LE TOUR DU HP-28 EN 80 PAGES ... OU PRESQUE

L'article ci-dessous est le résultat de plusieurs mois de travail de Paul Courbis sur la structure interne du HP-28C. Paul a fait l'effort de rédiger et de présenter dans JPC *l'état actuel de ses connaissances. Cela semblera peut-être un peu difficile d'abord. Mais pour expérimenter vous-mêmes, Paul a également fourni PEEK et POKE.*

La publication de ce travail et sa disponibilité constituent un enrichissement pour tous.

Félicitons tous Paul pour ce travail considérable, et souhaitons que beaucoup parmi vous continue dans la voie ainsi tracée.

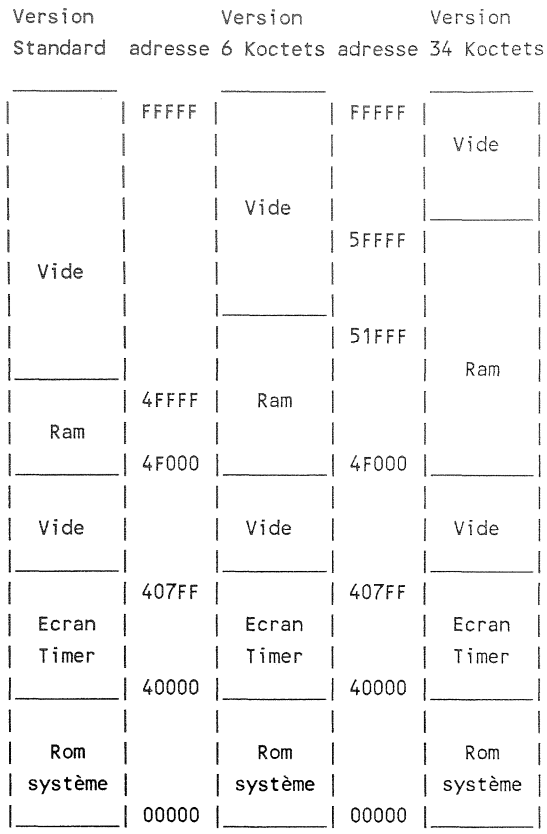
Le but de cet article est d'expliquer, ou plutôt de tenter d'expliquer, la manière de programmer en assembleur le HP-28C (c'est à dire en langage machine). Pour cela, une bonne connaissance de la structure interne de la machine est nécessaire et la première partie de cet article est consacrée à cela.

Tout d'abord, le microprocesseur du HP-28C est quasiment le même que celui du HP-71, ce qui permet de connaître les codes des différentes instructions du processeur. Il faut cependant savoir qu'une instruction nouvelle vient compléter celles présentes sur le HP-71 :

`PC=(A)` de code 808C. Cette instruction lit un groupe de 5 quartets situé à l'adresse contenue dans A et charge ces 5 quartets (c'est à dire une adresse) dans PC (Program Counter).

L'instruction RPL qui a permis l'accès à la programmation en assembleur est bien entendu `SYSEVAL` dont le rôle est exactement le même que `PC=(A)` : `SYSEVAL` prend l'entier binaire au niveau 1 de la pile, le tronque pour n'en garder que 5 quartets ; lit 5 quartets à cette adresse et les place dans le registre PC.

La structure de la mémoire est la suivante :



Le contenu de la mémoire est constitué de 4 types de groupes de quartets :

- ceux concernant l'écran ou le timer qui sont des zones mémoire d'entrée / sortie, quartets d'un type particulier,
- des données : suites de quartets représentant quelque chose. Par exemple, la valeur de correction de la routine d'horloge,
- des routines en langage machine,
- des objets RPL (comme ceux que l'utilisateur crée).

Il est à noter que ce genre de groupe de quartets se rencontre aussi dans la Rom ; en particulier, les instructions RPL sont des objets au même titre que ceux créés par l'utilisateur.

LES OBJETS

Pour bien comprendre le fonctionnement du HP-28C, il est nécessaire de bien connaître les différents types d'objets. Ils sont au nombre de 19 dont 9 peuvent être créés directement par l'utilisateur.

Ils commencent tous par un groupe de 5 quartets, le *prologue* qui indique leur type et, éventuellement, par des renseignements sur leur longueur, dimension...

Le tableau suivant résume les différents types d'objet ainsi que leur prologue. Ces objets sont ensuite détaillés.

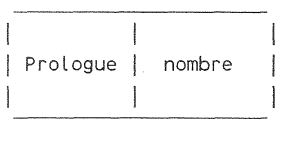
Objet	Prologue
Short integer (*)	02911
Real	02933
Extended Real (*)	02955
Complex	02977
Extended Complex (*)	0299D
Byte (*)	029BF
Premier objet inconnu (*)	029E1
Array	02A0A
Second objet inconnu (*)	02A2C
String	02A4E
Binary integer	02A70
List	02A96
Ram/Rom Pair (*)	02AB8
Algebraic	02ADA
Program	02C67
Assembly Code (*)	02C96
Global Name	02D12
Local Name (*)	02D37
Rom Pointer (*)	02D5C

(*) Ces objets sont impossibles à créer par l'utilisateur et apparaissent tous comme `System Object` (sauf Local Name).

Short integer

Prologue : 02911_{16}

Structure :

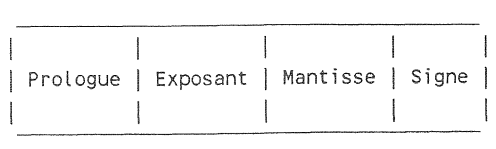


Exemple : 1192054321 est le short integer 12345.

Real

Prologue : 02933_{16}

Structure :

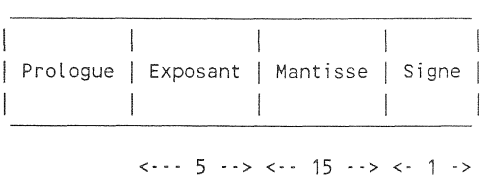


L'exposant et la mantisse sont en décimal codé binaire (DCB). L'exposant varie de 000 à 499 (positif) et de 999 à 501 (négatif). La mantisse contient 12 chiffres. Le signe est le signe de la mantisse : 0 si positif, 9 si négatif. Par exemple, $\pi \times 10^5$ est représenté, en valeur numérique, par : 339205009535629514130.

Extended Real

Prologue : 02955_{16}

Structure :

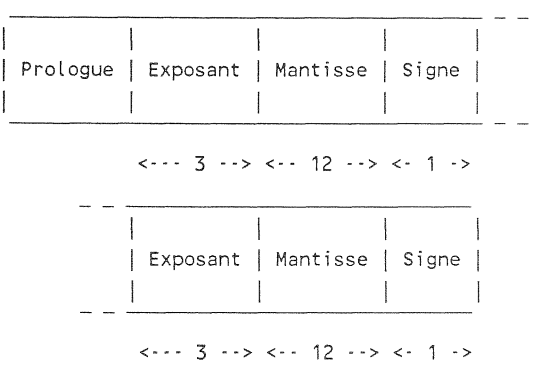


Même remarques que pour Real et exemple similaire avec 3 décimales de plus et un exposant compris entre -49999 et +49999.

Complex

Prologue : 02977_{16}

Structure :

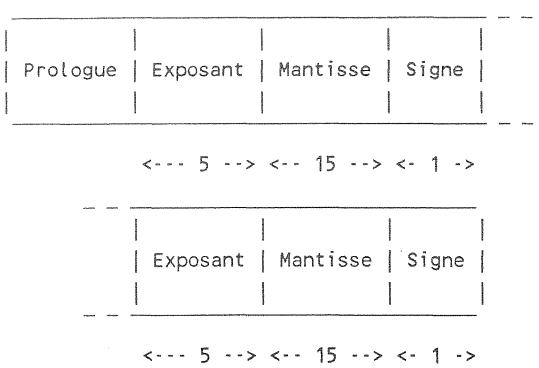


On trouve d'abord la partie réelle puis la partie imaginaire, sinon même structure que Real. Par exemple, `(10,20)` est représenté par : 77920100000000000001010000000000000020.

Extended Complex

Prologue : $0299D_{16}$

Structure :

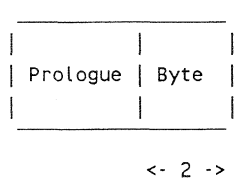


Même remarques que pour Real, exemple similaire à Complex avec 2 × 3 décimales en plus et des exposants à 5 chiffres de -49999 à +49999.

Byte

Prologue : $029BF_{16}$

Structure :



Par exemple, octet 01 : FB92010.

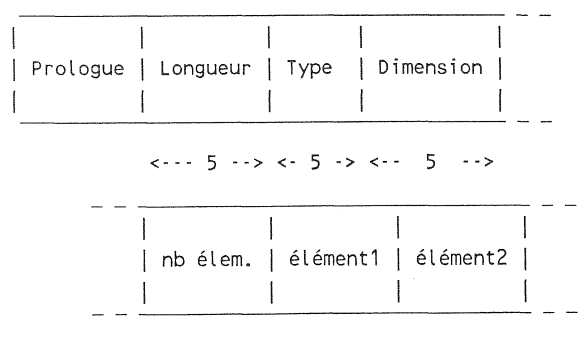
Premier objet inconnu

Prologue : $029E1_{16}$

Array

Prologue : $02A0A_{16}$

Structure d'un vecteur :



La *longueur* est le nombre de quartets à ajouter au pointeur RPL (D0) lorsqu'il pointe sur cette longueur pour qu'il pointe sur la fin de l'objet, c'est à dire la longueur + 5 (longueur de la longueur). Le *type* est soit réel soit complexe et est représenté par le prologue Real ou Complex. *Dimension* vaut 00001 dans le cas du vecteur (une seule dimension). *Nombre d'éléments* est le nombre de colonnes de la matrice 1 × n qu'est le vecteur.

La structure d'une matrice est quasiment la même. Cependant, *dimension* vaut 00002 et on a 2 × 5 quartets pour le nombre d'éléments (nombre de lignes / nombre de colonnes). Par exemple, `[[1 2] [3 4]]` est représenté par :

A0A20	prologue
95000	longueur
33920	type réel
20000	dimensions
20000	nombre de lignes
20000	nombre de colonnes
0000000000000010	premier réel
0000000000000020	deuxième réel
0000000000000030	troisième réel
0000000000000040	quatrième réel

Second objet inconnu

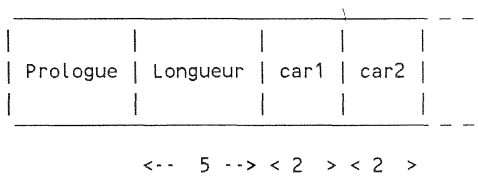
Prologue : $02A2C_{16}$

Cet objet, ainsi que le précédent inconnu, sont impossibles à charger par un `STO` ; ils provoquent un message `Range Exception`.

String

Prologue : $02A4E_{16}$

Structure :

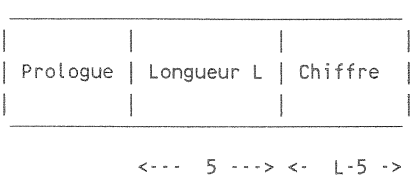


Longueur est définie comme pour Array et vaut également 2 × `SIZE(String)` + 5. Par exemple, `"PPC"` vaut : E4A20B0000050534.

Binary integer

Prologue : $02A70_{16}$

Structure :

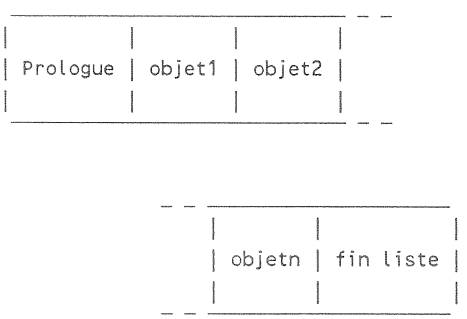


Habituellement L = 15_{16} pour les nombres entrés par l'utilisateur. Cependant, la valeur peut être différente ce qui permet un gain de place pour le stockage de petits entiers : #0 peut être codé 07A20600000. Par exemple, #123456789ABCDEF0 est représenté par : 07A20510000FEDCBA987654321.

List

Prologue : $02A96_{16}$

Structure :



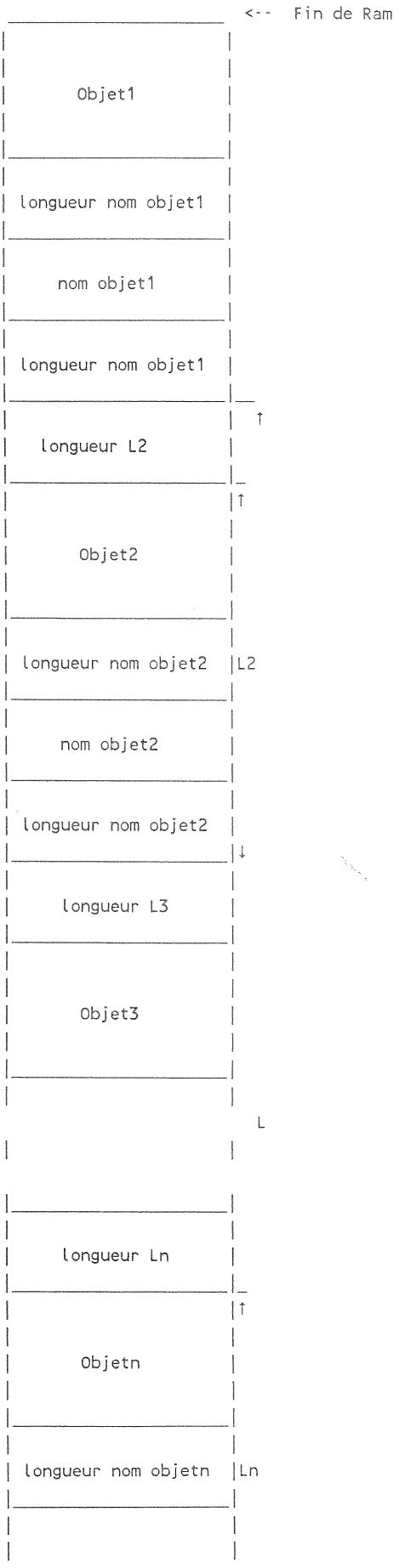
Fin de liste est le groupe de quartets 02F90. Par exemple, { #123456789ABCDEF0 "PPC" } est représenté par :

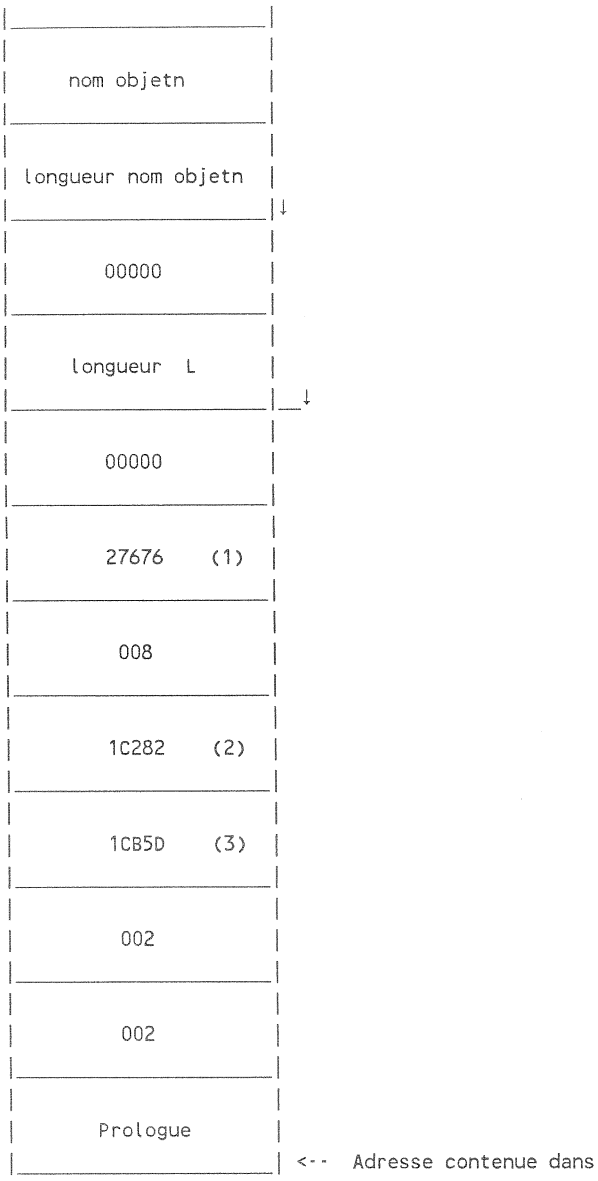
69A20	prologue List
07A20	prologue Binary integer
510000FEDCBA987654321	l'entier binaire
E4A20	prologue String
B0000050534	la chaîne
09F20	fin de liste

Ram/Rom Pair

Prologue : $02AB8_{16}$

Cet objet constitue en fait la Ram USER et sa structure est mal connue. Son explication est indissociable de celle de la structure de la partie de la Ram contenant les objets utilisateur. En voici la structure :





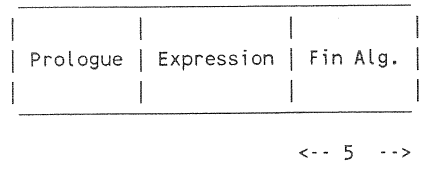
(1), (2) et (3) sont probablement des adresses concernant les instructions RPL situées en Rom.

Il est à noter qu'une telle structure permet le parcours de la Ram utilisateur dans les deux sens.

Algebraic

Prologue : $02ADA_{16}$

Structure :



Fin d'algèbre vaut 02F90. L'expression contient la suite des opérations à effectuer pour faire le calcul, en notation polonaise inversée. C'est un ensemble composite comme la liste. Par exemple, 1.33 × 7.42 + √74 sera représenté par :

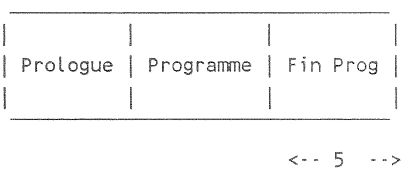
ADA20	prologue Algebraic
33920	prologue Real
0000000000003310	réel 1.33
33920	prologue Real
0000000000002470	réel 7.42
D9F81	+
33920	prologue Real
1000000000000470	réel 74
DD591	√
D9F81	+
09F20	fin d'Algebraic

Une telle structure permet de ne pas avoir à stocker de parenthèses : les calculs se font par l'intermédiaire de la pile.

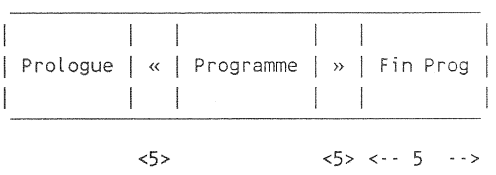
Program

Prologue : $02C67_{16}$

Structure :



Fin de programme vaut 02F90. Le programme est une suite d'objets : c'est donc un objet composite. Pour les programmes créés par l'utilisateur, la structure est plus complexe car elle est de la forme :



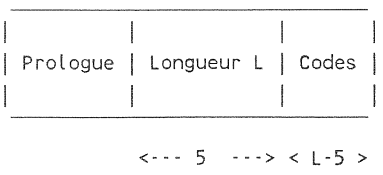
C'est encore un objet composite. Par exemple : « "PPC" #123456789ABCDEF0 » est représenté par :

76C20	prologue
A0F72	«
E4A20	prologue String
B0000050534	PPC
07A20	prologue Binary Integer
510000FEDCBA987654321	#123456789ABCDEF0
F1F72	»
09F20	fin de programme

Assembly Code

Prologue : $02C96_{16}$

Structure :

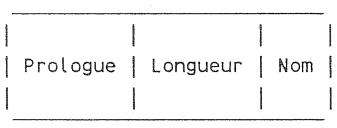


Codes représente la suite des codes des instructions du microprocesseur.

Global Name

Prologue : $02D12_{16}$

Structure :



Longueur est le nombre de caractères du nom. Par exemple, 'ABC' est représenté par : 21D2030142434.

Local Name

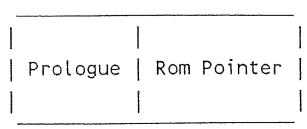
Prologue : $02D37_{16}$

Structure identique à un nom global, seul le prologue change.

Rom Pointer

Prologue : $02D5C_{16}$

Structure :



Rôle inconnu.

A ces 19 objets, il convient d'ajouter :
- les instructions RPL,
- les instructions internes du HP-28C, par exemple 1C285 qui vérifie la présence d'un élément dans la pile, le stocke dans LAST, si LAST est validé, ou affiche Too Few Arguments si la pile est vide. La liste de ces instructions ne peut encore être envisagée.
- les lettres de A à Z,
- les nombres de -9 à +9
- diverses constantes.

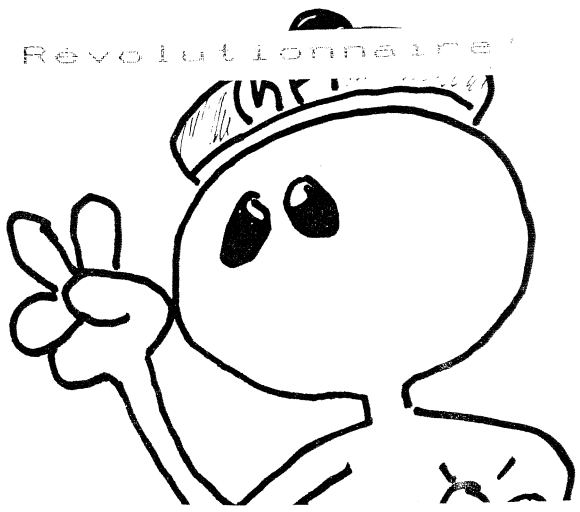
tous codés sur 5 quartets.

Une liste de certaines instructions RPL a été donnée dans JPC 47, en voici d'autres :

```
A : 2789F     U : 279CB     5 : 1E353
B : 278AE     V : 279DA     6 : 1E368
C : 278BD     W : 279E9     7 : 1E37D
D : 278CC     X : 279F8     8 : 1E392
E : 278DB     Y : 27A07     9 : 1E3A7
F : 278EA     Z : 27A16   π : 1E479 (approx)
G : 278F9    -9 : 1E464   π : 1E48E (étendu)
H : 27908    -8 : 1E44F  MAXR : 1E4A8
I : 27917    -7 : 1E43A -MAXR : 1E4BD
J : 27926    -6 : 1E425  MINR : 1E4D2
K : 27935    -5 : 1E410 -MINR : 1E4E7
L : 27944    -4 : 1E3FB     0 : 1E4FC (ext. réel)
M : 27953    -3 : 1E3E6     1 : 1E516 (ext. réel)
N : 27962    -2 : 1E3D1     2 : 1E530 (ext. réel)
O : 27971    -1 : 1E3BC     3 : 1E54A (ext. réel)
P : 27980     0 : 1E2EA     4 : 1E564 (ext. réel)
Q : 2798F     1 : 1E2FF     5 : 1E57E (ext. réel)
R : 2799E     2 : 1E314
S : 279AD     3 : 1E329
T : 279BC     4 : 1E33E
```

Ces objets sont tous considérés comme des instructions et sont en fait représentés par leur adresse d'exécution (utilisable par SYSEVAL).

Après ce petit (!) prologue, passons à présent au contenu de la mémoire vive ; la Rom (128 Koctets) étant encore loin d'être connue, il serait illusoire de vouloir la décrire.



ZONE D'ENTREE / SORTIE

407FF		
	Valeur du Timer	8 quartets
407F8		
	???	246 quartets
40701	Contraste	1 quartet
	???	33 quartets
406DF		
	Row Driver Waveforms	256 quartets
405E0		
405DF		
	Partie droite écran	480 quartets
40400		
	???	288 quartets
402DF		
	Partie gauche écran	616 quartets
40078		
	???	120 quartets
40000		

L'écran

Chaque pixel est représenté par un bit, soit donc 4 pixels par quartet. L'écran est codé colonne par colonne (les 8 premiers quartets représentent donc la première colonne), du haut vers le bas (le bit 0 du premier quartet représente donc le pixel en haut à gauche).

Row Driver Waveforms

C'est une série de 256 quartets indiquant au Display Driver le numéro de la ligne logique, située dans la Ram précédemment décrite, à afficher en une ligne physique donnée (sur le LCD proprement dit).

Appelons L_1 ... L_{32} les 32 lignes physiques et l_1 ... l_{32} les 32 lignes logiques. Chaque ligne physique est concernée par 32 bits (8 quartets) dans l'ordre L_1, L_{32}, L_2, L_{31}... Pour une ligne physique L_n donnée, on aura :
- si les 32 bits sont à 0, la ligne L_n sera éteinte.
- si plus d'un des 32 bits est à 1, la ligne L_n apparaîtra noire (plus ou moins foncée).
- si le $m^{ème}$ bit et lui seulement est à 1, la ligne l_m sera affichée en L_n.

L'état normal est donc (L_1 et L_{32} seulement) en bits :

10000000000000000000000000000000 L_1

00000000000000000000000000000001 L_{32}

soit donc, en hexadécimal : 10000000 00000008.

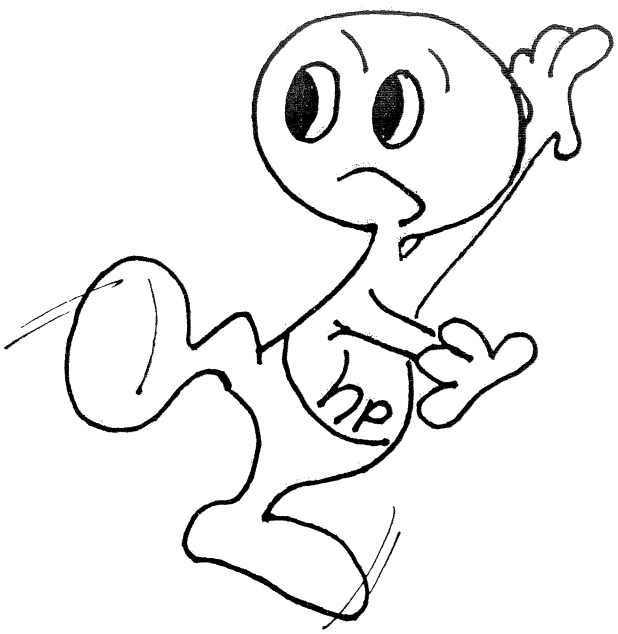
Il est à noter qu'un arrêt système après une modification de ces 256 octets remet le Row Driver Waveforms à la valeur standard.

Le contraste

C'est un quartet dont la valeur va de #0, contraste le plus faible, à #F, contraste le plus fort. Il est à noter que les touches [ON][+] et [ON][-] ne permettent d'accéder qu'aux valeurs #4 à #F.

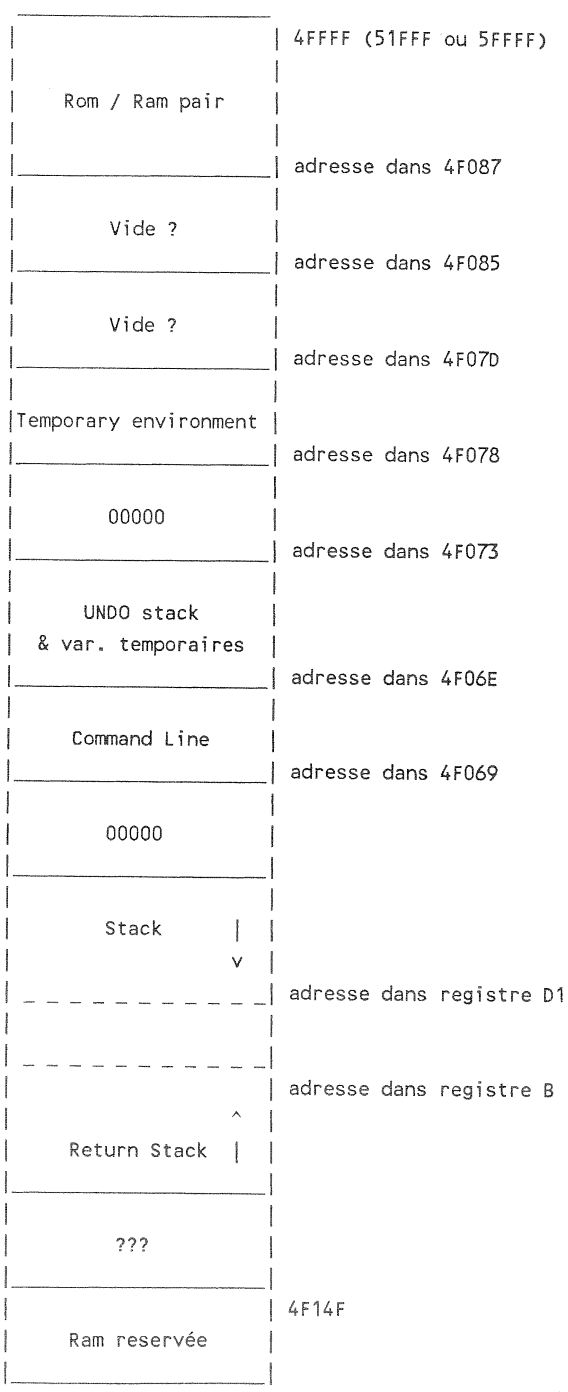
Le timer

C'est un groupe de 8 quartets dont la valeur va décroissant de #077FFF à #000000. La valeur exacte obtenue par #123E SYSEVAL est calculée en faisant la différence entre un offset de correction situé en Ram et la valeur du timer.



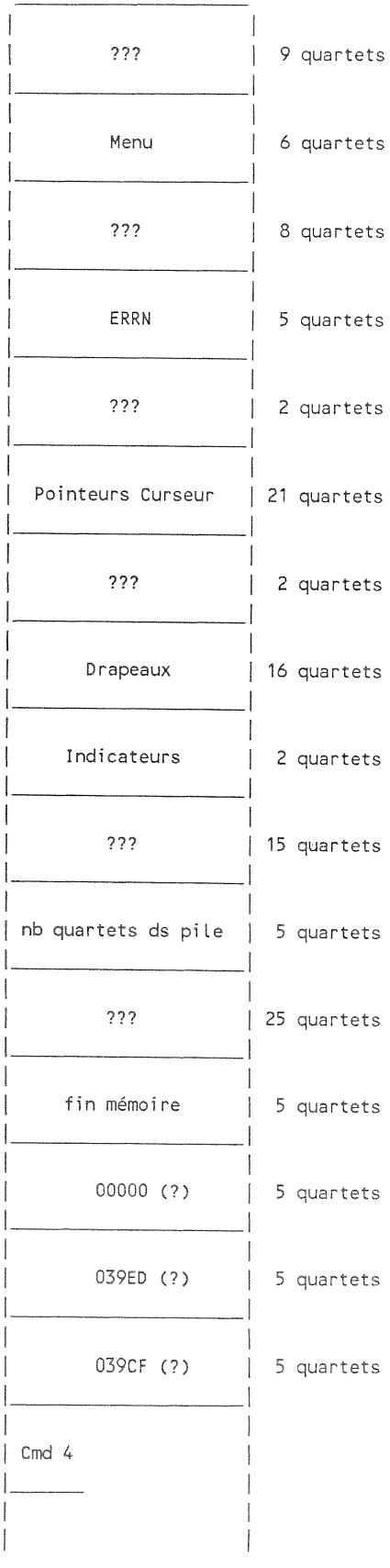
LA MEMOIRE

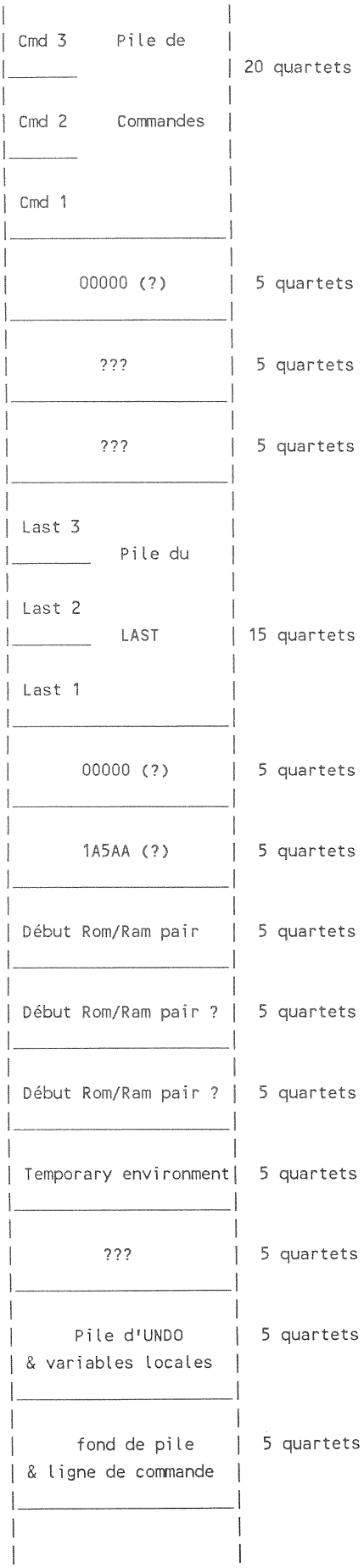
En version standard, elle va de 4F000 à 4FFFF, en version 6 K-octets de 4F000 à 51FFF et en version 34 K-octets de 4F000 à 5FFFF.

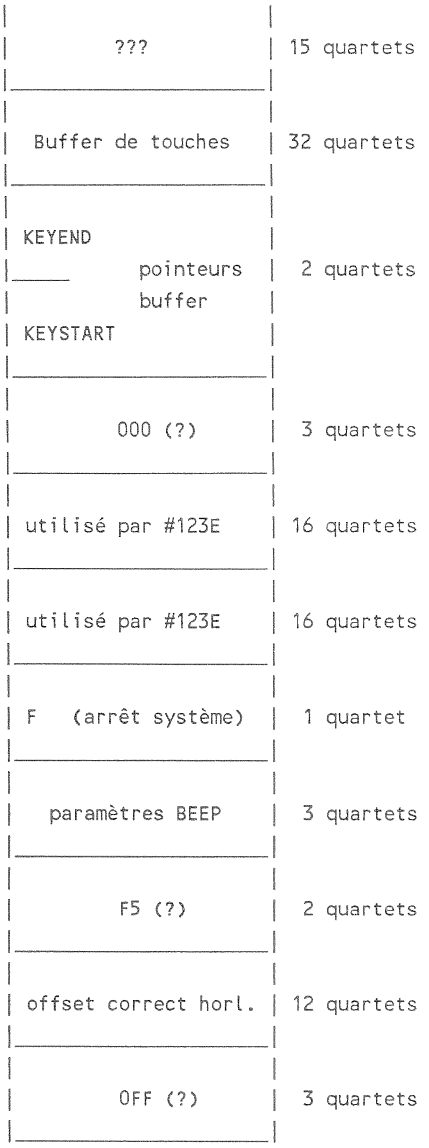


LA RAM RESERVEE

Les adresses sont spécifiées dans la description détaillée du contenu.







Offset de correction de l'horloge

(#4F003 à #4F00E) Cette série de 12 quartets sert au calcul du temps effectif lors de l'appel de #123E. On a *temps réel* = *offset* - *timer*, *timer* est lu dans la Ram d'entrée/sortie.

Paramètres de BEEP

(#4F011 à #4F013) Ces 3 quartets sont une référence pour l'émission d'un BEEP. En les modifiant, on fait varier tous les BEEP possibles à produire avec la machine.

Arrêt système et auto-tests

(#4F014) La valeur de ce quartet est normalement #F. En le mettant à 0, l'arrêt système [ON] [^] ainsi que les deux auto-tests n'ont plus aucun effet. A noter que l'extinction de la machine réarme cet indicateur et permet à nouveau l'arrêt système.

Stockage temporaire

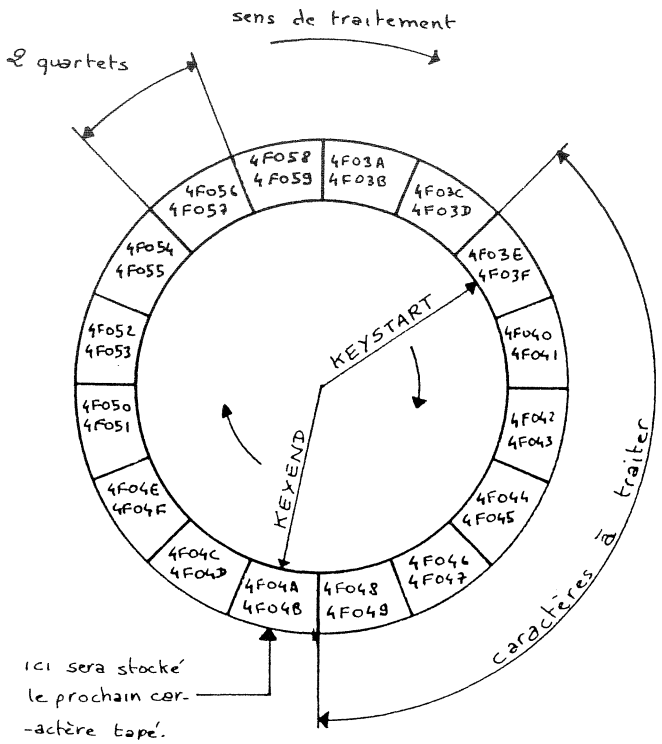
(#4F015 à #4F024 et #4F025 à #4F034). Ces deux zones sont utilisées par #0123E pour le stockage temporaire de certains registres. Elles sont sans doute utilisées de la même façon par d'autres routines.

Le buffer

(#4F038) contient KEYSTART : pointeur de début des touches à traiter.

(#4F039) contient KEYEND : pointeur de fin des touches à traiter.

(#4F03A à #4F059) contient les codes clavier des touches. Les touches restant à traiter vont de #4F03A + 2 × KEYSTART à #4F03A + 2 × KEYEND (non compris). En fait, la meilleure façon de représenter le buffer est de le comparer à une horloge dont les aiguilles seraient KEYSTART et KEYEND :

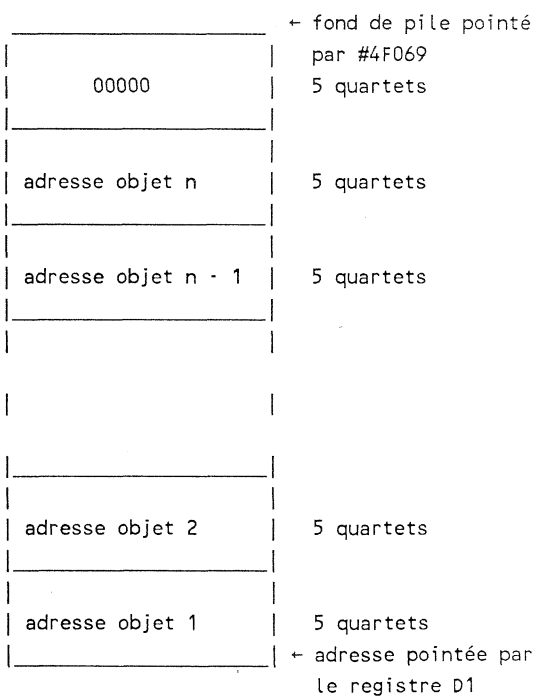


Attention : les codes stockés dans le buffer sont les codes clavier et non les codes ASCII. En voici la liste :

Touche	Code	Touche	Code	Touche	Code
#	18	'	2D	(	14
*	1A	+	02	,	01
-	0E	.	04	/	26
0	07	1	13	2	10
3	0D	4	1F	5	1C
6	19	7	2B	8	28
9	25	=	0B	A	41
B	42	C	48	D	47
E	46	F	44	G	35
H	36	I	3C	J	3B
K	3A	L	38	M	29
N	2A	O	30	P	2F
Q	2E	R	2C	S	1D
T	1E	U	24	V	23
W	22	X	20	Y	11
Z	12	[	16	{	17
«	0C	Space	06	α	08
LC	0A	EVAL	15	STO	21
EEX	31	DROP	32	Effac	33
CHS	34	ENTER	37	SOLV	3D
USER	3E	NEXT	3F	TRIG	40
Curs	43	↓	49	←	4A
→	4B	↑	4C	DEL	4F
INS	51	ON	7F	Shift	80

La pile

Les objets ne sont pas dans la pile, c'est leur adresse (sur 5 quartets) qui est stockée :

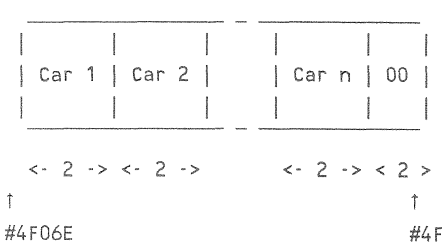


Ainsi pour obtenir l'adresse de l'objet situé au niveau 1 de la pile, suffit il de lire 5 quartets à partir de l'adresse contenue dans le registre D1.

Si, par exemple, le niveau 1 contient "PPC", à l'adresse ainsi lue, on trouvera E4A20B0000050534.

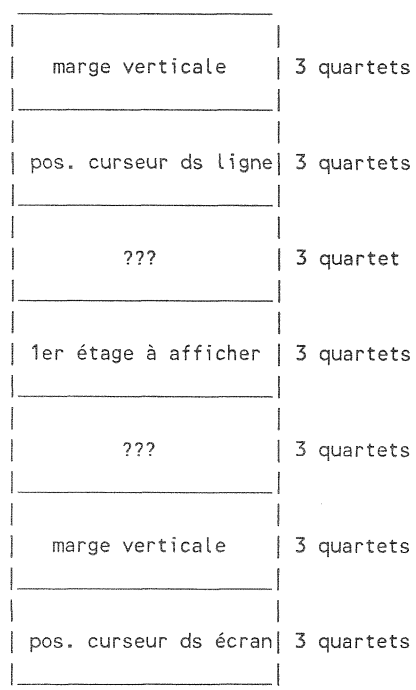
La ligne de commandes

Elle se situe entre l'adresse pointée par #4F069 et celle pointée par #4F06E (non inclus). La structure est la suivante :



Car i est le code ASCII du i^{ème} caractère de la ligne de commande. *n* vaut au minimum 23 si le nombre réel de caractères entrés est inférieur, ils sont completés à 23 par l'adjonction du ou plusieurs caractères de code 00.

Des informations sur le curseur sont stockées de #4F11D à #4F132 :



marge à gauche : nombre de caractères cachés à gauche. Par exemple, si *marge à gauche* vaut 3, `"12345" 1 DISP` donnera ...56.

position curseur dans la ligne de commande i : position du caractère sous le curseur dans la ligne de commande.

premier étage de la pile à afficher : premier étage visible de la pile .

marge verticale : numéro de la ligne à partir de laquelle il faut afficher. Par exemple, si *marge verticale* vaut 3, on aura :

```
"1
2
3
4" 1 DISP
```

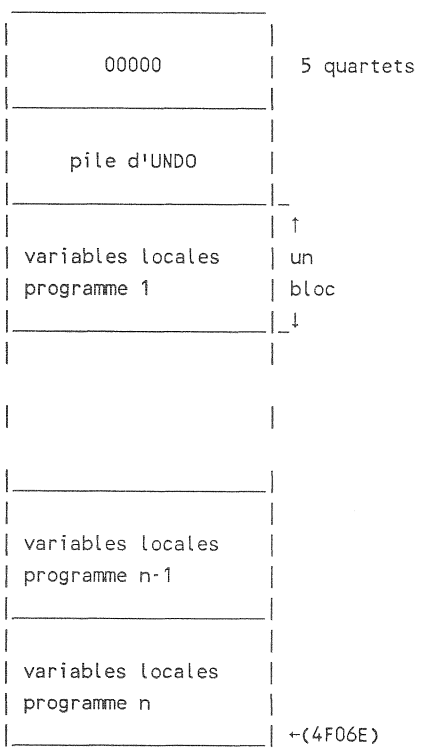
donnera :

```
3
4
```

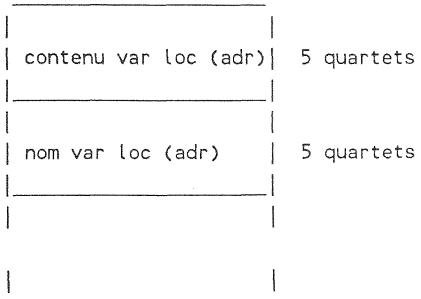
position horizontale dans écran : de 0 à 16_{16}.

La pile d'UNDO et des variables locales

Son adresse se trouve en #4F06E sur 5 quartets. La structure est la suivante :



Chaque bloc ayant la structure :



contenu var loc (adr)	5 quartets
nom var loc (adr)	5 quartets
longueur du bloc	5 quartets
identificateur	5 quartets

nom variable locale pointe sur un nom vide ('') dans le cas où le bloc est relatif à la pile d'UNDO.

identificateur vaut 00002 pour la pile d'UNDO et 00000 pour un bloc de variables locales.

Dans le cas de la pile d'UNDO, le premier nombre après un nom vide est le nombre d'éléments dans la pile.

Si on effectue des boucles FOR ... NEXT ou START ... NEXT, des variables locales sont créées contenant la valeur du compteur ; son nom est ''noname' dans le cas de la boucle START (impossible à entrer au clavier). Une variable ''stop' contient la valeur de fin de boucle (impossible à entrer au clavier). Une boucle est considérée dans ce cas comme un sous-programme. Au retour d'un sous-programme, les variables locales le concernant sont détruites.

Si UNDO n'est pas activé, le bloc UNDO n'existe pas.

Temporary environment

Adresse en #4F078 à #4F07C. C'est une zone qui renseigne le HP-28C sur le menu à exécuter.

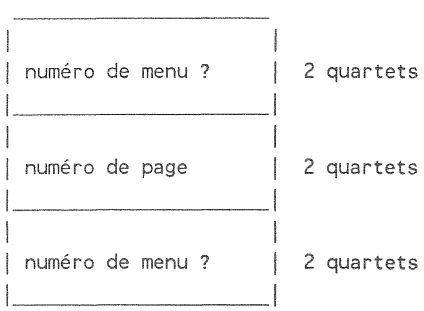
00000	5 quartets
0805A 08371 en mode édition	5 quartets
080BE	5 quartets
adresse 6ème touche	5 quartets
adresse 5ème touche	5 quartets
adresse 4ème touche	5 quartets
adresse 3ème touche	5 quartets
adresse 2ème touche	5 quartets
adresse 1ère touche	5 quartets
039D9	5 quartets
039D9	5 quartets
adresse disp 6	5 quartets
adresse disp 5	5 quartets
adresse disp 4	5 quartets
adresse disp 3	5 quartets
adresse disp 2	5 quartets
adresse disp 1	5 quartets
08C (?)	3 quartets

adresse disp n pointe sur une routine déterminant le nom à placer dans la ligne de commande en mode ALPHA. Si le menu n'est pas le menu USER, le nom en question sera affiché dans le menu.

Pour le menu USER les *adresses disp* sont dans l'ordre :
0E059, 0E07C, 0E0B3, 0E0FE, 0E117, 0E130
et d'exécution :
0DDBB, 0DDD9, 0DDF3, 0DE15, 0DEB3, 0DE51
Ces adresses ont pour rôle :
- pour l'affichage de placer dans la pile le nom du $n^{ème}$ programme (chaîne de caractères).

- pour les secondes de déclencher l'exécution du n^{ème} programme du menu.

Ces routines utilisent certainement les six quartets concernant aussi le menu et situés de #4F141 à #4F146.



Numéro de page est le numéro de l'objet à partir duquel il faut commencer le menu (0,6,C,...)
Numéro de menu semble être le numéro de menu à afficher. Voici la table relevée :

```
Array  : 12  Real    : 0D  Ctrl   : 0E  User : 01
Binary : 0B  Stack   : 08  Branch : 03  Mode : 06
Cmplx  : 13  Store   : 10  Test   : 0F  Logs : 04
String : 14  Algebra : 09  Trig   : 07  Stat : 05
List   : 0C  Print   : 11  Solv   : 0A  Plot : 15
                           Solvr  : 02 (Sous menu)
```

A laquelle on peut ajouter :

```
16 : Menu FORM1  (Colct Expan Level Exget [←] [→])
17 : Menu FORM2  (DNEG DINV *1 /1 ^1 +1 -1)
18 : Menu FORM3  (-() ← → ←M M→ ←A A→)
19 : Menu FORM4  (AF)
1A : Menu FORM5  (1/() ← → ←D D→ ←A A→)
1B : Menu FORM6  (-() L() ←M M→)
1C : Menu FORM7  (1/() E() ←D D→ ←A A→)
1D : Menu FORM8  (1/() E^ D→)
1E : Menu FORM9  (-() L* D→)
1F : Menu FORM10 (→())
00 : pas de menu
```

Rom/Ram pair

Adresse en #4F087 à #4F08B. Elle a déjà été étudiée précédemment (voir la rubrique sur les différents types d'objets).

La pile du LAST

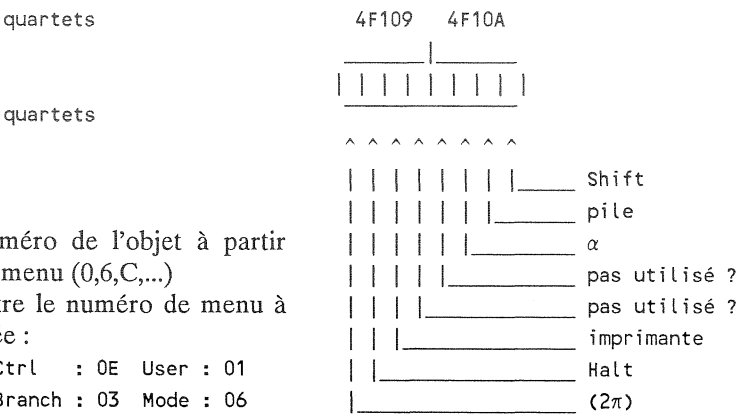
Adresse de #4F096 à 4F0A4. Elle contient les adresses de 1 à 3 objets (si il y a moins de 3 objets les adresses des objets non présents seront 00000). Ce sont les adresses des objets pris par la dernière fonction utilisée. Si LAST est invalidé, les 3 adresses seront 00000.

Pile de commandes

Adresse de #4F0B4 à #4F0C7. C'est une liste de 4 adresses, éventuellement à 00000, pointant sur des chaînes de caractères contenant les caractères d'une ligne de commande.

Stockage des valeurs des indicateurs

2 quartets de #4F109 à #4F10A.



Attention : changer ces valeurs ne conduit pas à un effet immédiat lors de l'exécution d'un programme : c'est seulement à la fin de ce programme que les indicateurs seront réactualisés en fonction de ces valeurs.

Les drapeaux sont de #4F10B à #4F11A. Ce sont 64 bits, chaque bit représentant un drapeau (Flag 1 : Bit 0 de #4F10B, Flag 64 : Bit 3 de #4F11A).

Adresses diverses

En #4F0D7 est inscrite l'adresse de fin de mémoire (#50000 en standard).

En #4F0F5 est inscrit le nombre de quartets dans la pile, c'est à dire 5 fois le nombre d'éléments dans la pile (5×DEPTH).

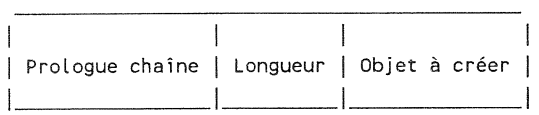
En #4F134 à #4F138 se trouve le numéro de la dernière erreur commise.

PROGRAMMATION EN ASSEMBLEUR

Après ce petit préliminaire, voici la deuxième partie de cet article : comment programmer en langage machine sur le HP-28C. Tout d'abord voici l'explication des programmes ASS et →LEX parus dans JPC numéro 50.

Le but recherché était de permettre la création de n'importe quel objet. L'idée retenue a finalement été un processus en deux étapes :

1- Mise en place de la suite de quartets constituant l'objet à créer sous forme de chaîne de manière à obtenir :



2- Cet objet étant placé dans la pile, la seconde étape consiste à décaler l'adresse contenue dans la pile de manière à ce que l'objet pointé soit l'objet créé et non plus la chaîne.

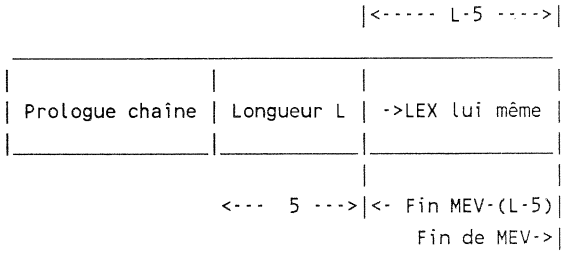
La première étape est réalisée par le programme ASS. La liste des codes hexadécimaux étant placée dans une chaîne, le programme prend 2 codes consécutifs et crée le CHR correspondant de manière à ce que les codes présents dans la Ram soient ceux désirés. Comme le microprocesseur l'attend, les codes ASCII sont stockés dans l'ordre (quartet de poids faible, quartets de poids fort), il faut inverser les 2 chiffres hexadécimaux avant de faire le CHR. Voici le listing de ASS :

```
« → LM                  On place la chaîne contenant les
                        codes hexadécimaux dans la
                        variable temporaire LM.
  « HEX                 Passage en mode hexadécimal.
    ""                  Chaîne vide : y seront ajoutés les
                        CHR représentant les codes.
   1'LM SIZE FOR X      Boucle pour prendre tous les
                        codes hexadécimaux.
    "#"                 préparation de la conversion de
                        deux nombres hexa sous forme de
                        chaîne en un nombre binaire.
      LM X DUP2         On prend deux chiffres hexa
      1 + DUP           et on les inverse. Exemple :
      SUB 3 ROLLD       si on entre "1234" pour X=1
      DUP SUB +         on obtient "21"
      + STR→            On ajoute "#" et on convertit en
                        un nombre binaire (ex "21" -> #21)
      B→R CHR           On prend le CHR correspondant,
                        dans l'exemple CHR(33) ; en
                        mémoire seront stockés, et
                        dans cet ordre, les codes 1 et 2
      +                 On ajoute CHR à la chaîne résultat
                        ("" au départ).
    2 STEP              Couple de caractères suivant.
» »
```

Le second programme, c'est →LEX, qui à partir de la chaîne contenant l'objet place l'objet dans la pile : ce programme ne pouvait être réalisé qu'en langage machine... Cycle infernal me direz vous ! Heureusement non. Là encore, on procède en plusieurs étapes :

- On transforme le listing des codes en un code exécutable grâce à ASS.

- On stocke →LEX (sous forme de chaîne) en début de menu USER. On connaît alors exactement l'endroit où il se trouve :



Ainsi en faisant (Fin de MEV-L+5) SYSEVAL, on pourra démarrer →LEX. Comme →LEX prend une chaîne dans la pile et retourne l'objet qu'elle contient, on place →LEX dans la pile avant de faire ce SYSEVAL. A ce stade, on a le vrai programme →LEX dans la pile : il ne reste plus qu'à le stocker dans →LEX ; on perd la chaîne qui n'est plus d'aucune utilité. Voici le listing de →LEX :

```
Code      Mnémonique      Commentaire
-------------------------------------------------
76C20     CON(5) #02C67   Objet : programme
5B3C1     CON(5) #1C3B5   Routine vérifiant la
                          présence d'au moins un
                          élément dans la pile.
69C20     CON(5) #02C96   Objet : Assembly code
      deb
E1000     CON(5) fin-deb  Longueur du Prgm
143       A=DAT1 A        A=^ Objet niv 1
133       AD1EX
179       D1=D1+ 10
133       AD1EX           A=^chaîne+10 =^Objet créé
141       DAT1=A A        Ecriture nouvelle adresse
                          dans la pile
142       A=DAT0 A        fin de programme :
164       D0=D0+ 5        cf explications plus bas
808C      PC=(A)
09F20     CON(5) #02F90   fin de l'objet programme
      fin
```

Il faut savoir que :
- D1 est l'adresse du sommet de la pile
- D0 est le pointeur RPL : à la fin de chaque programme assembleur, il faut toujours faire :

```
A=DATO A
DO=DO+ 5
PC=(A)
```

C'est en fait ceci qui permet l'exécution de l'objet suivant (ici 02F90, fin de structure). Une fois les codes entrés puis assemblés grâce à ASS, on procède comme expliqué plus haut. L'adresse du SYSEVAL à effectuer est :
#4FFCE (#50000 - #32) en version standard
#51FCE (#52000 - #32) en version 6 Ko
#5FFCE (#60000 - #32) en version 34 Ko
où #32 (50) est la longueur du programme →LEX.

Dorénavant pour créer un objet dans la pile aucun SYSEVAL n'est plus à faire. Par exemple, créons artificiellement "PPC" dans la pile :
Codes : E4A20B0000050534
Pour le créer : `"E4A20B0000050534" [ASS] [→LEX]`

A noter une chose intéressante : si on crée un nom (prologue : 02D12) comportant des caractères non acceptés au clavier (par exemple : espace, [,] ou tout autre caractère), il sera accepté pour un STO (la vérification de la bonne syntaxe d'un nom se fait à l'entrée et non lors du STO. Par exemple on peut créer un programme nommé 'A B C', entrez : `"21D20501402240234" [ASS] [→LEX].`

Il y aura alors 'A B C' dans la pile et le STO sera possible. De cette manière, on peut créer des programmes portant le même nom qu'une instruction RPL et ce programme sera prioritaire sur l'instruction mais seulement s'il est lancé du clavier et non exécuté par l'intermédaire du menu contenant l'instruction. Ainsi SYSEVAL et CLUSR peuvent être facilement recréées... Par exemple pour CLUSR :

```
<< "ET LE CODE ??? " 1 DISP 1000 .1 BEEP >>
"21D205034C4553525" [ASS] [→LEX] [STO]
```

Dorénavant, toute tentative pour faire un CLUSR ne pourra aboutir. Il est intéressant de remarquer qu'une fois le programme CLUSR créé, il est possible d'entrer directement le nom CLUSR sans passer par ASS. Pour revenir à un état normal : `'CLUSR' [PURGE].`

Dernier point : comment concevoir un Lex pour le HP-28C.

Tout dépend si vous désirez mixer des instructions RPL à de l'assembleur ou faire un programme assembleur seul. Dans le second cas, créer le programme assembleur en commençant par le prologue *Assembly code*. Dans le premier cas, réaliser une structure composite (prologue de programme) contenant les instructions RPL et un ou plusieurs Assembly codes.

Dans tout code assembleur :
- Sauvegarder les registres D0, D1, B et D si besoin est
- Terminer par la récupération éventuelle des dits registres puis par :

```
A=DATO
DO=DO+ 5
PC=(A)
```

Dans le cas de la structure composite programme, ne pas oublier le 02F90 terminal.

Je ne saurais trop vous conseiller l'achat d'un livre sur le HP-71B contenant les instructions avec leurs codes (HDS par exemple) ou la lecture des articles de Jacques Baudier dans JPC 46 page 25 et JPC 47 page 26.

A présent, c'est à vous ! Et envoyez vos réalisations au Journal...

En remerciant Pierre David et Janick Taillandier pour leur aide et leur patience, Graeme Cawsey pour sa liste des différents prologues et Sébastien Lalande pour son aide lors de l'analyse du Row Driver Waveform.

Paul Courbis (392)

NDLR : Il est bien entendu inutile de contacter HP à propos des informations contenues dans cet article. Il faut également noter que toutes les adresses données sont relatives au HP-28C version 1BB, elles peuvent être différentes sur les versions ultérieures.