le nombre de octets (1/2 oct. compris).

3- tapez chaque ligne de codes avec un espace entre chaque groupe de 4 caractères.
4- tapez la somme de contrôle. S'il y a erreur la ligne sera réaffichée pour correction après émission d'un BEEP
5- stockez le programme assemblé dans la variable donnée en tête.
6- si tout s'est bien déroulé vous pouvez purger tmpcod qui contient la chaîne de codes.

Programmes HP-71

MAKELEX2 est la nouvelle version de MAKELEX, qui nous a servi auparavant. Bien que totalement réécrit, afin de simplifier l'implantation de la nouvelle présentation des lignes par groupes de quatre quartets et le calcul du checksum, le principe du programme reste le même.

L'utilisation, identique à l'ancienne version est la suivante :

1- Lancer le programme : RUN MAKELEX2
2- Entrer la taille du fichier.
3- Entrer les listes de codes puis le checksum. En cas d'erreur corriger la ligne que vous venez d'introduire.
4- Une fois que toutes les lignes sont entrées, éteindre le HP-71, puis le rallumer.

Notez bien que vous devrez conserver l'ancienne version (MAKELEX) pour entrer les programmes parus dans les anciens JPC.

Programmes MS-DOS

Afin d'être utilisé par tous, nous avons décidé d'écrire un programme pouvant être exécuté avec le GWBASIC. Il devrait cependant être facile de le convertir pour un autre programme (QBASIC, Turbo BASIC...).

L'utilisation est sensiblement la même que pour MAKELEX2 :

1- Lancer le programme : GWBASIC MAKEDOS.BAS
2- Entrer le nom du fichier destination.
3- Entrer la taille du fichier.
4- Entrer les listes de codes puis le checksum (en prenant soin d'entrer les codes héxadécimaux en majuscules). En cas d'erreur corriger la ligne, en prenant soin de placer le curseur après le dernier caractère avant de taper sur la touche d'entrée.
5- Une fois que toutes les lignes sont entrées, sortir du GWBASIC en exécutant la commande SYSTEM. Le nouveau programme est immédiatement disponible.

Note : La taille du fichier résultant peut être supérieure d'un octet à ce qui est affiché dans le listing. Cela n'est pas un problème.

Ce mois ci...

Aujourd'hui, nous vous présentons dans ce premier *Coin des codes* les programmes HP-48 parus ce mois-ci, ainsi que les programmes HP-71 et MS-DOS (de moins de 1000 octets) parus dans les derniers JPC :

Programmes HP48

REV	Paru dans *JPC 80*
PIL	Paru dans *JPC 80*
POKE	Paru dans ce numéro
ADDR	Paru dans ce numéro
EDIV	Paru dans ce numéro
FFACT	Paru dans ce numéro

Lex HP-71

CERCLEX	Paru dans *JPC 78*

Programmes MS-DOS

TJR8	Paru dans *JPC 78*
FF	Paru dans *JPC 78*
TJPS	Paru dans *JPC 79*

Guy Toublanc (276)
Jacques Belin (123)

Programme MAKELEX2

```
 10 SFLAG -1 @ DIM E$[40],C$[40] @ LC OFF @ PURGE AH @ INPUT "Nb d'octets : ";N
 20 CREATE DATA AH,1,N-4 @ B=HTD(ADDR$("AH")) @ A=B+37 @ N=2*N+37 @ N1=N DIV 16
 30 L9=MOD(N,16)+(MOD(N,16) DIV 5) @ S=0 @ P$="---- ---- ---- ----"
 40 IF MOD(N,16)=0 THEN N1=N1-1 @ L9=LEN(P$)
 50 FOR X=0 TO N1
 60   IF X=N1 THEN P$=P$[1,L9]
 70   C$=P$
 80   M=S @ DISP DTH$(X)[3]; @ INPUT ": ",C$;C$ @ INPUT " sm = ","---";D$
 90   FOR Z=1 TO LEN(C$)
100     IF C$[Z,Z]#" " THEN M=M+(Z-(Z DIV 5))*NUM(C$[Z])
110   NEXT Z
120   IF D$#DTH$(MOD(M,4096))[3] THEN PRINT "Erreur de somme" @ BEEP @ GOTO 80
130   IF X=0 THEN E$=C$
140   IF X=1 THEN E$=E$&C$
150   IF X=2 THEN C$=C$[7]
160   IF X>1 THEN GOSUB 210
170   S=M
180 NEXT X
190 C$=E$ @ A=B @ GOSUB 210 @ CFLAG -1
200 END
210 FOR Z=1 TO LEN(C$)
220   IF C$[Z,Z]#" " THEN POKE DTH$(A),C$[Z,Z] @ A=A+1
230 NEXT Z @ RETURN
```

Programme MAKEDOS.BAS

```
 10 INPUT "Nom du fichier : ",NOM$ : INPUT "Nombre d'octets : ",N : N=N*2
 20 OPEN NOM$ FOR OUTPUT AS #1 : CLOSE #1 : KILL NOM$
 30 OPEN "bin.tmp" FOR OUTPUT AS #1 : S=0 : P$="---- ---- ---- ----"
 40 NLINES=N\16 : LENLAST=(N MOD 16)+((N MOD 16)\5)
 50 IF (N MOD 16)=0 THEN NLINES=NLINES-1 : LENLAST=LEN(P$)
 60 FOR X=0 TO NLINES
 70   IF X=NLINES THEN P$=LEFT$(P$, LENLAST)
 80   C$=P$
 90   X2$="00"+HEX$(X) : PRINT RIGHT$(X2$,3);":";
100   Y=CSRLIN : LOCATE Y,6 : PRINT C$; : LOCATE Y,6 : INPUT "",C$ : Y=Y-1
110   LOCATE Y,27 : PRINT " sm = ---" : LOCATE Y,33 : INPUT "",D$
120   M=S
130   FOR Z=1 TO LEN(C$)
140     IF MID$(C$,Z,1)<>" " THEN M=(M+((Z-(Z\5))*ASC(MID$(C$,Z,1)))) MOD 4096
150   NEXT Z
160   D2$="00"+HEX$(M) : D2$=RIGHT$(D2$,3)
170   IF D2$<>D$ THEN PRINT "Erreur de somme" : BEEP : GOTO 90
180   FOR Z=1 TO LEN(C$) STEP 2
190     IF MID$(C$,Z,1)=" " THEN Z=Z-1 : GOTO 230
200     CH=ASC(MID$(C$,Z,1))-48 : IF CH>9 THEN CH=CH-7
210     CL=ASC(MID$(C$,Z+1,1))-48 : IF CL>9 THEN CL=CL-7
220     PRINT#1,CHR$((16*CH)+CL);
230   NEXT Z
240   S=M
250 NEXT X
260 CLOSE #1 : NAME "bin.tmp" AS NOM$ : END
```

```
REV                          (HP48)
# EC5Ch              42.5 octets

     0123 4567 89AB CDEF    sm

000: D9D2 0ECE 81D0 040D    E3B
001: 9D20 FD55 0226 50FE    C70
002: D30E C370 2C23 0122    856
003: 7088 1303 3750 FE22    55E
004: 6433 70B9 F06B 2130    21F
005: B213 0                 514
```

```
PIL                          (HP48)
# 2DA2h              136 octets

     0123 4567 89AB CDEF    sm

000: D9D2 0FDE 8144 B46D    F0C
001: 9D20 B6CC 1E07 A29C    E3D
002: 2A25 F6A2 AEC8 1226    C57
003: 509F 1167 FE30 5AC2    AE1
004: 6CBD 30F6 E309 5D26    90A
005: C1C1 6E90 16CC D202    6F1
006: A000 8FD5 F301 0810    351
007: 18F1 4660 CC12 08FB    1F0
008: 9760 1471 3406 1691    D72
009: 7414 7135 179D 831A    ACD
00A: 0D75 3211 015F 115C    7A7
00B: 1171 161A 6C5E EAEB    87C
00C: 15C1 161C D5CD 111C    70D
00D: C461 15F1 15C1 1611    31B
00E: 71A6 C5EE 8F2D 7601    190
00F: 7407 1451 C48D 9415    EAA
010: 0442 30B2 130B 2130    A37
```

```
POKE                         (HP48)
# 2647h               75 octets

     0123 4567 89AB CDEF    sm

000: D9D2 0FDE 8111 9203    CCC
001: B000 D9D2 0322 3030    834
002: A50C CD20 F500 08F1    5C2
003: 4660 8FB9 7601 3014    1D3
004: 3131 174A D014 3818    E4D
005: F848 1CD8 1743 1935    B09
006: B115 B19E A808 1868    89B
007: 6158 0171 160C D54E    690
008: 8D34 1504 4230 B213    23A
009: 0B21 30                667
```

```
ADDR                         (HP48)
# B811h              35 octets

     0123 4567 89AB CDEF    sm

000: D9D2 05AA 81FE F309    F1F
001: 6C26 CCD2 0320 0014    AD9
002: 3E71 7414 7137 1741    67A
003: 4113 58D9 4150 CC95    469
004: 0B21 30                896
```

```
EDIV                         (HP48)
# EE66h              343 octets

     0123 4567 89AB CDEF    sm

000: D9D2 02BF 81D5 040D    E06
001: 9D20 8813 0D53 4590    A58
002: BA29 F345 3223 0CCD    8CA
003: 2090 0006 911B 2130    42A
004: 1204 0D9D 2013 236A    105
005: 9226 A217 02BC 8188    E7C
006: B26D 9D20 9612 6399    B66
007: 16D9 D200 09A2 9F34    906
008: 5322 30B2 130F 3126    51E
009: 3991 6D9D 20D5 3459    2AB
00A: F345 3223 0B21 302B    EE3
00B: C81B 2130 D9D2 0961    BE0
00C: 2639 916D 9D20 009A    95E
00D: 29F3 45CC D209 0000    5C3
00E: 6070 D534 5B21 30F3    2B9
00F: 1263 9916 D9D2 0CCD    25F
010: 2090 0006 940B 2130    DD2
011: 2BC8 1B21 30B2 1309    A18
012: FF30 D9D2 0009 A288    776
013: 1309 0BA2 9F34 5322    43B
014: 309F 345C CD20 3510    118
015: 08FC 1523 7731 1091    CD5
016: 317A 21AF 7111 978C    AC9
017: 1AF2 2030 2AF5 0D90    85E
018: 8BFA 7C97 C80E 465F    7C6
019: 0111 8408 8FF0 8502    4B3
01A: 0A80 8142 E80C FAC5    431
01B: 8140 D5AF 8101 0020    FEC
01C: 7760 B667 0607 6507    C07
01D: 2507 8407 A407 0407    7E4
01E: 2407 8307 E207 6307    409
01F: 620A C29F 3AD6 480A    2CE
020: 19A1 90CA 4D11 0810    F54
021: 100B 0657 1A19 42EA    D61
022: 194F DA19 4CD1 100C    B78
023: AC95 5081 0B1A 5CFA    AE7
024: 1AA4 E50F 860E 1B90    914
025: 80CF 2081 A011 80DF    71B
026: B1A5 CFA1 A0D9 1C00    550
027: A9AA C007 1199 7650    1DD
028: AF01 59F2 08D3 4150    E74
029: 1431 3317 915F F01D    C71
02A: 5345 B213 0B21 30      1CE
```

```
FFACT                        (HP48)
# 5139h            378.5 octets

     0123 4567 89AB CDEF    sm

000: D9D2 0E16 3278 BF13    DD8
001: 3920 1000 0000 0000    778
002: 0410 CFCE 13CE 22AF    79C
003: E22D 9D20 14BB 1B0B    636
004: C1B2 1305 BF22 D9D2    4C2
005: 078B F1ED 2A2E EDA1    55D
006: DBAA 1EED A16C 9B1E    5DB
007: D2A2 50FA 192C F178    45B
008: BF16 C9B1 EEDA 176B    46F
009: A192 CF13 2BA1 6C9B    3B3
00A: 1EED A190 DA19 2CF1    2DA
00B: 3392 0100 0000 0000    C8C
00C: 0021 0EED A187 2B19    ACB
00D: C2A2 76BA 16C9 B176    92D
00E: BA1F 0CB1 B969 1DBB    931
00F: F1B9 691C CD20 BD10    7B0
010: 08FB 9760 8FE3 1608    55B
011: F2D7 6014 3130 169A    1D0
012: F014 2102 1741 4313    CBD
013: 0169 142D 6BF2 BF2B    C90
014: F2BF 2BF2 108A F234    AAC
015: 5000 0C2C 68F8 DA60    96D
016: 5318 F2D7 6014 2164    58C
017: 808C 1321 018F 2D76    30A
018: 0111 1411 3134 C2A2    F67
019: 0145 1743 4500 00DA    BFE
01A: 118B F6BF 6BF6 BF6B    D5D
01B: F6C6 C214 5E21 748F    B6E
01C: C576 0119 1351 7930    6FD
01D: 115D 0AF2 10B1 1913    394
01E: 4169 118D 7AF0 1421    081
01F: 1A05 8FB9 BD01 13A7    EC6
020: 2041 44BF 6BF6 BF6B    F6E
021: F6BF 610B 164C F5BC    F49
022: 1138 A8D0 1401 10E4    BCB
023: 1001 1ACE 10A8 AE79    B08
024: 118B F6BF 6BF6 BF6B    C67
025: F6D5 D711 1130 169C    942
026: A131 1798 FC07 6011    5D5
027: 1130 169C 0131 179C    2BE
028: F310 315F 014C 1611    EDF
029: 70CF 5CE1 81AF 3D9D    F38
02A: 781F CF11 1131 1791    B62
02B: 4B14 E148 14D1 7118    7F9
02C: 1CF5 BE8F 2D76 0142    58F
02D: 1648 08CD BBF1 8DBF    6DF
02E: 1B21 305D F229 3632    384
02F: B213 0                 679
```

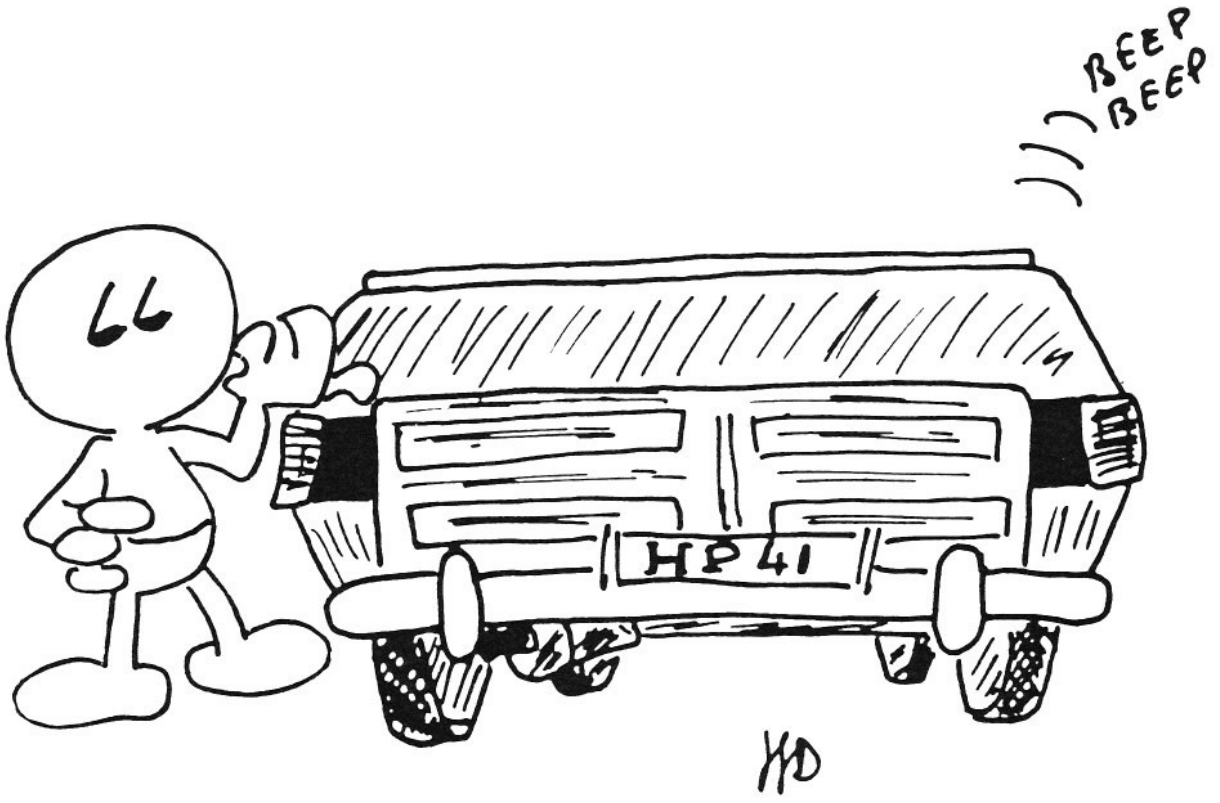
```
CERCLEX                (HP-71)
                     365 octets

     0123 4567 89AB CDEF   sm

000: 3454 2534 C454 8502   C0F
001: 802E 0000 0000 0000   5F1
002: ED20 0868 1910 0000   0BC
003: F020 0000 0000 0000   A58
004: 08D0 00DF 006A 000D   7AE
005: B349 4253 4C45 4817   43E
006: 2594 E474 911F F8FB   42D
007: C631 8424 7085 2F82   100
008: 4AF2 D617 F90C B086   F7D
009: 200F A038 D91F B08D   EC4
00A: 0745 0305 6600 304A   A91
00B: 0E10 A8FD 9630 7810   785
00C: 11A9 0A21 8FDC 6305   587
00D: F011 A49D 8D27 1308   2A1
00E: DC5E 208B FFFA BFFF   5B8
00F: 8F87 1F08 4384 17C6   31F
010: F470 FA85 1066 A10F   0E4
011: 8FFF 89FF F8F8 71F0   0F5
012: 8418 5373 4F47 0851   D67
013: FA10 A863 4006 DA07   AB7
014: 8BE7 010A DEEE 8FD3   BAD
015: DE0A D7D6 067F 0F10   9CD
016: 8780 F10B 8FBC 6311   7E4
017: 3018 4146 164D 7110   3C9
018: E2CE 108D 2109 112C   0BD
019: 4303 E206 1196 0407   C5E
01A: 9800 7DA2 430F B0E1   A98
01B: 195E 012A CED5 12AE   A37
01C: 9C6C 6CA2 0346 0000   5ED
01D: C206 119E 6109 112E   2C3
01E: 24DB 1198 A660 7C30   004
01F: 0707 CE06 10AD BD3A   FAE
020: 5FD7 4606 F7F0 78D8   E94
021: 4A80 119F A109 6D10   BD0
022: 119F A109 6420 8407   7ED
023: 0404 0E79 2011 AFA1   5C5
024: 0A7D 2049 D761 0119   22F
025: 12AF A109 8700 0850   E2B
026: 5EC1 29FA 1296 8200   AC5
027: 7031 1211 8E28 BF00   827
028: F2DA C4C4 C2C6 1321   59A
029: 30C2 1350 6111 11BC   248
02A: A340 8200 8BE4 C303   F83
02B: A850 E04B 0D81 C81C   E1A
02C: 137C 2135 3015 50A0   9F1
02D: 6A0D 59F1 5B08 71E0   7BF
02E: 0E0E 1590 0701 B8C0   4E8
02F: E0EB 8C15 1007 03F    E9E
```

```
TJR8.COM               (MS-DOS)
                     526 octets

     0123 4567 89AB CDEF   sm

000: E970 0100 0000 0000   9C2
001: B4CF C5C0 CCC8 D4B5   A23
002: C1CD C9DD D1D9 D8D0   AA8
003: DCD7 D3C2 CECA C3CB   C3D
004: EFDA DBD6 BBD2 3FBE   E0D
005: C4D5 C6C7 B7B6 F9FA   F24
006: B93F 3FF8 F7B8 FBFD   16C
007: 3F7F 3F7C 7CE0 A2A1   0D0
008: 3F3F 3F3F 3FBF BC3F   27E
009: 3F3F 3F7C F62B E2E1   252
00A: 3F3F 3F3F 3F3D 3FBA   33B
00B: E4E3 A4A5 A33F E5A6   2C5
00C: A73F 3F3F 3F7C E63F   337
00D: E7DE DFE8 EAE9 F3F1   481
00E: F0ED AEAD B2B1 B0A8   481
00F: 2DFE 5FF5 F4BD 3F3F   5AC
010: B3AB F23F 3F3F FCA0   645
011: EB02 544A 80FC 0574   3E8
012: 5A80 FC40 750D 83FB   300
013: 0475 08C6 0607 0100   E34
014: EB0F 9080 FC44 755F   CD6
015: 3C03 755B 83FB 0475   AA0
016: 5683 F900 7451 9C50   76D
017: 5351 5206 5755 560E   42B
018: 078B EA8B F132 FF3E   488
019: 8A5A FF80 FB7F 760C   45D
01A: 80EB 8026 8A87 0801   0F0
01B: 3E88 42FF 4E75 E85E   0DA
01C: 5D5F 075A 595B 589D   FD1
01D: EB1D 9080 FA7F 7617   E35
01E: 9C50 5356 1E0E 1F8A   D11
01F: DA80 EB80 32FF 8A97   C62
020: 0801 1F5E 5B58 9D2E   BBD
021: FF2E 0301 0D0A 544A   94C
022: 5238 2065 7374 2069   538
023: 6E73 7461 6C6C 822E   382
024: 0D0A 240D 0A54 4A52   0CE
025: 3820 6573 7420 6482   C87
026: 6A85 2069 6E73 7461   94D
027: 6C6C 822E 0D0A 2450   672
028: 6F75 7220 6C65 2064   2B0
029: 8273 6163 7469 7665   F31
02A: 7220 3A20 2054 4A52   B4B
02B: 3820 550D 0A24 544A   870
02C: 5238 2065 7374 2064   40C
02D: 8273 6163 7469 7682   07B
02E: 0D0A 24BE 8000 803C   DBC
02F: 0074 4346 4680 3C55   A57
030: 753C B821 35CD 2126   791
031: 817F 0254 4A75 2326   3F0
032: 8B16 0301 26A1 0501   F3C
033: 8ED8 B821 25CD 218C   DBF
034: C326 8E06 2C00 B449   B0A
035: CD21 8EC3 B449 CD21   9B1
036: 0E1F BA5E 02B4 09CD   926
037: 21B8 004C CD21 B821   6D1
038: 35CD 2126 817F 0254   383
039: 4A74 30B8 2135 CD21   0D1
03A: 8C06 0501 891E 0301   CA0
03B: BA88 01B8 2125 CD21   9FB
03C: BA0C 02B4 09CD 21BA   906
03D: 3F02 B409 CD21 BA0C   814
03E: 02B1 04D3 EA42 B800   593
03F: 31CD 21BA 2302 B409   2B1
040: CD21 BA3F 02B4 09CD   1E0
041: 21B8 004C CD21        323
```

```
FF.COM                 (MS-DOS)
                      10 octets

     0123 4567 89AB CDEF   sm

000: B20C B405 CD21 B44C   E9F
001: CD21                  0C4
```

```
TJPS.COM               (MS-DOS)
                     636 octets

     0123 4567 89AB CDEF   sm

000: E9E3 0100 0000 0000   9F8
001: 0D0A 1B26 6B30 531B   738
002: 266B 3153 1B26 6B32   422
003: 5300 0000 0000 0000   DAD
004: 0000 0000 0000 0000   72D
005: 0000 0000 0000 0000   0AD
006: 0000 0000 0000 0000   A2D
007: 0000 0000 0000 0000   3AD
008: 0000 0000 0000 0000   D2D
009: 0000 0000 0000 0000   6AD
00A: 0000 0000 0000 0000   02D
00B: 0000 0000 0000 0000   9AD
00C: 0000 0000 0000 0000   32D
00D: 0000 0000 0000 0000   CAD
00E: 0000 0000 0000 0000   62D
00F: 0000 0000 0000 0000   FAD
010: 0000 0000 0000 0000   92D
011: 0000 0000 0000 0000   2AD
012: 0000 0000 0000 0000   C2D
013: 0000 0000 0000 0000   5AD
014: 0000 0000 0000 0000   F2D
015: 0000 0000 0000 00EB   B08
016: 0250 539C 5053 5152   6DF
017: 551E 5606 57B8 5000   30C
018: 8ED8 33F6 803C 0175   019
019: 03E9 A800 C604 01FB   E16
01A: B840 008E D8BE 6300   BE2
01B: 8B1C 81FB B403 7506   924
01C: B800 B0EB 0490 B800   628
```

```
01D: B88E C0BE 4E00 8B3C  572
01E: BE4A 008B 1CBE 8400  36F
01F: 8A3C FEC7 8BEB 8CC8  4E1
020: 8ED8 80FB 2877 06BA  3B6
021: 0F01 EB0F 9080 FB50  1F9
022: 7706 BA0A 01EB 0490  F76
023: BA14 01B4 40BB 0400  BDB
024: B905 00CD 218B C5BE  B9F
025: 1901 268A 1580 FA20  8E1
026: 7302 B22E 8814 4647  599
027: 47FE C875 EDC6 040D  4BF
028: 46C6 040A B440 BB04  25C
029: 008B CD32 ED80 C102  03A
02A: BA19 01CD 218B CDFE  135
02B: CD8B E975 C8BA 0801  F55
02C: B440 BB04 00B9 0700  B75
02D: CD21 B850 008E D833  90C
02E: F6C6 0400 5F07 5E1F  74A
02F: 5D5A 595B 589D CF0D  787
030: 0A54 4A50 5320 6573  373
031: 7420 696E 7374 616C  0EE
032: 6C82 2E0D 0A24 0D0A  EFA
033: 544A 5053 2065 7374  AE9
034: 2064 826A 8520 696E  879
035: 7374 616C 6C82 2E0D  6F5
036: 0A24 506F 7572 206C  41A
037: 6520 6482 7361 6374  FFA
038: 6976 6572 203A 2020  B73
039: 544A 5053 2055 0D0A  851
03A: 2454 4A50 5320 6573  425
03B: 7420 6482 7361 6374  004
03C: 6976 820D 0A24 BE80  E15
03D: 0080 3C00 7443 4646  A09
03E: 803C 5575 3CB8 0535  738
03F: CD21 2681 7F02 5053  35A
040: 7523 268B 1603 0126  EE8
041: A105 018E D8B8 0525  C53
042: CD21 8CC3 268E 062C  A93
043: 00B4 49CD 218E C3B4  9A7
044: 49CD 210E 1FBA D102  827
045: B409 CD21 B800 4CCD  75F
046: 21B8 0535 CD21 2681  41F
047: 7F02 5053 742B 8C06  155
048: 0501 891E 0301 BAAF  051
049: 01B8 0525 CD21 BA7F  FB3
04A: 02B4 09CD 21BA B202  DAC
04B: B409 CD21 BA7F 02B1  C32
04C: 04D3 EA42 B800 31CD  A81
04D: 21BA 9602 B409 CD21  853
04E: BAB2 02B4 09CD 21B8  6C0
04F: 004C CD21            EC5
```



Le Journal JPC est le bulletin de liaison entre les membres de l'Association "PPC Paris", régie par la loi de 1901 et indépendante de tout constructeur ou société commerciale. Le Club est éditeur de JPC, et son siège social est au 56, rue Jean-Jacques Rousseau, 75001 Paris.

La maquette de ce numéro a été préparée et réalisée par Jacques Belin et Asdin Aoufi.

Les dessins sont de Jean-Jacques Dhénin et Paul Courbis.

Les informations et programmes parus dans ce journal sont publiées "Tels quels" et ne peuvent en aucun cas engager la responsabilité de Hewlett-Packard ou de PPC Paris. Hewlett-Packard se réserve le droit de ne pas répondre aux questions concernant le sujet de certains articles.

Les programmes publiés peuvent être utilisés librement. Cependant, ils ne peuvent être vendus ou fournis dans un ensemble commercialisé, sous quelque forme que ce soit, sans l'accord écrit de l'auteur ou de PPC Paris.

Directeur de la publication : Jacques Belin
Numéro ISSN : 0762 - 381X

Veuillez adresser toute correspondance à :
PPC Paris, BP 604, 75028 Paris Cedex 01.