INITIATION A L'ASSEMBLEUR

Olivier Nachba et Cedric Lecourt

POUR HP48 S - SX - G - GX

L'invitation au voyage...

angkor

INITIATION A L'ASSEMBLEUR SUR HP 48 S/SX ET G/GX

L'INVITATION AU VOYAGE

INITIATION A L'ASSEMBLEUR SUR HP 48 S/SX ET G/GX

L'INVITATION AU VOYAGE

Olivier NACHBA
&
Cédric LECOURT

Toutes les marques citées dans ce livre sont des marques déposées.

Première édition : Octobre 1994

© 1994, Angkor, Paris
ISBN 2-87892-008-2

Editions Angkor
6, rue Maître Albert
75005 Paris
Tel : (1)-43-25-45-20

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

La loi du 11 mars 1957 n'autorisant, aux termes des alinéas 2 et 3, de l'article 4, d'une part, que les "copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective", et, d'autre part, que les analyses et les courtes citations dans le but d'exemple et d'illustration, "toute représentation ou reproduction intégrale, ou partielle, faite sans le consentement de l'Auteur ou de ses ayant droit, ou ayant cause, est illicite" (alinéa premier de l'article 40).
Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles 425 et suivant du Code Pénal.

Olivier NACHBA
19 ans, élève de terminale S au lycée Jules Ferry.

Cédric LECOURT
20 ans, élève ingénieur à l'Ecole Centrale de Lille, ancien élève de mathématiques spéciales au lycée Condorcet, Paris.

Nous tenons à remercier :
François Xavier et **Anabelle ALVAREZ** pour leur ordinateur et leur aide.
Laurent BERNARD (Larry) pour son assembleur.
Cyrille de BREBISSON (**HP mâd**) pour son immense aide à tous les niveaux.
Stéphane COHEN pour la conception de la couverture.
Paul COURBIS pour les nombreux stylos détruits sur nos premiers jets.
Stéphane DOAT pour ses conseils et ses corrections.
Albert EINSTEIN pour la théorie de la Relativité Générale.
Etienne de FORAS pour ses conseils et suggestions.
Eric HOURI pour ses bons conseils.
La société **MAUBERT ELECTRONIC**.
Francis MUSEUX pour ses conseils, corrections.
Sylvie Putot et Corinne Thiébault pour avoir été d'excellentes débutantes.
Jean-Claude RICHARD pour ses nombreux conseils.
Gérald SQUELART (Warlock) pour ses suggestions.
Agnès et Eric TERRIEN pour leurs idées, commentaires.

NOTE AU LECTEUR

Cet ouvrage s'adresse aussi bien aux débutants qu'aux programmeurs expérimentés cherchant à aborder la programmation en assembleur sur HP48. Il comprend à la fois des chapitres d'initiation et des chapitres de programmation avancée.

La programmation en assembleur nécessite un utilitaire nommé "assembleur", permettant à la HP 48 de comprendre des programmes écrits dans ce langage. Il existe de nombreux assembleurs pour la HP48, à la fois sur celle-ci et sur des machines plus puissantes : APPLE MACINTOSH, IBM PC, COMMODORE AMIGA, ATARI ST... Ces assembleurs sont relativement similaires dans leurs syntaxe. L'assembleur de référence pour ce livre est celui donné dans la disquette d'accompagnement (voir chapitre 5).

Ce livre est divisé en quatre parties :

- Une introduction vous présentant les bases et vous familiarisant avec la programmation en assembleur. Des exercices corrigés se situent à la fin de chaque chapitre ;
- Une deuxième partie vous présentant la programmation avancée en langage assembleur ;
- Une bibliothèque de programmes prêts à l'emploi et documentés qui vous dévoileront quelques méthodes de programmation classiques ;
- Enfin, une série d'annexes vous présentant différentes informations utiles.

Il est important de noter que l'existence de différents modèles de HP48 (S/SX, toutes versions et G/GX, toutes versions) est prise en compte dans cet ouvrage. Sauf indication contraire, les informations données sont valables sur tous les modèles.

Nous vous souhaitons une bonne et agréable lecture !

TABLE DES MATIERES

Note au lecteur ..... Page 1

Table des matières ..... Page 3

Première partie

Les bases usuelles

Chapitre 1 : Introduction au calcul hexadécimal ..... Page 11

Chapitre 2 : Système de notation des bases ..... Page 21

Chapitre 3 : Conséquences de l'introduction des bases binaire et hexadécimale ..... Page 23

Chapitre 4 : Calculs élémentaires ..... Page 29

Deuxième partie

L'assembleur

Chapitre 5 : Qu'est-ce que l'assembleur et la syntaxe des assembleurs ..... Page 47

Chapitre 6 : Votre premier programme ..... Page 51

Chapitre 7 : Les boucles ..... Page 59

Chapitre 8: Les tests de touche ..... Page 65

Chapitre 9 : Les pointeurs et l'utilisation de la pile .. Page 75

Chapitre 10 : Le graphisme..... Page 83

Chapitre 11 : Récapitulatif 1 : Les registres ..... Page 91

Chapitre 12 : Récapitulatif 2 : Les calculs en assembleur..... Page 105

Chapitre 13 : Récapitulatif 3 : Les sauts ..... Page 113

Chapitre 14: Les erreurs à éviter, les "bons trucs"... Page 119

Troisième partie

Bibliothèque de programmes

Chapitre 15 : Avertissement ..... Page 129

Utilitaires :

->object ..... Page 133

object-> ..... Page 136

->MACRO ..... Page 138

DUMPER ..... Page 142

GRAPHICS VIEWER (SOFT) ..... Page 148

GRAPHICS VIEWER (HARD) ..... Page 151

DIGITCONV ..... Page 154

DIGITPLAY ..... Page 159

LABY ..... Page 162

ASM REPL ..... Page 164

GRIS ..... Page 170

CRNAME ..... Page 174

ASM NEWOB ..... Page 176

Jeux :

HPTRON ..... Page 178

LA BEBETE ..... Page 186

HPUISSANCE 4 ..... Page 205

Mathématiques :

DECOMP ..... Page 219

Annexes

Annexe 1 : Corrigés des exercices ..... Page 227

Annexe 2 : Liste des routines en ROM ..... Page 229

Annexe 3 : Liste des logiciels les plus utiles sur HP 48 ..... Page 239

Annexe 4 : Comment reprendre les logiciels de la disquette d'accompagnement ..... Page 245

Annexe 5 : Glossaire ..... Page 247

Annexe 6 : Index ..... Page 253

Première partie

Les bases usuelles

INTRODUCTION AU CALCUL HEXADECIMAL

Le but de ce chapitre est de vous familiariser avec les calculs dans la base hexadécimale. Il s'agit certainement d'un des chapitres les plus "mathématiques" du livre ; sa parfaite compréhension est toutefois indispensable pour la programmation en assembleur. Vous êtes prêts ? Prenez votre souffle, on plonge !

I) Notion de base

La première chose à comprendre est la notion de "base" : qu'est ce qu'une base ? A quoi sert-elle ? Du point de vue mathématique, il s'agit tout simplement d'un ensemble de symboles permettant de compter des éléments. C'est-à-dire qu'à chaque quantité, on associe un ou plusieurs symboles.

Prenons comme exemple la base 10 qui est la base naturelle dans laquelle nous comptons tous. Nous avons à notre disposition un jeu de dix symboles :

A zéro élément, on associe le symbole "0" ;

A un élément, le symbole "1" ;

A deux éléments, le symbole "2" ;

....

A neuf éléments, le symbole "9".

Nous avons donc utilisé dix symboles pour compter : { 0 1 2 3 4 5 6 7 8 9 }. Ces symboles peuvent être de forme quelconque, et en nombre quelconque (il en faut cependant au moins deux). Ainsi, on aurait pu, par exemple, choisir de compter avec les quatre symboles suivants : { a W ç & }. La technique de dénombrement aurait été la même :

A zéro élément on associe le symbole "a" ;

A un élément, le symbole "W" ;

A deux éléments, le symbole "ç" ;

A trois éléments, le symbole "&".

Nous aurions ainsi constitué une base à 4 symboles, plus communément appelée base 4.

Nous avons maintenant un ensemble de symboles à notre disposition. Cependant, cet ensemble est limité, c'est pourquoi il nous faut trouver un moyen de dénombrer des éléments en nombre plus important, si important que nos symboles ne suffisent plus pour les compter. Nous avions créé les chiffres. Nous allons maintenant créer les nombres.

Commençons par étudier la base 10 : comment, par exemple dénombrer les vingt-et-un éléments suivants :

o o o o o o o o o o o o o o o o o o o o o

Nos vingt-et-un éléments précédents seront regroupés comme ceci :

o o o o o o o o o o o o o o o o o o o o o

Nous pouvons maintenant compter nos éléments : nous avons "2" paquets et "1" élément. Ceci se note conventionnellement dans le sens Nombre de paquets - Nombre d'éléments, soit ici : "2 1" : Nous venons de créer 21 !

Il faut bien comprendre que, par la suite, le principe reste le même : lorsqu'on n'a plus assez de chiffres pour dénombrer nos paquets, on fait des paquets de paquets et ainsi de suite...

Ainsi, par exemple, en base 10, cent trente deux éléments se décomposent en :

- 1 paquet de 10 paquets de 10 éléments ;
- 3 paquets restants de 10 éléments ;
- 2 éléments restants.

On vient de créer "1 3 2" plus simplement noté 132.

Allons encore plus loin. Nous venons d'étudier le cas de la base 10, mais cela est similaire quelque soit la base. Ainsi, dans notre base précédente { a W ç & }, nous pouvons dénombrer treize éléments en les regroupant par quatre, quatre étant le plus petit nombre d'éléments qu'on ne sait pas compter. Treize, c'est trois paquets de quatre éléments, et un élément restant. A trois éléments, on associe le symbole "&", et à un élément, le symbole "W", treize s'écrira donc dans notre base " &W ".

Voilà, vous connaissez maintenant les caractéristiques propres à une base, tant sur le plan de ses fondements que de son dénombrement. Nous pouvons maintenant aborder la base qui nous intéresse soit l'hexadécimal.

II) Pourquoi introduire l'hexadécimal ?

Votre machine, comme tous les ordinateurs, ne sait traiter qu'une forme de signaux : des impulsions électriques. Elle ne sait même que très mal les gérer, car elle ne parvient à identifier que deux états : la présence ou l'absence de courant, ce qu'on symbolise par les deux chiffres 0 (aucun courant) et 1 (du courant). Tout en mémoire est donc stocké sous la forme :

1101011001010111000100011001011001010001010100 ...

Puisque la machine ne sait reconnaître que deux symboles (0 et 1), elle travaille en base 2, ou **binaire**. Elle effectue tous ses calculs dans cette base. Ainsi on peut introduire la notion de bit. Un bit peut prendre la valeur 1 ou 0 qui correspond à une impulsion électrique ou à l'absence d'impulsion électrique.

Compter en binaire donne la suite :

0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011,
1100, 1101, 1110...

Ce qui devient vite illisible, car il faut un nombre important de chiffres pour représenter des quantités même petites. Il a donc fallu trouver une autre base permettant, d'une part, de rendre les informations beaucoup plus lisibles, et, d'autre part, permettant de retrouver facilement l'écriture binaire des nombres étudiés.

On a donc choisi de regrouper les bits par 4 (soit 16 combinaisons possibles). On va par conséquent travailler en base 16 ou base **hexadécimale** pour répondre à ces besoins. En effet, tout nombre compris entre zéro et quinze s'écrit sous une forme unique :

$$n=1*a+2*b+4*c+8*d$$

avec a, b, c et d égaux à zéro ou un selon n.

De plus, la théorie mathématique précédente nous apprend que ce nombre s'écrit en binaire sous la forme "dcba". L'hexadécimal répond donc bien à nos attentes.

III) L'hexadécimal

Notre premier problème est le suivant : comment représenter un nombre en base 16 lorsque nous n'avons que dix chiffres ? Il nous faut nécessairement créer de nouveaux chiffres. Conventionnellement, on utilise les symboles suivants : { 0 1 2 3 4 5 6 7 8 9 A B C D E F }. On compte donc comme suit :

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, ... , 19,1A,1B,1C,... , 9D, 9E, 9F, A0, A1,...

Voilà ! Nous avons maintenant notre base hexadécimale. Etudions ses propriétés :

IV) Les propriétés de la base hexadécimale

1) Les conversions binaire-hexadécimal

Il est aisé de convertir un nombre binaire en hexadécimal :

Soit le nombre binaire "abcdefghij", donc avec a, b, c, d, e, f, g, h, i, j égaux à 0 ou 1. Convertissons-le en hexadécimal. Pour cela, on le coupe en morceaux de quatre chiffres, en partant de la droite, et en rajoutant des zéros à gauche si besoin est : "00ab" , "cdef" et "ghij". On convertit chacun de ces nombres en un chiffre hexadécimal grâce à la méthode suivante :

LE NOMBRE BINAIRE "abcd" SE REPRESENTE EN HEXADECIMAL PAR LE SYMBOLE CORRESPONDANT A : 1*d+ 2*c + 4*b+ 8*a

On obtient, pour chaque "morceau", un chiffre hexadécimal. Pour avoir la valeur voulue, dans cette base 16, il suffit de placer ces chiffres dans le même ordre que les parties.

Convertissons par exemple 10011101011011.

On le découpe : 0010 / 0111 / 0101 / 1011,

On convertit les chiffres grâce à la formule : 2 7 5 B,

On obtient donc la représentation hexadécimale: 275B.

2) Les conversions hexadécimal-binaire

On peut, de facilement convertir de l'hexadécimal en binaire, en utilisant la décomposition en somme de puissances de 2:

TOUT CHIFFRE HEXADECIMAL S'ECRIT DE FACON UNIQUE SOUS LA FORME 1*d+ 2*c+ 4*b+ 8*a; AVEC a, b, c, d EGAUX A 0 OU 1, ET SA REPRESENTATION BINAIRE EST : abcd

Ainsi, pour convertir un nombre hexadécimal en binaire, on transforme chacun de ses chiffres en binaire, et on garde le même ordre afin de créer le nombre binaire. Prenons un exemple concret pour que vous vous rendiez mieux compte de cette méthode :

Convertissons le nombre hexadécimal F85A

On le découpe : F / 8 / 5 / A,

On convertit chacun de ses chiffres :

F=1+2+4+8 donc en binaire, F s'écrit 1111;

8 = 8 donc en binaire, 8 s'écrit 1000;

5 =1+4 donc, en binaire, 5 s'écrit 0101;

A=2+8 donc, en binaire, A s'écrit 1010.

Enfin, on replace les nombres binaires dans le même ordre que les chiffres : 1111 1000 0101 1010, on obtient donc la valeur de F85A en binaire : 1111100001011010.

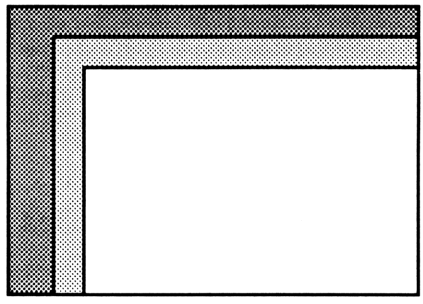
Remarque : les 0 situés à gauche du nombre obtenu peuvent être ôtés (00111100 se simplifie en 111100).

V) Conclusion

Nous voici arrivé à la fin de ce premier chapitre dédié à l'hexadécimal. Nous vous conseillons de bien le maîtriser avant de passer à la suite de l'ouvrage où ces notions seront supposées acquises. Cependant, si vous désirez vous entraîner à la manipulation des bases binaire et hexadécimale, nous vous conseillons de vous aider de votre machine avec les commandes : **BIN**, **DEC**, **HEX**, ainsi que du tableau suivant qui présente les valeurs hexadécimales, décimales et binaires de 0 à 255 (les plus utiles au programmeur) :

	B	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
	I	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
	N	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
		0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
BIN	HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000	0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0001	1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0010	2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
0011	3	48	49	450	51	82	53	54	55	56	57	58	59	60	61	62	63
0100	4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
0101	5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
0110	6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
0111	7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
1000	8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
1001	9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
1010	A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
1011	B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
1100	C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
1101	D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
1110	E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
1111	F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Le tableau ci-dessus peut être schématisé par le dessin suivant :



La partie gris foncé représente les chiffres binaires, la partie gris clair les chiffres hexadécimaux et la partie blanche les nombres décimaux. Un nombre décimal se trouve à l'intersection d'une ligne et d'une colonne représentant respectivement le début et la fin de

ce nombre dans les bases binaires et hexadécimales. Ainsi 151 s'écrit 10011110 en binaire, 97 en hexadécimal (voir tableau).

SYSTEME DE NOTATION DES BASES

Dorénavant, afin d'éviter d'avoir à donner pour chaque nombre sa base, nous adopterons le système de notation de la HP48, à savoir que les entiers seront représentés sous la forme # XXXXXa où "a" est la base de l'entier, indiquée par son initiale :

b représente la base binaire,

d la base décimale,

h la base hexadécimale.

Et XXXXX représente la valeur de cet entier dans cette base. Cela permettra donc des notations du type : # 10110011b = # 179d = # B3h.

Remarque : Pour les nombres décimaux, nous les présenterons dans la plupart des cas sous leur forme habituelle soit par exemple : 1234567890

Exercices :

A.2.1 : Convertissez les nombres suivant en décimal, si possible en utilisant le tableau :

09h ; # 01001010b ; # 1515h ; # 14h ; # 48h ; # 1101011101b

A.2.2 : Convertissez les nombres suivant en hexadécimal :

1515d ; # 10110100b ; # 111d ; # 230823d ; # 100100000b

A.2.3 : Convertissez les nombres suivant en binaire :

255d ; # FFh ; # 161d ; #24d ; #10h

CONSEQUENCES DE L'INTRODUCTION DES BASES BINAIRE ET HEXADECIMALE

Comme la base décimale a amené les notions de dizaine, centaine..., les bases binaire et hexadécimale amènent une multitude de nouvelles notions, dont la plus simple est le bit.

I) Le bit

Il s'agit de la plus petite unité que sait gérer la machine : le chiffre binaire. Un bit ne peut donc prendre que deux valeurs : 0 ou 1. Il amène les notions de quartet et d'octet.

II) Le quartet

Comme son nom le rappelle, le quartet est un ensemble ordonné de quatre bits. Il y a donc seize quartets possibles de # 0000b à # 1111b. Un quartet peut par conséquent prendre toutes les valeurs possibles comprises entre # 0h et # Fh, un quartet contient exactement un chiffre hexadécimal.

Le quartet est, avec le bit, l'entité la plus manipulée en assembleur sur HP48.

Le quartet est appelé en anglais Nibble ou Nybble.

III) L'octet

Comme son nom l'indique aussi, l'octet est un ensemble ordonné de huit bits. Il y a par conséquent 256 valeurs possibles pour l'octet, comprises entre # 00000000b et # 11111111b ; soit toutes les valeurs possibles entre # 0h et # FFh, un octet est donc un ensemble de deux chiffres hexadécimaux, donc de deux quartets. Bien que très utilisé sur la majeure partie des ordinateurs, l'octet est assez peu employé sur HP48.

L'Octet s'appelle aussi souvent Byte (à ne pas confondre avec Bit !).

Sur la HP les caractères sont codés sur 2 quartets. Ce qui permet d'en avoir 256. Ainsi la lettre "A" correspond au 65ème caractère.

IV) Le kilo-octet

Il s'agit d'un ensemble d'octets. Cependant, contrairement à la notion habituelle de kilo, le kilo-octet ne contient pas mille octets mais 1024, qui facilite les calculs en assembleur, un kilo-octet étant égal à # 400h octets. En fait, ceci réside dans le fait que tout en informatique est basé (par construction) sur des puissances de 2. Ainsi il est plus facile de faire des mémoires de 1024 octets que de 1000 octets

V) Le méga-octet, Le giga-octet...

De même que pour le kilo-octet, le méga-octet contient 1024 Kilo-octets soit 1048579 octets. Le giga-octet contient 1024 méga-octet soit 1073741824 octets...

VI) Repérage des bits dans un quartet, dans un octet

Nous venons de voir qu'un quartet contient quatre bits. Il est important de remarquer que l'ordre doit être respecté (# 1010b n'est pas égal à # 1100h par exemple). Afin de pouvoir facilement repérer les bits lors de certaines manipulations, on les numérote de droite à gauche.

Ainsi, si on étudie un quartet # $\alpha\beta\chi\delta$b, on va numéroter ses bits comme suit :

3	2	1	0
α	β	χ	δ

L'ordre de numérotation peut vous apparaître farfelu, pourquoi ne numérote-t-on pas conventionnellement de gauche à droite ? Pour répondre à cette question, il faut se souvenir de la décomposition en puissances de deux :

$$\#\ \alpha\beta\chi\delta b = 1*\alpha + 2*\beta + 4*\chi + 8*\delta$$

Cela peut s'écrire plus simplement si on fait paraître les puissances de deux. On obtient alors la formule suivante de décomposition d'un quartet suivante :

$$(2^0)*(\text{Bit 0}) + (2^1)*(\text{Bit 1}) + (2^2)*(\text{Bit 2}) + (2^3)*(\text{Bit 3}).$$

Elémentaire, non ? Le numéro attribué à chaque bit correspond tout simplement à la puissance de deux représentée par ce bit.

Cette notion peut tout aussi bien être étendue à l'octet. On numérote maintenant de zéro à sept, toujours de la droite vers la gauche, et on a toujours la formule de décomposition :

$$(2^0)*(\text{Bit } 0) + (2^1)(\text{Bit } 1) + (2^2)(\text{Bit } 2) + (2^3)(\text{Bit } 3) + (2^4)(\text{Bit } 4) + (2^5)(\text{Bit } 5) + (2^6)(\text{Bit } 6) + (2^7)(\text{Bit } 7).$$

7	6	5	4	3	2	1	0
η	γ	ϕ	ε	δ	χ	β	α

VII) Bit de poids faible - Bit de poids fort

Il ne s'agit pas, dans ce paragraphe, de vous apprendre de nouvelles notions, mais de vous donner quelques définitions d'objets couramment utilisés par le programmeur en assembleur.

Bit de poids faible : On appelle "bit de poids faible" le bit portant le numéro 0 de l'entité que l'on manipule (quartet, octet...).

Bit de poids fort : On appelle "bit de poids fort" le bit portant le plus grand numéro sur l'entité manipulée, à savoir le bit 3 pour un quartet, le bit 7 pour un octet, le bit 144 pour un entier de 145 bits...

Bits de poids faible : On appelle "n bits de poids faibles", les bits de numéro 0 à n-1 de l'entité manipulé.

Bits de poids forts : On appelle les n bits portant les numéros les plus élevés de l'entité manipulée (ex : les 2 bits de poids fort d'un octet sont les bits 7 et 6).

Quartet(s) de poids faible, quartet(s) de poids fort : De même, on peut définir ces notions sur les quartets. En numérotant les quartets de l'entité manipulée, comme on avait numéroté les bits.

VIII) Conclusion

Au terme de ce chapitre, vous devriez normalement être sensibilisé avec les notions de bit, quartet, octet qui seront particulièrement importantes par la suite. Encore une fois, si cela vous paraît flou, il serait préférable de lire et relire ce chapitre jusqu'à en obtenir une bonne maîtrise.

Exercices :

A.3.1 : Quelles sont les valeurs du bit de poids faible de chavun des octet suivants :

10h # 9d # 10110b

A.3.2 : Quelles sont les valeurs des bits de poids fort de chacun des octets suivants octets suivants :

123d # 7h # 1001001b

A.3.3 : Combien y a t-il de kilo-octet dans un méga-octet ? dans un giga-octet ?

CALCULS ELEMENTAIRES

Ce chapitre a pour but de vous sensibiliser, tout d'abord aux problèmes que posent les calculs en assembleur, puis aux différents types de calculs que l'on peut effectuer dans ce langage.

I) Les problèmes posés par les calculs en assembleur

Un nombre de chiffres limité

Lorsque nous calculons "à la main", le problème du nombre de chiffres ne se pose pas. Nous travaillons avec autant de chiffres que nécessaire, en en rajoutant si besoin est.

En assembleur, nous travaillons, comme nous le reverrons par la suite, sur des "variables" de taille limitée : un octet pour certaines, un quartet pour d'autres, cinq quartets pour d'autres encore... Il se pose donc rapidement un problème : IL EXISTE UN MAXIMUM POUR CHAQUE "VARIABLE". Ainsi, un octet ne peut contenir de valeur supérieure à # FFh, un quartet ne peut gérer un entier supérieur à # Fh ...

De plus, face à ce premier problème, deux autres inconvénients se présentent, dont le premier est :

L'inexistence d'entiers négatifs

En effet, toujours à cause du format des "variables", les nombres négatifs ne peuvent exister en assembleur, les variables ne pouvant contenir une valeur inférieure à # 0h.

Il existe enfin un dernier problème lors des calculs en assembleur :

L'impossibilité de gérer des nombres "non entiers".

Toujours de part la structure des "variables", il est impossible de gérer des nombres autres que des entiers naturels en assembleur.

Vous venez donc d'être sensibilisé aux principaux problèmes posés par la programmation en assembleur sur HP48. Cependant, ne vous inquiétez pas trop, car, même si cela peut paraître très contraignant, il s'avère, lors de la programmation, que ces "défauts" ne sont que peu, voire pas du tout, gênants.

II) La gestion circulaire des nombres

Il s'agit d'une conséquence de la taille limitée des "variables". Etudions-la sur plusieurs exemples.

Supposons que l'on travaille sur une variable contenant trois chiffres, c'est-à-dire dont la valeur maximale est # FFFh. Que se passe-t-il, par exemple, lorsqu'on calcule # FFFh + # 1h ?

En théorie, le résultat de ce calcul est #1000h ; cependant, on ne peut avoir d'entier de plus de trois chiffres. On ne compte donc pas # FFEh, # FFFh, # 1000h, # 1001h mais comme suit :

FFEh, # FFFh, # 000h, # 001h...

Il est essentiel de noter que, dès qu'il y a débordement, on continue après être revenu à # 0h. Ainsi le résultat de # FFFh + # 1h sera # 0h, mais surtout sachez bien qu'une addition comme # FFFh + # 8h va donner comme résultat # 7h.

De même, vous pouvez vous entraîner à compter, par exemple, # C00h + # 561h. Si vous n'avez pas trouvé # 161h, vous pouvez relire ce chapitre. Il va de soi que ce système de calcul est indépendant du nombre de chiffres étudiés (il faut faire attention au nombre de chiffres avec lequel on va compter). Ainsi, par exemple, en travaillant sur un octet, # FEh + # 3h = # 1h.

Pour la soustraction, le principe est le même. Supposons par exemple que l'on travaille sur trois quartets, c'est-à-dire douze bits, on aura le résultat suivant :

0h - # 1h = # FFFh

Encore une fois, le principe est identique quel que soit le nombre de chiffres. Exercez-vous par exemple à compter sur un octet # A4h-# B8h. Vous avez trouvé # ECh ? Bravo ! Nous pouvons maintenant passer aux calculs proprement dits.

III) Les calculs les plus simples

Nous allons traiter, dans ce paragraphe, des calculs les plus simples que l'on puisse effectuer sur des entiers naturels. A savoir l'addition, la soustraction, la multiplication et la division, bien que la multiplication et la division ne soient pas des calculs simples étant donné que dans la majeure partie des cas, on doit les programmer.

1) L'addition

Il s'agit de la même opération que celle que l'on effectue habituellement s'il n'y a pas de débordement. Dans le cas contraire, les nombres sont gérés de façon circulaire comme nous l'avons vu précédemment.

2) La soustraction

Là aussi, il s'agit de la même opération que celle que nous effectuons couramment, à la gestion des débordements près, les nombres négatifs n'existant pas.

3) La multiplication

Là encore, on effectue les mêmes calculs "qu'à la main" cependant, les débordements peuvent engendrer des erreurs dans le résultat (cela dépend du type de multiplication que l'on effectue). L'assembleur de la HP48 ne permet pas d'effectuer directement cette opération : elle devra être programmée.

4) La division

Puisqu'on ne sait gérer que des nombres de type entier, la division sur les entiers sera forcément euclidienne. Lorsqu'on divise a par b, on obtient le quotient q et le reste r tels que a=bq+r et $0 \leq r < b$. Il n'y a aucun risque de débordement puisque q et r sont nécessairement inférieurs à "a". L'assembleur de la HP48 ne permet pas d'effectuer directement cette opération : comme pour la multiplication, elle devra être programmée.

Nous venons de voir les calculs habituels sur les entiers hexadécimaux. Il existe cependant d'autres types d'opérations pouvant être effectuées simplement, dont les premières sont les décalages.

IV) Les décalages

Il s'agit ici de décaler des bits ou des ensembles de bits dans un entier, et ce, vers la gauche ou vers la droite.

1) Le décalage d'un bit à droite

Il s'agit du décalage le plus simple. Supposons que l'on travaille sur n bits. Lors du décalage, le bit de poids fort, donc le bit n-1, devient le bit n-2 , le bit n-2 devient le bit n-3... le bit 2 devient le bit 1, le bit 1 devient le bit 0. Enfin, le bit 0 est perdu et le nouveau bit n-1 est mis à zéro. Prenons l'exemple de # E3h. En binaire, il s'écrit # 11100011b. Le décalage le transforme donc en # 01110001b ou # 71h.

Ce que l'on peut remarquer ici c'est que, de par la décomposition en puissance de deux, le décalage d'un bit à droite correspond à une division par deux, le reste étant en théorie perdu (nous verrons au cours du chapitre sur la programmation qu'il est possible de savoir ce qui est sorti). En effet, le bit p qui était auparavant affecté à la valeur 2^p est maintenant affecté à la valeur 2^{p-1} en se transformant en bit p-1.

3	2	1	0
δ	χ	β	α

avant

3	2	1	0
0	δ	χ	β

après

2) Le décalage d'un bit à gauche

Il s'agit d'une opération similaire à la précédente : on décale tous les bits, mais, cette fois, vers la gauche. Pour un entier de n bits : le bit n-1 est perdu, le bit n-2 devient le bit n-1, le bit n-3 devient le bit n-2, ... , le bit 0 devient le bit 1 et le nouveau bit 0 est passé à 0. Reprenons l'exemple de # E3h. Il s'écrit en binaire # 11100011b, le décalage le transforme en # 111000110b soit # 1C6h.

Remarque : Si on travaille sur un octet, le résultat est # 11000110b = # C6h

On peut là aussi remarquer que cette opération correspond à une multiplication par deux, et ce, pour les mêmes raisons que précédemment.

3	2	1	0
δ	χ	β	α

avant

3	2	1	0
χ	β	α	0

après

3) Le décalage d'un quartet à droite

Il s'agit d'une opération similaire à la première. On décale ici d'un quartet (4 bits) au lieu d'un bit à droite. Si on étudie un entier de n quartets, le quartet n-1 devient le quartet n-2... Le quartet 1 devient le quartet 0. Le nouveau quartet n-1 est passé à 0, et l'ancien quartet 0 est perdu. Si on reprend l'exemple de # E3h, il devient après décalage # Eh.

Un quartet étant un ensemble de quatre bits, on peut remarquer qu'un décalage d'un quartet à droite correspond à quatre décalages d'un bit à droite. En conséquence, décaler d'un quartet à droite correspond à effectuer une division par seize. On peut ainsi faire des divisions par 32, 64, 128...

quartet 2				quartet 1				quartet 0			
11	10	09	08	07	06	05	04	03	02	01	00
λ	κ	φ	ι	η	γ	ϕ	ε	δ	χ	β	α

avant

quartet 2				quartet 1				quartet 0			
11	10	09	08	07	06	05	04	03	02	01	00
0	0	0	0	λ	κ	φ	ι	η	γ	ϕ	ε

après

4) Rotation d'un quartet à droite

Il s'agit d'une extension du décalage précédent. Ici, le quartet 0 n'est pas perdu mais devient le nouveau quartet n-1. Il y a donc un décalage circulaire des bits, ce qui correspond bien à une rotation. Reprenons notre exemple : # E3h. Si on travaille sur un octet, la rotation le transformera en # 3Eh. Cependant, si on travaillait sur trois quartets, il aurait été transformé en # 30Eh (sur 3 quartets, on a effectué la rotation sur # 0E3h !). Cette rotation ne correspond à aucune opération mathématique simple.

quartet 2				quartet 1				quartet 0			
11	10	09	08	07	06	05	04	03	02	01	00
λ	κ	φ	ι	η	γ	ϕ	ε	δ	χ	β	α

avant

quartet 2				quartet 1				quartet 0			
11	10	09	08	07	06	05	04	03	02	01	00
δ	χ	β	α	λ	κ	φ	ι	η	γ	ϕ	ε

après

5) Décalage d'un quartet à gauche

Le décalage d'un quartet à gauche est similaire au décalage d'un bit à gauche, les entités manipulées n'étant plus des bits mais des quartets. Si on travaille sur n quartets, le quartets n-1 est perdu, le quartet n-2 devient le quartet n-1... Le quartet 0 devient le quartet 1 et le nouveau quartet 0 est passé à zéro. Notre exemple : # E3h devient donc après décalage # E30h.

Le décalage d'un quartet à gauche correspond à quatre décalages d'un bit à gauche. Par conséquent, décaler d'un quartet à gauche, c'est multiplier par seize.

quartet 2				quartet 1				quartet 0			
11	10	09	08	07	06	05	04	03	02	01	00
λ	κ	φ	ι	η	γ	ϕ	ε	δ	χ	β	α

avant

quartet 2				quartet 1				quartet 0			
11	10	09	08	07	06	05	04	03	02	01	00
η	γ	ϕ	ε	δ	χ	β	α	0	0	0	0

après

6) Rotation d'un quartet à gauche

La rotation d'un quartet à gauche s'effectue selon le même principe que celle d'un quartet à droite : dans le décalage précédent, le quartet n-1 n'est pas perdu mais devient le nouveau quartet 0. Il y a donc décalage circulaire des quartets, il s'agit bien d'une rotation. Si on travaille sur un octet, la rotation de # E3h le transforme en # 3Eh. Par contre, si on travaille sur trois quartets, on obtient # E30h, car on a effectué la rotation sur # 0E3h. Cette rotation ne correspond à aucune opération mathématique simple.

quartet 2				quartet 1				quartet 0			
11	10	09	08	07	06	05	04	03	02	01	00
λ	κ	φ	ι	η	γ	ϕ	ε	δ	χ	β	α

avant

quartet 2				quartet 1				quartet 0			
11	10	09	08	07	06	05	04	03	02	01	00
η	γ	ϕ	ε	δ	χ	β	α	λ	κ	φ	ι

après

Voilà tout pour les opérations de décalage sur les entiers. Il existe un dernier type de calcul sur les entiers : les opérations logiques, directement issues de l'arithmétique Booléenne.

V) Les opérations logiques

Il s'agit d'effectuer des opérations du type Not, And, Or, Xor soit respectivement en français non, et, ou, ou exclusif. Etudions tout d'abord les résultats de ces opérations sur un bit.

1) Les opérations logiques sur un bit

Afin de bien comprendre le fonctionnement de ces opérations, il faut connaître la convention suivante : DANS UN BIT, 0 REPRESENTE LA VALEUR "FAUX" ET 1 LA VALEUR "VRAI". A l'aide de celles-ci, nous pouvons comprendre les opérations logiques suivantes. Les opérations suivantes sont presque toutes accessibles directement lors de la programmation.

a) L'opération NOT

Il s'agit de l'inversion "non" : "non a" est vraie si "a" est fausse, "non a" est fausse si "a" est vrai. Il suffit donc d'inverser le bit tout simplement :

Not 0 = 1,

Not 1 = 0.

b) L'opération AND (et)

Il s'agit du "et" logique standard : "a et b" est vrai si et seulement si "a" est vrai et "b" est vrai. De par la convention, on obtient les résultats suivants :

0 And 0 = 0,

0 And 1 = 0,

1 And 0 = 0,

1 And 1 = 1.

Ce qui se représente par la table de vérité :

And	0	1
0	0	0
1	0	1

c) L'opération OR (ou)

Il s'agit ici du "ou" habituel : "a ou b" est vrai si "a" est vrai ou "b" est vrai. On a donc les résultats suivants :

0 Or 0 = 0,

0 Or 1 = 1,

1 Or 0 = 1,

1 Or 1 = 1.

Ce qui se représente par la table de vérité :

Or	0	1
0	0	1
1	1	1

d) L'opération XOR (ou exclusif)

Il s'agit d'une opération moins habituelle : le "ou exclusif". "a ou b exclusivement" est vrai si "a" est vrai ou "b" est vrai mais tout les deux ne sont pas vrai simultanément. Cela peut aussi se dire ("a" est vrai ou "b" est vrai) et "a et b" est faux. On obtient donc la formule : **a Xor b = (a Or b) - (a And b)**. Attention le moins n'est pas une opération logique mais il est très facile d'accès ; nous retiendrons par conséquent cette formule qui est la plus simple à programmer. On a donc :

0 Xor 0 = 0,

0 Xor 1 = 1,

1 Xor 0 = 1,

1 Xor 1 = 0.

Ce qui se représente par la table de vérité :

Xor	0	1
0	0	1
1	1	0

Nous venons de voir les opérations logiques sur un bit, nous pouvons maintenant aborder celles sur un ensemble de bit : un quartet, un octet, cinq quartets...

2) Les opérations logiques sur un entier

Il est très facile d'effectuer une opération logique sur un entier lorsqu'on connaît les opérations sur un bit. En effet, il suffit d'effectuer l'opération sur chacun des bits de l'entier. Par exemple, pour calculer # 12h And # 58h, on convertit chacun des nombres en binaire, et on effectue le "And" sur chacun des bits de ceux-ci, bit 0 avec bit 0, bit 1 avec bit 1... Détaillons ce calcul.

12h And # 58h = # 00010010b And # 01011000b.

Pour le bit 0, 0 And 0= 0 ;

Pour le bit 1, 1 And 0 = 0 ;

Pour le bit 2, 0 And 0 = 0 ;

Pour le bit 3, 0 And 1 = 0 ;

Pour le bit 4, 1 And 1 = 1 ;

Pour le bit 5, 0 And 0 = 0 ;

Pour le bit 6, 0 And 1 = 0 ;

Pour le bit 7, 0 And 0 = 0.

Le résultat de # 12h And # 58h est donc # 00010000b ou # 10h. Etudions maintenant "# 12h Or # 58h". Ici encore, on effectue l'opération sur chacun des bits. On calcule donc :

12h Or # 58h = # 00010010b Or # 01011000b,

12h Or # 58h = # 01011010b = # 5Ah.

De même # 12h Xor # 58h = # 01001010b = # 4Ah. On vérifie donc bien que # 12h Xor # 58h = (# 12h Or # 58h) - (# 12h And # 58h).

Calculons enfin "Not # 12h". Si nous travaillons sur deux quartets, # 12h = # 00010010b. "Not # 12h" est donc égal à # 11101101b ou # EDh. Si nous avions travaillé sur trois quartets, on aurait obtenu :

Not # 12h = Not # 000000010010b = # 111111101101b = # FEDh.

VI) La complémentation à deux

Nous incluons ici le dernier type d'opération élémentaire : la complémentation à deux. Il s'agit d'une opération très simple que l'on peut rapprocher du calcul de l'opposé. Il s'agit en effet de calculer le nombre b tel que a + b = # 0h.

Ainsi, par exemple, le complément à deux de # 12h, lorsqu'on travaille sur un octet est # 0h - # 12h = # EEh. Si on avait travaillé sur trois quartets, on aurait obtenu # FEEh.

Remarque : Il existe une relation simple entre la complémentation à 2 et le non logique. Si on note '-A' le complément à 2 de A, on a la relation : **Not A = -A-1**

VII) Conclusion

Au terme de ce chapitre, vous connaissez la majeur partie des opérations élémentaires qu'il est possible d'effectuer sur des entiers binaires ou hexadécimaux. Ce ne sont cependant pas les seules et l'on peut par exemple introduire la racine carrée...

Il s'agit cependant des seules opérations que vous pourrez effectuer "facilement" en assembleur.

Exercices :

A.4.1 : Tentez de transformer #1011b en #10110000b de deux manières différentes.

A.4.2 : Calculez # 41d And #1010b

A.4.3 : Résoudre # 61d Or # ..h = # 111101b

Deuxième partie

L'assembleur

QU'EST-CE QUE L'ASSEMBLEUR ET LA SYNTAXE DES ASSEMBLEURS

1) Généralités sur le langage machine et l'assembleur

Le langage machine est le langage le plus proche de la machine car il est directement exécuté par le microprocesseur. Par conséquent il est très rapide. Cependant il est constitué d'une suite de chiffres hexadécimaux, ce qui le rend illisible par les pauvres humains que nous sommes. Nous allons donc remplacer ces suites de chiffres par des mnémoniques, c'est à dire des petites séquences correspondant aux actions faites par le microprocesseur.

Un utilitaire, nommé assembleur va s'occuper de traduire ces mnémoniques en hexadécimal ou en objet Code (type d'objet de la HP48 contenant du langage machine).

L'opération consistant à traduire du langage machine en assembleur s'appelle l'assemblage. L'opération inverse est le désassemblage. On peut le faire à la main ou avec des programmes qui s'appellent respectivement "assembleur" et "désassembleur". Dans cet ouvrage nous ne verrons pas le codage des instructions, nous laisserons la fastidieuse tâche d'assemblage à ces utilitaires.

Remarque : Cet utilitaire se trouve sur la disquette d'accompagnement

Le système d'exploitation (le RPL dans le cas de la HP 48) ne peut pas contrôler les agissements d'un programme en langage-machine et ne peut donc pas l'arrêter en cas de défaillance.

De plus sur les HP 48, il n'y a pas de différence entre la mémoire vive (mémoire d'exécution des programmes) et la mémoire de stockage des variables : toute erreur dans un programme en langage machine peut provoquer la perte des données. Ainsi nous vous conseillons de protéger de l'écriture (en utilisant le loquet derrière chaque carte) vos cartes (non mergées) si vous en avez.

2) La syntaxe de l'assembleur fourni avec le livre

Avant de commencer, il est bon de savoir que LASM S/SX et G/GX ont exactement la même syntaxe, ainsi tous ce qui sera dit dans les paragraphes suivants conviendra aux deux assembleurs.

Ces assembleurs prennent une chaîne de caractères comme argument. Elle est constituée d'une suite d'instructions, de commentaires et de directives d'assemblage, séparés par des caractères de séparation (les retours à la ligne), et terminée par un retour à la ligne et un arobas (@ ; chr 64 : accessible par la séquence de touches 'α', '↱' et 'ENTER'). L'assembleur différencie les minuscules des majuscules. Les deux labels Boucle et BOUCLE sont donc différents (mais il n'est pas conseillé d'utiliser cette possibilité, car il sera plus compliqué de retrouver une éventuelle erreur).

Le caractère de séparation est le retour à la ligne (chr 10 : accessible par la séquence de touches '↱' et '.').

Certaines instructions nécessitent un ou plusieurs paramètres ; ces instructions sont séparées de leurs paramètres par un ou plusieurs espaces. Le nombre d'espace importe peu.

L'assembleur dispose d'instructions modifiant la manière dont la traduction des mnémoniques en langage-machine (nombres hexadécimaux) sera faite. Ces instructions commencent par un "**!**" et seront détaillées ultérieurement.

3) Les Macros

On a souvent besoin d'inclure des données dans un programme assembleur, on peut le faire, soit en tapant un $ (chr 36 : accessible par la séquence de touches 'α', '↰' et '4') suivi des codes hexadécimaux, soit en tapant "ɑ" (chr 162 : accessible par la séquence de touches 'α', '↱' et '4') suivi des codes ASCII. Mais ce n'est pas pratique, et cela augmente la taille des sources qui deviennent de plus en plus long à éditer.

La solution, ce sont les macros. Une macro, c'est un bloc de données quelconques qui seront incluses à l'assemblage dans le code assemblé. Le système d'exploitation de la HP 48 obligeant à stocker des objets en mémoire, il faut emmagasiner les données dans une structure d'objet quelconque. L'assembleur utilise l'objet chaîne de caractères. Cependant vous n'êtes pas obligé de stocker des chaînes de caractères, l'assembleur s'occupe de les transformer lui-même. Le programme "→**MACRO**" de la bibliothèque de programme vous explique comment l'assembleur réalise cette action.

Les macros doivent être stockées dans un sous répertoire du répertoire HOME nommé **Macro**. Respectez absolument le fait que dans le nom 'Macro', la première lettre soit une majuscule et les autres des minuscules.

La syntaxe d'inclusion d'une macro est la suivante

```
/Nom.de.macro
```

4) Comment entrer un programme

Un programme en assembleur s'écrit sur votre machine dans des chaînes de caractères accessibles par la séquence : ↱ et -.

Dans les programmes des chapitres suivants, il ne sera fait aucune allusion au fait que les programmes sont inclus dans des chaînes de caractères. Ce sera par conséquent à vous de les inclure dedans.

Remarque : Si vous voulez savoir comment faire pour récupérer les données de la disquette reportez-vous à l'annexe 5 traitant de ce sujet.

Pour assembler vos programmes, mettez la chaîne de caractères où se trouvent les sources de votre programme sur la pile et tapez LASM.

Attention : Un programme pour pouvoir être assemblé doit impérativement être terminé par un retour à la ligne puis un arobas (@).

VOTRE PREMIER PROGRAMME

Dans ce chapitre nous allons aborder des notions indispensables qu'il vous faudra absolument assimiler pour pouvoir aller plus loin. Pour cela nous vous conseillons de lire ce chapitre attentivement puis de le reprendre une seconde fois en portant une attention spéciale aux notions de registre et de champ. Si vous voulez mettre toute les chances de votre côté, nous vous conseillons de recopier l'exemple sur votre HP. Ensuite reprenez le programme et modifiez-le. Ce dernier exercice est de loin le meilleur pour mémoriser les instructions et leur comportement. Cette méthode est bien sûr valable pour tous les prochains chapitres et programmes qui suivent.

ATTENTION : -Sauvez vos données car vous risquez de faire des erreurs graves.

-N'oubliez pas qu'une carte protégée en écriture ne pourra en aucun cas être effacée de manière logicielle.

Voici donc votre premier programme en assembleur qui n'est qu'une simple boucle. Ne vous affolez pas, il est expliqué en détails dans les pages qui suivent.

```
%%% BOUCLE POUR L'INTRODUCTION %%%

LC 12345              % Le registre Ca contient la valeur #12345h

 *BCL.1               % Création d'un label
C=C-1 A               % Décrémentation de Ca

 ?C#0 A               % Si C est différent de 0 alors on va au
 GOYES BCL.1          %label "BCL.1"
```

```
A=DAT0 A                % Retour au RPL avec les trois prochaines
D0=D0+ 5                %instructions
PC=(A)
@                       % Arobas permettant à l'assembleur de savoir
                        %ou arrêter l'assemblage
```

programme 1.1

Commentons ligne par ligne le programme 1.1:

Les commentaires

```
%%% boucle pour l'introduction %%%
```

La première ligne est un commentaire : lors de l'assemblage cette ligne sera ignorée par l'assembleur. Vous pouvez et vous devez en mettre le plus possible dans vos sources car ils vous servent à vous repérer dans un programme. Prenons un exemple concret : vous faites un programme X. Vous décidez trois mois plus tard de faire des rajouts, de corriger des petits bugs. Sans l'aide des commentaires vous ne vous y retrouverez jamais. Ceci est vrai pour tous les langages et plus particulièrement pour l'assembleur qui nécessite un très grand nombre d'instructions pour réaliser une action. Pour conclure, si vous désirez inclure un commentaire vous devez mettre le signe "%" avant ce dernier, tout ce qui suit jusqu'à la fin de la ligne est ignoré.

Registres et chargement de constantes

```
LC 12345
```

A la deuxième ligne, l'instruction "**LC**" (**L**oad **C**) permet de charger une valeur dans le registre de calcul C. Ici on charge la

valeur hexadécimale # 12345h. Il faut savoir que la HP (plus précisément le microprocesseur SATURN) dispose de 4 registres de calcul (**A, B, C** et **D**), qui servent à réaliser des opérations comme des incrémentations, des décrémentations, des additions ou des soustractions. Ce sont des sortes de variables. "LC" affecte par conséquent la valeur # 12345h au registre de calcul C. Un registre de calcul peut être considéré comme une petite zone de mémoire de 16 quartets, interne au microprocesseur, qui permet de stocker n'importe quelle valeur entière.

Les labels

```
*BCL.1
```

A la troisième ligne la séquence "***BCL.1**" est un label. Les labels sont utilisés par l'assembleur pour savoir où continuer l'exécution du programme lorsqu'on lui en donne l'ordre. Un peu plus loin vous verrez l'instruction "**GOYES BCL.1**". Cette instruction signifie : si la condition précédente est vérifiée va au label "BCL.1" et exécute les instructions qui le suivent. Les labels sont introduits par le signe "*****" avant leur nom. Attention les labels ne doivent en aucun cas comporter d'espace. Cependant vous pouvez, comme vous avez pu le constater, utiliser des points dans un nom de label.

Décrémentations

```
C=C-1 A
```

A la quatrième ligne on a l'instruction "**C=C-1 A**". Le but de cette instruction est d'ôter 1 au registre de calcul C. Cependant le "**A**" après la séquence "**C=C-1**" doit sûrement vous poser problème.

Avant tout, il faut savoir que tous les registres de calcul ont une taille de 16 quartets. Cependant on n'est pas dans l'obligation de travailler avec tout le registre. On peut très bien ne travailler qu'avec 1, 2 ou 5 quartets par exemple.

On utilise pour cela des parties de registre appelées des champs. A la fin de l'explication du programme 1.1 nous verrons un tableau récapitulatif de tous les champs disponibles et une description de chacun d'entre eux.

On utilise les champs pour plusieurs raisons qui sont en particulier la vitesse d'exécution. L'utilisation d'un champ A, 5 quartets, sera plus rapide que pour un champ W, 16 quartets, mais aussi pour des raisons de calculs : on peut, par exemple, n'avoir besoin de décrémenter une valeur que sur 2 quartets et être dans l'obligation de ne pas toucher au reste du registre.

Pour des raisons de simplicité d'écriture on utilisera dorénavant la notation suivante : en majuscule le registre et en minuscule le champ. Ce qui donne par exemple Cb pour le registre C champ B. Dans la suite du livre pour définir un registre de champ quelconque on utilisera par exemple pour le registre C de champ A la notation Ca. On utilisera aussi l'abréviation q pour remplacer quartet(s).

Tests

```
?C#0 A
GOYES  BCL.1
```

A la cinquième et sixième lignes se trouvent deux instructions qui vont ensemble ; plus généralement sachez qu'une instruction débutant par un point d'interrogation est un test et est toujours suivie, après un retour à la ligne, d'un "**GOYES label**". Les instructions "**?C#0 A**" et "**GOYES BCL.1**" signifient : si le registre

de calcul C est différent (#) de 0 sur le champ A (5 quartets), alors va au label "**BCL.1**". Il s'agit donc d'un saut conditionnels.

Sortie d'un programme en langage machine

```
A=DAT0 A
D0=D0+ 5
PC=(A)
```

Les trois dernières instructions servent à sortir du programme. Ces instructions sont systématiquement employées pour sortir de tous les programmes en assembleur. Ne tentez pas de les comprendre tout de suite. Sous leur apparence simple elles cachent un mécanisme complexe. Elles vous seront expliquées dans quelques chapitres.

Fin d'un programme en assembleur

```
@
```

A l'extrême fin du programme vous pouvez remarquer qu'il y a un "@". Ce symbole est utilisé par l'assembleur pour savoir où se termine le code du programme. Ce symbole doit être le dernier du programme. En particulier, il ne doit être suivi ni d'un espace, ni d'un retour à la ligne.

En résumé le programme fait : tant que Ca (C champ A) est différent de 0 on décrémente Ca (C champ A). Quand Ca est égal à 0 on sort du programme et on revient dans le système courant de la HP 48.

Les champs

Voici le tableau des champs et leur description :

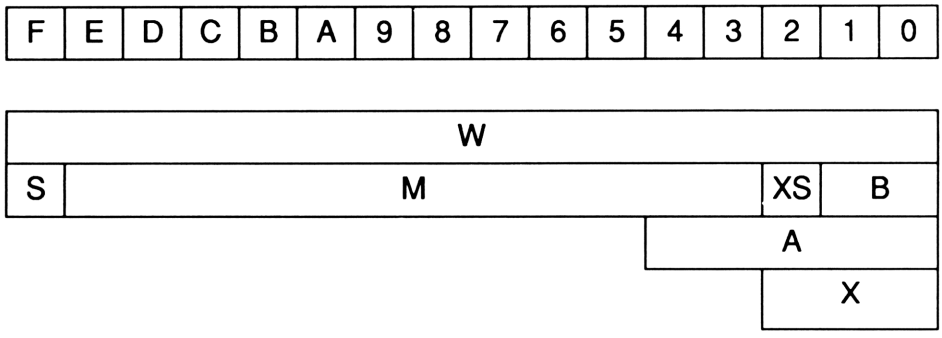


TABLEAU DES CHAMPS

Notez que les quartets sont numérotés de 0 à 15, le premier quartet (0) étant le plus à droite.

Commençons par le champs **W** comme **W**ide. Ce champ affecte tout le registre. On travaille donc sur le registre entier (64 bits).

Le registre **B** comme **B**yte (octet) n'affecte que les deux quartets de poids faible. Si on avait la valeur # A455C4945400000000h dans le registre C et qu'on place la valeur # 7Bh en utilisant le champ **B** on aura dans le registre C la valeur # A455C494540000007Bh

Le champ **A** comme **A**ddress correspond aux cinq premiers quartets du registre (quartets 4 à 0).

Le champ **X** correspond aux trois premiers quartets du registre (quartets 2 à 0).

Le champ **XS** correspond au troisième quartet du registre (quartet n°2), c'est à dire le quartet de poids fort du champ X du registre.

Le champ **M** correspond du quatrième au quinzième quartet (quartets 14 à 3).

Et enfin le champ **S** correspond au dernier quartet du registre (quartet n°15).

Attention, il existe deux autres champs. Les champs **P** et **WP**. Ces deux champs, étroitement liés, sont de longueur et de position modifiables par le programmeur. Nous les reverrons plus longuement lors de leur utilisation dans le chapitre 11.

Nous avons vu dans ce chapitre :

Les commentaires

Les labels

Les registres de calculs

Les champs

Une décrémentation d'un registre

Un test

Exercices :

B.6.1 : Combien de fois la boucle du programme 1.1 est-elle effectuée ?

B.6.2 : Modifiez le programme pour qu'il boucle 1000 fois.

LES BOUCLES

Les boucles sont très utilisées dans tous les langages, que ce soit en Pascal, en Basic ou en C. Les boucles, comme leur nom l'indique, servent à répéter plusieurs fois la ou les mêmes instructions. On peut créer une boucle pour faire une pause au milieu d'un programme ou, pour un jeu, par exemple, afin de répéter plusieurs fois les mêmes séquences. Voici donc d'autres programmes commentés qui vous dévoileront la plupart des instructions permettant de réaliser des boucles.

```
%%% boucle avec utilisation de la carry %%%


LC 01000              % Ca contient la valeur # 1000h

*BCL.1                % Création d'un label

C=C-1 A               % Décrémentation de Ca

GONC BCL.1            % Si la carry est à 0 alors retour au label
                      %BCL.1, si elle est à 1 on continue le
                      %programme

A=DAT0 A              % Retour au RPL avec les trois prochaines
D0=D0+ 5              %instructions
PC=(A)

@                     % Arobas permettant à l'assembleur de savoir
                      %ou arrêter l'assemblage
```

programme 2.1

La Retenue (CARRY)

Cette boucle est basée sur l'utilisation d'un registre drapeau nommé CARRY. En anglais "carry" signifie retenue. Il faut savoir que la carry a exactement le même rôle que la retenue de

l'addition, de la soustraction ou de la multiplication. Il n'y en a qu'une seule pour tous les registres. Sa valeur est mise à jour lors de toutes les opérations de calcul. Elle passe à 1 (on dit qu'elle est armée) si il y a une retenue (par exemple si on décrémente la valeur # 0h). Si il n'y a pas de retenue, elle est mise à 0 (on dit qu'elle est désarmée).

La retenue est armée lorsque les opérations d'addition, soustraction, incrémentation, décrémentation donnent lieu à des débordements (lors du calcul sur un octet #FFh+#1h donne # 0h : la Carry est armée. Si on avait travaillé sur trois quartets, on aurait obtenu # 100h. Il n'y aurait pas eu de débordement, la Carry aurait donc été désarmée).

Utilisation de la CARRY

Maintenant que vous avez compris le rôle de la carry vous ne devriez pas avoir de problème à comprendre l'instruction "**GONC**" du programme 2.1. Cette instruction signifie "**GO** if **N**o **C**arry" ce qui correspond, en français, à "va si il n'y a pas de retenue". Cette instruction veut donc dire que si la carry est à 0 alors on va au label "**BCL.1**", sinon on poursuit l'exécution du programme.

Attention : lorsqu'on utilise la carry, on boucle une fois de plus que la valeur initiale du compteur, c'est-à-dire que dans le programme 2.1 on va itérer # 01001h fois.

```
%%% boucle utilisant la carry %%%

LA 100                  % Ax contient la valeur # 100h

*BCL.1                  % Création d'un label
 A=A+3 X                % incrémentation de 3 de Ax
  GOC LBL.2             % Si la carry est à 1 on va au label "LBL.2"
                        %sinon on continue
GOTO BCL.1              % On va au label "BCL.1"

  *LBL.2                % Création d'un label
```

```
A=DAT0 A                % Retour au RPL avec les trois prochaines
D0=D0+ 5                %instructions
PC=(A)

@                       % Permet à l'assembleur de savoir
                        %ou arrêter l'assemblage
```

programme 2.2

Voici donc un second programme qui, comme le premier, est une boucle. Cependant, il est un peu plus complexe, tant par sa structure que par les nouvelles instructions qui sont présentées.

Le début du programme commence, comme tout les programmes de ce livre par un commentaire. Dorénavant nous ne porterons plus aucune attention aux commentaires.

Le vrai début du programme est donc l'instruction "**LA 100**" (**L**oad **A**). Elle doit certainement vous rappeler l'instruction "**LC**" vue dans les programmes précédents, en effet, cette instruction permet de charger une valeur dans le registre A.

Remarque : Pour une question de vitesse il est conseillé, pour mettre à 0 les registres, d'utiliser les instructions : A=0 (champ) ; B=0 (champ) ; C=0 (champ) ; D=0 (champ). Ainsi au lieu de faire un LA 00000 on fera un A=0 A. Le deuxième avantage de cette instruction est de pouvoir mettre à 0 les registres B et D alors qu'ils n'ont pas d'instruction de chargement.

Un petit peu plus loin dans le programme vous avez l'instruction "**A=A+3 X**", cette instruction sert à incrémenter le registre Ax (champ X de A) de trois.

Saut inconditionnel (GOTO ; GOLONG)

L'instruction "**GOC LBL.2**" (**GO**to if **C**arry) signifie : va au label si la carry est à 1. Ici, si la carry est à 1 alors on va au label "LBL.2",

sinon on continue et on va arriver à l'instruction "**GOTO BCL.1**" qui correspond à aller au label "BCL.1" sans aucune condition. On assimile à l'instruction "GOTO" l'instruction "GOLONG" qui a le même effet que "GOTO" mais que l'on utilise pour de longs sauts c'est à dire que si l'on a beaucoup d'instructions entre le label et le "GOTO" on mettra un "**GOLONG**" (**GO**to **LONG**). Lorsque vous programmerez, utilisez toujours des "GOTO" et si l'assembleur émet un message d'erreur comme "Saut trop long", remplacez le par un "GOLONG".

Pour achever l'explication du programme 2.2 voici son rôle : il stocke la valeur # 100h dans Ax pour commencer. Il incrémente de 3 Ax (champs X de A). Si la carry est mise à 1 on sort, sinon on continue d'incrémenter Ax (champs X de A) tant que la carry n'est pas à un.

Maintenant réfléchissons un peu : combien de fois la boucle va t-elle s'exécuter ? En d'autre termes : combien de fois exécuterons-nous GOTO BCL.1 ?

Premier passage : A vaut # 103h
Deuxième passage : A vaut # 106h
Troisième passage : A vaut # 109h
Quatrième passage : A vaut # 10Ch

.
.
.

Avant dernier : A vaut # FFDh
Dernier : A vaut # 000h

Nous avons donc exécuté # 4FFh fois l'instruction "GOTO BCL.1"

Pour terminer ce chapitre sur les boucles voici deux exemples supplémentaires récapitulant les notions vues :

```
%%% DOUBLE BOUCLE %%%

LA 01000              % Aa contient la valeur # 1000h
C=0 B                 % Cb contient la valeur # 0h

*DEBUT.PRG            % Création d'un label

C=C-1 B               % Décrémentation de 1 de Cb
 GONC DEBUT.PRG       % Si il n'y a pas de carry on retourne au
                      %label "DEBUT.PRG"
A=A-1 A               % Décrémentation de 1 de Ab
 ?A#0 A               % Si Aa est différent de 0 on retourne au
 GOYES DEBUT.PRG      %label "DEBUT.PRG"

A=DAT0 A              %  Retour  au  RPL  avec  les  trois  prochaines
D0=D0+ 5              %instructions
PC=(A)

@                     % Permet à l'assembleur de savoir
                      %où arrêter l'assemblage
```

programme 2.3

```
%%% Double boucle sur le même registre %%%

LC F000000000010000  % Ca contient la valeur # 1000h et Cs
                     %contient la valeur # Fh
*DEBUT.PRG           % Création d'un label "DEBUT.PRG"
C=C-1 A              % Décrémentation de 1 de Ca
 GONC DEBUT PRG      % Si la carry est à 1 on va au label
                     %"DEBUT.PRG"
C=C-1 S              % Décrémentation de 1 de Cs
 ?C#0 S              %  Si  Cs  est  différent  de  0  on  retourne  au
GOYES DEBUT.PRG      %label "DEBUT.PRG"

A=DAT0 A             % Retour au RPL avec les trois prochaines
D0=D0+ 5             %instructions
PC=(A)

@                    % Permet à l'assembleur de savoir
                     %ou arrêter l'assemblage
```

programme 2.4

Ce que nous avons vu dans ce chapitre

La retenue (CARRY)

Les sauts conditionnels

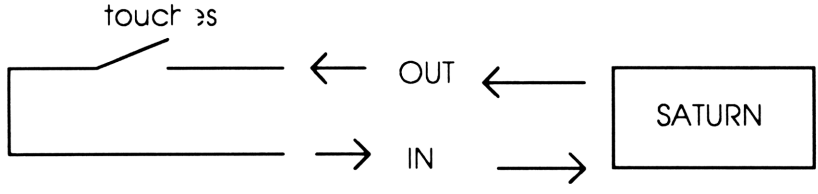
Les sauts inconditionnels

Exercice :

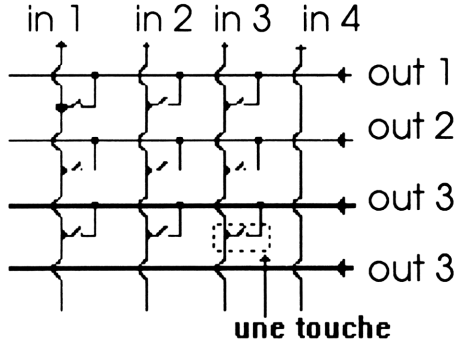
B.7.1 : Combien de fois boucle le programme 2.3 ? Le programme 2.4 ?

LES TESTS DE TOUCHES

Le titre semble assez explicite ; cependant qu'est-ce que tester une touche ? Physiquement on envoie du courant dans une des lignes de touches de la HP et on teste si le courant sort a la colonne voulue. C'est ainsi que pour envoyer du courant on met une valeur dans le masque de sortie (le masque de sortie représente la valeur que l'on envoie dans le clavier et qui est représenté par "**OUT**") et pour voir ce qui sort on regarde la valeur du masque d'entrée (représenté par "**IN**"). Une touche correspond grossièrement au circuit suivant:



Si l'on envoie du courant sur la ligne OUT et si l'on a pas appuyé sur une touche on n'obtiendra aucun courant au niveau du IN. Cependant si la touche est pressée le courant passera et on aura un signal au niveau du IN.



A	B	C	D	E	F
002 / 0010	100 / 0010	100 / 0008	100 / 0004	100 / 0002	100 / 0001
MTH	PRG	CST	VAR	↑	NXT
004 / 0010	080 / 0010	080 / 0008	080 / 0004	080 / 0002	080 / 0001
'	STO	EVAL	←	↓	→
001 / 0010	040 / 0010	040 / 0008	040 / 0004	040 / 0002	040 / 0001
SIN	COS	TAN	√x	y^x	1/x
008 / 0010	020 / 0010	020 / 0008	020 / 0004	020 / 0002	020 / 0001
ENTER		+/-	EEX	DEL	←
010 / 0010		010 / 0008	010 / 0004	010 / 0002	010 / 0001
α	7	8	9	÷	
008 / 0020	008 / 0008	008 / 0004	008 / 0002	008 / 0001	
↰	4	5	6	*	
004 / 0020	004 / 0008	004 / 0004	004 / 0002	004 / 0001	
↱	1	2	3	-	
002 / 0020	002 / 0008	002 / 0004	002 / 0002	002 / 0001	
ON	0	.	SPC	+	
/ 8000	001 / 0008	001 / 0004	001 / 0002	001 / 0001	

Couples OUT / IN des touches du clavier

Le tableau ci-contre doit vous paraître incompréhensible pour le moment. N'ayez pas peur, vous comprendrez très vite son utilité lorsque vous aurez étudié les quelques programmes qui suivent.

```
%%% programme de test de la touche DROP (←)

INTOFF

*PAS.SORTIR          % Création d'un label

LC 010               % Cx contient la valeur # 010h qui correspond
                     %au couple IN de la touche DROP du clavier
OUT=C                % On stocke dans le couple OUT la valeur de
                     %Cx
GOSBVL 01160         % GOSBVL 01160 remplace l'instruction C=IN

 ?CBIT=0 0           % Teste si le bit 0 de C est à 0. Si c'est le
 GOYES PAS.SORTIR    %cas on sait que la touche drop n'est pas
                     %appuyée en conséquence on va au label
                     %PAS.SORTIR

INTON

A=DAT0 A             % Retour au RPL avec les trois prochaines
D0=D0+ 5             %instructions
PC=(A)

@                    % Arobas permettant à l'assembleur de savoir
                     %ou arrêter l'assemblage
```

programme 3.1

Dans ce programme, de nouvelles instructions ont fait leur apparition. Nous ne les prendrons pas comme d'habitude dans leur ordre d'apparition mais nous les étudierons dans un ordre plus thématique.

```
INTOFF
INTON
```

Commentons tout d'abord les instructions "**INTOFF**" et "**INTON**". Ces deux instructions servent respectivement à interdire les interruptions et à les ré-autoriser. Vous devez sûrement vous

demander ce qu'est une interruption. Une interruption c'est comme la sonnerie d'un téléphone. On est en train de faire quelque chose le téléphone sonne et on va décrocher. On perd par conséquent du temps sur le travail que l'on accomplit. Quand ça sonne, la HP fait un "**GOSBVL 0000F**" (l'instruction "**GOSBVL**" est un appel à un sous programme situé à une adresse fixe. Elle sera expliquée plus tard.). A cette adresse se trouve une routine très complexe qui ralentira tout programme en langage machine. Par conséquent au début de programmes ayant besoin de tester des touches, on interdit les interruptions (car dès qu'on appuie sur une touche, les interruptions s'exécutent) et à la fin, on les ré-autorise, soit par un "**INTOFF**" puis un "**INTON**", soit de manière beaucoup plus forte (inhibition de toutes les touches, même "ON", ou bien "ON-C" ou encore "ON-A et F") par un "**GOSBVL 01115**" pour les interdire puis un "**GOSBVL 010E5**" pour les ré-autoriser.

Il est donc raisonnable de commencer par utiliser "**INTOFF**" et "**INTON**" et de les remplacer par les deux "**GOSBVL**" pré-cités lorsque votre programme sera parfaitement au point.

ATTENTION : N'oubliez jamais de ré-autoriser les interruptions en fin de programme.

GOSBVL 01160

Cette instruction correspond à un point d'entrée dans la mémoire morte de la HP 48. Il faut savoir que la HP, comme tous les ordinateurs, est programmée, à la base, en langage machine (seul langage compréhensible par le microprocesseur). On peut par conséquent utiliser des routines pré-programmées par les développeurs HP. On peut appeler ces routines de deux manières. La première est celle utilisée dans le programme 3.1, avec un "**GOSBVL**" suivi de l'adresse sur 5 quartets. La seconde est une autre instruction que l'on utilise de la même manière que le "**GOSBVL**" et qui est l'instruction "**GOVLNG**". La différence entre ces deux instructions est simple. L'instruction "**GOSBVL**" signifie qu'il y aura un retour au programme où a été effectué l'appel et l'instruction "**GOVLNG**" ne revient pas dans le programme d'où a

été lancé l'appel. Pour ne pas vous embêter à chercher des points d'entrée, nous avons trouvé pour vous les plus importants (voir annexe 2).

Revenons au problème initial, pourquoi a-t-on mis un "**GOSBVL 01160**" au lieu de mettre l'instruction qu'elle représente, c'est à dire "**C=IN**" ? Tout simplement car le microprocesseur SATURN est buggué : cette instruction ne fonctionne pas correctement lorsqu'elle est sur une adresse impaire.

Dans le programme 3.1 l'instruction de test "**?CBIT=0 0**", est utilisée pour tester si le bit 0 de C est à 0. Par exemple si on avait voulu tester si le bit 11 du registre A était à 1 on aurait utilisé l'instruction "**?ABIT=1 11**". Comme toujours cette instruction conditionnelle est suivie d'un saut (GOYES) qui sera effectué si le test est vérifié.

Maintenant il s'agit de comprendre comment on teste la touche DROP. Premièrement, à l'aide du tableau des couples, OUT / IN des touches du clavier, on détermine, (à la case correspondante à la touche DROP), le couple OUT / IN correspondant à la touche étudiée (←). Le premier est celui que l'on va mettre avant le "**OUT=C**". Le deuxième est celui que l'on va tester après le "**IN**". Pour cet exemple on va donc tester si le bit 0 du registre C est à 0, car si vous lisez le deuxième chiffre de la touche DROP, vous remarquez que ce nombre est le 0001. Or 0001 correspond dans un champ à mettre le bit 0 à 1.

En définitive dans cet exemple on va donc tester si la touche n'est pas appuyée. Si c'est le cas on retourne au label de départ "PAS.SORTIR". Si la touche est appuyée on sort du programme assembleur par les trois instructions qui doivent commencer à vous être familières après avoir re-autorisé les interruptions par l'instruction "INTON".

Voici un exemple, plus complexe dans sa structure, qui reste néanmoins facile à comprendre.

```
%%% tests de touches imbriqués

GOSBVL 01115          % Inhibe les interruptions
A=0 A                 % Mise à 0 de Aa
C=0 A                 % Mise à 0 de Ca

*TCH.1                % Création d'un label
LC 002                % Cx contient la valeur # 002h
GOSBVL 01EEC          % On appelle une routine faisant
                      %succesivement un OUT=C puis un C=IN
LA 0008               % Aa contient la valeur # 00008h
?C#A A                % Teste si Ca est différent de Aa
GOYES TCH.1           % Si le teste précédent est vérifié on va au
                      %label TCH.1
*TCH.5                % Création d'un label
LC 004                % Cx contient la valeur # 004h
OUT=C                 % On fait un OUT=C
GOSBVL 0115A          % On appelle la routine faisant un A=IN
LC 0004               % Ca contient la valeur # 00004h
?A#C A                % Teste si Aa est différent de Ca
GOYES TCH.5           % Si le test précédent est vérifié on va au
                      %label TCH.5

*TCH.9                % Création d'un label

LC 008                % Cx contient la valeur # 008h
OUT=C                 % On fait un OUT=C
GOSBVL 01160          % On appelle la routine faisant un C=IN
?CBIT=0               % On teste le deuxième bit de C
GOYES TCH.DROP        % Si le test précédent est vérifié on va au
                      %label FIN.PROG

GOTO TCH.9            % On va sans condition au label TCH.9

*TCH.DROP             % Création d'un label
LC 010                % Cx contient la valeur # 010h
OUT=C                 % On fait un OUT=C
GOSBVL 01160          % On appelle la routine faisant un C=IN
LA 00001              % Aa contient la valeur # 00001h
A=A&C A               % On fait un OU logique entre Aa et Ca. Aa
                      %contiendra le résultat
?A=0 A                % On teste si Aa est égal à 0
GOYES  TCH.DROP       % Si le teste précédent est vérifié on va au
                      %label TCH.DROP

*FIN.PROG             % Création d'un label
GOSBVL 010E5          % ré-autorise les interruptions
A=DAT0 A              % Retour au RPL avec les trois prochaines
D0=D0+ 5              %instructions
PC=(A)

@                     % Arobas permettant à l'assembleur de savoir
                      %où arrêter l'assemblage
```

programme 3.2

Ce nouveau programme ne fait pas grand chose, il utilise les différentes manières de tester des touches. Toutes les méthodes sont expliquées en commentaire.

Test de la touche ON

Dans le programme suivant, nous allons tester la touche "ON". Celle-ci diffère des autres par la façon dont on la teste. Le test de cette touche ne nécessite pas de masque de sortie. Il suffit juste de lire la valeur du registre IN et de tester si le seizième bit (n°15) est à un.

Voici donc un petit programme testant seulement la touche "ON"

```
%%% programme de teste de la touche "ON"

GOSBVL 01115          % Inhibe les interruptions

*DEBUT
GOSBVL 01160          % On fait un C=IN
?CBIT=0 15            % Test du seizième bit de C
GOYES DEBUT           % Si il n'est pas à 1 on va au label "DEBUT"

GOSBVL 010E5          % Re-autorise la totalité des interruptions

A=DAT0 A              % Retour au RPL avec les trois prochaines
D0=D0+ 5              %instructions
PC=(A)

@                     % Arobas permettant à l'assembleur de savoir
                      %où arrêter l'assemblage
```

programme 3.3

Remarque : Ici on utilise les GOSBVL 01115 et 010E5 au lieu de INTOFF et INTON car ces derniers ne permettent pas de tester la touche ON et par conséquent d'arrêter le programme en cours.

Tester simultanément plusieurs touches

Pour finir ce chapitre voici un programme supplémentaire testant trois touches qui doivent être pressées en même temps pour sortir du programme.

```
%%% teste des touches F, L, S simultanément

INTOFF

*DEBUT

LC 100              % Masque d'entré de la touche F
OUT=C
GOSBVL 01160        % On fait un "C=IN"
?CBIT=0 0           % On teste si le bit 0 est à 0
GOYES DEBUT         % Si le bit 0 est à 0 on va au label "DEBUT"

LC 080              % Masque d'entrèe de la touche L
OUT=C
GOSBVL 01160        % On fait un "C=IN"
?CBIT=0 0           % On teste si le bit 0 est à 0
GOYES DEBUT         % Si le bit 0 est à 0 on va au label "DEBUT"


LC 008              % Masque d'entréé de la touche S
OUT=C
GOSBVL 01160        % On fait un "C=IN"
?CBIT=0 4           % On teste si le bit 4 est à 0
GOYES DEBUT         % Si le bit 0 est à 0 on va au label "DEBUT"

INTON

A=DAT0 A            % Retour au RPL avec les trois prochaines
D0=D0+ 5            %instructions
PC=(A)

@                   % Arobas permettant à l'assembleur de savoir
                    %où arrêter l'assemblage
```

programme 3.4

Récapitulatif des notions vues dans ce chapitre

Le test des touches

Le test de la touche "ON"

Les appels absolus

Exercice :

B.8.1 : Ecrire un programme qui boucle tant que ENTER ou ON n'est pas appuyé.

LES POINTEURS ET L'UTILISATION DE LA PILE

Pour comprendre plus facilement ce qu'est un pointeur, on peut considérer que la mémoire de la HP (mémoire morte, ROM et mémoire vive, RAM) est comme une longue rue avec de nombreuses maisons. Chaque maison a une boîte aux lettres avec à l'intérieur un quartet. Un pointeur peut être assimilé à un facteur qui se placerait devant la boîte aux lettres qu'on lui aurait indiqué par son adresse.

A partir du moment où le pointeur est à l'endroit qu'on lui a indiqué, on peut lire ce qu'il y a dans la mémoire et recopier son contenu, dans le registre de calcul A ou C ou, au contraire, déposer le contenu du registre A ou C dans cette "boite aux lettres". Le microprocesseur SATURN dispose de deux pointeurs **D0** et **D1**. Pour la lecture ou l'écriture de quartets, on dispose d'une série d'instructions où **DAT0** représente la "maison" pointée par D0 et **DAT1** représente la "maison" pointée par D1.

Remarque : pour un champ contenant plusieurs quartets (par exemple W ou A qui contiennent respectivement 16 et 5 quartets), le 1er quartet (quartet n°0) est déposé dans la première boite au lettre, le deuxième dans la suivante (en D0+1 ou D1+1) et ainsi de suite.

Sauvegarde et restauration des registres RPL

Avant de commencer un programme il faut savoir que les deux pointeurs **D0** et **D1**, ainsi que les registres de calcul **Ba** et **Da** sont utilisés par le système. Il faut par conséquent sauvegarder

leurs valeurs avant de commencer à les utiliser dans un programme et récupérer leurs valeurs avant de sortir (à moins de vouloir modifier leurs valeurs pour agir sur le système). Pour sauvegarder et restaurer les valeurs des pointeurs et des registres on utilise respectivement le "**GOSBVL 0679B**" et le "**GOSBVL 067D2**".

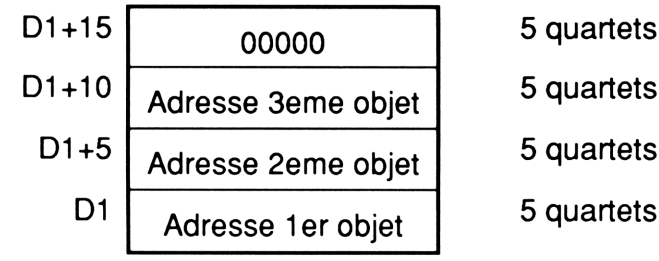
Nous avons vu que les deux pointeurs et que les registres Ba et Da étaient utilisés par le système. Mais quelle utilisation en fait-il ?

Explication de la sortie d'un programme

D0, est utilisé par la HP 48 pour pointer sur le prochain objet RPL à exécuter. C'est pour cette raison qu'on finit toujours un programme en langage machine par les instructions "**A=DAT0 A**", "**D0=D0+ 5**", "**PC=(A)**". Cette séquence lit l'adresse du prochain objet à exécuter ("**A=DAT0 A**"). On déplace le pointeur sur l'objet suivant ("**D0=D0+ 5**") et enfin on fait un "**PC=(A)**". Cette dernière instruction lit 5 quartets (une adresse) à l'adresse Aa et continue l'exécution du programme à l'adresse ainsi lue (adresse d'un programme en assembleur capable de traiter l'objet dont le contenu suit).

Relation entre la pile et le pointeur D1

La pile ne contient pas les objets (de tailles quelconques), mais leurs adresses (de tailles fixes : 5 quartets). D1 est utilisé par la HP pour pointer sur la pile. Il pointe sur le premier niveau. Lorsqu'on avance de 5 quartets (soit l'instruction **D1=D1+ 5**) on monte d'un niveau de pile. Lire 5 quartets à l'adresse pointée par D1 (par exemple l'instruction "**C=DAT1 A**") renvoie l'adresse de l'objet sur la pile. Voici un schéma représentant 4 niveaux de pile et avec 3 objets sur les 3 premiers niveaux. Remarquez que la pile se termine par la séquence 00000.



Relation entre le système et le registre Ba

Ba pointe sur ce qu'on appelle la pile des retours du langage RPL (cases de 5 quartets).

Relation entre le système et le registre Da

Da contient la mémoire libre en blocs de 5 quartets.

Voici donc un programme qui réalise l'équivalent d'un "SWAP". Attention on ne teste pas le nombre d'objets sur la pile, son utilisation sur une pile disposant de moins de deux objets est par conséquent dangereuse.

```
%%%  Equivalent d'un SWAP
C=DAT1 A              % On met l'adresse au niveau 1 de la pile
                      %dans Ca
D1=D1+ 5              % On monte d'un niveau de pile

A=DAT1 A              % On met l'adresse au niveau 2 de la pile
                      %dans Aa
DAT1=C A              % On met au niveau 2 de la pile l'adresse de
                      %l'objet qui était au niveau 1

D1=D1- 5              % On descend d'un niveau de pile
DAT1=A A              % On met au niveau 1 de la pile l'adresse de
                      %l'objet qui était au niveau 2
```

```
A=DAT0 A                % Retour au RPL avec les trois prochaines
D0=D0+ 5                %instructions
PC=(A)
@                       % Arobas permettant à l'assembleur de savoir
                        %où arrêter l'assemblage.
```

programme 4.1

Ce dernier programme ne nécessite pas de sauvegarde ni de restauration des registres D0, D1, B, D car en définitive on n'a pas changé la valeur de ces derniers registres.

Le principe de ce programme est simple. Il met dans un premier temps l'adresse de l'objet au premier niveau de la pile dans Ca. Ensuite on monte d'un niveau de pile, on met l'adresse de l'objet dans Aa et on écrit l'adresse contenue dans Ca à la place de l'ancienne. Pour finir on redescend d'un niveau, on place la valeur de Aa à la place de l'ancienne et on sort du programme

Voici un deuxième programme qui réalise l'équivalent d'un "DROP". Attention dans cet exemple on va tester si il y a un objet sur la pile.

```
%%% DROP AVEC TEST

C=DAT1 A                % Ca contient l'adresse du premier objet sur
                        %la pile
?C=0 A                  % Si Ca est égal à 0 on va au label "FIN"
GOYES FIN               %car la fin de la pile est marquée par une
                        %adresse nulle

D1=D1+ 5                % On monte d'un niveau de pile
D=D+1  A                % On libère une case mémoire de 5 quartets

*FIN                    % Création d'un label

A=DAT0 A                %  Retour au RPL avec les trois prochaines
D0=D0+ 5                %instructions
PC=(A)

@                       % Arobas permettant à l'assembleur de savoir
                        %où arrêter l'assemblage
```

programme 4.2

Dans ce programme on ne sauvegarde pas les registres car on modifie volontairement leurs valeurs pour réaliser le DROP.

Le programme suivant transforme les minuscules en majuscules dans une chaîne de caractères placée au premier niveau de la pile. Si il n'y a pas de chaîne de caractère sur le premier niveau de la pile le programme sort simplement sans créer d'erreur.

Remarque : la structure d'une chaîne est la suivante. Un prologue est le code

Prologue	(5 quartets) :	02A2C
Taille hors prologue	(5 quartets) :	taille
Contenu	(suite d'octet)	

```
%%% TRANSFORMATION DE MINUSCULES EN MAJUSCULES


INTOFF                  % Interdiction des interruptions

GOSBVL 0679B            % Sauvegarde des registres

C=DAT1 A                % Lecture de l'adresse de l'objet au niveau 1
                        %de la pile
D1=C                    % D1 pointe sur l'objet au niveau 1 de la
                        %pile
A=DAT1 A                % Aa contient le prologue de l'objet au
                        %niveau 1 de la pile
LC 02A2C                % Ca contient la valeur # 02A2Ch
?C=A  A                 % On teste si Aa est égal à Ca
GOYES BON.OBJET         % Si c'est le cas on va au label "BON.OBJET"

*FIN                    % Création d'un label

INTON                   % ré autorisation des interruptions

GOSBVL 067D2            % récupération de la valeur de départ des
                        %registres

A=DAT0 A                % Retour au RPL avec les trois prochaines
D0=D0+ 5                %instructions
PC=(A)
```

```
*BON.OBJET              % Création d'un label

D1=D1+ 5                % On passe le prologue
C=DAT1 A                % On lit la taille de l'objet en quartet
                        %moins 5 quartets
C=C-5  A                % On retire la taille du prologue soit 5
                        %quartets
CSRB   A                % On divise par 2 Ca ce qui permet de
                        %connaître le nombre de caractères de la
                        %chaîne
D=C    A                % Da contient le nombre de caractères de la
                        %chaîne
D=D-1  A                % On décrémente Da car on va utiliser une
                        %boucle avec la carry
GONC PAS.CHAINE.VIDE    % Si la carry est à 1 on va au label
                        %"PAS.CHAINE VIDE"
GOTO FIN

 *PAS.CHAINE.VIDE       % Création d'un label

D1=D1+ 5                % D1 pointe 5 quartets plus loin soit sur le
                        %premier caractère

*BCL.PRIN               % Création d'un label

A=DAT1 B                % Ab contient le numéro du caractère pointés
                        %par D1
LC 61                   % Cb contient la valeur # 61h
?C<=A B                 % On teste si le numéro du caractère est plus
                        %grand que # 61h
GOYES MIN.TST.1.OK      % Si c'est le cas on va au label
                        %"MIN.TST.1.OK"
GOLONG CHR.SUIVANT      % On va sans condition au label "CHR.SUIVANT"

*MIN.TST.1.OK           % Création d'un label

LC 7A                   % Cb contient la valeur # 7Ah
?C>=A B                 % Teste si Cb est supérieur ou égal Ab
GOYES MIN.TST.2.OK      % Si c'est le cas on va au label
                        %"MIN.TST.2.OK"

GOLONG CHR.SUIVANT      % On va sans condition au label "CHR.SUIVANT"

*MIN.TST.2.OK           % Création d'un label

%%% Création des majuscules

A=A-16 B                % On décrémente Ab de 16
A=A-16 B                % On décrémente à nouveau Ab de 16
DAT1=A B                % On stocke la majuscule à la place de la
                        %minuscule

 *CHR.SUIVANT           % Création d'un label

D1=D1+ 2                % On avance d'un caractère

D=D-1 A                 % On décrémente Da
```

```
  GONC BCL.PRIN        % Si la carry est à un on va au label
                       %"BCL.PRIN"


INTON                  % Ré-autorisation des interruptions

GOSBVL 067D2           % Restauration des registres B, D, D0, D1

A=DAT0 A               % Retour au RPL avec les trois prochaines
D0=D0+ 5               %instructions
PC=(A)

@
```

programme 4.3

Ne soyez pas affolé par ce programme sous prétexte qu'il est long. Vous êtes enfin devant un programme en assembleur de taille respectable. Il commence par pointer sur l'adresse de l'objet au niveau 1. On regarde si c'est bien une chaîne de caractères en testant son prologue. Si celui-ci est correct on prend la taille de la chaîne et on calcule sa longueur en caractères. Ainsi on va effectuer une boucle qui va lire la valeur d'un caractère et va soustraire 32 à sa valeur si c'est une minuscule car les majuscules et les minuscules sont séparées par 32 caractères, "A" a pour code 65 et "a" : 97).

Ce que nous avons vu dans ce chapitre

Les registres Ba, Da

Les pointeurs D0, D1

Structure de la pile système

Exercice :

B.9.1 : Ecrire un programme réalisant la fonction RPL CLEAR.

LE GRAPHISME

Le graphisme est un terme qui a l'inconvénient de regrouper une foule de notions différentes.

1) L'affichage d'un écran

Il est essentiel de faire la différence entre ce qui est visible sur l'écran de la HP et les données graphiques contenues dans la mémoire de la machine. Les données graphiques sont une suite de codes hexadécimaux stockés en mémoire. Pour faciliter la compréhension des affichages d'écrans entiers il faut savoir (au moins superficiellement) comment fonctionne l'affichage à partir des données contenues en mémoire. Cet affichage dépend des valeurs qui ont été stockées dans les zones d'échanges de données avec les périphériques d'affichage.

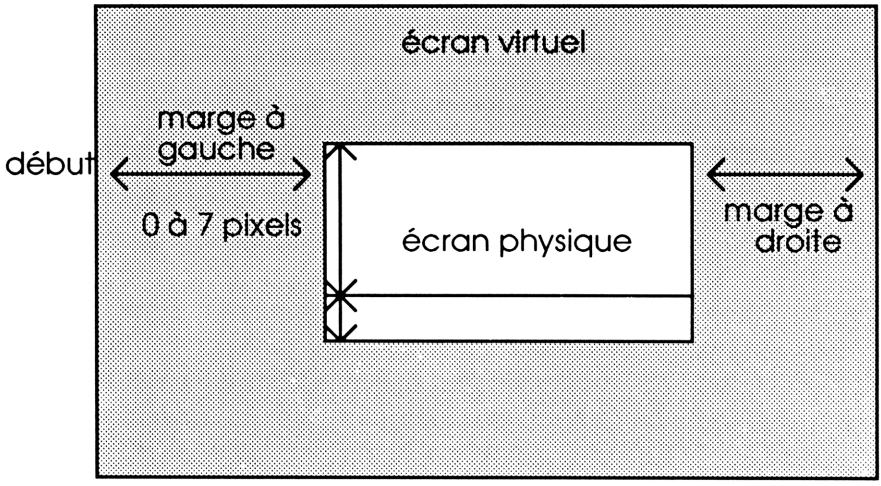
Comme un écran de télévision, l'écran de la HP est rafraîchie ligne par ligne. La première ligne de l'écran est la ligne 0 et la dernière est la ligne 63 (le numéro de la ligne en cours de rafraîchissement est lisible en #00128h sur 6 bits). Pour réaliser un affichage complet de l'écran, les coprossesseurs d'affichage agissent comme suit :

Ils initialisent un pointeur (non accessible au programmeur) à l'adresse de début de l'écran graphique (qui leur est donnée en #00120h).

On retrouve ce pointeur devant une série de bits. Les coprossesseurs vont en passer un certain nombre (nombre lisible

sur les 3 premiers bits en #00100h et appelé marge à gauche). Puis affichent les 131 bits suivants.

Ce pointeur avance de 34 quartets si la marge à gauche est inférieure à 4, et 35 quartets sinon. Il avance ensuite du nombre de quartets donné en #00125h appelé marge à droite (que l'on ne peut qu'écrire). Ce pointeur est initialisé (si l'on a pas encore affiché 64 lignes) à la valeur donnée en #00130h (Adresse des menus).



Quelques précisions

L'adresse donnée en #00120h (qui n'est pas lisible) doit être paire, en cas d'adresse impaire, le bit 0 est ignoré, ainsi, si on y met l'adresse #80021h, les coprossesseurs liront #80020h. Il faudra alors rétablir l'affichage grâce aux marges droite et gauche.

La valeur écrite en #00125h (ne peut être lue) peut être négative (#FFFh représente -1) mais la somme de sa valeur et de la marge à gauche divisée par 4 doit être paire (par exemple #FFFh et #Ch), ce qui veut dire qu'une ligne d'un objet graphique est de taille paire (ceci vient en grande partie du fait que les copossesseurs d'affichage travaillent en 8 bits).

Si la hauteur menu est mise à # 3Fh, il n'y aura pas d'affichage des menus. La nouvelle hauteur de menu stockée durant l'affichage n'est prise en compte que si l'on a pas encore passé la ligne en question (c'est à dire que l'on est pas en train d'afficher les menus). Il est impossible d'afficher les menus à partir de la ligne 0.

De même que l'adresse de début de l'écran, l'adresse de début des menus doit être paire. Une adresse impaire ne peut être rectifiée car il n'y a pas de marge gauche et droite pour les menus.

Etudions à présent comment afficher un écran.

Supposons que nous disposions d'un écran de x*y pixels débutant à l'adresse : @écran. Pour afficher cet écran à partir du pixel { X Y } il va nous falloir procéder en différentes étapes :

Tout d'abord, on calcule le nombre de quartets d'une ligne du grob (objet qui contient le graphique à afficher) : L = (x+3)/4 (+1 si (x+3)/4 est impair, par exemple à l'aide du programme suivant :

```
% R0a contient @écran, c'est à dire l'adresse du premier quartet
%contenant les données graphiques.
% R1a contient x
% R2a contient y
% R3a contient X
% R4a contient Y

%% Calcul de L
C=R1 A                  % x
C=C+3 A                 % x+3
CSRB A
CSRB A                  % (x+3)/4
C=C+1 A
CBIT=0 0                % On arrondit à la valeur paire par excès, ce
                        %qui correspond bien à ce que l'on veut
```

Ensuite, on calcule la marge à gauche, stockée en # 00100h. Il faut y mettre 8+X (X correspondant au nombre de pixels à

passer) plus 4 si l'adresse de l'écran est impaire, par exemple à l'aide du programme suivant :

```
% Les registres de sauvegarde contiennent les mêmes valeurs que
%précédemment

%% Calcul de la marge à gauche
A=R0 P                  % dernier quartet de @Ecran pour en tester la
                        % parité
C=R3 A                  % X
?ABIT=0 0               % Si @Ecran est impaire, on rajoute 4
GOYES APRES
C=C+4 P

*APRES
CBIT=1 3                % Revient à additionner 8, et à garder le
                        %résultat sur un quartet
D0= 00100
DAT0=C 1                % Attention, la marge à gauche est stockée
                        %sur UN quartet
```

Puis, il nous faut calculer l'adresse du premier octet de l'écran effectivement visible, stockée en # 00120h. Cette valeur est A = L*Y+X/4+@écran, calculée par exemple à l'aide du programme suivant

```
% Les registres de sauvegarde contiennent toujours les mêmes
%valeurs, Ca contient L

%% Calcul de la valeur en # 00120h
A=R4 A
B=0 A
*BOUCLE                 %Calcul de L*Y à l'aide d'une simple boucle
?A=0 A                  %Le résultat est stocké dans Ba
GOYES FINI
B=B+C A
A=A-1 A
GOTO BOUCLE
*FIN
C=R3 A                  %Calcul de X/4
CSRB A
CSRB A
C=C+B A                 %L*Y+X/4
A=R0 A
C=C+A A                 %L*Y+X/4+@Ecran
D0= 00120
DAT0=C A                %On écrit l'adresse de l'écran en #120h
```

Enfin, il nous faut calculer la marge à droite, qu'il faut ensuite écrire en # 00125h sur trois quartets. La valeur en est L-34 (-1 si @écran est impair), que l'on peut calculer à l'aide de ceci :

```
% Les registres de sauvegarde contiennent toujours les mêmes
%valeurs, Ca contient L, Aa contient @écran
C=C-16 A              % L-34
C=C-16 A              %
C=C-2  A              %
?ABIT=0 0             % On teste la parité de @écran
GOYES APRES
C=C-1 A
*APRES
D0= 00125             % On écrit la marge à droite en #125h
DAT0=C X
```

Ceci peut paraître complexe, et ça l'est. Toutefois, dans le cas général, on se débrouille pour avoir un écran en adresse paire et d'une taille standard. Ce qui donne pour valeurs en #00120h l'adresse de l'écran, en #00100h la valeur #8h (marge gauche nulle) et en #00125h la valeur #000h (marge droite nulle).

Maintenant que nous savons comment afficher une zone mémoire sur l'écran LCD. Il nous reste à la trouver. Ici trois options s'offrent à nous : créer notre propre zone mémoire, en utiliser une déjà créée (options à utiliser lorsque l'on désire ensuite travailler dans ces écrans) ou afficher directement l'écran contenu dans le programme (inclus en Macro dans l'objet code par exemple).

En réalité, on ne crée pas une zone mémoire, on en réserve une que l'on affiche ensuite. Malheureusement, cette zone peut être aussi bien en adresse paire qu'impaire. Pour palier à cet ennui, on réserve un quartet de plus que nécessaire, ainsi, si la zone est allouée en adresse paire, on ne fait rien, et si elle est allouée en adresse impaire, on commence un quartet plus loin et on dispose donc d'une zone de mémoire en adresse paire. De plus, comme la HP n'aime pas du tout voir "traîner" un objet sans prologue, au lieu de réserver juste une zone de mémoire, on réserve, par exemple, une chaîne de caractères. Ainsi pour réserver un écran "standard" (131*64) on procédera comme suit :

```
LC 00881                % Un écran 131*64 : # 880h quartets
GOSBVL 05B7D            % Réservation d'une chaîne de Ca caractères
ADOEX                   % Echange D0 et A
A=A+1 A                 % On rend Aa paire (supérieur ou égale à son
ABIT=0 0                %ancienne valeur)
```

Ce qui a pour effet de réserver un écran de #880h quartets dont l'adresse paire est dans Aa.

On peut utiliser une zone de mémoire déjà réservée comme l'écran de pile par exemple (toutefois, l'écran de pile ne fait pas 131*64 mais 131*56. On peut agrandir cet écran à la taille de 131*64 grâce à l'external # 4E2CFh SYSEVAL). Le PICT aussi peut être utilisé ; son avantage est de pouvoir être mis à la taille que l'on veut grâce à l'instruction RPL PDIM. Ces deux zones de mémoire sont évidemment toujours en adresse paire (la HP se charge de cela).

Pour afficher un grob contenu dans l'objet code, il faut donc écrire les bonnes valeurs en #00120h, #00100h et #00125h.

Dans les deux premiers cas, nous n'avons fait que préparer une zone mémoire où afficher notre graphique, il nous faut maintenant copier notre graphique dans la zone de mémoire dont nous voyons l'image à l'écran. Ceci peut paraître stupide mais une image graphique est faite pour être modifiée (dans la plupart des cas), par exemple cette image peut être un fond d'écran sur lequel on fera se déplacer des petits vaisseaux. **Un affichage d'un écran n'est rien d'autre qu'une recopie des données** symbolisant le graphique contenu dans votre code dans la zone que nous avons réservé. Cette recopie se fera comme suit.

```
GOSUBL APRES_GROB       % On fait un saut après l'image
                        % graphique. Analysons ce que fait un GOSUB:
                        % il met dans la pile de retour l'adresse de
                        % l'instruction qui le suit puis saute au
                        % label.
/Notre.graphique        % Macro qui contient le graphique
*.APRES_GROB
C=RSTK                  % puis on récupère dans Ca l'adresse qui a
```

```
                    % été stockée dans la pile de retours, c'est
                    % à dire l'adresse de début des donnes
                    % graphique.
D1=C                % D1 pointe sur le début du graphique.
A=R0 A              % Je suppose que j'ai stocké dans R0a
                    % l'adresse de la zone que j'ai réservée.
D0=A                % D0 pointe au début de la zone réservée.
LC 3F               % je dois recopier 64 lignes (mais
                    % j'initialise mon compteur a #3Fh car
                    % j'utilise une boucle en GONC.
*Boucle
  A=DAT1 16         % Recopie des 64 premier pixels de la ligne
  DAT0=A 16
  D0=D0+ 16         % passage des 64 pixels déjà traités
  D1=D1+ 16
  A=DAT1 16         % Recopie des 64 second pixels de la ligne
  DAT0=A 16
  D0=D0+ 16         % passage aux 64 pixels déjà traités
  D1=D1+ 16
  A=DAT1 B          % Recopie des 8 dernièrs pixels de la ligne
  DAT0=A B
  D1=D1+ 2          % passage a la ligne suivante
  D0=D0+ 2
C=C-1 B
GONC Boucle
```

Les "sprites"

Un "sprite" n'est rien d'autre qu'un graphique de petite taille et peut être assimilé à une feuille de papier (opaque ou calque) que l'on pose sur notre bureau (le bureau étant évidemment l'écran). Il suffit, pour afficher un sprite à l'écran, de recopier les données le constituant dans le graphique de l'écran (voir la routine d'affichage graphique donnée dans le programme **ASM REPL**).

Remarque : On appelle affichage HARD un affichage qui utilise l'adresse en # 00120h. Ainsi l'affichage SOFT est une simple copie de mémoire.

Ce que nous avons vu dans ce chapitre

L'affichage HARD

L'utilisation de la marge droite

L'utilisation de la marge gauche

Changement de la hauteur des menus

L'affichage graphique dans les menus

L'affichage SOFT

Exercice :

B.10.1 : Ecrire un programme qui noirci l'écran.

RECAPITULATIF 1
LES REGISTRES

Dans les chapitres précédents, vous avez abordé les notions de registre et de champ par l'exemple. Il est maintenant temps d'étudier en profondeur ces notions, et de voir les instructions du microprocesseur SATURN les concernant.

I) Les registres de calcul

Ce sont les registres permettant les calculs courants en assembleur. Ils se nomment **A**, **B**, **C**, **D** et, comme nous l'avons vu précédemment, il sont divisés en champs :

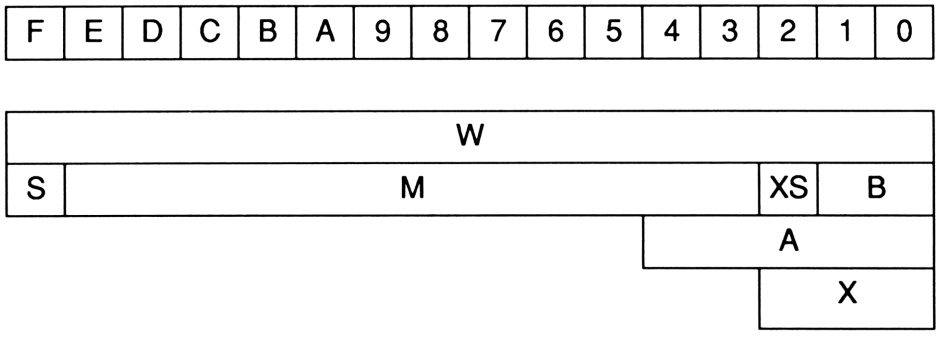


TABLEAU DES CHAMPS

Vous avez pu le comprendre dans les exemples précédents, les champs permettent de découper les registres de calculs en différents sous-registres. Dans le chapitre "Votre premier

programme", vous avez vu les champs **A**, **B**, **M**, **S**, **W**, **X**, **XS**, vous avez aussi appris l'existence des champs **P** et **WP**. Il est maintenant temps de vous expliquer l'emploi de ces deux derniers.

1) Le champ **P**

Il s'agit d'un champ de 1 quartet personnalisable. Il s'agit en effet du seul champ pouvant être déplacé à votre guise dans le registre. Etudions cela par l'exemple.

Supposons que vous voulez lire un quartet à l'adresse # 80000h, et que vous voulez que ce résultat soit stocké dans le quartet 5 de C, comment faire ? La solution réside dans l'emploi du champ **P**, à l'aide du morceau de programme suivant :

```
D1= 80000
P= 5
C=DAT1 P
```

Explicitons les opérations effectuées : tout d'abord, on fait pointer D1 sur 80000 grâce à l'instruction **D1= 80000** . Ensuite, on charge la valeur 5 dans P. Cela définit alors le champ P comme étant le quartet numéro cinq dans n'importe quel registre de calcul. L'instruction suivante, **C=DAT1 P** fonctionne comme C=DAT1 avec n'importe quel champ. Elle remplit le champ avec la valeur lue à l'adresse pointée par D1. Puisque le champ P à une longueur de un quartet, on a donc bien le résultat voulu, le quartet 5 de C contient un quartet lu à l'adresse # 80000h.

Il s'agit du premier emploi du champ P. Ce n'est cependant pas le seul. Les instructions **LA** et **LC** sont en effet conditionnées par le champ P. Dans les exemples précédents, vous avez vu, par exemple que LC 123 stocke # 123h dans les quartets 0 à 2 du registre C. C'est parce qu'à l'entrée d'un programme assembleur, P vaut 0. Si on avait fait :

```
P= 2
LC 123
```

On aurait alors chargé 123 dans les quartets 2,3 et 4 de C. **LC** charge donc C à partir du quartet dont le numéro est contenu dans P. De même, **LA** charge à partir du quartet P et ce de façon cyclique dans le registre.

Voici des exemple concrets qui vont vous permettre de mieux comprendre comment les chargements de valeurs sont conditionnés par le champs P. Nous affecterons dans tous nos exemple la valeur #12345h dans Ca, avec Cw qui était initialement nul :

Avec P=0 :

Cw : 0000000000012345

Avec P=1

Cw : 0000000000123450

Avec P=2

Cw : 0000000001234500

Avec P=15

Cw : 5000000000001234

Remarque importante : Nous l'avons vu précédemment, à l'entrée d'un programme assembleur, P vaut 0. A la sortie, il faut impérativement rétablir cette valeur à l'aide de l'instruction **P= 0**.

2) Le champ **WP**

Il s'agit d'un autre champ personnalisable. Celui-ci est à position fixe, mais de longueur variable. Le champ **WP** contient les quartets P à 0 du registre. Ainsi, par exemple pour lire 6 quartets en

80000h et les stocker dans les quartets 0 à 5 de C, on utilisera une routine du type :

```
D1= 80000
P= 5
C=DAT1 WP
```

Simple, non ? **C=DAT1 WP** charge 6 quartets (de P=5 à 0) dans le registre C.

Voilà pour l'emploi des champs P et WP. Vous devez certainement vous demander quelle peut en être l'utilité. Voici un exemple d'utilisation. Supposons que vous vouliez lire le contenu d'un Binary Integer (entier de la forme #.....) au niveau un de la pile. et le stocker dans Cw. Ceci pose problème dans la mesure où le Binary Integer peut être de taille variable. La structure de cet objet est la suivante :

Prologue	(5 quartets)	:	02A4E
Taille hors prologue	(5 quartets)		
Valeur de l'entité	(n quartets)		

On va alors utiliser un programme du type suivant :

```
GOSBVL 0679B          % Sauvegarde des registres
C=DAT1 A              % On lit l'adresse de l'objet au niveau 1
D1=C                  % On pointe dessus
D1=D1+ 5              % On passe 5q. D1 pointe donc sur la longueur
                      %hors-prologue
C=DAT1 A              % On lit la longueur
C=C-6 A               % On retire 6. On a donc le nombre de
                      %chiffres du BI moins 1
P=C    0              % On stocke ce nombre dans P
C=0 W                 % On initialise Cw à zéro
D1=D1+ 5              % On fait pointer D1 sur les chiffres
C=DAT1 WP             % On lit P+1 chiffres, soit exactement tous
                      %les chiffres
P= 0                  % On réinstalle P à zéro
....
```

Compris ? Pas évident, d'autant plus que l'instruction **P=C n** vous est inconnue. Etudions-la ainsi que toutes ses "soeurs".

Exemple

Prenons l'exemple de P= 8 et regardons sur le tableau des champs ce que l'on obtient pour le champs P, WP et tous les autres champs :

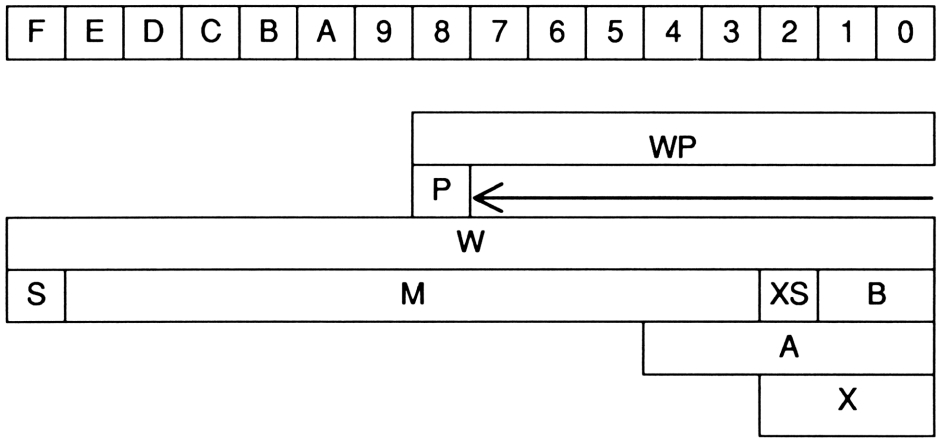


TABLEAU COMPLET DES CHAMPS

3) Les instructions sur P et C

Ces instructions mettent en jeu P et le registre C. Elles permettent d'échanger des valeurs. Elles sont au nombre de quatre :

- **C=P n** : Affecte la valeur de P au quartet numéro n de C.
- **P=C n** : Affecte la valeur du quartet numéro n de C à P.

- **CPEX n** : Echange les valeurs de P et du quartet numéro n de C.
- **C+P+1** : Ajout de P+1 à Ca

II) Les registres de sauvegarde

Il existe deux types de registres de sauvegarde : les registres R0, R1, R2, R3, R4 et la pile RSTK.

1) Les registres R0, R1, R2, R3, R4

Il s'agit de registres en tous points identiques aux registres A, B, C ou D. Il en diffère cependant dans la mesure ou l'on peut juste lire ou écrire leur contenu. Il est impossible d'y effectuer le moindre calcul. Ces registres ne fonctionnent qu'avec A et C. Ils admettent un jeu d'instruction très réduit :

- Des instructions d'affectation : **A=R0 Champ**, **C=R0 Champ**, **R3=C Champ**,.... (Toutes les combinaisons possibles de A ou C avec R0, R1, R2, R3 ou R4). Ces instructions permettent d'affecter à un champ de A ou C la valeur du même champ de R0..R4, et de même d'affecter la valeur d'un champ de A ou C au même champ de R0..R4.
- Des instructions d'échange : **AR0EX (Champ)**, **CR1EX (Champ)**... (De même, toutes les combinaisons possibles entre A ou C et R0, R1, R2, R3 ou R4. Le nom du registre A ou C étant toujours avant celui du registre de sauvegarde.). Ces instructions permettent d'échanger la valeur d'un champ de A ou C avec le même champ de R0, R1, R2, R3 ou R4.

Il s'agit des seules instructions disponibles sur R0, R1, R2, R3 et R4. Il s'agit donc bel et bien de registres de sauvegarde. En général, ils sont employé comme suit : on affecte à R0..R4 des valeurs importantes par l'intermédiaire des registres C et A, ce qui

permet de libérer A, B, C et D. On s'arrange alors pour toujours laisser les registres de calcul à la disposition des sous-routines, afin d'avoir une certaine marge de manoeuvre.

2) La pile **RSTK**

Il s'agit d'un ensemble de huit adresses permettant de sauvegarder les points où doit revenir le programme après une instruction **GOSUB**, **GOSUBL**, **GOSBVL**. Etudions le sur un exemple :

```
...
GOSUB Sous-routine
LC 01000
....
*Sous-routine
...
RTN
```

Supposons qu'avant l'instruction GOSUB Sous-routine, RSTK soit comme suit :

8 :	02410
7 :	01975
6 :	54677
5 :	97980
4 :	34267
3 :	0000F
2 :	79644
1 :	65432

Lors de l'instruction **GOSUB Sous-routine**, Le SATURN stocke au premier niveau l'adresse de **LC 01000** et va exécuter le label *Sous-routine (remarquez que, comme la pile de la HP48, les éléments suivants sont décalés). On dit qu'il y a "empilage".

8 :	01975
7 :	54677
6 :	97980
5 :	34267
4 :	0000F
3 :	79644
2 :	65432
1 :	**Adr. LC 01000**

Lorsque l'instruction **RTN** est rencontrée, le SATURN lit dans RSTK l'adresse de **LC 01000** et va continuer le programme en cet endroit. Il y a alors "dépilage" sur RSTK, le niveau entrant étant mis à # 00000h

00000
01975
54677
97980
34267
0000F
79644
65432

Remarque importante : Puisque RSTK est limité à 8 niveaux, et de par la construction de la machine, on sera limité à 7 **GOSUB**, **GOSUBL** ou **GOSBVL** imbriqués. De plus, lorsque les interruptions sont autorisées, on ne peut plus utiliser que 6 **GOSUB** imbriqués (car les interruptions consomment 2 nivaux de pile) .

3) Utilisation avancée de **RSTK**

En plus de la gestion automatique de RSTK par le SATURN, il existe deux instructions permettant de lire ou d'écrire sur RSTK :

- **C=RSTK** : Lit le niveau le plus bas de RSTK et stocke cette valeur dans Ca, le dépilage est effectué exactement comme lorsque le processeur rencontre un **RTN**. Etudions ceci sur un exemple :

Avant, RSTK est comme suit :

02410
01975
02404
01974
34267
0000F
79644
65432

Le SATURN rencontre l'instruction **C=RSTK**, Ca contient alors la valeur du niveau le plus bas soit **# 65432h**. Il y a alors dépilage, RSTK devient :

00000
02410
01975
02404
01974
34267
0000F
79644

RSTK=C : Pose Ca au niveau le plus bas de RSTK, à la manière du SATURN lorsqu'il rencontre un **GOSUB**. Voici un exemple

Supposons qu'en un point du programme Ca vaut **# 12345h** et que RSTK soit comme suit :

02410
01975
02404
01974
34267
0000F
79644
65432

Le microprocesseur rencontre alors l'instruction **RSTK=C**, il empile alors la valeur de Ca dans la pile :

01975
02404
01974
34267
0000F
79644
65432
12345

Ce sont les deux seules instructions permettant une gestion manuelle de RSTK. Elles servent dans deux cas :

- Tout d'abord, ceci permet une sauvegarde temporaire de Ca. Attention cependant que la valeur soit bien dépilée avant le prochain RTN, l'adresse de retour après le GOSUB devant être directement accessible par le SATURN.
- Ensuite, ceci permet une gestion "simple" des données dans un programme, gestion que vous avez dû aborder dans la partie "Graphismes". Revoyons-en la méthode. On utilise une routine du type suivant :

```
...
GOSUB Apres_données
/Données
*Après_données
C=RSTK
...
```

Cette routine utilise le **GOSUB** pour stocker dans RSTK l'adresse de début des données (Premier quartet après l'instruction GOSUB Après_données). Après GOSUB, le programme rencontre l'instruction **C=RSTK**, l'adresse du premier quartet des données est donc stockée dans Ca, ce qui permet tout type d'exploitation. Il s'agit là de la meilleure méthode pour lire des données propres à un Code.

III) Les registres-drapeaux

Ce sont des registres d'un bit qui ne peuvent donc prendre que deux valeurs, 0 ou 1. Ils seront donc utilisés comme des interrupteurs. Ils sont au nombre de seize et sont dénommés ST.

1) Affecter une valeur à un registre-drapeau

Ces registres ne peuvent contenir que deux valeurs, 0 ou 1. Il existe donc deux instructions :

- **ST=0 n** : Affecte la valeur 0 au registre n, n compris entre 0 et 15,

- **ST=1 n** : Affecte la valeur 1 au registre n, n toujours compris entre 0 et 15.

2) Tester un drapeau

Il existe deux tests, d'égalité à zéro ou un :

- **?ST=0 n** : Teste si le registre n vaut 0. Comme tout test, cette instruction doit être suivie d'un **GOYES**.

- **?ST=1 n** : Teste si le registre n vaut 1. Ici aussi, cette instruction doit être suivie d'un **GOYES**.

3) Utilisation des registres-drapeaux

Il faut tout d'abord noter une chose :

LES DRAPEAUX 12, 13, 14 et 15 SONT UTILISES DE FACON INTERNE PAR LA MACHINE. IL EST DONC FORTEMENT DECONSEILLE D'EN MODIFIER LA VALEUR.

Nous avons donc à notre disposition 12 drapeaux, de 0 à 11. Il sont particulièrement intéressants pour sauvegarder un état ou il ne peut y avoir que deux possibilités. Prenons par exemple une routine qui ne doit s'exécuter qu'une fois sur deux. On pourra alors utiliser une routine du type suivant :

```
ST=0 0                  % Le drapeau 0 va nous servir d'indicateur.,
                        %on l'initialise
....
GOSUB ROUTINE           % Ce qui appelle la routine.
....
*ROUTINE
?ST=0 0                 % Si le drapeau 0 est à 0, on exécute, en
                        %allant à OK
GOYES OK
ST=0 0                  % Sinon, il vaut 1, on le met à 0 pour la
                        %fois suivante, et on exécute rien
RTN
*OK                     % S'il valait 0, on le met à 1 et on exécute
                        %le programme
ST=1 0
...                     % Programme à exécuter une fois sur deux
RTN
```

Voilà donc la méthode. On "bascule" l'interrupteur à chaque appel à la routine, celle-ci étant exécutée que pour une valeur (0 ou 1 au choix). De nombreuses autres utilisations des registres drapeaux sont bien sûr possibles.

IV) Conclusion

Voici la fin de ce chapitre sur les registres. Il convient de noter deux choses :

- Ces registres ne sont pas les seuls disponibles. Il en existe encore quelques autres qui vous seront présentés dans le chapitre "Les calculs en assembleur".
- Les instructions présentées ne sont pas les seules. Vous en découvrirez d'autres par la suite. Sachez tout de même que ce sont les plus utiles au programmeur.

RECAPITULATIF 2
LES CALCULS EN ASSEMBLEUR

Il ne s'agit pas ici de vous présenter la théorie du calcul, déjà abordée dans la première partie de ce livre, mais de vous présenter le jeu d'instructions permettant d'effectuer ces calculs.

I) Le registre CARRY, la gestion des débordements

Nous avons déjà abordé ce registre dans le chapitre 7 de ce livre. Revenons en détail sur son fonctionnement.

Il convient tout d'abord de comprendre ce qu'est la CARRY, il s'agit d'une retenue fonctionnant tout à fait à la manière de la retenue que l'on pose lorsque l'on fait des calculs à la main. Elle intervient lors des opérations d'addition ou de décrémentation sur un ou deux registres.

1) Armement, désarmement de la Carry

Rappelez vous la gestion circulaire les nombres : nous avions vu que les registres se comportaient à la manière d'un compteur kilométrique. Supposons par exemple que l'on calcule sur un champ B ou **deux quartets**. On compte comme suit :

00, 01.., 09, 0A...FC, FD, FE, FF, **00**...

Ce qu'il faut comprendre, c'est que lorsque le "compteur fait un tour" (passage de # FFh à # 0h), la Carry est **armée**, dans le cas contraire, elle est **désarmée**. Elle est donc modifiée à chaque calcul. Il faut savoir que la même chose a lieu dans le sens décroissant, le passage de # 0h à # FFh arme la Carry.

Il ne faut cependant pas perdre de vue que ces changements de la Carry dépendent du champ. Sur un champ B, # FFh + # 1h vaut # 0h et la Carry est armée. Sur un champ plus grand, par exemple A, le registre contient alors # 100h et la Carry est désarmée.

2) Instructions entre deux registres changeant la Carry

Les seules instructions modifiant la Carry sont celles permettant des additions ou soustractions entre deux registres, par exemple :

A=A+C A , A=A+A W, D=D-C B, C+P+1...

3) Instructions sur un registre modifiant la Carry

Il s'agit, là aussi, des instructions d'incrémentation et décrémentation, comme :

A=A+15 A, D0=D0+ 4, D1=D1- 13, B=B-1 P, P=P-1...

4) Le bug du SATURN sur la Carry

Il est important de savoir que les instructions d'incrémentation et de décrémentation sur un champ d'un quartet (**P**, **S**, **XS**), d'une valeur supérieure à 1 (A=A+2 S ; C=C+3 P ...) sont bugguées. Lorsqu'une de ces instructions arme la Carry, celle-ci est propagée sur tout le registre. **IL FAUT DONC EVITER TOUTE OPERATION**

D'INCREMENTATION OU DECREMENTATION DE PLUS D'UNE UNITE SUR UN QUARTET.

Voici l'utilisation de la Carry pendant les calculs. Nous verrons plus tard qu'il en existe une autre lors des tests. Etudions maintenant une notion nouvelle : les modes de calculs.

II) Les deux modes de calcul

Dans les chapitres précédents, nous vous avons montré que tous les calculs en assembleur étaient effectués dans la base hexadécimale. Ceci était pour vous habituer à la base seize, mais le SATURN est aussi capable de calculer en base dix. Pour cela, on dispose de deux commandes : **SETHEX** pour choisir le mode hexadécimal et **SETDEC** pour choisir le mode décimal. En mode décimal, les opérations de calcul sont effectuées comme suit :

Supposons que Aa contienne les quartets "00008", et que l'on soit en mode décimal. Aa contient alors # 00008d. Si on ajoute 2 à Aa, par exemple avec A=A+2 A, Aa contiendra alors les quartets "00010", représentation de # 00010d. La Carry est gérée comme en mode hexadécimal.

Attention : Aucune conversion n'est effectuée lorsque l'on passe d'un mode à l'autre. Pour reprendre notre exemple précédent, si on repasse en mode hexadécimal par **SETHEX**, Aa contiendra toujours "00010", mais ce sera cette fois la représentation de # 00010h soit # 16d !!!

Remarque : Puisqu'à l'entrée d'un programme assembleur, le mode de calcul est l'hexadécimal, celui-ci doit obligatoirement être rétabli à la sortie par la commande SETHEX.

Voyons maintenant les calculs directement accessibles en assembleur.

III) Opérations directement accessibles sur le SATURN

Il existe différents types d'instructions sur le SATURN : additions, soustractions, décalages, rotations et enfin les opérations logiques.

1) Additions et soustractions sur un registre

Ces instructions permettent d'ajouter ou retrancher une valeur comprise entre 1 et 16 à un registre ou un pointeur : A, B, C, D sur n'importe quel champ, D0, D1, et enfin P. Le résultat est indépendant du mode de calcul (SETDEC ou SETHEX), l'addition ou la soustraction se fait toujours en mode hexadécimal. La syntaxe est différente pour les registres de calcul et pour les autres :

- Pour A, B, C ou D, la syntaxe est du type **A=A+13 A**, **B=B-8 M**... (Pas d'espace entre le plus et la valeur)
- Pour D0, D1 ou P, la syntaxe est du type **P=P+ 15**, **D0=D0- 3** ...

Remarque : Les opérations sur un champ de un quartet sont bugguées (Voir ci-dessus)

2) Addition et soustraction entre deux registres

Ces instructions permettent d'ajouter ou de retrancher la valeur d'un registre de calcul (A, B, C ou D) à un autre registre de calcul (A, B, C ou D). Ici encore, le résultat dépend du mode de calcul. Leur syntaxe est du type **D=D+C XS**, **A=A+B A**...

3) Décalages sur un registre

Ces instructions existent sur n'importe quel champ des registres A, B C ou D. Etudions-les sur le champ A de A.

- Décalage d'un bit à gauche : **A=A+A A** (Rappelez-vous l'équivalence avec une multiplication par 2).
- Décalage d'un bit à droite : **ASRB A** (Shift-Right-Bit) (Equivalent à une division par deux).
- Décalage d'un quartet à gauche : **ASL A** (Shift Left) (Equivalent à une multiplication par seize).
- Décalage d'un quartet à droite : **ASR A** (Shift Right) (Equivalent à une division par seize).

4) Rotations sur un registre

Ces instructions existent sur A,B,C et D uniquement sur le champ W. La syntaxe est la suivante:

- Rotation d'un quartet à gauche : **ASLC** (Shift-Left-Circular).
- Rotation d'un quartet à droite : **ASRC** (Shift-Right-Circular).

5) Opérations logiques entre deux registres

Là encore, ces opérations n'existent qu'entre A, B, C et D, sur n'importe quel champ. Il n'existe que deux opérations : Et logique et Ou logique :

- Le Et logique est représenté par "&", la syntaxe est donc du type **A=A&B A**, **C=C&A WP**...
- Le Ou logique est représenté par "!", la syntaxe est donc du type **D=D!C W**, **B=B!A S**...

Nous avons étudié ici toutes les opérations de calcul directement disponibles sur le SATURN. Etudions finalement comment accéder à d'autres opérations.

IV) Effectuer des opérations non directement accessibles

Il existe deux moyens d'effectuer d'autres opérations :

1) Programmer nos calculs à l'aides des opérations élémentaires du SATURN

Il s'agit là de transformer une opération compliquée en une suite d'additions, soustractions, décalages... Supposons par exemple que l'on veuille multiplier Aa par 34. La méthode la plus rapide est de remarquer que 34*Aa = 16*(2*Aa)+2*Aa. On utilise alors la routine suivante :

```
%% Multiplier Aa par 34
A=A+A A              % On multiplie Aa par deux
B=A A                % On sauvegarde 2*Aa
ASL A                % On multiplie maintenant Aa par 16, soit par
                     %32 depuis le début
A=A+B A              % Enfin, on rajoute 2*Aa, finalement, on
                     %abouti bien a 34*Aa
```

Prenons d'autres exemples. Dans la première partie du livre, nous avons vu que A Xor B= (A Or B)-(A And B). On peut alors programmer le Xor de Aa et Ca comme suit :

```
%% Calculer A Xor Ca
B=A A                % Ba = Aa
B=B&C A              % On calcule Aa And Ca
C=C!A A              % On calcule Aa Or Aa
C=C-B A              % En enfin, on calcule (Aa Or Ca)-(Aa And Ca)
```

2) Utiliser des routines existant en Rom

Pour avoir à éviter de reprogrammer à chaque fois les opérations de multiplication et de division, les ingénieurs de chez HP ont prévu plusieurs routines, par exemple :

- Une multiplication de Aa par Ca, en **# 03991h** (donc à appeler par **GOSBVL 03991**). Cette routine stocke le résultat dans Ba. Attention : les débordements ne sont pas gérés.

- Une multiplication de Aw par Cw en **# 53EE4h** (par **GOSBVL 53EE4**). Le résultat est stocké dans Aw et Cw. Attention car ici encore, les débordements ne sont pas gérés.

- Enfin, une division de Aw par Cw, en **# 65807h** (**GOSBVL 65807**). Le quotient est stocké dans Aw et Cw, le reste dans Bw.

De nombreuses autres routines existent : voir l'annexe 2.

IV) Conclusion

Pour conclure ce chapitre, il faut noter qu'en général, les routines en Rom sont assez rapides, mais que dans de nombreux cas particuliers, les reprogrammer peut s'avérer plus efficace. Par exemple pour multiplier par trois Aa, on préférera utiliser une routine ci-dessous plutôt qu'utiliser la routine en Rom.

```
%% Multiplication par trois sur un champ A
B=A A                  % Sauvegarde
A=A+A A                % Multiplication par deux
A=A+B A                % Puis 3 !
```

RECAPITULATIF 3
LES SAUTS

Ce court chapitre a pour but de vous présenter les différentes types de sauts et de retours.

I) Sauts avec ou sans retour ?

Comme dans tout langage non structuré, il existe deux types de sauts.

1) Les sauts avec retour

Il s'agit de sauts permettant de revenir après l'exécution d'une sous-routine. Ceux-ci sont composés de deux instructions : l'instruction d'appel **GOSUB** ou **GOSUBL** et l'instruction de retour **RTN**. Etudions ceci sur un exemple :

```
%% Exemple de saut avec retour
...                    % Début du programme
GOSUB LABEL            % L'instruction qui effectue le saut
LC 01000               % Instruction qui sera effectuée après le RTN
...
*LABEL                 % La destination du saut
...
RTN                    % L'instruction indiquant au SATURN de
                       %revenir après le GOSUB
```

2) Les sauts sans retour

Il s'agit tout simplement de sauter en un point du programme. Ceci s'effectue avec **GOTO** et **GOLONG**

Ce programme effectue donc les opérations suivantes : Il va au label "LABEL", exécute tout ce qui se trouve jusqu'à l'instruction RTN. Lorsque celle-ci est rencontrée, il revient après l'instruction GOSUB soit sur LC 01000. On peut remarquer que le comportement est absolument identique si l'on utilise **GOSUBL** à la place de GOSUB ou **RTNCC** et **RTNSC** à la place de RTN, à la différence près que RTNCC désarme la Carry (ReTurN Clear Carry), RTNSC arme la Carry (ReTurN Set Carry).

II) Sauts relatifs, sauts absolus

Il existe deux types de sauts en assembleurs :

1) Les sauts relatifs

Il s'agit de sauts à l'intérieur d'un programme. On utilise donc les instructions **GOTO**, **GOLONG**, **GOSUB**, **GOSUBL** suivies d'un nom de Label.

2) Les sauts absolus.

Il s'agit de sauts à des routines extérieures au programme, en général à des programmes en Rom. On utilise pour cela les deux instructions **GOSBVL** (GOSuB Very Long) et **GOVLNG** (Goto Very LoNG) suivies de l'adresse du programme. Comme leurs noms l'indiquent, GOSBVL correspond à GOSUB, GOVLNG à GOTO. Par exemple GOSBVL 0679B va exécuter la routine en # 0679Bh (Voir l'annexe pour une liste de routines en Rom).

III) GOTO contre GOLONG, GOSUB contre GOSUBL

Il est très facile de savoir quelle commande utiliser. Le langage machine est fait de telle façon que GOTO permette les petits sauts et GOLONG les grands sauts. GOTO est cependant plus rapide que GOLONG. Lorsqu'on programme, on utilise donc la méthode suivante :

- On programme le saut avec GOTO.
- On essaye d'assembler. En cas d'erreur ("Too Long Jump"), on reprend le source et on remplace l'instruction par GOLONG.

Pour les sauts avec retour, la méthode est absolument identique. GOSUB est à GOTO ce que GOSUBL est à GOLONG.

IV) Les sauts conditionnels

1) Les sauts après un test

Nous l'avons vu dans les chapitres précédents, toute instruction de test est suivie d'un **GOYES Label**. Il s'agit d'un saut conditionnel. Si le test est vérifié, le programme effectue un **GOTO Label**, sinon, il saute le GOYES et passe donc à l'instruction suivante. Remarquez que le saut est bien sans retour. Si on veut appeler Label avec un retour possible, on utilisera une routine du type suivant :

```
%% On veut aller à Label si C#0 A
...
?C=0 A                  % Si le test opposé est vérifié, on saute le
                        %Gosub
GOYES Pas_Label         % Au cas contraire, on a bien C#0 A, et
                        %l'instruction exécutée est GOSUB Label
```

```
GOSUB Label             % On a bien un saut avec retour.
*Pas_Label
...
*Label                  % La sous-routine que l'on veut appeler
...
RTN
...
```

Remarque : Il est intéressant de noter qu'un test modifie la valeur de la Carry. Si celui-ci est vérifié, elle est armée, sinon, elle est désarmée. On peut donc envisager une sous-routine renvoyant le résultat d'un test sur la Carry, avec une routine du type suivant :

```
...
?C=0 A
RTNYES                  % On met à la place du "GOYES" habituel un
                        %"RTNYES" qui correspond à un "RTN" si le
                        %test est vérifié
RTN
...
```

Dans tous les cas, après le test, on a retour après le Gosub ayant appelé la routine. Cependant, si le test est vrai, la Carry est armée, sinon elle est désarmée.

2) Les sauts sur la Carry

Il existe un autre type de saut conditionnel : Les sauts avec test sur la Carry. Il existe pour cela deux instructions : **GOC** (**GO**to if **C**arry) et **GONC** (**GO**to if **N**o **C**arry).

GOC Label effectue un GOTO Label si et seulement si la Carry est armée.

GONC Label effectue un GOTO Label si et seulement si la Carry est désarmée.

V) Conclusion

Vous connaissez maintenant tous les types de sauts possibles en assembleur. Bien que leur emploi ait l'air extrêmement facile, Il est courant de faire de petites erreurs de programmation lorsqu'on les utilise, erreurs que nous allons aborder dans le chapitre suivant.

LES ERREURS A EVITER LES "BONS TRUCS"

Ce chapitre n'est pas à proprement parler un chapitre de cours. Il s'agit plutôt d'une liste des erreurs les plus courantes lors de la programmation assembleur, des moyens de les corriger, puis de "trucs" couramment utilisés.

I) Les erreurs à éviter

1) Les points "dangereux"

Nous l'avons vu dans les chapitres précédents, il existe certaines conditions à l'entrée d'un programme assembleur qui doivent être rétablies à la sortie. Rappelons-les :

A la sortie d'un programme assembleur, il ne faut pas oublier :

De rétablir P à 0 à l'aide de P= 0,

De rétablir le mode de calcul hexadécimal, à l'aide de SETHEX,

Enfin, de rétablir les registres Ba, Da, D0 et D1 à l'aide de GOSBVL 067D2 (sauf si leurs valeurs sont modifiées volontairement).

De ré-autoriser les interruptions.

2)Les erreurs sur les sauts

Il existe deux types d'erreurs courantes sur les sauts : la première se rencontre dans une sous-routine avec retour. Supposons que juste avant le retour, vous vouliez appeler Label. On aurait alors tendance à programmer :

```
...
GOSUB Label
RTN
```

Cela peut cependant être amélioré en se rappelant que les adresses de retour sont empilées dans RTSK. Il suffit donc de remplacer les deux lignes de programme précédentes par :

```
...
GOTO Label
```

Ceci conduit exactement au même résultat, mais est meilleur car plus rapide, moins gourmand en mémoire et surtout car cela libère un niveau dans la pile RSTK.

La deuxième erreur courante se trouve lors des tests. On rencontre souvent des opérations du type :

```
...
?C=0 A
GOYES Pas_Label
GOTO Label
*Pas_Label
...
```

Ceci peut dans la plupart des cas être amélioré en utilisant le test opposé. On programmera donc plutôt :

```
...
?C#0 A
GOYES Label
...
```

Sauf si un GOLONG est nécessaire.

De même, les erreurs sur la Carry peuvent être éliminées :

```
...
GOC Pas_Label
GOTO Label
*Pas_Label
...
```

sera remplacé par :

```
...
GONC Label
...
```

3) Les erreurs de calcul

Lorsqu'on programme en assembleur, on peut par exemple être amené à charger # 2h * # 123h dans Ca. Naturellement, on programmerait :

```
...
LC 00123              % On charge # 123h
C=C+C A               % On le multiplie par 2
...
```

Cela peut être amélioré en remarquant que la multiplication peut se faire en dehors du programme. # 2h * # 123h = # 246h. On programmera donc à la place des instructions précédentes :

```
...
LC 00246
...
```

Une autre erreur de calcul serait d'employer le registre de calcul A lorsque le registre C est disponible. Il s'agit d'une erreur pour deux raisons :

- Le registre A fonctionne avec B et C, mais pas du tout avec D, alors que C fonctionne indifféremment avec les trois registres de calculs cités ci-dessus.
- Des mesures de temps nous montrent que les instructions sur A sont plus lentes que les mêmes opérations sur C. Notamment, LA est beaucoup plus lent que LC. Lorsqu'on programme, on évite au maximum LA et LC qui sont des instructions lentes, et si leur emploi est obligatoire, on fait tout notre possible pour utiliser LC plutôt que LA. Habituez-vous à refuser toute utilisation de cette instruction, sauf dans les cas extrêmes où seul A est disponible.

Une dernière erreur est d'utiliser les champs B, X ou XS lorsqu'on n'utilise aucune autre partie du champ A du registre de calcul, en effet, par construction du SATURN, les instructions sur le champ A sont plus rapides que les mêmes instructions sur d'autres champs. Attention aussi aux champs qui se recoupent.

II) Les "bons trucs"

Il existe certains trucs que l'on acquièrt avec l'expérience. Nous allons maintenant vous en présenter quelques-uns des plus courants :

1) Charger un registre

Supposons que l'on veuille charger # 12h dans Cw. L'idée naturelle serait de programmer cela comme suit :

```
...
LC 0000000000000012
...
```

Il est facile de voir les défauts de cette méthode : Cela est gourmand en mémoire, peu lisible et fastidieux à programmer car il faut compter le nombre de zéros tapés. Une méthode plus propre est de remplacer cette unique instruction par :

```
...
C=0 W
LC 12
...
```

Beaucoup plus lisible, non ? Le deuxième truc se rencontre lorsque l'on veut charger un quartet quelconque de C. Supposons par exemple que l'on veuille stocker # 3h dans le quartet 12. L'idée naturelle est la suivante :

```
...
P= 12
LC 3
...
```

Cette méthode est certes rapide, mais impropre car elle fait perdre la valeur de P. Il est donc préférable d'utiliser :

```
...
CPEX 12
P= 3
CPEX 12
```

ou encore, si l'on a que faire de la valeur de P :

```
...
P= 3
C=P 12
...
```

2) Utilisation de P comme compteur de boucle

Lorsqu'on effectue moins de seize boucles, pour accélérer le programme, plutôt qu'utiliser un champ P, S ou XS d'un champ de calcul. Par exemple, pour faire 13 boucles, on utilise une routine du type :

```
...
P= 12
*BOUCLE
...
P=P-1
GONC BOUCLE
...
```

Il est facile de voir les limitations de la méthode. Si des instructions dont le fonctionnement dépend de P sont présentes dans la routine, leur comportement risque d'être altéré. Attention aussi à ne pas oublier de rétablir la bonne valeur de P après la boucle (dans la plupart des cas P= 0) car sinon, le programme risque de se comporter de façon incompréhensible.

III) Conclusion

Pour conclure, nous allons vous donner un dernier conseil. Le langage machine n'étant pas contrôlé par le système, les bugs peuvent conduire à des pertes de mémoire partielles ou totales, pertes extrêmement fréquentes lorsque l'on débute. Nous vous conseillons donc fortement d'archiver vos données avant d'évaluer tout objet code dont vous n'êtes pas absolument sûr. Il est particulièrement énervant de travailler sur un programme pendant une demi-heure, et de tout perdre à cause d'un bête bug, alors attention !

Troisième partie

Bibliothèque de programmes

AVERTISSEMENT

Dans ce chapitre, tous les programmes sont présentés de la manière suivante :

-Titre

-Explication sur l'utilité du programme

-Programme en lui même

-CRC de l'objet 'Code'

-Commentaires et approfondissement

-Programme assemblé

-CRC du programme assemblé

Le titre

A coté du titre se trouvent des étoiles. Celles-ci indiquent, suivant leur nombre, la difficulté de programmation soit :

* : Très facile

** : Facile

*** : Moyen

**** : Difficile

***** : Très difficile

Le programme

Dans certains cas, les adresses sur S, SX sont différentes de celles sur G, GX. Pour les notifier nous utiliserons la notation suivante :

```
D0= (7050E : S
     8068D : G)
```

Dans cet exemple, vous pouvez constater que la notation ne respecte pas la syntaxe LASM. Ce sera par conséquent à vous de prendre la bonne adresse. Si vous avez une HP 48 S, SX vous recopierez dans votre programme la ligne suivante :

```
D0= 7050E
```

Si vous avez une HP 48 G, GX vous copierez la ligne suivante :

```
D0= 8068D
```

N'oubliez surtout pas qu'il n'est pas nécessaire de taper les commentaires.

Le CRC du programme

Le système ne contrôle pas les agissements d'un programme assembleur. En conséquence, toute erreur de frappe dans un programme peut provoquer des pertes de données. Afin de parer à ces erreurs, nous vous proposons la méthode suivante :

1) Tapez votre programme et sauvegardez-le dans une variable.

2) Assemblez-le en corrigeant si besoin est les erreurs détectées par l'assembleur

3) Dupliquez l'objet 'Code' sur la pile (avec 'ENTER') puis faites un 'BYTES', un 'DROP' et vérifiez que le nombre obtenu correspond avec notre calcul de CRC.

ATTENTION : Le CRC est dans certains cas différent entre les séries S et G.

4) Si le CRC ne correspond pas, vous avez fait soit une erreur de saisie, soit vous avez délibérément changé le programme.

Le mieux est de récupérer les programmes depuis la disquette. La marche à suivre est donnée en annexe 4.

Le programme assemblé

Les programmes assemblés sont présentés sous la forme d'une suite de chiffres hexadécimaux avec un espace tous les 5 caractères. Pour les utiliser vous devez les recopier dans une chaîne de caractères sans espace ni retour à la ligne. Ensuite calculez le CRC de la chaîne à l'aide de la commande BYTES, puis lancez le programme RPL **GASS** (voir ci-dessous) ou le programme assembleur **->OBJECT** donné en bibliothèque de programme.

GASS (S et G)

```
«
"GROB 8 " OVER SIZE 2 / + " " + SWAP + OBJ→
# 4017h SYSEVAL # 56B6h SYSEVAL DROP NEWOB
»
```

CRC : # 1DB3h (S et G)

Remarque : Ce programme a été repris (avec l'autorisation de l'auteur) de l'ouvrage intitulé "VOYAGE AU CENTRE DE LA HP 48 G/GX".

->OBJECT *

Nous l'avons vu précédemment, tout, en mémoire est stocké sous forme hexadécimale. Il peut donc être intéressant de transformer une chaîne de chiffres en objet. C'est ce que fait ce programme. Il prend sur la pile une chaîne contenant l'objet en langage machine (par exemple "C2A207000023") et renvoie l'objet correspondant (dans notre exemple : " "). ->OBJECT sera utilisé par la suite pour assembler tous les programmes donnés sous forme de codes hexadécimaux.

Remarque : ->objet est une version assembleur de GASS

Attention : l'utilisation au hasard sur une chaîne ne correspondant pas à un objet peut conduire à des "Memory Lost". Voir "VOYAGE AU CENTRE DE LA HP 48" pour plus de détails sur les objets et leur structure.

```
%% ->OBJECT

GOSBVL 0679B          % Save-Reg
C=DAT1 A              % On lit l'adresse de l'objet au niveau 1
R0=C A                % On sauvegarde l'adresse de début de
                      %l'objet.
D1=C                  % On pointe sur cet objet
A=DAT1 A              % On en lit les 5 premiers quartets, donc le
                      %prologue
LC 02A2C              % On charge C avec le prologue d'une String
?C=A A                %On vérifie le prologue
GOYES PAS_ERR         % C'est bien une chaîne ? Okay, pas
                      %d'erreur. Sinon
*ERR                  % Routine d'émission du message "Bad
                      %Argument Type"
LA 00202              % On charge Aa avec le numéro du message
                      %d'erreur
GOSBVL 067D2          % On récupère les registres.
GOVLNG 05023          % On appelle la routine d'émission du
                      %message. C'est fini.

*PAS_ERR
```

```
D1=D1+ 5                % On pointe sur les 5 quartets suivants, donc
                        %la longueur
C=DAT1 A                % On lit la longueur
C=C-5 A                 % moins 5 pour avoir la mémoire occupée par
                        %les caractères
CSRB A                  % On divise par 2 car un caractère occupe 2
                        %quartets.
?C=0 A
GOYES ERR               % Si Ca vaut 0, la chaîne est vide, on
                        %émet une erreur.
R1=C A                  % On sauvegarde la taille.
GOSBVL 039BE            % On réserve Ca quartets pour créer l'objet.
CD0EX                   % D0 pointe sur le début de la zone mémoire
                        %réservée
D0=C
R2=C A                  % On sauvegarde cette adresse.
C=R0 A                  % On récupère l'adresse de début de la
                        %chaîne.
D1=C
D1=D1+ 10               % D1 pointe sur le contenu
C=R1 A                  % On récupère la taille de l'objet
C=C-1 A                 % moins 1 pour la boucle.

*BOUCLE                 % Boucle de création
A=DAT1 B                % on lit 2 quartets, donc un caractère
LC 30
A=A-C B                 % On retire #30h car "0" est le caractère
                        %#30h
LC 09
?C>=A B                 % Si on a obtenu un quartet entre 0 et 9,
                        %"Ok", sinon on a un caractère A..F
GOYES OK
LC 07                   % On retire donc #7h car le code ASCII de A
                        %est #41h
A=A-C B                 % On obtient donc un quartet entre A et F

*OK
DAT0=A P                % On recopie le quartet obtenu
D1=D1+ 2                % On passe au caractère suivant
D0=D0+ 1                % Et au quartet suivant
B=B-1 A                 % On décrémente le compteur, et on
GONC BOUCLE             %reboucle si on n'a pas fini

*FIN
GOSBVL 067D2            % On récupère les registres.
C=R2 A                  % On récupère l'adresse de début de l'objet
                        %créé
DAT1=C A                % On le pose sur la pile à la place de
                        %la chaîne
A=DAT0 A                % Fin traditionnelle
D0=D0+ 5
PC=(A)

@
```

CRC : # 44C1h (S et G)

Exemple d'utilisation de ->OBJECT

Assemblez l'objet suivant (Attention, les espaces et sauts de lignes sont donnés pour vous faciliter la lecture, il ne doivent pas être saisis).

```
C2A20  13000  C4729  6E667  96471  64796  F6E60   21657
0267F  69716  7656
```

CRC : # 9468h (S et G)

OBJECT-> *

Le programme précédent transforme une chaîne de chiffres en objet. Celui-ci effectue exactement l'opération inverse. Il prend sur la pile un objet quelconque et renvoie son image mémoire, c'est-à-dire la liste de codes hexadécimaux le représentant.

```
%% OBJET->

GOSBVL 0679B          % Save-Reg
C=DAT1 A              % On lit l'adresse de l'objet au niveau 1
D=C A                 % On la stocke dans Da
R1=C A                % et dans R1a
D0=C                  % D0 pointe sur cet objet
GOSBVL 03019          % Calcule l'adresse du premier quartet
                      %après cet objet
CD0EX                 % On la met dans Ca
C=C-D A               % Calcul de la longueur totale de l'objet
                      %(@Fin-@Debut)
R2=C A                % On stocke dans R2a
C=C+C A               % fois 2 pour la taille en quartets de
                      %la String à créer
GOSBVL 05B7D          % On crée la String
C=R1 A                % On récupère l'adresse de début de l'objet
D1=C                  % D1 pointe dessus
C=R2 A                % On récupère la longueur de l'objet
B=C A                 % On l'installe dans Ba
B=B-1 A               % On enlève 1
GOC FIN               % Si Carry donc si la longueur valait 0,
                      %on arrête

*BOUCLE
A=0 B
A=DAT1 P              % On lit un quartet de l'objet
LC 30                 % On ajoute #30h car #30h est le code
                      %ASCII de '0'
A=A+C B
LC 39                 % Si on a un code ASCII supérieur à # 39h,
                      %c'est qu'on a
?C>=A B               %un quartet de type A..F
GOYES OK

LC 07                 % On ajoute donc 7 pour se placer sur
                      %les caractères de
A=A+C B               %type 'A'..'F'
```

```
*OK
DAT0=A B              % On écrit le caractère.
D1=D1+ 1              % On pointe sur le quartet suivant
D0=D0+ 2              % Et le caractère suivant
B=B-1 A               % On décrémente le compteur
GONC BOUCLE           % Et on reboucle si on n'a pas fini.

*FIN                  % Fin du programme
GOSBVL 067D2          % On rétablit les registres
A=R0 A                % Adresse de début de la String
DAT1=A A              % On l'installe à la place de l'objet au
                      %niveau 1
A=DAT0 A              % Fin traditionnelle
D0=D0+ 5
PC=(A)

@
```

CRC : # 902h (S et G)

Exemple d'utilisation de OBJECT->

Comment est codée la chaîne "Coucou" en mémoire ? Pour cela, on utilise la méthode suivante :

On crée la chaîne ("Coucou") et on la pose au niveau 1.

On utilise OBJECT->, on obtient alors :

"C2A201100034F65736F657"

Remarque : On peut alors utiliser ->OBJECT sur la chaîne obtenue, on recrée ainsi l'objet.

```
CCD20  5C000  8FB97  60147  81AF0  81351  4334C  2A208
A2A18  08242  02008  F2D76  08D32  05017  41478  18FA4
819F2  8AA4D  81AF0  98FEB  93013  61348  1AF0A  81AF1
81351  7981A  F19CE  14B31  03B6A  31909  EA903  170B6
A1500  17116  0CD59  D8F2D  76081  AF1A1  45142  16480
8C
```

CRC : # 50D7h (S et G)

->MACRO *

Nous l'avons vu dans la partie précédente, l'insertion d'objet dans un programme assembleur se fait à l'aide de **macros**. Il faut cependant distinguer l'insertion d'objet sous leur forme standard et l'insertion du contenu de ces objets. Ainsi, pour un grob, le prologue et les indications de longueur sont inutiles. On stocke seulement les quartets représentant les informations graphiques.

De plus, les données stockées dans les macros ne doivent pas être des chaînes de chiffres hexadécimaux mais des chaînes de caractères, chaque caractère représentant deux quartets. Il faut donc effectuer une transformation de l'objet en chaîne de caractères.

Toutes ces opérations sont automatisées dans ce programme qui, de manière générale transforme un objet en chaîne servant de **macros**. Il gère de plus les trois objets String, Grob et Code de la façon suivante :

- Pour la **String**, il saute le Prologue et la longueur afin de ne garder que le contenu.

- De même pour le **Grob** où il passe les vingts premiers quartets afin de ne récupérer que les informations graphiques.

- Enfin, pour le **Code**, il saute le Prologue et la longueur afin de ne garder que les instructions langage-machine. Cela permet entre autre d'insérer un programme ou un morceau de programme.

Ce programme va rapidement vous devenir indispensable.

```
%%% ->MACRO %%

GOSBVL 0679B          % Save-Reg
C=DAT1 A              % On lit l'adresse de l'objet au niveau 1
?C#0 A                % Cette adresse est-elle non nulle ?
GOYES OK              % Si oui, on a bien un objet. Sinon, la pile
                      %est vide, erreur
LA 00201              % On charge # 201h, numéro de "Too Few
                      %Argument"
GOSBVL 067D2          % On récupère les registres
GOVLNG 05023          % On émet le message et on sort
*OK                   % Dans le cas ou on a bien un objet
R1=C A                % On sauvegarde son adresse
D=C A                 % dans Da aussi !
D0=C                  % On fait pointer D0 dessus
GOSBVL 03019          % Zou, D0 contient l'adresse du premier
                      %quartet après
CD0EX                 % l'objet, on la stocke dans Ca
C=C-D A               % On calcule la taille de l'objet
                      %(@Fin-@Début)
GOSUB TESTE_OBJET     % On va tester si on n'est pas sur un objet
                      %particulier
R2=C A                % On sauvegarde la taille de l'objet
GOSBVL 05B7D          % On réserve une chaîne de la taille de
                      %l'objet
CD0EX                 % D0 pointe sur le contenu de la chaîne
D1=C                  % Magique ! Maintenant c'est D1
C=R1 A                % On récupère l'adresse de début de l'objet
D0=C                  % D0 pointe dessus
C=R2 A                % On récupère la taille
GOSBVL 0670C          % Et Hop, on recopie l'objet de l'adresse de
                      %D0 à celle de D1
GOSBVL 067D2          % On récupère les registres
A=R0 A                % On place la chaîne au niveau 1, à la place
                      %de l'objet
DAT1=A A
A=DAT0 A              % Fin traditionnelle
D0=D0+ 5
PC=(A)

*TESTE_OBJET          % Routine qui teste si l'objet n'est pas
                      %String, Grob, ou Code
D=C A                 % On sauvegarde Ca qui contient la longueur
                      %de l'objet
C=R1 A                % On récupère l'adresse de début de l'objet
D1=C                  % On fait pointer D1 dessus
A=DAT1 A              % On lit cinq quartets donc le prologue
LC 02A2C              % Prologue d'une chaîne
?C#A A                % Pas le même prologue ?
GOYES PAS_CHAINE      % Alors ce n'est pas une chaîne, on saute

*PASSE_10             % Routine qui passe 10 quartets
                      %(Prologue+Longueur)
C=R1 A                % On récupère l'adresse de début
C=C+10 A              % On passe ce qui est inintéressant
R1=C A                % On re-sauvegarde cette adresse
C=D A                 % On récupère la longueur
```

```
C=C-10 A                % On la diminue des 10 quartets qu'on a passé
RTN                     % Salut, j'ai fini mon boulot

*PAS_CHAINE             % Si on n'avait pas une chaîne
LC B1E                  % Ca contenait 02A2C, il contient maintenant
                        %02B1E, prologue
?C#A A                  % D'un grob. C'est le même prologue ?
GOYES PAS_GROB          % Non, alors on saute
C=R1 A                  % Même Topo que précédemment mais avec 20
                        %quartets à sauter
C=C+16 A
C=C+4 A
R1=C A
C=D A
C=C-16 A
C=C-4 A
RTN

*PAS_GROB               % Si ce n'était pas non plus un grob
LC DCC                  % Ca contient maintenant 02DCC, le prologue
                        %d'un Code
?C#A A                  % Ce n'est pas un Code ?
GOYES PAS_CODE          % Alors on saute !
GOTO PASSE_10           % On a 10 quartets à sauter, on réutilise la
                        %routine précédente

*PAS_CODE               % Ce n'est pas non plus un Code ! Alors ce
                        %n'est pas un objet
                        %particulier
C=D A                   % On rétablit la longueur
RTN                     % Et salut, je ne peux rien pour vous !

@
```

CRC : # 6928h (S et G)

Modifications possibles de ce programme

Il peut être effectué différentes modifications :

Gérer d'autres objets comme les 'Library Data', les 'Binary Integer'...

Gérer d'autres fonctions, par exemple transformer une suite de quartets (chaîne de chiffres hexadécimaux) en macro équivalente.

A vous de voir...

```
CCD20  40100  8FB97  60147  8AEA1  80824  10200  8F2D7
608D3  20508  1AF09  D7134  8F910  30136  EB734  081AF
0A8FD  7B501  36135  81AF1  91348  1AF1A  8FC07  608F2
D7608  1AF10  14114  21648  08CD7  81AF1  91351  4334C
2A208  A6E18  1AF19  818F2  981AF  09DB8  18FA9  0132E
1B8A6  A281A  F1981  8F2F8  18F23  81AF0  9DB81  8FAF8
18FA3  0132C  CD8A6  6067A  FDB01
```

CRC : # D0FDh (S et G)

DUMPER **

Ce programme permet de visionner la mémoire sous forme hexadécimale. Si il n'y a pas d'objet sur la pile le programme commence à l'adresse # 00000h. Si il y en a un on va à l'adresse de début de l'objet.

Les touches sont :

- Plus (pour avancer d'un quartet)
- Moins (pour reculer d'un quartet)
- Drop (pour sortir)

REMARQUE : Ce programme nécessite de mettre une fonte dans le répertoire Macro sous le nom 'FONT'. Le programme en RPL suivant se charge de créer le Grob de la fonte. Il ne vous restera qu'à utiliser le programme ->MACRO.

```
«
# 8h # 80h BLANK 0 9
  FOR I # 0h I # 8h * 2 →LIST I 48 + CHR 2 →GROB REPL
  NEXT 0 5
  FOR I # 0h I # 8h * # 50h + 2 →LIST I 65 + CHR 2 →GROB REPL
  NEXT
»
```

CRC : # DCBDh (S et G)

```
%%%     DUMPER

%% R0a  :  @ ECR
%% R1a  :  @ OBJ
%% R2a  :  @ FONT


GOSBVL 0679B
GOSBVL 01115
```

```
C=DAT1 A
C=C-1 A
R1=C A                  % R1a contient l'adresse de l'objet au niveau
                        %1 de la pile

D0= (8068D : G
     7050E : S)
C=DAT0 A
R0=C A                  % R0a contient l'adresse de l'écran


GOSUBL APRES.FONT
/FONT
 *APRES.FONT
C=RSTK
R2=C A                  % R2a contient l'adresse du début de la fonte


 *BCL.PRIN

LC 001                  % Test de la touche plus
GOSBVL 01EEC
?CBIT=0 0
GOYES PAS.PLUS          % Si la touche n'est pas enfoncée on va au
                        %label PAS.PLUS

C=R1  A
C=C+1 A                 % On avance d'un quartet dans la mémoire
R1=C  A                 % On stocke cette avance

*PAS.PLUS


LC 002                  % Test de la touche moins
GOSBVL 01EEC
?CBIT=0 0
GOYES PAS.MOINS         % Si la touche n'est pas enfoncée on va au
                        %label PAS.MOINS
C=R1  A
C=C-1 A                 % On recule d'un quartet dans la mémoire
R1=C  A                 % On stocke ce recule

*PAS.MOINS

LC 010                  % Test de la touche DROP
GOSBVL 01EEC
?CBIT=0 0
GOYES PAS.FIN           % Si elle n'est pas appuyée on ne sort pas du
                        %programme
GOSBVL 010E5
GOSBVL 067D2
A=DAT0 A
D0=D0+ 5
PC=(A)

*PAS.FIN
```

```
GOSUBL AFF.ADR          % On appelle la routine qui affiche l'adresse
                        %où l'on est

LC 01000                % On fait une boucle de pause
*PAUSE
C=C-1 A
GONC PAUSE

GOSUB AFF.CHR           % On affiche les caractères hexadécimaux de
                        %la mémoire

GOTO BCL.PRIN           % On retourne au label BCL.PRIN


%%% DEBUT DES GOSUBS

*AFF.CHR

LA F                    % On pointe sur le dernier caractère
                        %hexadécimal à afficher
B=A    P
C=R1   A
C=C+1  A
C=C+15 A
R1=C   A

*AFF.CHR.2

C=R1  A
D0=C
C=C-1 A
R1=C  A

*BCL.PRIN.N.CHR

C=0 A
C=DAT0 P                % On lit le chiffre hexadécimal
CSL A                   % On le multiplie par 16 de sorte que l'on
                        %puisse pointer sur le graphique du chiffre
                        %correspondant car il ne faut pas oublier
                        %qu'un grob de caractère est codé ici sur 16
                        %quartets
A=R2 A
C=C+A A
GOSUBL AFF.ECR

B=B-1 P                 % On va boucler 16 fois
GONC AFF.CHR.2
RTN                     % On revient au programme source

*AFF.ECR                % On affiche le grob pointer par Ca
D1=C
C=R0 A
D0=C
C=0 A
C=B P
C=C+C A
AD0EX
```

```
C=C+A A
D0=C

LC 7                    % On boucle 8 fois

*BCL.WV                 % Boucle d'affichage

A=DAT1 B
DAT0=A B
D0=D0+ 16
D0=D0+ 16
D0=D0+ 2
D1=D1+ 2

C=C-1 P
GONC BCL.WV
RTN

*AFF.ADR

B=0 P
D0= (80100 : G
     70300 : S)

C=R1 A
C=C+1 A
DAT0=C A                % On va stocker dans la RAM réservée
                        %l'adresse où l'on est

 *BCL.AFF.ADR           % Cette routine est à peu de chose prés
                        %similaire à la précédente

A=0 A
A=B P
LC (80104 : G
    70304 : S)
C=C-A A
D0=C
A=DAT0 P                % On récupère les chiffres hexadécimaux un à
ASL A                   %un
C=R2 A
C=C+A A
GOSUB AFF.CHR.ADR       % On appelle la routine d'affichage des
                        %caractères de l'adresse
B=B+1 P
LC 5
?B#C P
GOYES BCL.AFF.ADR

RTN                     % On retourne au programme principal

*AFF.CHR.ADR

D1=C

A=R0 A                  % On récupère l'adresse de l'écran
LC 00408                % On ajoute # 408h pour être plus ou moins au
```

```
C=C+A A                 %milieu de l'écran

D0=C
C=0 A
C=B P
C=C+C A
ADOEX
C=C+A A
D0=C
LC 7                    % On boucle 8 fois

*BCL.OP                 % Création d'un label

A=DAT1 B
DAT0=A B
D0=D0+ 16
D0=D0+ 16
D0=D0+ 2
D1=D1+ 2

C=C-1 P
GONC BCL.OP

RTN                     % On retourne à la boucle principale

@
```

CRC : # 1605h (G) # 9572h (S)

Ce programme est simple du point de vue programmation. Il met en avant une technique qui consiste à sauvegarder des données dans une zone de mémoire particulière. Cette zone, qui s'appelle la RAM réservée est donnée dans ses moindres détailles dans l'ouvrage intitulé: "Voyage au centre de la HP 48"

- Extensions possibles:

Visionner la mémoire de 16 quartets en 16 quartets en appuyant sur deux autres touches.

Pouvoir avoir une deuxième ligne avec les caractères ASCII

Avoir la possibilité d'écrire en mémoire

```
CCD20  DC200  8FB97  608F5  11101  47CE8  1AF09  1BD86
08146  81AF0  88E00  10E01  19151  3111E  00040  60404
04040  E000E  01101  C0201  0F100  E0110  1E001  11E00
080C0  A090F  18080  00F11  0F001  0111E  000C0  2010F
01111  E000F  10180  40202  02000  E0111  1E011  11E00
0E011  11E10  18060  00E01  111F1  11111  100F0  1111F
01111  F000E  01110  10101  1E000  70901  11111  90700
0F110  10F01  010F1  00F11  010F0  10101  00007  81AF0
A3210  08FCE  E1080  8A001  81AF1  9E681  AF093  22008
FCEE1  0808A  00181  AF19C  E81AF  09320  108FC  EE108
08A0A  18F5E  0108F  2D760  14216  4808C  8E890  03400
010CE  5DF74  006C7  F8082  0FA88  81AF1  9E681  8F2E8
1AF09  81AF1  9134C  E81AF  09D21  560F2  81AF1  2C28E
8000A  0D55D  01135  81AF1  8134D  2A89C  6132C  21343
0714B  14816  F16F1  61171  A0E5A  E01A8  11B00  10881
AF19E  6144D  0A843  44010  8E213  41520  F081A  F1AC2
7D00B  05305  9054D  01135  81AF1  03480  400C2  134D2
A89C6  132C2  13430  714B1  4816F  16F16  1171A  0E5AE
01
```

CRC : # B53Ah (G)

```
CCD20  DC200  8FB97  608F5  11101  47CE8  1AF09  1BE05
07146  81AF0  88E00  10E01  19151  3111E  00040  60404
04040  E000E  01101  C0201  0F100  E0110  1E001  11E00
080C0  A090F  18080  00F11  0F001  0111E  000C0  2010F
01111  E000F  10180  40202  02000  E0111  1E011  11E00
0E011  11E10  18060  00E01  111F1  11111  100F0  1111F
01111  F000E  01110  10101  1E000  70901  11111  90700
0F110  10F01  010F1  00F11  010F0  10101  00007  81AF0
A3210  08FCE  E1080  8A001  81AF1  9E681  AF093  22008
FCEE1  0808A  00181  AF19C  E81AF  09320  108FC  EE108
08A0A  18F5E  0108F  2D760  14216  4808C  8E890  03400
010CE  5DF74  006C7  F8082  0FA88  81AF1  9E681  8F2E8
1AF09  81AF1  9134C  E81AF  09D21  560F2  81AF1  2C28E
8000A  0D55D  01135  81AF1  8134D  2A89C  6132C  21343
0714B  14816  F16F1  61171  A0E5A  E01A8  11B00  30781
AF19E  6144D  0A843  44030  7E213  41520  F081A  F1AC2
7D00B  05305  9054D  01135  81AF1  03480  400C2  134D2
A89C6  132C2  13430  714B1  4816F  16F16  1171A  0E5AE
01
```

CRC : # C350h (S)

GRAPHICS VIEWER (SOFT) *

Ce programme sert à visionner la mémoire sous forme graphique. Il est assez primaire à ce niveau de programmation. Cependant de nombreuses options peuvent être rajoutées.

Les touches sont :

+ et - pour avancer et reculer dans la zone de mémoire

DROP pour sortir

```
% Effacer impérativement les menus

GOSBVL 0679B
INTOFF

D0= (8068D : G
     7050E : S)
C=DAT0 A
R0=C A                % R0a contient l'adresse de l'écran

C=DAT1 A              % On récupère l'adresse de l'objet au niveau
D1=C                  %1 de la pile si il n y en a pas l'adresse
                      %est à # 00000h


 *BCL.PRIN

LC 001                % Test de la touche plus
GOSBVL 01EEC
?CBIT=0 0
GOYES PAS.PLUS

D1=D1+ 16             % On avance d'une ligne (32 quartets)
D1=D1+ 16
D1=D1+ 2


 *PAS.PLUS

LC 002                % Test de la touche moins
```

```
GOSBVL 01EEC
?CBIT=0 0
GOYES PAS.MOINS


D1=D1- 16              % On recule d'une ligne
D1=D1- 16
D1=D1- 2


 *PAS.MOINS




LC 010                 % On teste la touche drop
GOSBVL 01EEC
?CBIT=0 0
GOYES PAS.OUT


INTON                  % On sort du programme
GOSBVL 067D2
A=DAT0 A
D0=D0+ 5
PC=(A)

 *PAS.OUT

GOSUBL CP.D1.D0        % On appelle la routine copiant le graphique
                       %à l'écran

  GOTO BCL.PRIN        % On retourne à la boucle principale


 *CP.D1.D0             % Routine de copie d'écran

CD1EX
D1=C
R1=C A

C=R0 A
D0=C
LA 3F                  % On va boucler 64 fois pour afficher les 64
                       %lignes constituant l'écran

*BCL.AFF               % On ne copie dans cette boucle qu'une seule
                       %ligne
C=DAT1 16
DAT0=C 16
D0=D0+ 16
D1=D1+ 16

C=DAT1 16
DAT0=C 16
D0=D0+ 16
D1=D1+ 16
```

```
C=DAT1 B
DAT0=C B
D0=D0+ 2
D1=D1+ 2

 A=A-1 B
 GONC BCL.AFF          % Si la carry n'est pas armée on va au label
                       %BCL.AFF
 C=R1 A
D1=C

RTN

@
```

CRC : # 1605h (G) # E688h (S)

Ce programme est très utile car il va vous permettre de comparer la différence de vitesse entre l'affichage SOFT de ce programme et l'affichage HARD du programme suivant. Ainsi pour les modifications possibles reportez-vous au programme suivant.

```
CCD20  5E000  8FB97  60808  F1BD8  60814  681AF  08147
13532  1008F  CEE10  808A0  B017F  17F17  13220  08FCE
E1080  8A0B0  1CF1C  F1C13  20108  FCEE1  0808A  07180
808F2  D7601  42164  808C8  E4000  699F1  37135  81AF0
981AF  18134  80821  F315F  F15CF  16F17  F15FF  15CF1
6F17F  14F14  C1611  71A6C  54D81  AF191  3501
```

CRC : # 8A93h (G)

```
CCD20  5E000  8FB97  60808  F1BE0  50714  681AF  08147
13532  1008F  CEE10  808A0  B017F  17F17  13220  08FCE
E1080  8A0B0  1CF1C  F1C13  20108  FCEE1  0808A  07180
808F2  D7601  42164  808C8  E4000  699F1  37135  81AF0
981AF  18134  80821  F315F  F15CF  16F17  F15FF  15CF1
6F17F  14F14  C1611  71A6C  54D81  AF191  3501
```

CRC : # A676h (S)

GRAPHICS VIEWER (HARD) *

Ce programme utilise les mêmes touches que le précèdent.

```
GOSBVL 0679B
INTOFF

D0= 00120              % D0 pointe sur l'adresse de l'écran

C=DAT1 A               % Ca va contenir l'adresse de l'objet au
                       %niveau 1 de la pile
D1=C                   % D1 pointe sur l'adresse au niveau 1 de la
                       %pile

 *BCL.PRIN

LC 001                 % On teste si le touche touche + est appuyée
GOSBVL 01EEC
?CBIT=0 0
GOYES PAS.+

D1=D1+ 16              % On avance d'une ligne (34 quartets)
D1=D1+ 16
D1=D1+ 2

GOSUB CP.D1.D0         % On appelle la routine CP.D1.D0

*PAS.+

LC 002                 % On teste si la touche - est appuyée
GOSBVL 01EEC
?CBIT=0 0
GOYES PAS.-

D1=D1- 16              % On recule d'une ligne
D1=D1- 16
D1=D1- 2

GOSUB CP.D1.D0         % On appelle la routine CP.D1.D0

*PAS.-


LC 010                 % On teste si la touche DROP est appuyée
GOSBVL 01EEC
?CBIT=0 0
GOYES PAS.DROP
```

```
INTON                   % On re-autorise les interruptions

GOSBVL 01C7F
GOSBVL 067D2
A=DAT0 A
D0=D0+ 5
PC=(A)


*PAS.DROP

GOSUBL CP.D1.D0

GOTO BCL.PRIN


 *CP.D1.D0              % Dans cette version du viewer on a modifié
                        %la routine d'affichage de l'écran.

CD1EX                   % On reprend l'adresse contenu dans D1
D1=C
DAT0=C A                % Et on la stocke dans D0


RTN

@
```

CRC : # C363h (S et G)

Ce programme n'a que très peu d'intérêt avec le peu d'options qu'il a actuellement. Cependant en partant de cette idée vous pouvez créer un programme fort utile qui permettrait de visionner tout les grobs et d'avoir la possibilité de mettre, soit leur adresse, soit le grob lui-même sur la pile.

```
CCD20  1A000  8FB97  60808  F1B02  10014  71353  21008
FCEE1  0808A  0F017  F17F1  71795  03220  08FCE  E1080
8A0F0  1CF1C  F1C17  93032  0108F  CEE10  808A0  E1808
08FF7  C108F  2D760  14216  4808C  8E400  06A8F  13713
51440  1
```

CRC : # F14Bh (S et G)

DIGITSOUND ***

La HP48 dispose de l'un des meilleurs buzzers disponibles sur le marché des ordinateurs de poche. Malheureusement, son utilisation standard est limitée à l'émission de Bips plus où moins gracieux. DIGITSOUND est un ensemble de deux programmes permettant de jouer des sons échantillonnés sur votre calculatrice. Les programmes sont les suivants :

DIGITCONV : Un utilitaire transformant un son huit bits de micro-ordinateur en Library Data exploitable par la routine suivante

DIGITPLAY : Un programme jouant les sons convertis par DIGITCONV à une vitesse définie par l'utilisateur.

Etudions tout d'abord DIGITCONV, premier de ces deux utilitaires.

DIGITCONV ***

Comme nous l'avons dit plus tôt, ce programme convertit des sons huit bits d'un maximum de 11 kilo hertz en un format exploitable par DIGITPLAY. Nous avons testé cet utilitaire sur des sons provenant de machines telles qu'un PC, un Machintoch, ou un Amiga.

Le format de sons huit bits est certainement l'un des plus répandus sur micro-ordinateurs. Pour référence, citons que ce programme fut originellement écrit pour les "Soundtracks" de l'Amiga, il a été testé avec succès sur le format .WAV du PC et il devrait certainement fonctionner avec les échantillons de l'Atari ST ou du Macintosh.

Pour comprendre la routine de conversion, il convient de faire un peut de théorie du son sur micro-ordinateur. Pour émettre un son, un ordinateur peut déplacer la membrane d'un haut-parleur en un nombre fixé de positions. Des ordinateurs comme l'Amiga ou l'Atari ST dispose de 2^8=256 niveaux. C'est pourquoi on parle de sons huit bits. Sur le HP48, la chose est beaucoup plus limitée ; on ne dispose que de deux niveaux, accessibles à l'aide de OUT :

- Effectuer un OUT 800 émet un "click" sur le haut parleur ;
- Effectuer un OUT 0 coupe le haut parleur.

L'idée consiste alors à étudier le son de l'ordinateur, et à émettre un "click" que si le niveau de la membrane dépasse 128 (Le milieu des positions possibles).

Enfin, il faut connaître le codage des échantillons sur ordinateurs. Ceci est très simple, les différents niveaux de

membranes sont stockés les uns à la suite des autres, sans aucune autre information. La routine de conversion est alors très élémentaire : on lit un niveau, on le convertit en un bit, ce qui est suffisant car un bit peut prendre deux positions, et on recommence jusqu'à la fin.

```
%% DIGITCONV

GOSBVL 0679B
C=DAT1 A              % On regarde l'objet au niveau 1
D1=C

A=DAT1 A
LC 02A2C              %c'est bien une chaîne ?
?C=A A
GOYES OK

*BAT                  %si non, on émet "Bad Argument Type"
LA 00202
GOSBVL 067D2

GOVLNG 05023          % Message d'erreur

*OK                   %si oui
D1=D1+ 5              % On va lire la longueur de l'objet
CD1EX
D1=C                  % On sauvegarde la position dans la chaîne
RSTK=C

C=DAT1 A              % On lit la longueur
C=C-5 A               % On calcule le nombre de quartets
                      %d'informations
?C=0 A                % Si c'est nul, la chaîne est vide, on ne
                      %peut convertir
GOYES BAT             %on émet "Bad Argument Type"

CSRB A                % Sinon, on calcule la taille du son à créer
CSRB A                % C'est la taille de la chaîne divisé par
CSRB A                %huit car on remplace un octet par un bit
R4=C A                % On sauvegarde la taille
GOSBVL 05B7D          %et on crée une chaîne de la longueur
                      %nécessaire
C=RSTK                % On récupère l'adresse dans la chaîne
C=C+5 A               %et on va pointer sur le contenu
D1=C
LC 80                 % # 80h=# 128d, la valeur par rapport à
                      %laquelle
D=C B                 %on va faire les comparaisons
C=R4 A                % On récupère le nombre échantillons à
C=C-1 A               %convertir, moins un pour boucler sur
B=C A                 %sur la Carry

*BOUCLE               %La boucle principale
```

```
A=0 A                   %On lit un échantillon
C=DAT1 B                % On compare avec la référence
?C<D B
GOYES PASSE0            %et si nécessaire, on modifie le bit adéquat
                        %dans le son créé
ABIT=1 0

*PASSE0
D1=D1+ 2                % de même pour échantillon suivant
C=DAT1 B
?C<D B
GOYES PASSE1
ABIT=1 1

*PASSE1
D1=D1+ 2                % Puis pour le troisième
C=DAT1 B
?C<D B
GOYES PASSE2
ABIT=1 2

*PASSE2
D1=D1+ 2                % Et enfin pour le quatrième
C=DAT1 B
?C<D B
GOYES PASSE3
ABIT=1 3

*PASSE3
D1=D1+ 2
DAT0=A P                % On inscrit les quatre bits convertis dans
D0=D0+ 1                %la chaîne de conversion
B=B-1 A                 %A-t-on fini la conversion
GONC BOUCLE

C=R0 A                  %Ca y est, on a fini
D1=C                    % On change le prologue de la chaîne en un
LC 02B88                %prologue Library Data, c'est plus joli
DAT1=C A                %a l'affichage

GOSBVL 067D2            % Une fin standard
A=R0 A                  % On remplace le son par sa conversion
DAT1=A A

A=DAT0 A                % Et on sort
D0=D0+ 5
PC=(A)
@
```

CRC : # 6A8h (S et G)

Programme assemblé (S et G)

```
CCD20  50100  8FB97  60147  13514  334C2  A208A  2A180
82420  2008F  2D760  8D320  50174  13713  50614  7818F
A48AA  1D819  F2819  F2819  F281A  F0C8F  D7B50  07818
F2413  53108  AE781  AF1CC  ED5D0  14F9E  37080  85017
114F9  E3708  08511  7114F  9E370  80852  17114  F9E37
08085  31711  50016  0CD54  B81AF  18135  3488B  20145
8F2D7  6081A  F1014  11421  64808  C
```

CRC : # 7A2Ah (S et G)

Emploi de DIGITCONV

Ce programme est très simple d'emploi :

- Tout d'abord, chargez un son huit bits de votre ordinateur sur le HP48 (Ne tenez pas compte d'un message d'erreur éventuel lors de transferts par XModem);
- Posez l'objet sur la pile, il doit normalement être contenu dans une chaîne;
- Exécutez DIGITCONV. Quelques instants plus tard, vous obtenez un objet Library Data. Il s'agit de votre son codé dans un format utilisable par DIGITPLAY;
- Vous pouvez enfin effacer l'objet initial, celui-ci ne vous est plus d'aucune utilité.

Voilà, nous possédons notre son dans un format exploitable. Il va maintenant nous falloir le jouer, et cela à l'aide de DIGITPLAY.

DIGITPLAY **

DIGITPLAY est absolument élémentaire. Il lit chaque bit du son et émet ou n'émet pas un "click" sur le buzzer selon le cas. Un binary integer pris sur la pile lui permet de déterminer la vitesse à laquelle le son sera joué. En effet, celui-ci représente le nombre de clicks à effectuer pour chaque information. Par exemple, si la vitesse est # 5h, le programme agira comme suit :

- Si le bit en cours vaut 1, celui-là fait cinq clicks
- Si le bit vaut 0, le programme fait cinq OUT 0

La vitesse de lecture est donc d'autant plus grande que le Binary Integer est petit. Pour référence, une valeur comprise entre # 6h et # 8h permet de retrouver la vitesse du son sur micro.

```
%% PLAY
% R1a : Sauvegarde de la vitesse

GOSBVL 0679B
C=DAT1 A              % L'objet au niveau 1 est-il un Binary
                      %Integer ?
D0=C
A=DAT0 A
LC 02A4E
?C=A A
GOYES OK1

*BAT                  % Si le type est mauvais, on émet "Bad
                      %Argument Type"
LA 00202
GOSBVL 067D2
GOVLNG 05023

*OK1
```

```
D0=D0+ 5              % On lit le Binary Integer
C=DAT0 A
C=C-6 A
P=C 0
D0=D0+ 5

A=0 W
A=DAT0 WP
P= 0
R1=A A                % On sauvegarde sa valeur
D1=D1+ 5              % On vérifie maintenant le niveau 2
C=DAT1 A              %c'est bien un objet Library Data ?
D1=C
A=DAT1 A

LC 02B88
?C#A A
GOYES BAT
D1=D1+ 5              % On va lire sa longueur, pour savoir combien
C=DAT1 A              %d'échantillons il faut jouer
C=C-6 A
B=C A                 % Ba sera le compteur
D1=D1+ 5              % On pointe sur les échantillons
*BOUCLE
C=DAT1 1              % On lit le premier "ensemble" de quatre
                      %échantillons
?CBIT=0 0             % Selon le bit 0, on va émettre un "click"
GOYES PAS_PLAY0       %ou éteindre le haut-parleur
GOSUB PLAY
GOTO APRES0


*PAS_PLAY0
GOSUB PAS_PLAY
*APRES0
C=DAT1 1
?CBIT=0 1             % De même pour le bit 1
GOYES PAS_PLAY1
GOSUB PLAY
GOTO APRES1
*PAS_PLAY1
GOSUB PAS_PLAY

*APRES1
C=DAT1 1
?CBIT=0 2             % Puis pour le bit 2
GOYES PAS_PLAY2
GOSUB PLAY
GOTO APRES2

*PAS_PLAY2
GOSUB PAS_PLAY
*APRES2
C=DAT1 1
?CBIT=0 3             % Puis pour le bit 3
GOYES PAS_PLAY3
```

```
GOSUB PLAY
GOTO APRES3
*PAS_PLAY3
GOSUB PAS_PLAY
*APRES3
D1=D1+ 1              % On pointe sur échantillon suivant
B=B-1 A               % Sons restant à jouer
GONC BOUCLE           % C'est fini ?
GOSBVL 067D2          % Si oui, on sort en ôtant les deux objets
D1=D1+ 10             %de la pile
D=D+2 A
A=DAT0 A
D0=D0+ 5
PC=(A)

*PLAY                 La routine émettant un "click"
LC 800                %Le masque pour OUT
GOTO SON              % On appelle la routine qui fait les OUT

*PAS_PLAY             % La même chose pour ne pas émettre de
                      %"click"
C=0 X                 % Le masque 0 pour OUT

*SON                  % La routine effectuant R1a+1 OUT
A=R1 A
*BOUCLE_SON           % Totalement triviale, non ?
OUT=C
A=A-1 A
GONC BOUCLE_SON
RTN
@
```

CRC : # 8998h (S et G)

Programme assemblé

```
CCD20  E1100  8FB97  60147  13414  234E4  A208A  2A180
82420  2008F  2D760  8D320  50164  14681  8FA58  0D016
4AF01  52120  81AF0  11741  47135  14334  88B20  8A60B
17414  7818F  A5D51  7415F  0808A  0A07F  60670  07070
15F08  08A1A  07850  67007  95015  F0808  A2A07  14067
00724  015F0  808A3  A07A2  06700  7B201  70CD5  E98F2
D7601  79818  F3114  21648  08C32  00866  00AB2  81AF1
1801C  C5AF0  1
```

CRC : # FAD7h (S et G)

Utilisation de DIGITPLAY

Ce programme prend deux objets sur la pile :

- Au niveau 2 un son sous forme de Library Data
- Au niveau 1 la vitesse sous forme de Binary Integer

En guise d'exemple, vous pouvez charger le son RAIDERS.SON du répertoire DIGITSOUND de la disquette de programme. Celui-ci est déjà au format exploitable par DIGITPLAY. Posez-le sur la pile, tapez # 6h et lancez DIGITPLAY, vous devriez reconnaître un air célèbre.

Améliorations possibles de l'ensemble DIGITSOUND

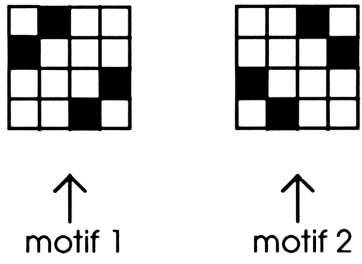
L'une des améliorations les plus évidentes est l'écriture de routines de conversion pour des sons de formats différents, comme par exemple les sons 16 bits des cartes sonores les plus récentes.

Une autre idée peut être la création de filtres lors de la conversion, pour éviter les grésillements par exemple. Il existe pour cela différents algorithmes pouvant être portés sur la HP48.

Enfin, l'idée la plus folle pourrait être la création d'un échantillonneur pour la HP48. Les micro-ordinateurs familiaux utilisent l'interface série pour cela. Il est donc envisageable d'utiliser celle de la machine pour le même résultat.

LABY *

Ce programme crée un labyrinthe aléatoire à partir de la superposition de deux motifs. Ce programme n'a pas de grand intérêt en lui-même mais il va peut être vous permettre de prendre conscience de certaines astuces de programmation. Ce programme utilise deux graphiques qui sont :



Ils seront mis dans le programme sous la forme d'un LC αβχδ

LABY a été écrit par Etienne De FORAS

```
GOSBVL 0679B
GOSBVL 01C31          % Fait pointer D0 sur le premier quartet de
                      %l'écran

D1= 00103             % D1 pointe sur l'adresse précédent le
                      %checksum
LC 0D                 % Sur 14 lignes de hauteur
D=C A
*B1
 LC 20                % Sur 33 colonnes
 B=C B
 *B2
  A=DAT0 B
  LC 2184             % Motif n°1
  A=DAT1 A            % Lecture dans le générateur de checksum
  A=A+A A             % On considère le bit 0 comme aléatoire
  GONC MOTIF1
```

```
 LC 4812              % Motif n°2
 *MOTIF1
 P= 3
 *B3                  % Ecriture des 4 quartets, les uns sur les
                      %autres
  DAT0=C P
  D0=D0+ 16
  D0=D0+ 16
  D0=D0+ 2
  P=P-1
 GONC B3
 P= 0

 AD0EX                % Met D0 sur le carré adjacent à écrire
 LC 00087
 A=A-C A
 D0=A

 B=B-1 B
GONC B2               % Caractère suivant en X

AD0EX                 % Met D0 sur le carré de la ligne suivante
LC 00067
A=A+C A
D0=A

D=D-1 B
GONC B1

GOVLNG 138B9          % Restauration des registres et retour au RPL

@
```

CRC : # BF2Ch (S et G)

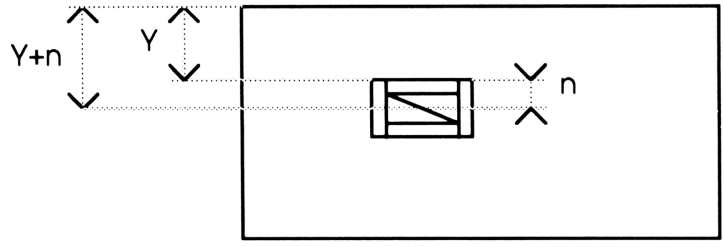
Il n'y a aucune amélioration notable pour ce programme.

```
CCD20  58000  8FB97  608F1  3C101  F3010  031D0  D7310
2AE51  4A334  81214  3C458  03321  84231  54016  F16F1
610D5  0F201  32347  8000E  A130A  6D50C  13234  76000
CA130  A6F54  A8D9B  831
```

CRC : # 9D83h (S et G)

ASM REPL ****

Une routine d'affichage n'est en réalité qu'une recopie du morceau de mémoire contenant l'objet à afficher dans la partie de mémoire contenant l'écran. Malheureusement les 2 graphiques (écran et objet à afficher) n'ont pas la même taille ; ce qui va obliger à recopier ligne à ligne la ligne n du graphique dans la ligne Y+n de l'écran.



Après cette première complication vient la deuxième. Un quartet contient 4 pixels (4 bits) ! Ce qui va obliger (pour afficher au pixel près) à décaler le grob à afficher de 0, 1, 2 ou 3 pixels, avant de le recopier.

Ce programme fait rebondir un grob 16*16 à l'écran.

Remarque : Il faut insérer l'équivalent d'un grob 16*16 dans le répertoire 'Macro' sous le nom **'TARGET'** (vous pouvez par exemple utiliser le programme **->MACRO**).

ASM REPL a été écrit par Cyrille de BREBISSON

```
%%%%%%%%%%%%%%%%%%%%
%     regs. using    %
%                    %
% R0 : adr ECR       %
% R1 : X             %
% R2 : Y             %
% R3 : Delta X       %
% R4 : Delta Y       %
%%%%%%%%%%%%%%%%%%%%

GOSBVL 0679B
GOSBVL 01115          % Inhibition des interruptions

%%%%%%%%%%%%%%%%%%%%%%%%
D1= (7050E : S        % Adresse de l'écran sur HP 48 S, SX
     8068D : G)       % Adresse de l'écran sur HP 48 G, GX
A=DAT1 A
R0=A A

A=0   A               % Mise à 0 de Aa
R1=A  A               % Mise à 0 de R1a
R2=A  A               % Mise à 0 de R2a
A=A+1 A               % Mise à 1 de Aa
R3=A  A               % Mise à 1 de R3a
R4=A  A               % Mise à 1 de R4a

GOSUB AFICHE          % On appelle la routine AFICHE

%%%%%%%%%%%%%%%%%%%%%%%%

*BOUCLE

LC 01000              % Je fais une pause pendant que le Graphique
                      %est allumé
*PAUSE
C=C-1 A
GONC PAUSE

GOSUB AFICHE          % Et j'efface le graphique en question

%%%%%%%%%%%%%%%%%%%%%%%%
A=R1 A
LC 00073
B=C A
C=R3 A
A=A+C A
R1=A A
?A#0 A
GOYES .PAS.GAUCHE
C=-C A
*.PAS.GAUCHE

?A#B A
GOYES .PAS.DROITE
C=-C A
*.PAS.DROITE
```

```
R3=C A

A=R2 A
LC 00030
B=C A
C=R4 A
A=A+C A
R2=A A
?A#0 A
GOYES .PAS.HAUT
C=-C A
*.PAS.HAUT

?A#B A
GOYES .PAS.BAS
C=-C A
*.PAS.BAS

R4=C A


%%%%%%%%%%%%%%%%%%%%%%%% Test de la touche DROP

LC 010
GOSBVL 01EEC
?CBIT=0 0
GOYES PAS_DROP
GOSBVL 067D2
A=DAT0 A
D0=D0+ 5
PC=(A)

*PAS_DROP

GOSUB AFICHE

GOTO BOUCLE


*AFICHE
C=R2 W
C=C+C A              % Multiplication par 22h qui doit devenir
                     %classique pour vous
B=C A                % Elle permet de trouver l'adresse de la
                     % Y^iemme ligne de écran
CSL A
C=C+B A

A=R1 W
B=A P                % mise en mémoire des 2 derniers bits de X
                     %pour le décalage de pixels
ASRB A
ASRB A
A=A+C A              % Aa = Y*22 + X/4, c'est à dire que x
                     % contient la distance en quartets entre le
                     %début de l'écran et le quartet contenant le
                     %premier pixel ou afficher.
```

```
C=R0 A
A=A+C A
D0=A                    % D0 pointe sur le quartet ou afficher

LC 3
C=C&B P                 % On ne garde que les 2 derniers bits de X,
                        % On module X par 4
P=C 0                   % Cs contient le décalage à effectuer en
                        %pixel
C=P 15
P= 0

GOSUB APRES.TARGET      % On saute les data du grob à afficher
/TARGET
*APRES.TARGET
 C=RSTK                 % On récupère leur adresse.

D1=C                    % D1 pointe sur le grob à afficher
LC 0F                   % On doit boucler 16 fois (attention à la
                        %conversion hexadécimale)
                        % On ne stocke pas 10h car le test de fin de
                        %boucle va se faire par le test de la
                        %retenue
D=C B                   % Db contient le compteur

*BOUCLE.AFICHE
A=0 A                   % mise à 0 de Aa pour que les pixels de plus
                        %que l'on affiche soit blancs (une ligne du
                        %grob fait 4q et on en affiche 5)
A=DAT1 4                % On lit une ligne du grob
B=C S
B=B-1 S                 % On décale pour être précis au pixel près,
                        %car sur 1 quartet on a 4 bits donc 4 pixels
GOC FIN.DECALE
A=A+A A                 % Décalage d'un pixel (multiplication par 2
                        %en binaire)
B=B-1 S
GOC FIN.DECALE
A=A+A A
B=B-1 S
GOC FIN.DECALE
A=A+A A                 % Au maximum on décale de 3 pixels
*FIN.DECALE

C=DAT0 A                % On lit le graphique à l'écran
B=C A
C=C!A A                 % On fait un XOR logique avec le graphique
                        %que l'on veut afficher
B=B&A A
C=C-B A

DAT0=C A                % On affiche le résultat à l'écran

D1=D1+ 4                % On passe à la ligne suivante sur le grob à
                        %afficher
D0=D0+ 16               %et sur l'écran aussi
D0=D0+ 16
```

```
D0=D0+ 2

D=D-1 B                 % Décrémentation du nombre de lignes qu'il
                        %reste à afficher
GONC BOUCLE.AFICHE      % Si le nombre était différent de 0 on
                        %recommence
RTN
% et voila vous avez fini la routine d'affichage !!!!!

@
```

CRC : # 88D3h (G) # 6D6Bh (S)

Modifications possibles de la routine d'affichage

Cette routine peut être modifiée pour être réemployée dans d'autres programmes. Les changements à faire sont les suivants :

Hauteur du Grob : Celle-ci est stockée dans Db. Il suffit simplement de modifier la valeur de ce registre.

Largeur du Grob : Il suffit de changer les instructions de lecture : ici, l'instruction **A=DAT1 4** lit quatre quartets soit seize pixels d'une ligne, les décalages sont effectués sur un champ A. Si l'on veut gérer des Grobs plus larges, il convient de passer sur un champ plus grand comme W. Attention cependant que ce genre de routine ne permet pas de gérer des graphismes d'une taille supérieure à 61 pixels, les décalages étant effectués directement dans les registres du microprocesseur.

Des animations sans image de fond de peu de sprites à la fois

Cette routine permet des animations à l'aide de la méthode suivante :

Afficher le ou les sprites, modifier les coordonnées et recommencer est la technique la plus simple, mais aussi la moins propre. Entre autre si l'on veut animer beaucoup de sprites, on va avoir l'impression qu'ils clignotent.

On crée un sprite avec 4 pixels blancs autour puis on l'affiche en REPL aux différentes positions.

La technique des XOR est employée dans ce programme. D'après la table de vérité des XOR faire X xor Y xor Y = X donc en affichant 2 fois un grob en XOR au même endroit on l'efface !!!

```
CCD20  1B100  8FB97  608F5  11101  FD860  81438  1AF00
D081A  F0181  AF02E  481AF  0381A  F047E  90340  0010C
E5DF7  E8081  AF113  43700  0D581  AF1BC  A81AF  018AC
40FA8  A440F  A81AF  0B81A  F1234  03000  D581A  F1CCA
81AF0  28AC4  0FA8A  440FA  81AF0  C3201  08FCE  E1080
8A031  8F2D7  60142  16480  8C740  0656F  11AC6  D5F2C
9111A  88819  F0819  F0CA8  1AF18  CA130  3030E  0580D
080CF  20704  0FFFF  FFFFF  FFFFF  FFFFF  FFFFF  FFFFF
FFFFF  FFFFF  FFFFF  FFFFF  FFFFF  FFFFF  FFFFF  07135
31F0A  E7D01  5B3AC  5A4D4  41C4A  4D4C0  C4A4D  440C4
146D5  0EFA0  EF4E9  14417  316F1  6F161  A6F5D   B01
```

CRC : # 4E0Eh (G)

```
CCD20  1B100  8FB97  608F5  11101  FE050  71438  1AF00
D081A  F0181  AF02E  481AF  0381A  F047E  90340  0010C
E5DF7  E8081  AF113  43700  0D581  AF1BC  A81AF  018AC
40FA8  A440F  A81AF  0B81A  F1234  03000  D581A  F1CCA
81AF0  28AC4  0FA8A  440FA  81AF0  C3201  08FCE  E1080
8A031  8F2D7  60142  16480  8C740  0656F  11AC6  D5F2C
9111A  88819  F0819  F0CA8  1AF18  CA130  3030E  0580D
080CF  20704  0FFFF  FFFFF  FFFFF  FFFFF  FFFFF  FFFFF
FFFFF  FFFFF  FFFFF  FFFFF  FFFFF  FFFFF  FFFFF  07135
31F0A  E7D01  5B3AC  5A4D4  41C4A  4D4C0  C4A4D  440C4
146D5  0EFA0  EF4E9  14417  316F1  6F161  A6F5D   B01
```

CRC : # D2B6h (S)

GRIS **

Ce programme va vous permettre de créer des images en quatre couleurs soit le blanc, le gris clair, le gris foncé et le noir. Pour fonctionner, ce programme prend comme argument sur la pile un "Graphic 131 * 128" sur la pile. Pour sortir du programme, il faudra actionner la touche DROP.

```
GOSBVL 0679B
INTOFF

LC 3F                  % En # 00128h est codé sur 2 bits la hauteur
D0= 00128              %des menus. Nous allons les faire disparaître
DAT0=C B               %en écrivant à cet endroit # 3Fh

A=DAT1 A               % On prend l'adresse de l'objet au premier
                       %niveau de la pile
A=A+10 A               % Cet objet est censé être un graphique, on
A=A+10 A               %passe par conséquent les 20 premiers
                       %quartets constituant sa structure

R0=A   A               % R0a contient l'adresse des données du
                       %graphique

LC 00880
C=C+A A

R1=C A                 % R1a pointe sur # 880h quartets plus loin
                       %soit le 2eme grob
?CBIT=0 0              % On teste si l'objet est en adresse paire
GOYES PAIRE            % On va changer les pointeur graphique en
                       %fonction de la parité ou de l'imparité du
                       %graphique
D0= 00125
LC FFF
DAT0=C X
D0= 00
LC D
DAT0=C 1
 GOTO PRE.BCL

*PAIRE

C=0 X
D0= 00125
DAT0=C X
```

```
D0= 00
LC 8
DAT0=C 1

*PRE.BCL

D0= 00120            % D0 pointe sur l'adresse du bitmap écran
D1= 00128            % D1 pointe sur l'adresse ou se trouve la VBL

*BCL.PRIN

GOSUB VSYNC          % On appelle la routine d'attente de VBL

C=R0 A
DAT0=C A             % On affiche le premier graphique

GOSUB VSYNC
C=R1 A
DAT0=C A             % On affiche le deuxième graphique

GOSUB VSYNC
C=R1 A
DAT0=C A             % On affiche le deuxième graphique

LC 010               % On teste si la touche DROP est enfoncée
GOSBVL 01EEC         %si c'est le cas on sort du programme
 ?CBIT=0 0
 GOYES PAS.OUT

INTON
GOSBVL 01C7F         % On rétablit les pointeurs graphiques
GOSBVL 067D2
A=DAT0 A
D0=D0+ 5
PC=(A)


 *PAS.OUT

GOTO BCL.PRIN

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

*VSYNC               % Routine d'attente de la VBL (voir les
                     explications après le programme)

C=DAT1 B
CBIT=0 6
CBIT=0 7
 ?C=0 B
 GOYES VSYNC

*VSYNC2

C=DAT1 B
CBIT=0 6
CBIT=0 7
```

```
 ?C#0 B
 GOYES VSYNC2

RTN

@
```

CRC : # 2C56 (S et G)

La syncro VBL

Pour créer des niveaux de gris vous devez attendre que le balayage de l'écran passe à 0. Si vous ne l'attendez pas vous n'obtiendrez que de très mauvais niveaux de gris. Ainsi pour savoir où en est le balayage de l'écran, on lit les 6 premiers bits en # 128h. C'est pour cette raison que dans le programme on va mettre à 0 les bits 6 et 7 pour pouvoir tester la nullité du balayage écran sur un champs B. Cependant il y a un petit problème lorsqu'on fait des niveaux de gris. L'affichage HARD étant si rapide que parfois entre deux affichages, le balayage de l'écran n'aura pas eu le temps de passer à 1. C'est pour cette raison que l'on teste tout d'abord que le balayage écran est différent de zéro puis si il est égal à 0.

Remarque : Vous pouvez utiliser cette routine pour faire des scrollings fluides.

Programme assemblé

```
CCD20  2F000  8FB97  60808  F31F3  1B821  0014C  14381
8F098  18F09  81AF0  03408  800C2  81AF0  9808A  0611B
52100  32FFF  15436  110AB  21B52  10015  431B0  21001
F8210  07F40  81AF1  81447  24081  AF191  44753  081AF
19144  32010  8FCEE  10808  A0718  0808F  2D760  14216
4808C  60BF1  4F808  86808  8796A  0F14F  80886  80887
96E0F  01
```

CRC : # F0AAh (S et G)

CRNAME **

Ce programme prend sur la pile une chaîne de caractères et la transforme en Global Name. Ce qui va vous permettre de créer des variables inaccessibles par l'utilisation "standard" de la machine.

CRNAME a été écrit par Stéphane DOAT

```
% D1= SUR LE Global Name
% D0= SUR LA CHAINE


GOSBVL 0679B
%%%%%%%%%%%%%%%%%%%%
C=DAT1 A
D1=C
RSTK=C                  % Vérification que l'objet au niveau 1 est
                        %une string
A=DAT1 A
LC 02A2C
?C=A    A
GOYES   OK_GLOBAL_NAME
A=0 A
GOVLNG 0357C
*OK_GLOBAL_NAME

%%%%%%%%%%%%%%%%%%%%
D1=D1+ 5
C=0 A                   % Taille de la string
C=DAT1 A
R1=C    A
%%%%%%%%%%%%%%%%%%%%
B=C     A
BSRB    A
B=B-2   A
A=0     A
A=-A-1 B
A=A+1   X               % Calcul de la taille du global name
?B<A    A               % Condition : nombre de caractères < 256
GOYES   OK_MAKE
A=0     A
A=-A-1 A
GOVLNG 0357C
*OK_MAKE
```

```
%%%%%%%%%%%%%%%%%%%%%%

C=C+2  A
GOSBVL 039BE          % RESROOM
%%%%%%%%%%%%%%%%%%%%%%

AD0EX
R0=A    A
AD1EX                 % Ecriture du prologue
LC 02E48
DAT1=C  A
D1=D1+  5
%%%%%%%%%%%%%%%%%%%%%%
A=R1    A
A=A-5   A
B=A     A
ASRB    A
DAT1=A  B
                      % Ecriture de la taille du Global Name +
                      %calcul de la taille de celui-ci
D1=D1+  2
%%%%%%%%%%%%%%%%%%%%%%%%
C=RSTK
D0=C
D0=D0+  10            % Ecriture des caractères
BCEX    A
GOSBVL  0670C
%%%%%%%%%%%%%%%%%%%%%%%%
*FIN
GOSBVL 067D2
A=R0   A
DAT1=A A
A=DAT0 A
D0=D0+ 5
PC=(A)

@
```

CRC : # 5304h (S et G)

La réservation de mémoire

Ce programme utilise une technique consistant à réserver de la mémoire de façon à écrire dessus des données qui pourrons éventuellement être mises sur la pile.

Programme assemblé (S et G)

```
CCD20  9C000  8FB97  60147  13506  14334  C2A20  8A2B0
D08DC  75301  74D21  4781A  F09D5  819F1  818F9  1D0BE
CB348  B0D0D  0FC8D  C7530  818F2  18FEB  93013  281AF
00133  3484E  20145  17481  AF118  18F84  D8819  F0149
17107  13416  9DD8F  C0760  8F2D76  081AF1  01411  42164
808C
```

CRC : # 9C5Ch (S et G)

ASM NEWOB **

Ce programme fait l'équivalent d'un NEWOB.

Ce programme a été écrit par Stéphane DOAT.

```
GOSBVL  0679B
C=DAT1  A
?C#0    A              % Vérifie le premier niveau de la pile
GOYES   OK_ONE_LEVEL
GOVLNG  05143          % Sortie sans toucher à la pile
*OK_ONE_LEVEL
RSTK=C                 % Sauvegarde de l'adresse de l'objet
D0=C
GOSBVL  03019          % NEXTOB dont le but est de calculer la
                       %taille de l'objet, étant donné que D0 pointe
                       %à la fin de celui-ci
AD0EX
C=RSTK
RSTK=C
C=A-C   A              % Calcul de la taille de l'objet
                       %(ADRESSE 1 - ADRESSE 2 = TAILLE)
RSTK=C
GOSBVL  039BE          % RESROM
AD0EX
D1=A
R0=A    A              % Sauvegarde de l'adresse de l'objet
C=RSTK
D0=C
C=RSTK
CD0EX
GOSBVL  0670C          % Copie de mémoire

A=R0    A
GOSBVL  067D2
DAT1=A  A
A=DAT0 A               % Ecriture de l'adresse de l'objet à la
D0=D0+ 5               %pile du premier niveau de la pile
PC=(A)                 % Retour au RPL
@
```

Ce programme n'utilise aucune notion nouvelle. Un newob n'est en fait qu'une recopie d'un objet à un autre endroit de la mémoire.

Programme assemblé (S et G)

```
CCD20  07000  8FB97  60147  8AE90  8D341  50061  348F9
10301  32070  6EE06  8FEB9  30132  13181  AF000  71340
71368  FC076  081AF  108F2  D7601  41142  16480  8C
```

CRC : # 1E17h (S et G)

HPTRON ****

HPTron est un petit jeu très intéressent du point de vue de la programmation. Il utilise des scrollings hard et un affichage hard. Ce logiciel ne nécessite pas de graphisme car il utilise uniquement des affichages de pixels.

Le but du jeu est simple. Vous êtes en déplacement constant et vous devez éviter les traînées noires que vous laissez derrière vous. Les touches sont :

2 : Bas

8 : Haut

4 : Gauche

6 : Droite

HPTRON a été écrit par Cyrille de BREBISSON.

```
%%%
```

```
CBIT=0 0                %je réserve un quartet de plus pour pouvoir
R0=C A                  %afficher sur une adresse paire )


% J'efface l'écran
D1=C
D0=C
LC 3FF
A=0 W
*EF

DAT1=A W
D1=D1+ 16

C=C-1 X
 GONC EF

                        % Je mets un carré autour de mon écran,
                        %remarquez que je le fait en déplaçant mon
                        %pointeur de gauche à droite, puis de haut en
                        %bas puis de droite à gauche et enfin de bas
                        %en haut. De plus j'emploie 2 systèmes pour
                        %faire mes boucles: un compteur sur Cb et un
                        %compteur sur P
A=A-1 W
P= 3

*Haut
DAT0=A W
D0=D0+ 16
P=P-1

 GONC Haut

P= 0

D0=D0- 1
LC FFE

*Droite

DAT0=C XS
D0=D0+ 16
D0=D0+ 16
D0=D0+ 16
D0=D0+ 16

C=C-1 B

GONC Droite
 D0=D0+ 1

P= 3
*Bas

D0=D0- 16
DAT0=A W
P=P-1
```

```
GONC Bas
P= 0

LC FFE

*Gauche

DAT0=C XS
D0=D0- 16
D0=D0- 16
D0=D0- 16
D0=D0- 16


C=C-1 B
 GONC Gauche

GOSBVL 01115          % Interdiction des interruptions

D0= 00128             % Je descends les menus
LC F
DAT0=C 1

                      % Initialisation des X, Y,
LC 00080
C=0 S
R1=C W
R2=C A

                      % Ma boucle principale
*GROSSE_BOUCLE
GOSUB ECRAN
GOSUB AVANCE
GOSUB PIXON
GONC Pas_Perdu
GOSBVL 010E5
GOSBVL 01C7F          % On rétablit l'écran
GOSBVL 067D2          % On sort
A=DAT0 A
D0=D0+ 5
PC=(A)

*Pas_Perdu

GOSUB TESTE_TOUCHE
LC 01000

*Pause
C=C-1 A
GONC Pause

GOTO GROSSE_BOUCLE


%%%%%%%%%%%%%%%%%%%%%%

*ECRAN
```

```
                        % Calcul des pointeurs écran en Hard
A=R1 A                  % Le pixel est ( si possible ) au milieu de
                        %l'écran. Donc je calcule les coordonnées du
LC 41                   %pixel en haut à droite de écran visible en
A=A-C B                 %fonction des coordonnées de mon pixel.
GONC Pas_Sur_La_Gauche
 A=0 A
*Pas_Sur_La_Gauche
LC 7D
?A<=C B
GOYES Pas_Sur_La_Droite
 A=C B
*Pas_Sur_La_Droite
 B=A A

A=R2 A                      % Idem mais pour le Y
LC 20
A=A-C B
GONC Pas_En_Haut
 A=0 A
*Pas_En_Haut
LC C0
?A<=C B
GOYES Pas_En_Bas
 A=C B
*Pas_En_Bas


%%%%%%%%%%%%%%%%%%  Calcul de l'adresse du pixel

ASL A
A=A+A A
A=A+A A
C=B A
CSRB A
CSRB A
C=C+A A
A=R0 A
C=C+A A
D0= 00120
DAT0=C A


%%%%%%%%%%%%%%%%%%  Calcul du décalage ( Marge à droite )


LC 7
C=C&B P
CBIT=1 3            % Ecran Allumé
D0= 00
DAT0=C 1

%%%%%%%%%%%%%%%%%%  Calcul de la marge à gauche
LA 01E
?CBIT=0 2           % Pas en adresse paire
GOYES Pas_Paire
A=A-1 B
```

```
*Pas_Paire
D0= 25
DAT0=A X
RTN
%%%%%%%%%%%%%%%%%%

*PIXON

A=R2 A                  % Pour faire un pixon, il faut calculer
ASL A                   %l'adresse du quartet contenant le bit a
A=A+A A                 %mettre à 1 puis déterminer quel est le bit
A=A+A A                 %du quartet à mettre à 1
C=R1 A
B=C P
CSRB A
CSRB A
A=A+C A
C=R0 A
A=A+C A
D0=A                    % D0 pointe sur le q en question
                        %Et la, attention les yeux, infâme bidouille
                        %pour charger le bit a mettre a 1 dans Cp en
                        %fonction du décalage

LC 3
C=C&B P
C=-C P
P=C 0
LC 8421
P= 0


%%%% Et ici, je mets réellement le bit à 1 en mémoire


A=DAT0 B
B=A P
A=A!C P
DAT0=A B

                        % Puis, je teste si le pixel était allumé (
                        %pour renvoyer la carry le cas échéant )
B=B&C P
?B#0 P
RTNYES
RTN



%%%%%%%%%%%%%%%%%%%%%%


*AVANCE

                        % Modifie les coordonnées dans R1 et R2 en
                        %fonction de la direction donnée dans R1s

A=R1 W
```

```
A=A-1 S
GOC Dir_Haut
A=A-1 S
GOC Dir_Bas
A=A-1 S
GOC Dir_Droite
                        % Dir_Gauche
A=A+1 A
R1=A A
RTN

*Dir_Droite
A=A-1 A
R1=A A
RTN

*Dir_Haut
A=R2 A
A=A-1 A
R2=A A
RTN

*Dir_Bas
A=R2 A
A=A+1 A
R2=A A
RTN


%%%%%%%%%%%%%%%%


*TESTE_TOUCHE
                        % Je teste les touches et modifie la
                        %direction le cas échéant
C=R1 S
LC 004                  % Test de la touche 6
GOSBVL 01EEC
?CBIT=0 1
GOYES Pas_Droite
P= 3
C=P 15
P= 0
*Pas_Droite

?CBIT=0 3               % Test de la touche 4
GOYES Pas_Gauche
P= 2
C=P 15
P= 0
*Pas_Gauche

LC 002
GOSBVL 01EEC
?CBIT=0 2
GOYES Pas_Bas
P= 1
```

```
C=P 15
P= 0
*Pas_Bas

LC 008
GOSBVL 01EEC
?CBIT=0 2
GOYES Pas_Haut
C=P 15
*Pas_Haut
R1=C S
RTN
@
```

CRC : # C9E0h (S et G)

Commentaires

TRON s'avère particulièrement intéressant par les techniques qu'il met en jeu, techniques réutilisables dans d'autres programmes :

Scrolling Hard avec l'emploi de la marge à gauche et de la marge à droite.

Gestion de pixels.

Modifications possible du programme

Il peut être apporté de nombreuses améliorations comme :

Gestion d'un deuxième joueur, au clavier ou par la machine ;

Calculs de scores, par exemple le nombre de pixels allumés avant de perdre ;

Amélioration de la gestion des touches, possibilité de se déplacer en diagonale, possibilité de se déplacer à

plusieurs vitesses (et donc de faire monter plus ou moins vite le score)...

Programme assemblé (S et G)

```
CCD20  FA200  8FB97  60341  00408  FD7B5  0136E  68088
081AF  08135  13432  FF3AF  01517  17FA3  E55FA  7C231
50716  F0D56  F2018  032EF  F1542  16F16  F16F1  6FA6E
5CE16  02318  F1507  0D56F  2032E  FF154  218F1  8F18F
18FA6  E5CE8  F5111  01B82  10030  F15C0  34080  00AC2
10981  AF0A7  E3078  217AC  05128  F5E01  08FF7  C108F
2D760  14216  4808C  77413  40001  0CE5D  F61CF  81AF1
13114  B6A54  0D031  D79EA  50AEA  D881A  F1231  02B6A
540D0  310C9  EA50A  EAF0C  4C4D9  819F2  819F2  C281A
F10C2  1B021  00144  3070E  05808  93190  015C0  80822
E1080  8A250  A6C19  52150  30181  AF12F  0C4C4  81AF1
9A858  19F28  19F2C  A81AF  18CA1  30303  0E05B  8A80D
03312  48201  4AA88  0E0E1  480E0  190D0  00111  1A4C4
22A4C  4C2A4  C4C0E  481AF  0101C  C81AF  C1018  1AF12
CC81A  F0201  81AF1  2E481  AF020  181A4  19324  008FC
EE108  08A1A  02380  CF208  08A3A  02280  CF203  22008
FCEE1  0808A  2A021  80CF2  03280  08FCE  E1080  8A260
80CF8  1A409  01
```

CRC : # 98FBh (S et G)

LA BEBETE *****

La Bébète est un jeu d'arcade stratégie en quatre niveaux de gris. Le but est d'enfermer la vilaine Bébète entre quatre murs afin de lui empêcher tout déplacement. Pour ce faire, vous pouvez vous déplacer suivant les quatre directions (Haut ; Bas ; Gauche ; Droite) et pousser les murs à votre guise. La Bébète, elle, peut se déplacer suivant les quatre directions, et les diagonales si le passage est possible. Elle se déplace deux fois plus lentement que vous et son contact est mortel. Le jeu s'achève d'une des deux façons suivantes :

-La Bébète vous touche, vous avez perdu.

-Vous enfermez la Bébète entre quatre murs (Haut ; Bas ; Gauche ; Droite), vous gagnez.

Pendant le jeu, les touches sont :

- DROP : Abandon,
- 8 : Haut,
- 2 : Bas,
- 4 : Gauche,
- 6 : Droite.

Vous commencez en haut, à gauche. La Bébète commence elle en bas, à droite.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  R0a : @ Ecran                 %
%  R1a : @ Buffer Ecran          %
%  R2a : @ Table                 %
%  R3b : Joueur X 0-31           %
%  R3s : Joueur Y 0-15           %
%  R4b : Bébète    X 0-31        %
%  R4s : Bébète    Y 0-15        %
%                                %
%  BUFFER Rs232c                 %
%  80400 : @Joueur (G,GX)        %
%  70300 : @joueur (S,SX)        %
%  70305 : @Bète   (S,SX)        %
%  80405 : @Bète   (G,GX)        %
%                                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

GOSBVL 0679B
GOSBVL 01115
LC 02401              % Je réserve de la mémoire pour quatre
                      %écrans, pour une table
GOSBVL 05B7D          % 16*32. Un quartet de plus au cas où l'écran
                      %serait à une adresse impaire.
AD0EX                 % On place l'écran à une adresse paire
A=A+1 A
ABIT=0 0
R0=A A                % R0a contient l'adresse du premier écran 4
                      %gris (2 Ecrans)
LC 01100
A=A+C A
R1=A A                % R1a pointe sur le deuxième écran 4 Gris
C=C+A A
R2=C A                % R2a pointe sur la table.
D0=C
D1= 00128             % On efface les menus.
LC 3F
DAT1=C B              % On va installer les valeurs de départ dans
                      %cette table
GOSUBL APRES_INIT
$2000110000000001
$0010100000000011
$0000000010001101
$0100100011000000
$1011010000000110
$0000010111000010
$1000000010000100
$0000110000000000
$0000000000100000
$0000010110000000
$1001000000000000
$0001100001011000     % Codage : 0 : Vide,1 : Mur, 2 : Joueur, 3 :
$0001010000100000     %Bébète
$0000100000001110
$0000000000000000     % De manière générale, plutôt que d'utiliser

$1000000000000000     %$ et les données, utiliser des Macros,
$0000100000000000     %beaucoup plus lisibles.
```

```
$0000110100100000    % Le $ est employé ici pour vous permettre la
$0000000000000000    %saisie.
$0001000000000001
$0100001000100000
$0000001000000010
$0010000000000100
$0000100001000000
$1000000000000100
$0001000011100000
$0000000000000000
$0001100010000011
$0010000000000010
$0100000001000000
$0000100000001000
$0110000010000003
*APRES_INIT          % Ici, on recopie les données dans la table
C=RSTK
D1=C
LC 1FF
*Initialise
 A=DAT1 1
 DAT0=A 1
 D0=D0+ 1
 D1=D1+ 1
 C=C-1 X
GONC Initialise      % Voilà, c'est fait, on va maintenant
                     %procéder à une deuxième
A=R0 A               % Initialisation : on efface la dernière
                     %colonne visible de l'écran
LC 00020             % Où il ne sera jamais rien affiché, pour la
                     %finition.
A=A+C A
D0=A
LC FF
A=0 P
*Init2               % Voilà la boucle qui installe des 0 sur les
                     %quatre écrans
 DAT0=A P
 D0=D0+ 16
 D0=D0+ 16
 D0=D0+ 2
 C=C-1 B
GONC Init2
C=0 S                % Maintenant, on initialise les positions du
                     %joueur et de la Bébète.
C=0 B
R3=C W
C=C-1 S
LC 1F
R4=C W
D0= (80400 : G
     70300 : S)      % Et on sauvegarde l'adresse dans la table
                     %contenant leurs positions respectives
A=R2 A
DAT0=A A
LC 001FF
C=C+A A
```

```
D0=D0+ 5
DAT0=C A
ST=0 0                  % On initialise les deux drapeaux : 0 pour
                        %savoir si le jeu est fini, 1 pour savoir si
                        %la Bébète doit se déplacer
ST=0 1

*GROSSE_BOUCLE          % La boucle principale.
GOSUB DESSINE_ECRAN     % Remarquez que, pour faciliter la lecture,
GOSUB PERMUTE           %chaque tâche est gérée par une sous-routine,
GOSUB AFFICHE_SCREEN    %la boucle principale étant constituée
GOSUB FIN?              %uniquement d'appels à ces sous-routines.
GOSUB TEST
GOSUB DEPLACE

?ST=0 0                 % Voici donc l'utilisation du drapeaux 0, on
                        %reboucle
GOYES GROSSE_BOUCLE     %uniquement s'il est toujours à 0. Sinon,
                        %c'est la fin
GOSBVL 01C7F            % Initialisation de l'écran
GOSBVL 010E5            % On ré-autorise les interruptions
GOSBVL 067D2            % Et on sort
A=DAT0 A
D0=D0+ 5
PC=(A)

*PERMUTE                % 1ere sous-routine qui permute l'écran
                        %visible et le buffer
C=R0 A                  % Remarquez qu'aucun affichage n'est
                        %effectué. On se contente
CR1EX A                 %de changer les références. La routine
                        %d'affichage s'occupera d'afficher le bon
                        %grob.
R0=C A
RTN

*AFFICHE_SCREEN         % Voici la routine d'affichage,
                        %relativement "bizarre".
D0= 00120               % On affiche le premier écran (Gris clair).
A=R0 A
DAT0=A A
LC 00880                % On pointe sur le second
A=A+C A
D0= 28                  % On attend une VBL
GOSUB WAIT0
GOSUB WAIT1

D0= 00120               % On affiche le second écran. Et on s'en
                        %retourne. Cela signifie qu'il va falloir
DAT0=A A                %occuper le processeur pendant deux
RTN                     %VBLs avant de rappeler cette routine.

*WAIT0                  % Sous-routine attendant la ligne 0
C=DAT0 B
CBIT=0 6
CBIT=0 7
?C#0 B
```

```
GOYES WAIT0
RTN

*WAIT1                % Deuxième sous-routine attendant la ligne 1
C=DAT0 B
CBIT=0 6
CBIT=0 7
?C=0 B
GOYES WAIT1
RTN

*DESSINE_ECRAN        % La routine d'affichage principale. Elle
                      %trace la zone de jeu en fonction des données
C=R1 A                %de la table. Elle trace sur le buffer
D0=C                  %écran, qui sera affiché une fois le tracé
                      %fini
C=R2 A
D=C A
B=0 S                 % On initialise les compteurs, Bs pour les
                      %colonnes Bb pour les lignes.
B=B-1 S               % On va tracer un écran après l'autre,
                      %colonne par colonne
LC 1F
B=C B
*BOUCLE_ECRAN_1       % Première sous-routine traçant l'écran "gris
                      %clair"
C=D A
D=D+1 A
D1=C
A=DAT1 P
GOSUB APRES_PIECES_1
$0000                 % Les données graphiques. Chaque ligne
                      %correspond
$F99F                 %à une pièce : Blanc, mur, joueur, Bébète
$6996
$FFFF
*APRES_PIECES_1
C=RSTK
*BOUCLE_POSITIONNE_1
A=A-1 P
GOC FINI_1
C=C+4 A
GOTO BOUCLE_POSITIONNE_1
*FINI_1
D1=C
C=DAT1 4
P= 3
*BOUCLE_AFFICHE_1
DAT0=C P
D0=D0+ 16
D0=D0+ 16
D0=D0+ 2
P=P-1
GONC BOUCLE_AFFICHE_1
P= 0
B=B-1 S
```

```
GONC BOUCLE_ECRAN_1  % On finit la colonne puis on pointe sur la
                     %suivante
CD0EX                % puis on pointe sur la suivante
LA 0087F
C=C-A A
RSTK=C
LC 10                % Ici, on regarde combien de colonnes on a
                     %afficher.
?C#B B               % Si c'est 16 (#10h), on a travaillé pendant
                     %deux VBLs, il faut
GOYES PAS_AFF_1      %donc appeler la routine d'affichage afin
                     %d'"entretenir" les
GOSUB AFFICHE_SCREEN %niveaux de gris

*PAS_AFF_1
C=RSTK
D0=C
B=B-1 B
GONC BOUCLE_ECRAN_1
GOSUB AFFICHE_SCREEN % De même, on va refaire un affichage
A=R1 A               % On point sur le deuxième écran
LC 00880
A=A+C A
D0=A
C=R2 A               % On repointe sur le début de la table
D=C A
B=0 S                % On ré-intialise les compteurs
B=B-1 S
LC 1F
B=C B

*BOUCLE_ECRAN_2      %et on recommence le traçage sur le deuxième
                     %Grob
C=D A
D=D+1 A
D1=C
A=DAT1 P
GOSUB APRES_PIECES_2
$0000
$FEE0
$6996
$F99F
*APRES_PIECES_2
C=RSTK
*BOUCLE_POSITIONNE_2
A=A-1 P
GOC FINI_2
C=C+4 A
GOTO BOUCLE_POSITIONNE_2
*FINI_2
D1=C
C=DAT1 4
P= 3
*BOUCLE_AFFICHE_2
DAT0=C P
D0=D0+ 16
D0=D0+ 16
```

```
D0=D0+ 2
P=P-1
GONC BOUCLE_AFFICHE_2
P= 0
B=B-1 S
GONC BOUCLE_ECRAN_2
CD0EX
LA 0087F
C=C-A A
RSTK=C
LC 10
?C#B B
GOYES PAS_AFF_2
GOSUB AFFICHE_SCREEN
*PAS_AFF_2
C=RSTK
D0=C
B=B-1 B
GONC BOUCLE_ECRAN_2
RTN

*DEPLACE              % La routine qui gère les déplacements de la
                      %Bébète
?ST=1 1               % Si le drapeau 1 est à 1, on déplace
GOYES OK_DEPLACE      %sinon, on le met à un pour déplacer au
                      %prochain appel
ST=1 1
RTN
*OK_DEPLACE           % En cas de déplacement
ST=0 1                % Tout d'abord, on passe le drapeau à 0, pour
                      %le prochain appel
A=R3 S                % Première comparaison : Le joueur est-il
                      %plus haut que la Bébète
C=R4 S                %SI oui, il faudrait que la Bébète monte
?C<=A S
GOYES PAS_HAUT_2      % Sinon, on laisse tomber le déplacement vers
                      %le haut
D0= (80405 : G
     70305 : S)       % On va voir si la case du dessus est libre
C=DAT0 A
C=C-1 A
D1=C
A=DAT1 P
?A#0 P
GOYES PAS_HAUT_2      % Si elle n'est pas libre, on laisse tomber
C=C-1 S               % Sinon, on modifie les coordonnées, la table
                      %et la sauvegarde
R4=C S                % dans le Buffer Rs232 pour que la Bébète
                      %soit une case plus haut
LA 03
DAT1=A B
DAT0=C A
*PAS_HAUT_2
A=R3 S                % Ici, on fait de même vers le bas
C=R4 S
?C>=A S
```

```
GOYES PAS_BAS_2
D0= (80405 : G
     70305 : S)

C=DAT0 A
D1=C
C=C+1 A
D1=C
A=DAT1 P
?A#0 P
GOYES PAS_BAS_2
C=C+1 S
R4=C S
LA 30
D1=D1- 1
DAT1=A B
DAT0=C A
*PAS_BAS_2           % Maintenant, c'est vers la droite
A=R3 B
C=R4 B
?C>=A B
GOYES PAS_DROITE_2
B=C B
D0= (80405 : G
     70305 : S)
C=DAT0 A
C=C+16 A
D1=C
A=DAT1 P
?A#0 P
GOYES PAS_DROITE_2
BCEX B
C=C+1 B
R4=C B
BCEX B
LA 3
DAT1=A P
D1=D1- 16
A=0 P
DAT1=A P
DAT0=C A
*PAS_DROITE_2        % Enfin, vers la gauche
A=R3 B
C=R4 B
?C<=A B
GOYES PAS_GAUCHE_2
B=C B
D0= (80405 : G
     70305 : S)
C=DAT0 A
C=C-16 A
D1=C
A=DAT1 P
?A#0 P
GOYES PAS_GAUCHE_2
BCEX B
```

```
C=C-1 B
R4=C B
BCEX B
LA 3
DAT1=A P
D1=D1+ 16
A=0 P
DAT1=A P
DAT0=C A
*PAS_GAUCHE_2
RTN

*TEST                % Routine gérant les touches et les
                     %déplacements du joueur.
LC 010               % Tout d'abord, la touche DROP
GOSBVL 01EEC
?CBIT=0 0
GOYES PAS_DROP
ST=1 0               % Le drapeau de la fin
*PAS_DROP
LC 002               % Ensuite la touche 2
GOSBVL 01EEC
?CBIT=0 2
GOYES PAS_BAS
C=R3 S               % On va voir si on peut descendre.
C=C+1 S              % On vérifie tout d'abord qu'on n'est pas au
                     %bord
GOC PAS_BAS
D0= (80400 : G       % La boucle suivante vérifie qu'on peut se
     70300 : S)      %déplacer.

C=DAT0 A             % Le déplacement peut être impossible pour
                     %deux raisons : Il y a des murs jusqu'au bord
C=C+1 A              % On ne peut donc pas les pousser plus loin

B=C A
D1=C                 % Ou bien, il y a la Bébète derrière les
                     %murs. On ne peut pas pousser non plus
*BOUCLE_BAS
A=DAT1 P
?A=0 P
GOYES FIN_BOUCLE_BAS
LC 3
?A=C P
GOYES PAS_BAS
C=C+1 S
GOC PAS_BAS
D1=D1+ 1
GOTO BOUCLE_BAS
*FIN_BOUCLE_BAS      % Si on arrive là, le déplacement est
                     %possible. On déplace alors le joueur ainsi
C=R3 S               %que tous les murs qui doivent l'être.
C=C+1 S
R3=C S
C=DAT0 A
D1=C
C=B A
```

```
DAT0=C A
A=0 P
*BOUCLE_BAS_2          % Voici la boucle qui déplace tout.
C=DAT1 P
?C=0 P
GOYES FINI_BAS
DAT1=A P
D1=D1+ 1
A=C P
GOTO BOUCLE_BAS_2
*FINI_BAS              % On a fini, puisqu'on a le droit qu'à un
                       %déplacement par appel, on arrête les tests.
DAT1=A P
RTN
*PAS_BAS
LC 008                 % Ici, on a la même chose vers le haut
GOSBVL 01EEC
?CBIT=0 2
GOYES PAS_HAUT
C=R3 S
C=C-1 S
GOC PAS_HAUT
D0= (80400 : G
     70300 : S)

C=DAT0 A
C=C-1 A
B=C A
D1=C
*BOUCLE_HAUT
A=DAT1 P
?A=0 P
GOYES FIN_BOUCLE_HAUT
LC 3
?A=C P
GOYES PAS_HAUT
C=C-1 S
GOC PAS_HAUT
D1=D1- 1               % On décrémente D1
GOTO BOUCLE_HAUT
*FIN_BOUCLE_HAUT
C=R3 S
C=C-1 S
R3=C S
C=DAT0 A
D1=C
C=B A
DAT0=C A
A=0 P
*BOUCLE_HAUT_2
C=DAT1 P
?C=0 P
GOYES FINI_HAUT
DAT1=A P
D1=D1- 1
A=C P
```

```
GOTO BOUCLE_HAUT_2
*FINI_HAUT
DAT1=A P
RTN
*PAS_HAUT
LC 004                  % Puis vers la gauche.
GOSBVL 01EEC
?CBIT=0 3
GOYES PAS_GAUCHE
C=R3 B
C=C-1 B
D=C B
GOC PAS_GAUCHE
D0= (80400 : G
     70300 : S)
C=DAT0 A
C=C-16 A
B=C A
D1=C
*BOUCLE_GAUCHE
A=DAT1 P
?A=0 P
GOYES FIN_BOUCLE_GAUCHE
LC 3
?A=C P
GOYES PAS_GAUCHE
D=D-1 B
GOC PAS_GAUCHE
D1=D1- 16
GOTO BOUCLE_GAUCHE
*FIN_BOUCLE_GAUCHE
C=R3 B
C=C-1 B
R3=C B
C=DAT0 A
D1=C
C=B A
DAT0=C A
A=0 P
*BOUCLE_GAUCHE_2
C=DAT1 P
?C=0 P                  % On teste si Cp est égal à 0
GOYES FINI_GAUCHE
DAT1=A P
D1=D1- 16
A=C P
GOTO BOUCLE_GAUCHE_2
*FINI_GAUCHE
DAT1=A P
*PAS_GAUCHE
LC 004                  % Et enfin, la droite
GOSBVL 01EEC
?CBIT=1 1
GOYES DROITE
GOLONG PAS_DROITE
*DROITE
```

```
C=R3 B
C=C+1 B
D=C B
LC 20
?C=D B
GOYES PAS_DROITE
D0= (80400 : G
     70300 : S)
C=DAT0 A
C=C+16 A
B=C A
D1=C
*BOUCLE_DROITE
A=DAT1 P
?A=0 P
GOYES FIN_BOUCLE_DROITE
LC 3
?A=C P
GOYES PAS_DROITE
D=D+1 B
LC 20
?C=D B
GOYES PAS_DROITE
D1=D1+ 16            % On incrémente D1 de 16
GOTO BOUCLE_DROITE
*FIN_BOUCLE_DROITE
C=R3 B
C=C+1 B
R3=C B
C=DAT0 A
D1=C
C=B A
DAT0=C A
A=0 P
*BOUCLE_DROITE_2
C=DAT1 P
?C=0 P
GOYES FINI_DROITE
DAT1=A P
D1=D1+ 16
A=C P
GOTO BOUCLE_DROITE_2
*FINI_DROITE
DAT1=A P
*PAS_DROITE
RTN

*FIN?                % La routine vérifiant la Bébète. Si autour
D0= (80405 : G       %se trouve le joueur, on a perdu.
     70305 : S)
                     % Bp compte le nombre de murs autour, si ce
C=DAT0 A             %compteur est à quatre, la bébète est
                     %enfermée, on a gagné.
D0=C
B=0 P
LC 1
```

```
D=C P
C=C+1 P
A=C P
D0=D0+ 1
C=DAT0 P
?C=A P
GOYES FIN
?C#D P
GOYES PAS_1
B=B+1 P
*PAS_1
D0=D0- 2
C=DAT0 P
?C=A P
GOYES FIN
?C#D P
GOYES PAS_2
B=B+1 P
*PAS_2
D0=D0- 15
C=DAT0 P
?C=A P
GOYES FIN
?C#D P
GOYES PAS_3
B=B+1 P
*PAS_3
D0=D0+ 16
D0=D0+ 16
C=DAT0 P
?C=A P
GOYES FIN
?C#D P
GOYES PAS_4
B=B+1 P
*PAS_4
LC 4
?B=C P
GOYES FIN
RTN

*FIN                  % La routine de fin.
ST=1 0                % On passe le drapeau à 1 pour interrompre la
                      %Grosse_Boucle
D0= 0010B             % On allume tous les indicateurs pour
                      %indiquer la fin
LC BF
DAT0=C B
GOSUB WAIT_NO_TOUCHE % On attend que toutes les touches soit
                      %relâchées
GOSUB WAIT_TOUCHE     % On attend une touche
GOSUB WAIT_NO_TOUCHE % Et on attend qu'elle soit relachée
RTN
*WAIT_NO_TOUCHE       % Routine attendant que les touches soient
                      %relachées.
GOSUB TOUCHE
```

```
?C#0 A
GOYES WAIT_NO_TOUCHE
RTN

*WAIT_TOUCHE          % Routine attendant une touche.
GOSUB TOUCHE
?C=0 A
GOYES WAIT_TOUCHE
RTN
*TOUCHE               % Sous routine des deux précédentes.
GOSUB AFFICHE_SCREEN % Elle réaffiche les niveaux de gris, puis
D0= 00128             %teste les touches
GOSUB WAIT0           % On appelle ces routines pour attendre les
GOSUB WAIT1           %deux VBLs entre deux affichages
GOSUB WAIT0
GOSUB WAIT1
LC 001FF
GOSBVL 01EEC
RTN
@
```

Commentaires

Ce programme est très instructif car il présente de nombreuses méthodes :

Gestion de niveaux de gris dans un programme : Nous l'avons déjà vu, pour afficher un Grob en niveaux de gris, on le décompose en deux écrans. Le premier est affiché pendant une VBL, le second pendant deux VBLs. La technique utilisée ici est la suivante : On affiche le premier Grob, on attend une VBL et on met alors le second Grob à l'écran. On a alors deux VBLs à notre disposition pour les calculs. La routine devra cependant être obligatoirement être rappelée avant le deuxième balayage sous peine de clignotements intempestifs. La méthode convient ici car le programme effectue relativement peu d'opérations. Celle-là pourrait cependant être affinée en utilisant la VBL d'affichage du premier Grob pour des calculs, plutôt qu'attendre que celle-ci ait lieu. On aura alors quasiment tout le temps machine à notre disposition.

Le DOUBLE BUFFER : Même si elles sont très rapides en assembleur, les opérations d'affichage ne sont pas instantanées. Cela conduit souvent à des désagréments tels que les

clignotements de sprites... Pour remédier à ces problèmes, on utilise le Double Buffer. La technique consiste à effectuer toutes les opérations d'affichage sur un deuxième Grob invisible à l'observateur puis, lorsque tout est mis en place, à rendre cet écran visible. On évite alors tout désagrément car les opérations de traçage sont restées invisibles.

Utilisation des pointeurs pour créer de nouvelles "Variables" : Le programme utilise tous les registres disponibles sur le Microprocesseur, cela ne lui est cependant pas suffisant. On crée alors de nouvelles variables à l'aide des pointeurs : Dans une zone inutile au programme, on va écrire nos variables, sur la longueur nécessaire. On utilise pour cela généralement le Buffer Rs232c, dont les données nous importent peu si l'on n'effectue aucun transfert. On a donc à notre disposition une zone de 512q pour les sauvegardes. Ici, par exemple, on utilise 5 quartets en # 80400h et en # 80405h pour sauvegarder deux adresses. Il va de soi que le Buffer Rs232c n'est pas la seule possibilité, on peut par exemple envisager une String vierge de longueur connue dans laquelle on va écrire. Attention cependant que dans ce cas, l'objet soit bien en Ram et non sur une carte protégée en écriture.

Gestion d'une table : Tout, dans ce jeu s'articule autour d'une table contenant toutes les informations nécessaires aux calculs de positions, de mouvements... Ce genre de gestion est toujours intéressant car il permet de remplacer toutes les opérations d'affichage par des calculs sur la table. L'affichage est à chaque fois entièrement refait. Cela permet de plus d'avoir des repères de temps constants entre deux événements, les opérations graphiques étant toujours les mêmes.

Gestion d'une intelligence artificielle : Il s'agit souvent d'une des opérations les plus délicates, il faut en effet s'arranger pour que l'ennemi réagisse de façon cohérente. Il faut comprendre que l'intelligence recherchée n'est pas toujours compliquée, et il faut s'interroger sur ce qui motive notre comportement. Ici, par exemple, notre but est de nous rapprocher du joueur. On regarde donc dans quelle direction il est, et on se déplace dans cette même direction (Voir programme HPUISSANCE 4 pour un autre exemple).

Améliorations possibles du programme

Il est possible d'apporter de nombreuses améliorations à ce programme. On peut par exemple envisager un système de scores basés sur le temps mis par le joueur pour enfermer la bébête. Plus le temps est court, plus le score est élevé. On peut alors rajouter une table des records au jeu.

On peut aussi améliorer le contrôle du déplacement de la Bébête et imaginer alors une succession de niveaux dans lesquels elle se déplacerait de plus en plus vite.

On peut enfin améliorer la finition, en rajoutant par exemple une pause, en reprogrammant les déplacements pour que ceux-ci s'effectuent au pixel près, ou en prévoyant la possibilité pour un second joueur de jouer la Bébète.... Comme vous le voyez, ce programme peut être grandement amélioré, à vos assembleurs...

Programme assemblé (G)

```
CCD20  45900  8FB97  608F5  11103  41042  08FD7  B5013
2E480  08408  1AF00  34001  10CA8  1AF01  C281A  F0A13
48E00  20200  01100  00000  00100  10100  00000  00110
00000  00100  01101  01001  00011  00000  01011  01000
00001  10000  00101  11000  01010  00000  01000  01000
00011  00000  00000  00000  00000  10000  00000  01011
00000  00100  10000  00000  00000  01100  00101  10000
00101  00001  00000  00001  00000  00111  00000  00000
00000  00100  00000  00000  00000  00100  00000  00000
00011  01001  00000  00000  00000  00000  00001  00000

00000  01010  00010  00100  00000  00001  00000  00100
01000  00000  00100  00001  00001  00000  01000  00000
00001  00000  10000  11100  00000  00000  00000  00000
00110  00100  00011  00100  00000  00001  00100  00000
10000  00000  01000  00001  00001  10000  01000  00030
71353  2FF11  5B015  80160  170A3  E5EE8  1AF10  34020
00CA1  3031F  FA801  50016  F16F1  61A6E  5FEAC  281A4
```

```
0BAE2  81A60  BA4E8  1A40C  31F18  1A60C  1B004  0881A
F1214  034FF  100C2  16414  48408  411B8  21003  0F15C
07CA0  7C407  C5078  75702  37DD1  8605E  1B821  00307

15401  FD860  81471  90214  48F5E  0108F  2D760  14216
4808C  81AF1  881AF  2981A  F0801  1B021  0081A  F1014
03408  800CA  1B821  00701  07610  1B021  00140  0114E
96EAF  0114E  96AAF  0181A  F1913  481AF  1AD7A  C1A4D
31F1A  E5DBE  71351  53070  10000  0F99F  6996F  FFF07
A0C4C  0818F  2363F  F1351  5F323  15401  6F16F  1610D
50F20  A4D5E  A1368  0824F  7800E  20631  01965  60782
F0713  4A6D5  58791  F81AF  11340  8800C  A1308  1AF1A
D7AC1  A4D31  F1AE5  DBE71  35153  07010  0000F  EE069
96F99  F07A0  C4C08  18F23  63FF1  3515F  32315  4016F

16F16  10D50  F20A4  D5EA1  36808  24F78  00E20  63101
96560  708E0  7134A  6D558  01871  70851  01841  81A41
381A4  1C9CE  031B5  04081  46CE1  35153  090C8  1A4E8
1A40C  80821  30149  14481  A4138  1A41C  9CA63  1B504
08146  135E6  13515  3090C  B1B46  81A40  C8082  1031C
01491  4481A  61381  A61C9  EA74A  E51B5  04081  46818
F2F13  51530  90C82  AEDB6  681A6  0CAED  80820  31510
1CFA8  01510  14481  A6138  1A61C  9EE74  AE51B  50408
14681  8FAF1  35153  090C8  2AEDA  6E81A  60CAE  D8082
03151  017FA  80151  01440  13201  08FCE  E1080  8A050

85032  2008F  CEE10  808A2  7781A  41BB4  64B61  B0040
8146E  6D513  51530  90871  30390  294B4  64341  7065E
F81A4  1BB46  81A40  B1461  35D91  44A80  15709  0A011
51017  0A8A6  CEF15  10013  28008  FCEE1  0808A  27781
A41BA  4E4B6  1B004  08146  CED51  35153  09087  13039
0294A  4E434  1C065  EF81A  41BA4  E81A4  0B146  135D9
144A8  01570  90A01  15101  C0A8A  6CEF1  51001  32400
8FCEE  10808  A3C78  1A61B  A6EAE  74D61  B0040  81468
18FAF  D5135  15309  08713  03902  74A6F  4141C  F65EF
81A61  BA6E8  1A60B  14613  5D914  4A801  57090  A0115

101CF  A8A6C  EF151  03240  08FCE  E1080  8B180  8CA80
081A6  1BB66  AE731  02963  371B0  04081  46818  F2FD5
13515  30908  D1303  902D4  B6731  02963  1417F  6FDF8
1A61B  B6681  A60B1  46135  D9144  A8015  7090A  01151
017FA  8A6CE  F1510  011B5  04081  46134  A8130  1A87B
06A8A  16015  60902  35907  50B05  18115  60902  F3907
50B05  18E15  60902  B2907  50B05  16F16  F1560  90241
90750  B0530  49014  00185  01BB0  10031  FB14C  7A007
```

```
11072  00017  2108A  E9F01  77008  AA9F0  17E2A  1B821
00775  A7D5A  7F4A7  55A34  FF100  8FCEE  1001
```

CRC : # 519Bh (G)

Programme assemblé (S)

```
CCD20  B3900  8FB97  608F5  11103  41042  08FD7  B5013
2E480  84081  AF003  40011  0CA81  AF01C  281AF  0A134
1F821  0031F  314D8  E0020  20001  10000  00000  10010
10000  00000  11000  00000  10001  10101  00100  01100
00001  01101  00000  00110  00000  10111  00001  01000
00001  00001  00000  01100  00000  00000  00000  00010
00000  00001  01100  00000  10010  00000  00000  00001
10000  10110  00000  10100  00100  00000  00100  00000
11100  00000  00000  00000  10000  00000  00000  00000

10000  00000  00000  01101  00100  00000  00000  00000
00000  00100  00000  00001  01000  01000  10000  00000
00100  00000  10001  00000  00000  10000  00100  00100
00001  00000  00000  00100  00010  00011  10000  00000
00000  00000  00000  11000  10000  01100  10000  00000
00100  10000  00010  00000  00001  00000  00100  00110
00001  00000  03071  3532F  F115B  01580  16017  0A3E5
EE81A  F1034  02000  CA130  31FFA  80150  016F1  6F161
A6E5F  EAC2A  E210B  A4E31  F110C  1B003  0781A  F1214
034FF  100C2  16414  48408  4175A  07430  74407  17579

1376D  18605  E8FF7  C108F  5E010  8F2D7  60142  16480
8C81A  F1881  AF298  1AF08  011B0  21008  1AF10  14034
08800  CA198  27010  70201  B0210  01400  114E8  08868
08879  6E0F0  114E8  08868  08879  6A0F0  181AF  19134
81AF1  AD7AC  1A4D3  1F1AE  5DBE7  13515  30701  00000
F99F6  996FF  FF07A  0C4C0  818F2  363FF  13515  F3231
54016  F16F1  610D5  0F20A  4D5EA  13680  824F7  800E2
06310  19656  0771F  07134  A6D55  8780F  81AF1  13408
800CA  13081  AF1AD  7AC1A  4D31F  1AE5D  BE713  51530
70100  000FE  E0699  6F99F  07A0C  4C081  8F236  3FF13

515F3  23154  016F1  6F161  0D50F  20A4D  5EA13  68082
4F780  0E206  31019  65607  F6E07  134A6  D5580  18717
08510  18418  1A413  81A41  C9CE0  31B50  30714  6CE13
```

```
51530  90C81  A4E81  A40C8  08213  01491  4481A  41381
A41C9  CA631  B5030  71461  35E61  35153  090CB  1B468
1A40C  80821  031C0  14914  481A6  1381A  61C9E  A74AE
51B50  30714  6818F  2F135  15309  0C82A  EDB66  81A60
CAED8  08203  15101  CFA80  15101  4481A  61381  A61C9
EE74A  E51B5  03071  46818  FAF13  51530  90C82  AEDA6
E81A6  0CAED  80820  31510  17FA8  01510  14401  32010

8FCEE  10808  A0508  50322  008FC  EE108  08A27  781A4
1BB46  4B61B  00307  146E6  D5135  15309  08713  03902
94B46  43417  065EF  81A41  BB468  1A40B  14613  5D914
4A801  57090  A0115  10170  A8A6C  EF151  00132  8008F
CEE10  808A2  7781A  41BA4  E4B61  B0030  7146C  ED513
51530  90871  30390  294A4  E4341  C065E  F81A4  1BA4E
81A40  B1461  35D91  44A80  15709  0A011  5101C  0A8A6
CEF15  10013  24008  FCEE1  0808A  3C781  A61BA  6EAE7
4D61B  00307  14681  8FAFD  51351  53090  87130  39027
4A6F4  141CF  65EF8  1A61B  A6E81  A60B1  46135  D9144

A8015  7090A  01151  01CFA  8A6CE  F1510  32400  8FCEE
10808  B1808  CA800  81A61  BB66A  E7310  29633  71B00
30714  6818F  2FD51  35153  0908D  13039  02D4B  67310
29631  417F6  FDF81  A61BB  6681A  60B14  6135D  9144A
80157  090A0  11510  17FA8  A6CEF  15100  11B50  30714
6134A  81301  A87B0  6A8A1  60156  09023  59075  0B051
81156  0902F  39075  0B051  8E156  0902B  29075  0B051
6F16F  15609  02419  0750B  05304  90140  01850  1BB01
0031F  B14C7  A0071  10720  00172  108AE  9F017  7008A
A9F01  7D1A1  B8210  0734A  735A7  B3A7B  4A34F  F1008
FCEE1  001
```

CRC : # FB1Ch (S)

HPUISSANCE 4 ****

HPUISSANCE 4 est la version HP48 d'un célèbre jeu de réflexion où il faut aligner quatre pions en ligne, colonne ou diagonale. Il vous permet de jouer à deux joueurs où contre votre machine. Il effectue tous les tests d'alignement et vous averti lorsque la partie est finie. Il utilise les touches suivantes :

La touche **DROP** pour abandonner la partie en cours.

Les flèches ← et → pour déplacer le pion.

La flèche ↓ pour poser le pion.

Enfin, la touche **ENTER** pour demander une aide à la machine. Elle place alors le pion dans la colonne où elle jouerait. Le coup n'est cependant pas joué. Vous pouvez toujours vous déplacer et jouer ailleurs.

Pour jouer contre la machine, il suffit de demander systématiquement à la machine son "avis", et de la faire jouer dans la colonne qu'elle indique. Amusez-vous bien

```
%%%
```

```
LC 007B9              % Réservation d'un écran 131*56, d'une table
GOSBVL 05B7D          %9*7 plus 1 quartet si l'écran est à une
                      %adresse impaire
AD0EX                 % Ici, on place l'écran à une adresse paire
A=A+1 A
ABIT=0 0
R0=A A                % R0a contient l'adresse de l'écran
LC 00770
C=C+A A
R1=C A                % R1a contient l'adresse de la table
GOSBVL 01115          % Interdiction des interruptions
LC 7A
A=R0 A
D0=A
A=0 W
*INIT                 % J'efface l'écran et j'initialise la
                      %table à 0
DAT0=A W
D0=D0+ 16
C=C-1 B
GONC INIT
P= 7
DAT0=A WP
P= 0
LC 100                % Initialisation : Joueur = 0, Colonne = 1
R2=C X
C=R0 A
D0= 00120             % J'installe l'écran
DAT0=C A
A=0 A
LA 90                 % On fait maintenant pointer R0a plus
                      %loin, au milieu de l'écran
C=C+A A               %pour centrer la grille
R0=C A
*BOUCLE               % La boucle principale
 GOSUB JOUE           % Le joueur joue
 GOSUB TEST           % On teste s'il n'y a pas quatre pions
                      %alignés
 C=R2 B               % On passe au joueur suivant (0 ou 1)
 C=C+1 B              % Regardez la méthode pour faire une addition
                      %modulo 2
 CBIT=0 1
 R2=C B
GOTO BOUCLE           % et c'est tout pour la boucle
*FIN                  % La routine de fin
D0= 0010B             % J'allume les indicateurs  pour indiquer la
                      %fin
LC BF
DAT0=C B
GOSUB PAS_TOUCHE      % J'attends que touche soit relachée
GOSUB UNE_TOUCHE      % J'attends une touche
GOSUB PAS_TOUCHE      % Tu vas la relâcher , oui ?
GOSBVL 01C7F          % On réinstalle l'écran
GOSBVL 010E5          % On ré autorise les interruptions
GOSBVL 067D2          %et enfin, on sort
A=DAT0 A
D0=D0+ 5
```

```
PC=(A)
*PAS_TOUCHE          % La routine qui attend que toutes les
LC 001FF             %touches soit relâchées
GOSBVL 01EEC
?C#0 A
GOYES PAS_TOUCHE
RTN
*UNE_TOUCHE          % La routine qui attend une touche
LC 001FF
GOSBVL 01EEC
?C=0 A
GOYES UNE_TOUCHE
RTN
*DESSINE_ECRAN       % La routine qui re-dessine la grille d'après
                     %la table
A=R1 A               % On va dessiner ligne par ligne
A=A+10 A
B=A A                % Ba pointe sur la prochaine "case" à
                     %dessiner
A=R0 A
D0=A
P= 15
LC 5                 % Cs contient le nombre de lignes -1
P= 0
*BOUCLE_DES1         % La boucle qui trace ligne après ligne
LC 06                % 7 colonnes -1 pour la retenue
D=C B
*BOUCLE_DES2         % La boucle qui dessine une ligne
C=B A                % On récupère la position dans la table
B=B+1 A              % On passe à la position suivante
D1=C
A=DAT1 1             % Et on lit ce qui se trouve dans la table
GOSUB APRES_PIECES   %les 3 pièces différentes
$FF3C181818183CFF    % Case vide (0 dans la table)
$FF3C99DBDB993CFF    % Case + Pion noir (1 dans la table)
$FF3C995A5A993CFF    % Case + Pion blanc (2 dans la table)
*APRES_PIECES
C=RSTK               % On va faire pointer D1 sur la bonne pièce
D1=C
*BOUCLE_POSITIONNE
A=A-1 P
GOC POSITIONNE
D1=D1+ 16
GOTO BOUCLE_POSITIONNE
*POSITIONNE          % Voilà, c'est fait
A=DAT1 16            % On lit la pièce
P= 7                 % 8 lignes pour un Grob, -1 pour la retenue
*BOUCLE_DESSINE
DAT0=A B             % on pose 2 quartets = 8 pixels
ASR W                % On installe la ligne suivante du Grob dans
                     %Ab
ASR W
D0=D0+ 16            % Et on pointe sur la ligne suivante de
                     %l'écran
D0=D0+ 16
D0=D0+ 2
```

```
P=P-1
GONC BOUCLE_DESSINE
P= 0
ADOEX                  % On repositionne à la colonne suivante sur
                       %l'écran
LC 0010E
A=A-C A
D0=A
D=D-1 B                % On affiche les 7 colonnes
GOC FINI_DES2
GOLONG BOUCLE_DES2
*FINI_DES2
B=B+2 A                % On passe les 2 colonnes non visibles
ADOEX                  % On repositionne D0 au début de la ligne
                       %suivante
LC 00102
A=A+C A
D0=A
C=C-1 S
RTNC
GOTO BOUCLE_DES1
*COLONNE               % Routine mettant dans Cb l'altitude
                       %disponible dans
A=R1 A                 %la colonne R2xs, et faisant pointer D0 sur
                       %cette colonne dans la table
C=0 A
LC 36
A=A+C A
C=0 A
C=R2 XS
CSR A
CSR A
A=A+C A
D0=A
C=0 B                  % On part du bas
*BOUCLE_COLONNE        %et on monte tant que c'est occupé
A=DAT0 P
D0=D0- 9
C=C+1 B
?A#0 P
GOYES BOUCLE_COLONNE
D0=D0+ 9
RTN
*JOUE                  % Routine pour le jeu d'un joueur
ST=0 1                 % Pour tester si la grille n'est pas pleine
D=0 XS                 % Numéro de la colonne que l'on va tester
*BOUCLE_TEST           % La boucle testant qu'il reste de la place
D=D+1 XS               % Colonne suivante
C=D XS
R2=C XS
GOSUB COLONNE          % Altitude ?
LA 07                  % Si c'est 7, la colonne est pleine
?A=C B
GOYES SAUTE_HP
ST=1 1                 % Sinon, il reste de la place. Le drapeau est
                       %armé
```

```
*SAUTE_HP
LC 700                  % A-t-on tout testé ?
?C#D XS
GOYES BOUCLE_TEST
?ST=1 1                 % Si le drapeau n'a jamais été armé, tout est
GOYES PAS_FIN           %plein
GOLONG FIN              %le jeu est fini
*PAS_FIN
C=0 XS
C=C+1 XS
R2=C XS

*BOUCLE_JOUE            % La boucle recherchant la première colonne
GOSUB COLONNE           %libre pour y afficher le pion
AD0EX
D0=A
R3=A A
LA 7
?A#C P                  % Si ce n'est pas 7, c'est moins, il reste de
GOYES PAS.COLONNE       %la place c'est fini
GOTO DROITE             % Sinon, on passe à la colonne suivante
GOTO BOUCLE_JOUE        %et on relance le test

*PAS.COLONNE
C=R2 P                  %On place le pion du joueur
C=C+1 P                 %Attention, dans la table on place 1 pour le
                        %joueur 0
DAT0=C P                %2 pour le joueur 1
GOSUB DESSINE_ECRAN     %Et on re-dessine la grille
A=R3 A
D0=A

*BOUCLE_TOUCHE          %On va tester les touches
LC 010                  % DROP ? Si oui, fini !
GOSBVL 01EEC
 ?CBIT=0 0
 GOYES PAS_DROP

GOLONG FIN
*PAS_DROP
LC 010                  % ENTER ? Si oui, on va à HP, la routine de
                        %reflexion
GOSBVL 01EEC
 ?CBIT=0 4
 GOYES PAS_ENTER
GOTO HP
*PAS_ENTER
LC 040                  % Test du curseur
GOSBVL 01EEC
?CBIT=1 0               % Droite ?
GOYES DROITE
?CBIT=0 1               % Bas ?
GOYES PAS_BAS
GOSUB PAS_TOUCHE
RTN
*PAS_BAS
?CBIT=1 2               % Enfin, gauche ?
```

```
GOYES GAUCHE
GOTO BOUCLE_TOUCHE
*DROITE                 % Routine déplaçant le pion vers la droite
C=0 P                   % On efface l'ancien pion
DAT0=C P
A=R2 XS
LC 700                  % Si on est sur la colonne 7, on est le plus
                        %à droite
?C#A XS
GOYES OK_DROITE
A=0 XS                  % On va alors sur la colonne 1
*OK_DROITE
A=A+1 XS
R2=A XS
GOSUB RALENTI           %On ralentit pour la maniabilité(Un comble !)
GOTO BOUCLE_JOUE        % Et on recommence tout (Affichage du pion,
                        %touches ...)
*GAUCHE                 % La même chose vers la gauche
C=0 P
DAT0=C P
A=R2 XS
LC 100
?C#A XS
GOYES OK_GAUCHE
LA 800

*OK_GAUCHE
A=A-1 XS
R2=A XS
GOSUB COLONNE
LA 7
?A#C P
GOYES OK_GAUCHE_2
A=R2 XS
GOTO OK_GAUCHE

*OK_GAUCHE_2
GOSUB RALENTI
GOTO BOUCLE_JOUE
*RALENTI                % Le ralentissement, une simple boucle
LC 04000                % A modifier si vous voulez changer la
                        %vitesse de réponse des touches
*BOUCLE_RALENTI
C=C-1 A
GONC BOUCLE_RALENTI
RTN


*TEST                   % La routine testant les pions alignés
C=R2 P                  % Joueur+1 dans la table
C=C+1 P
D=C P
A=0 A
A=A+1 A
B=0 P                   % Bp va contenir le nombre de pions alignés
GOSUB SOUS_TEST         % Vers la gauche
```

```
A=A-2 A
C=R3 A
D0=C
GOSUB SOUS_TEST       % Vers la droite
LC 3                  % En tout plus de trois pions alignés avec le
                      %nôtre ? Si oui, c'est gagné !
?B<C P
GOYES SUITE_TEST
GOLONG FIN


*SUITE_TEST           % Sinon, on teste vers le bas ( Vertical )
C=R3 A
D0=C
A=0 A
LA 9
B=0 P
GOSUB SOUS_TEST
LC 3
?B<C P
GOYES SUITE_TEST_2
GOLONG FIN

*SUITE_TEST_2         % Puis diagonale dans un sens
C=R3 A
D0=C
A=0 A
LA A
B=0 P
GOSUB SOUS_TEST
A=A-16 A
A=A-4 A
C=R3 A
D0=C
GOSUB SOUS_TEST
LC 3
?B<C P
GOYES SUITE_TEST_3
GOLONG FIN


*SUITE_TEST_3         % Et enfin, dans l'autre
C=R3 A
D0=C
A=0 A
LA 8
B=0 P
GOSUB SOUS_TEST
A=A-16 A
C=R3 A
D0=C
GOSUB SOUS_TEST
LC 3
?B<C P
RTNYES
GOLONG FIN
```

```
*SOUS_TEST              % Sous routine pour les tests. Elle teste le
                        %nombre de pions
CD0EX                   % alignés par déplacement de Aa dans la table
C=C+A A
D0=C
C=DAT0 P
?C#D P
RTNYES
B=B+1 P
GOTO SOUS_TEST


*HP                     % La routine de réflexion (Voir après le
                        %listing)
C=0 P
DAT0=C P
P= 2
LC 4
P= 0
R2=C XS
GOSUB COLONNE
LA 1
?A#C P                  % Si Ap est différent de Cp on va au label
GOYES PAS_COL4          %PAS_COL4

GOTO FIN_HP

*PAS_COL4
P= 2
LC 5
P= 0
R2=C XS
GOSUB COLONNE
LA 1
?A#C P
GOYES PAS_COL5
GOTO FIN_HP
*PAS_COL5
C=0 W
C=C+1 S
R4=C W
B=C W
*BOUCLE_HP
B=0 A
C=B S
CSLC
CSLC
CSLC
R2=C XS
GOSUB COLONNE
LA 7
?A#C P
GOYES PAS_FINI_COLONNE
GOLONG FINI_COLONNE

*PAS_FINI_COLONNE
B=B+1 A
```

```
C=R2 B
C=C+1 B
CBIT=0 1
R2=C B
GOSUB GAGNE?
GONC SAUTE
LA 02000
B=B+A A

*SAUTE
GOSUB COLONNE          % On appelle la routine COLONNE
LA 6
?C>=A P
GOYES INC
D0=D0- 9
GOSUB GAGNE?
GOC SAUTE2

*INC
LA 01000
B=B+A A

*SAUTE2
C=R2 B
C=C+1 B
CBIT=0 1
R2=C B
C=C+1 P
D=C P
GOSUB COLONNE
CD0EX
R3=C A
D0=C
A=0 A
A=A+1 A
A=0 S
GOSUB SOUS_TEST_2
C=R3 A
D0=C
A=A-2 A
GOSUB SOUS_TEST_2
GOSUB INCREMENTE       % On appelle la routine d'incrémentation
C=R3 A
D0=C
A=A+11 A
A=0 S
GOSUB SOUS_TEST_2
GOSUB INCREMENTE
C=R3 A
D0=C
A=0 W
LA A
GOSUB SOUS_TEST_2
C=R3 A
D0=C
A=A-16 A
```

```
A=A-4 A
GOSUB SOUS_TEST_2
GOSUB INCREMENTE
C=R3 A
D0=C
A=0 W
LA 8
GOSUB SOUS_TEST_2
C=R3 A
D0=C
A=A+16 A
GOSUB SOUS_TEST_2
GOSUB INCREMENTE
*FINI_COLONNE
A=B W
C=R4 W
?A<=C A
GOYES PAS_CHANGE
R4=A W
*PAS_CHANGE
B=B+1 S
P= 15
LC 8
P= 0
?B=C S
GOYES FINI_HP
GOLONG BOUCLE_HP


*FINI_HP
C=R4 S
CSLC
CSLC
CSLC
R2=C XS
*FIN_HP
GOSUB PAS_TOUCHE
GOLONG BOUCLE_JOUE
*GAGNE?              % routine semblable à TEST pour savoir si
                     %l'adversaire peut gagner
AD0EX                % mais qui renvoie son résultat sur la Carry.
                     %Armée si l'adversaire peut gagner
R3=A A               %effacée dans l'autre cas
D0=A
C=R2 P
C=C+1 P
D=C P
A=0 A
A=A+1 A
A=0 S
GOSUB SOUS_TEST_2
A=A-2 A
C=R3 A
D0=C
GOSUB SOUS_TEST_2
P= 15
```

```
LC 3
P= 0
?A<C S
GOYES SUITE_GAGNE
RTNSC
*SUITE_GAGNE
C=R3 A                  % Ca contient la valeur de R3a
D0=C
A=0 A
LA 9
A=0 S
GOSUB SOUS_TEST_2
?A<C S
GOYES SUITE_GAGNE_2
RTNSC
*SUITE_GAGNE_2
C=R3 A
D0=C
A=0 A
LA A
A=0 S
GOSUB SOUS_TEST_2
A=A-16 A
A=A-4 A
C=R3 A
D0=C
GOSUB SOUS_TEST_2
?A<C S
GOYES SUITE_GAGNE_3
RTNSC
*SUITE_GAGNE_3
C=R3 A
D0=C
A=0 A
LA 8
A=0 S
GOSUB SOUS_TEST_2
A=A-16 A
C=R3 A
D0=C
GOSUB SOUS_TEST_2
?A>=C S
RTNYES
RTNCC
*SOUS_TEST_2            % Comme SOUS_TEST, mais travaille sur As au
CD0EX                   %lieu de Bp
C=C+A A
D0=C
C=DAT0 P
?C#D P
RTNYES
A=A+1 S
GOTO SOUS_TEST_2
*INCREMENTE             % Augmente le compteur de points selon le
                        %nombre de pions alignés
A=A-1 S
```

```
RTNC
B=B+5 A
A=A-1 S
RTNC
B=B+3 A
A=A-1 S
RTNC
LC 10000
B=B+C A
RTN
@
```

CRC : # D0A7h (S et G)

La routine de réflexion de ce jeu

Ce jeu contient un algorithme d'intelligence artificielle. Le programme évalue dans n'importe quelle position la colonne où il lui parait le plus intéressant de jouer. L'assembleur Saturn ne permettant de faire que des calculs simples, tout est construit à l'aide d'un système de "scores". A chaque colonne est attribué un "score", le programme joue finalement dans la colonne ayant tout simplement le plus de points. Détaillons le décompte :

1) On initialise le compteur dans Ba.

2) Si la colonne n'est pas pleine, on ajoute 1, ce pour éviter que la machine puisse jouer dans une colonne vide. Les colonnes pleines ayant un total de 0, elles seront toujours éliminées face aux colonnes disponibles qui totaliseront au moins 1.

3) On regarde ensuite si l'adversaire peut gagner dans cette colonne. Si oui, on augmente le compteur de 2000

4) Puis, on regarde, si l'adversaire peut gagner au coup suivant, si l'on joue dans cette colonne. Si non, on rajoute 1000 points au score.

5) Enfin, on regarde le nombre de pions que l'on alignerait si l'on jouait dans cette case. On teste en ligne, colonne et diagonale. Les points attribués sont, par test :

5 pour deux pions, 8 pour trois pions, 10000 pour quatre pions et plus.

Remarquez que les points attribués ne le sont pas au hasard : on favorise tout d'abord le gain lorsqu'on le peut, puis le blocage de l'adversaire si lui peut gagner. On défavorise ensuite fortement les coups conduisant à la victoire de l'opposant. Enfin, on joue en favorisant les ouvertures de jeu : deux fois deux pions alignés est considéré comme meilleur que trois pions alignés (ils totalisent 2*5 = 10 points face à 8). A l'aide de cette méthode relativement simple, on obtient un niveau de jeu convenable.

Amélioration possible de ce programme

Ce programme ne possède que peu d'animations, il semble donc relativement facile de reprogrammer l'affichage en niveaux de gris.

Rajouter des animations, par exemple la chute du pion dans la colonne.

Rajouter un mode "Tournoi". On donne à la machine le nombre de joueurs. Elle s'occupe de gérer l'ordre des matches, les éliminations. Et enfin, elle dresse un classement des joueurs.

Et enfin, pourquoi ne pas améliorer la réflexion. Evaluer plusieurs coups à l'avance par exemple.

Programme assemblé (S et G)

```
CCD20  74700  8FB97  60349  B7008  FD7B5  0132E  48084
081AF  00340  7700C  281AF  098F5  11103  1A781  AF101
```

```
30AF0  15071  6FA6E  55F27  15012  03200  181A3  0A81A
F181B  02100  144D0  80821  09C28  1AF08  74817  0E281
A61AB  66808  8181A  60A63  EF1BB  01003  1FB14  C7720
78307  F108F  F7C10  8F5E0  108F2  D7601  42164  808C3
4FF10  08FCE  E108A  EFE01  34FF1  008FC  EE108  AAFE0
181AF  11818  F09D8  81AF1  01302  F3052  03160  AE7D9
E5135  15B07  030FF  3C181  81818  3CFFF  F3C99  DBDB9
93CFF  FF3C9  95A5A  993CF  F0713  5A0C4  9017F  66FF1

5BF27  148BF  4BF41  6F16F  1610D  5BE20  13234  E0100
EA130  A6F48  08C97  FF818  F1113  23420  100CA  130A4
E4006  25F81  AF11D  23163  CAD28  1A21A  F6F6C  A130A
E2152  0188B  6690C  3F168  01841  AA3B2  7AAB8  1A20A
74BF8  08217  09625  08513  20079  279D8  71808  C26EF
AA2B2  681A2  0A708  F1321  3081A  F0380  82079  06A06
C7060  EF81A  01AB0  61540  718E8  1AF13  13032  0108F
CEE10  808A0  808C0  0EF32  0108F  CEE10  808A4  606E9
13204  08FCE  E1080  8B0A1  808A1  80700  E0180  8B213
68AFA  82154  081A2  12320  07926  50AA0  B2481  A2027

D4060  4FA82  15408  1A212  32001  926A0  80822  008A2
C81A2  02759  E8082  07906  C081A  21261  EF740  067FE
34000  40CE5  DF018  1A01A  B06A8  7D0E4  A817F  B0818
F8181  AF1B1  347CA  03039  85808  C40DF  81AF1  B134D
08082  09A81  76803  03985  808CE  DCF81  AF1B1  34D08
0820A  A8170  60818  F8F81  8F838  1AF1B  13477  40303
98580  8CF9C  F81AF  1B134  D0808  208A8  17120  818F8
F81AF  1B134  7E003  03985  008C6  6CF13  6C213  41560
90700  B056B  EFA82  15402  23042  081A2  0A746  D8082
01906  6064A  12230  52081  A20A7  44D80  82019  06606

481AF  2B461  0CAF5  D1AC9  81281  28128  1A20A  711D8
08207  90680  8C811  0E581  A61AB  66808  8181A  60A7C
315E0  80824  00020  C873D  C8082  0698A  C0188  77114
E0808  24000  10C88  1A61A  B6680  88181  A60AB  06A87
749C1  3681A  F0B13  4D0E4  AC070  A181A  F1B13  4818F
817D8  171A1  81AF1  B1348  18F0A  AC073  71778  181AF
1B134  AF080  820A7  95181  AF1B1  34818  F8F81  8F837
04174  5181A  F1B13  4AF08  08208  76218  1AF1B  13481
8F0F7  31177  21AF4  11C8B  A5010  4B452  F3082  09418
08C4A  EF81A  41C81  28128  1281A  20A77  BA8CA  2CF13

281AF  03130  81A01  AB06A  87D0E  4AC07  AA081  8F818
1AF1B  13477  902F3  03209  C2400  281AF  1B134  D0808
209AC  07170  9C240  0281A  F1B13  4D080  820AA  C0725
```

```
0818F  8F818  F8381  AF1B1  34793  09C24  00281  AF1B1
34D08  08208  AC07A  10818  F8F81  AF1B1  34770  09CE0
00313  6C213  41560  90700  B446B  EFA4C  40081  8F14A
4C400  818F1  2A4C4  00340  0001C  101
```

CRC : # BD90h (S et G)

DECOMP ***

Décomposer un entier en produit de facteurs premiers est un travail intéressant, mais qui demande beaucoup de temps lorsque l'on travaille en RPL, d'où l'idée d'effectuer cette tâche à l'aide d'un programme en langage machine. DECOMP est un programme qui prend un Binary Integer sur la pile, et qui en renvoie la décomposition sous forme de Binary Integer.

Par exemple, exécuter DECOMP sur # 4886718345d (# 123456789h) renvoie en 0,8s # 3d, # 3d, # 5d, # 23d et # 4721467d, ce qui signifie que la décomposition de 4886718345 est 3*3*5*23*4721467 (Oui, oui, 4721467 est bien premier !)

L'algorithme utilisé par DECOMP est fort simple :

- Tout d'abord, on extrait tous les facteurs 2 possibles

- Puis on divise par tous les nombres impairs jusqu'à ne plus obtenir que 1 ou la racine carrée du nombre restant à décomposer. Si on arrive à celle-ci, le nombre est premier, on le renvoie, la décomposition est finie.

```
%% Décomposition en produit de facteurs premiers
% Registres utilisés :
% R0a et R1a utilisés par les routines en Rom appelées
% R2a : Nombre à décomposer
% R3a : Nombre impair par lequel on va essayer de diviser R2a
% R4a : Carré de R3a

GOSBVL 0679B
C=DAT1 A                % Tout d'abord, on vérifie qu'on a bien un
D0=C                    %Binary integer sur la pile
```

```
A=DAT0 A
LC 02A4E
?C=A A
GOYES OK

*BAT                    % Sinon, on émet "Bad Argument Type"
GOSBVL 067D2
LA 00202
GOVLNG 05023

*OK
D0=D0+ 5                % On lit le contenu avec une méthode déjà vue
                        %précédemment
C=DAT0 A
D0=D0+ 5
C=C-6 A
P=C 0
C=0 W
C=DAT0 WP
P= 0
?C=0 W                  % Si l'entier est nul, on ne peut le
                        %décomposer
GOYES BAT               % On sort avec l'erreur "Bad Argument Type"
A=0 B                   % Une première boucle pour compter les
                        %facteurs 2
*TESTE_2                % On utilise Ab comme compteur car n vaut au
                        %plus 2^64-1
?CBIT=1 0               % On teste le bit 0. s'il est nul, le nombre
                        %est pair
GOYES FINI
CSRB W                  % S'il est bien pair, on le divise par deux
A=A+1 B                 %on incrémente le compteur
GOTO TESTE_2            %et on recommence

*FINI                   % Les divisions par 2 sont finies
R2=C W                  % on sauvegarde l'entier restant à décomposer
GOSBVL 067D2            % Puisqu'on ne l'avait pas fait avant, on
D1=D1+ 5                % enlève l'entier de la pile
D=D+1 A
GOSBVL 0679B
C=A B

*BOUCLE_2               % Une boucle pour poser les facteurs 2 sur la
                        %pile
?C=0 B
GOYES APRES_2
RSTK=C
LA 2
GOSBVL 067D2            % On récupère les registres
GOSBVL 5422C            % On pose # 2h ( Awp avec P=0)
GOSBVL 0679B            % Et on resauvegarde
C=RSTK
C=C-1 B
GOTO BOUCLE_2

*APRES_2                 % Maintenant, on va tester les autres
                        %facteurs
```

```
C=0 W                    % Le premier nombre impair par lequel on va
                        %diviser
LC 3                     % c'est 3. Zou, dans R3w
R3=C W
LC 9                     % Son carré, c'est 9, dans R4w
R4=C W

*BOUCLE_PRINCIPALE
C=0 W                    % On teste si l'entier vaut 1
LC 1
A=R2 W
?A#C W                   % Si oui, c'est fini
GOYES APRES_FIN

*FIN
GOSBVL 067D2
A=DAT0 A
D0=D0+ 5
PC=(A)

*APRES_FIN               % Sinon
C=R3 W                   % On le divise par le nombre impair en cours
GOSBVL 65807             % La division en Rom

?B#0 W                   % Si le reste est nul, c'est un diviseur
GOYES NOMBRE_SUIVANT
R2=C W                   % alors on sauvegarde le quotient
A=R3 W                   %et on pose le diviseur sur la pile
GOSBVL 067D2             % Le même topo que pour le facteur 2
                        %précédemment
P= 15
GOSBVL 5422C
P= 0
GOSBVL 0679B
GOTO BOUCLE_PRINCIPALE % et on recommence les divisions

*NOMBRE_SUIVANT          % Si ce n'est pas un diviseur
C=R3 W                   % On incrémente le diviseur impair de 2, pour
B=C W                    %passer au suivant
C=C+2 W
R3=C W
B=B+B W                  % Et on calcule son carré en remarquant que
B=B+B W                  % (n+2)^2=n^2+2*n+4, on calcule donc le
C=R4 W                   % nouveau R4w en calculant R4w+2*R3w+4
A=R2 W
?C<A W                   % si le carré dépasse l'entier étudié, on a
                        %dépassé
GOYES PAS_DERNIER        %la racine carrée, c'est fini
GOSBVL 067D2             % On pose  ce dernier diviseur sur la pile
P= 15                    %Et on sort
GOVLNG 0596D

*PAS_DERNIER             % Sinon, on sauvegarde le nouveau carré
C=C+B W
C=C+4 W
R4=C W
```

```
GOTO BOUCLE_PRINCIPALE     % et on relance la recherche
@
```

CRC : # 1519h (S et G)

Programme assemblé (S et G)

```
CCD20  25100  8FB97  60147  13414  234E4  A208A  2A18F
2D760  80824  20200  8D320  50164  14616  4818F  A580D
0AF21  56120  97A9C  AE080  8B0C0  81EB6  462FF  10A8F
2D760  174E7  8FB97  60AE6  96A82  06808  2028F  2D760
8FC22  458FB  97600  7A6E6  8DFAF  23031  0B309  10CAF
23011  12976  318F2  D7601  42164  808C1  1B8F7  08569
7D521  0A113  8F2D7  602F8  FC224  5208F  B9760  62BF1
1BAF5  81872  110BA  75A75  11C11  29F62  18F2D  7602F
8DD69  50A79  81872  310C6  27F
```

CRC : # B21Dh (S et G)

Améliorations possibles de ce programme

Ce programme, bien que très rapide, peut encore être accéléré :

- Il utilise trois routines en Rom pour poser un chiffre sur la pile (Load-Reg - Push_BI, Save-Reg), programmer une routine plus adaptée permettrait un gain de temps appréciable.
- Il utilise la Routine de division en Rom, celle-ci n'est pas parfaite, et il est surement possible d'en faire une plus rapide, mieux adaptée au problème.
- Enfin, il utilise le plus simple des algorithmes de décomposition. Des algorithmes plus élaborés permettent un gain important.

Annexes

CORRIGES DES EXERCICES

Voici le corrigé des exercices.

A.1.1 : # 09d ; # 74d ; # 5397d ; # 20d ; # 72d ; # 861d

A.1.2 : # 5EBh ; # B4h ; # 6Fh ; # 385A7h ; # 120h

A.1.3 : # 11111111b ; # 11111111b ; # 10100001b ; #00010000b

A.2.1 : Il y a au moins deux méthodes différentes : une rotation d'un quartet à gauche, ou quatre rotations d'un bit à gauche.

A.2.2 : # 41d and # 1010b = #8d = # 8h = # 1000b.

A.2.3 : # 61d or # FFh = # 111101b.

B.6.1 : Le programme 1.1 boucle 74565 fois.

B.6.2 : Pour que le programme 1.1 boucle mille fois, il faut remplacer la séquence "**LC 12345**" par "**LC 003E8**"

B.7.1 : Le programme 2.3 boucle 1048321 fois et le programme 2.4 va boucler 65537 fois.

B.8.1 : Voici un programme possible pour boucler tant que ENTER ou ON ne sont pas appuyés

```
GOSBVL 01115
*DEBUT
LC 010
GOSBVL 01160
?CBIT=1 4
GOYES FIN
?CBIT=0 15
GOYES DEBUT
*FIN
GOSBVL 010E5
A=DAT0 A
D0=D0+ 5
PC=(A)
@
```

B.9.1 : Voici le programme réalisant la fonction CLEAR

```
*BCL.PRIN
C=DAT1 A
?C=0 A
GOYES FIN
D1=D1+5
D=D+1 A
GOTO BCL.PRIN
*FIN
A=DAT0 A
D0=D0+ 5
PC=(A)
@
```

B.10.1 : Voici le programme qui noircit l'écran

```
GOSBVL 0679B
D0= (7050E : S
     8068D : G)
C=DAT0 A
D0=C
LA 87
C=0 W C=C-1 W
*BCL.PRIN
DAT0=C 16
D0=D0+16
A=A-1 B
GONC BCL.PRIN
GOVLNG 05143
@
```

LISTE DES ROUTINES EN ROM

Dans cette annexe, nous allons vous présenter une liste de programmes en langage machine situés en Rom. Ces routines sont repérées par leurs adresses, et on les appelle donc par les commandes **GOSBVL Adresse** ou **GOVLNG Adresse**. L'emploi de ces routines présente à la fois des avantages et des inconvénients.

Parmi les avantages, on peut citer une plus grande facilité de programmation (le programme est déjà tout fait), une grande lisibilité lorsqu'on a la liste de ces routines sous les yeux, et une réduction de la taille du programme (Un sous-programme étant remplacé par un simple **GOSBVL Adresse**).

Parmi les inconvénients, on peut citer une lisibilité médiocre sans liste, une certaine lenteur car un **GOSBVL** ou un **GOVLNG** est toujours plus long qu'un simple **GOSUB** ou **GOTO**.

Cependant, certaines de ces routines étant relativement compliquées, ces adresses peuvent être particulièrement utiles. Elles permettent aussi de réaliser des programmes relativement élaborés sans grande difficulté lorsqu'on débute.

La présentation de ces routines sera du type suivant :

SAVE-REG (# 0679Bh) (D0, D1, B, C, D) réalise la sauvegarde des registres D0, D1, Ba et Da dans une zone réservée de la mémoire vive. Les anciennes valeurs sont écrasées.

On trouve donc, dans l'ordre :

- 1 : **SAVE-REG** : Nom donné habituellement à cette routine

- 2 : (# 0679Bh) : L'adresse oùm se situe ce programme.

- 3 : (D0, D1, B, C, D) : Les registres utilisés par la routine. (GOVLNG) indique que cette routine est à appeler par **GOVLNG**. Les registres utilisés sont donc sans intérêt.

- 4 : Enfin, un commentaire sur la fonction effectuée.

1) Sauvegarder et restaurer des registres

Il existe des routines permettant des sauvegardes et restaurations totales, d'autres permettant de ne récupérer que certaines valeurs.

SAVE-REG (# 0679Bh) (D0, D1, B, C, D) : réalise la sauvegarde des registres D0, D1, Ba et Da dans une zone réservée de la mémoire vive. Les anciennes valeurs sont écrasées.

LOAD-REG (# 067D2h) (D0, D1, B, C, D) : récupère les valeurs sauvegardées lors du dernier Save-Reg.

2) Sortir d'un programme

Il existe une multitude de routines utilisables pour sortir d'un programme. Ces routines sont à appeler par **GOVLNG**.

EXIT (# 2D564h) (GOVLNG) : exécute la séquence de retour au RPL A=DAT0 A, D0=D0+ 5, PC=(A)

LOAD-REG_EXIT (# 5143h) (GOVLNG) : récupère les registres sauvegardés et exécute Exit.

ERROR_A (# 05023h) (GOVLNG) : à utiliser après Load-Reg, sort du programme en affichant le message d'erreur dont le numéro est dans Aa.

STOP (# 10FDBh) (GOVLNG) : à utiliser après Load-Reg, sort du programme en exécutant une erreur non récupérable par IFERR. Cette routine interrompt donc tout programme en cours.

PUSH-SB_Aa_EXIT (# 357Ch) (GOVLNG) : à utiliser après Load-Reg, pose un System Binary représentant Aa sur la pile et exécute Exit.

PUSH-SB_R0a_EXIT (# 357Fh) (GOVLNG) : à utiliser après Load-Reg, pose un System Binary représentant R0a sur la pile et exécute Exit.

PUSH-BI_Awp_EXIT (# 596Dh) (GOVLNG) : à utiliser après Load-Reg, pose un Binary Integer sur la pile représentant Awp, appelle Exit.

PUSHA_EXIT (# 3A86h) (GOVLNG) : à utiliser après Load-Reg, pose l'objet dont l'adresse est dans Aa sur la pile et appelle Exit.

3) Lire des données sur la pile

Il est possible de lire diffèrents types de données sur la pile. Toutes ces routines doivent être exécutées après Load-Reg afin que les registres pointent sur la pile.

POP-SB_Aa (# 06641h) (D1, A, C, D) : Lit un System Binary au niveau 1 de la pile. Sa valeur est stockée dans Aa et l'entier est éliminé.

POP-SB_Ca_Aa (# 3F5Dh) (D1, A, C, D) : Lit deux System Binaries sur la pile. La valeur de l'entier au niveau 1 est stockée dans Ca, celle de l'entier au niveau 2 dans Aa. Les deux System Binaries sont éliminés.

POP-REAL_Aw (# 5CFBh) (D0, D1, A, C, R0) : Lit le contenu d'un Réel sur la pile, le place dans R0w et Aw.

4) Poser des données sur la pile

Il est aussi possible de déposer des données sur la pile. Comme précédemment, afin que les registres pointent sur la pile, ces routines doivent être appelées après Load-Reg.

PUSH_SB_R0a (# 06537h) (D0, D1, A, B, C, D) : Pose le contenu de R0a sous forme de System Binary au niveau 1 de la pile.

PUSH_SB_R0a_R1a_(# 06529h) (D0, D1, A, B, C, D, R0) : Pose de même les contenus de R0a et R1a sur la pile sous forme de System Binaries. R0a au niveau 2, R1a au niveau 1.

PUSH_BI_Awp (# 5422Ch) (D0, D1, A, B, C, D, R0, R1) : Pose Awp sous forme de Binary Integer au niveau 1 de la pile.

5) Gérer le clavier

Il existe différentes routines permettant de simplifier la gestion du clavier. Il faut noter deux choses :

- Les instructions C=IN et A=IN étant bugguées, il faut TOUJOURS les remplacer par les GOSBVLs correspondants.

- Les routines relatives à la gestion du buffer clavier ne fonctionnent que lorsque les interruptions sont autorisées.

A_IN (# 0115Ah) (A) : Réalise l'opération A=IN. Cet appel est obligatoire pour réaliser cette fonction car cette instruction est buggée.

C_IN (# 01160h) (C) : Réalise l'opération C=IN, cette instruction étant également bugguée.

C_OUT_IN (# 01EECh) (C) : Réalise successivement les opérations OUT=C et C=IN.

EMPTY_BUFFER (# 00D57h) (D1, C) : Vide le buffer clavier.

KEY_IN_BUFFER (# 04999h) (D1, C) : Teste la présence de touches en attente dans le buffer clavier. La carry est effacée si celui-ci est vide.

READ_KEY (# 04840h) (D1,C) : Teste la présence d'une touche dans le buffer clavier. Passe la Carry à 1 si le buffer est vide. Dans le cas contraire, celle-ci est effacée, le code de la touche à traiter est stocké dans Cb, le buffer est remis a jour.

PUSH_KEY (# 00D9Fh) (D1, A, C) : Pose la touche dont le code est dans Cb dans le buffer clavier. Cette routine,

généralement totalement inutile s'avère cependant particulièrement utile pour débugguer des routines de test de touche.

6) Gérer l'affichage

Il existe différentes routines permettant de réaliser des opérations graphiques.

D0_PIX_IN_GROB (# 01C31h) (D0, A) : Fait pointer D0 et Aa sur le premier quartet du grob courant.

D0_PIX_IN_MENU (# 01C58h) (D0, A) : Fait pointer D0 et Aa sur le premier quartet du grob du menu courant.

CONFGRAPH (# 01C7Fh) (D0, D1, A, B, C, D) : Replace les marges, écrans, menu d'après les valeurs stockées en Ram réservée.

7) Gérer la mémoire

Il existe plusieurs types de routines différentes : des routines de réservation de mémoire, des routines de création d'objets et des routines permettant des tests : test de mémoire disponible, test de taille de port...

GARB_COLL (# 0613Eh) (D0, D1, A, B, C, D) : Libère de la mémoire en détruisant tous les objets temporaires dont la HP48 n'a plus utilité.

COPY_DN (# 670Ch) (D0, D1, A, C) : Copie Ca quartets pointés par D0 à partir de l'adresse contenue dans D1, D0 et D1 pointent sur l'adresse de début de la zone.

COPY_UP (# 66B9h) (D0, D1, A, C) : De même que COPY_DN mais dans le cas ou D0 et D1 pointent sur la fin de la zone.

RES_ROOM (# 039BEh) (D0, D1, A, B, D) : Réserve Ca quartets. L'adresse de début de la zone réservée est stockée dans D0. Si la mémoire disponible est insuffisante, il y a sortie du programme et émission du message d'erreur "Insuffisant Memory".

RES_STR (# 05B7Dh) (D0, D1, A, B, C, D, R0) : Réserve une chaîne de caractères dont le contenu fait Ca quartets. L'adresse de la chaîne créée est stockée dans R0a, son contenu est pointé par D0. Comme pour RES_ROOM, si la mémoire disponible est insuffisante, il y a sortie du programme et émission du message d'erreur "Insuffisant Memory".

UNRES_STR (# 16671h) (D0, D1, A, B, C, D, R0, R1) : Réalise l'opération inverse de RES_STR. Libère une zone mémoire réservée par RES_STR. Prend en entrée l'adresse de début de Chaîne dans R0a et l'adresse du premier quartet inutile dans D0.

FREE_MEM (# 06806h) (D0, A, C) : Calcule la mémoire disponible et renvoie cette valeur dans Ca. Attention : la valeur renvoyée n'est valable que dans la mesure où les registres ont bien été sauvegardés auparavant.

SKIP (# 03019h) (D0, A, C) : Prend en entrée l'adresse d'un objet dans D0. Renvoie dans D0 l'adresse du premier quartet après cet objet.

CR_XLIB (g/gx uniquement : # 7B79h) (D0, D1, A, B, C, D) : Crée un objet Xlib Name dont le numéro de Librairie est dans R2x, numéro de commande dans R3x. Renvoie l'adresse de l'objet créé dans Aa.

CR_GLOBAL_NM (g/gx uniquement : # 7BAFh) (D0, D1, A, B, C, D, R2) : Crée un objet Global Name. Les arguments sont

pointés par D0 : d'abord le nombre de caractères sur deux quartets, puis chaque caractère stocké par son code ASCII. L'adresse de l'objet créé est renvoyée dans Aa.

PORT_MEM (# 1DA1h) (D0, A, C) : Teste la taille d'un port non protégé en écriture. Prend en entrée l'adresse de début de port dans Ba, Renvoie dans D0 l'adresse du premier quartet après ce port.

BASE_D0,A (s/sx : # C1B0h, g/gx : # C1C5h) (D0, A) : Renvoie dans le cinquième quartet de D0 et de Aa l'adresse de base de la Ram interne. (# 70000h ou # F0000h pour les s/sx, # 80000h, # 70000h ou # F0000h pour les g/gx)

BASE_D0,C (s/sx : # C192h, g/gx : # C1A7h) (D0, C) : De même avec D0 et Ca.

BASE_D1,A (s/sx : # C1A1h, g/gx : # C1B6h) (D1, A) : De même avec D1 et Aa.

BASE_D1,C (s/sx : # C183h, g/gx : # C198h) (D1, C) : De même avec D1 et Ca.

8) Effectuer des calculs.

Il existe différentes routines en Rom permettant d'effectuer des calculs non directement accessibles par le SATURN. Ces routines sont généralement relativement rapides.

MULT_Aa,Ba (# 03991h) (A, B, C) : Réalise la multiplication des registres Aa et Ca. Le résultat est stocké dans Ba, les débordements ne sont pas gérés.

MULT_Aw,Bw (# 53EE4h) (A, C, D) : Réalise la multiplication des registres Aw et Cw. Le résultat est placé dans ces deux registres. Les débordements ne sont pas gérés.

DIV_Aw,Cw (# 65807h) (A, B, C) : Divise Aw par Cw, le quotient est stocké dans Aw et Cw, le reste dans Bw. Un débordement est impossible.

DIV5_Ca (# 6A8Eh) (A, C, D) : Divise par cinq Ca. Le résultat est renvoyé dans ce même registre.

Aw_SL5 (# 0D5F6h) (A) : Réalise cinq fois l'opération ASL sur le champ W de A.

Aw_SR5 (# 0D5E5h) (A) : De même, réalise cinq fois l'opération ASR sur le champ W de A.

Cw_SL5 (# 0D618h) (C) : Réalise cinq fois l'opération CSL sur le champ W de C.

Cw_SR5 (# 0D607h) (C) : De même, réalise cinq fois l'opération CSR sur le champ W de C.

CRC_D0 (# 5981h) (D0, D1, A, C) : Calcule le CRC de Aa quartets pointés par D0. Le résultat est stocké dans Aa.

9) Autres routines utiles

Il existe encore d'autres routines utiles.

BEEP (# 017A6h) (D0, D1, A, B, C, D, R0, R1, R2, R3) : Emet un bip dont la fréquence est dans Da, et dont la durée en

millisecondes est dans Ca. Cette émission n'a lieu que si les bips sont autorisés. (A l'aide du drapeau -56).

INTOFF2 (# 01115h) (C) : Interdit toute forme d'interruption.

INTON2 (# 010E5h) ( ) : Ré autorise les interruptions après un appel à INTOFF2 en # 01115h.

10) Conclusion

Cette liste n'est évidemment pas exhaustive et il existe une multitude d'autres routines en Rom. Il est possible, en se munissant d'un désassembleur d'en rechercher. Il s'agit d'un travail demandant beaucoup de patience et de rigueur. Il est cependant très intéressant car il permet de comprendre les techniques employées par les ingénieurs de Hewlett Packard face à certains problèmes courants. Il permet aussi d'acquérir un certain recul face aux programmes en langage machine.

LISTE DES LOGICIELS LES PLUS UTILES SUR HP 48

Nous allons vous présenter ici une liste des logiciels les plus utiles sur HP48. La plupart sont distribués sous forme de Free-ware. Quelques autres sont en Share-Ware.

1) Traitements de textes

A vrai dire, il n'existe que peu de traitements de textes sur HP48.

Mined par Elendril. : un bon éditeur de texte en moins de 10 Ko. Son utilisation est aisée et il comprend les fonctions de base d'un traitement de texte : couper ; copier ; coller. A notre connaissance, il n'est disponible que sur Hp48s/sx et aucune version g/gx ne semble prévue.

String Writer par J.Y. Avenard : la référence en matière de traitement de texte sur HP48. La version 4.2 fait un peu moins de 20 Ko et elle cumule toutes les fonctions que l'on est en droit d'attendre d'un traitement de texte : couper, copier, coller, tabulation, table ASCII, saut à une ligne donnée, fonctions de recherche et de remplacement ultra-rapides, possibilité d'afficher de 8 à 10 lignes... Il s'agit d'un Share-Ware et il vous en coûtera donc 50F, cependant, il s'agit d'un très bon investissement car il permet l'édition de sources assembleurs en toute aisance. A ne manquer sous aucun prétexte. La version 4.2 tourne indifféremment sur s/sx ou g/gx.

2) Programmation

Il existe deux types de programmations : La programmation RPL-External et la programmation Assembleur. Il existe de très bon logiciels pour cela.

Dev Library par E. de Foras : la référence pour la programmation RPL-External. Cette librairie apporte de nombreuses commandes permettant d'en faciliter l'accès, comme un "Externaliseur-Desexternaliseur", une commande INCLUDE permettant d'insérer le contenu de variables dans un programme sans avoir besoin de les retaper... Cette librairie facilite aussi la gestion de certains objets non gérés en standard par la pile, comme les System-Binaries, Les Long Real... Une très bonne acquisition sur un peu moins de 5 Ko. Il existe deux versions différentes : une version s/sx et une version g/gx.

FTL Dessas par DTS : Un complément intéressant pour LASM. Un désassembleur temps réel. Il s'avère particulièrement utile pour désassembler la Rom. Il fait un peu moins de 5Ko. Différentes adaptations en ont été faites sur g/gx.

KAsm : l'auteur nous en est inconnu. Un très petit programme (moins de 1 Ko) mais très utile. Il permet de rendre les listings de désassemblage réassemblables.

3) Gestion de fichiers, compactage

PCT par M. Vogel et R. Duchesne : LE gestionnaire de fichiers. Sur un peu moins de 20 Ko, il permet de très facilement gérer le menu Var et les ports. Il dispose de nombreuses fonctions permettant la copie, destruction, réorganisation de fichiers... De plus, il dispose d'un menu Custom permettant d'insérer ses propres fonctions dans la librairie. Un must à ne surtout pas manquer..

Redundance Fighter par L. Vieweg : même s'il se fait vieux, RF (c'est son nom) reste la référence en la matière. Relativement rapide pour les compactages et dé compactage, son gain est très appréciable : de 30 à 70 % selon le type d'argument. La version originale tourne sur HP48s/sx et fait environ 3 Ko. La HP48g/gx dispose, elle, de différentes adaptations.

4) Communication

FIRe 2 par Elendril : un logiciel gérant son propre protocole pour les transmissions infra-rouges. Le gain est appréciable car les transmissions s'effectuent au moins dix fois plus vite qu'avec Kermit.

HP BBT par D. Peretti : un logiciel gérant le protocole BBT. Il fonctionne aussi bien par câble que par infra-rouges. Il est relativement rapide. Il s'avèrent particulièrement utile dans les opérations de téléchargement car la plupart des serveurs français n'utilisent que ce protocole.

5) Graphisme

IFF Viewer par Atlas : un logiciel intéressant car il permet la conversion de fichiers graphiques au format IFF quatre couleurs en grobs spéciaux. Ces grobs peuvent être visualisés à l'aide d'un deuxième utilitaire de la librairie gérant l'affichage en niveaux de gris. (Comme dans le programme GRIS de la bibliothèque de programme). Sa taille étant très réduite (environ 3 Ko), ne pas le posséder est inexcusable. Il existe sur s/sx et g/gx.

True Color par Atlas : la prolongation d'IFF Viewer sous la forme d'un éditeur graphique entièrement en niveaux de gris. Son utilisation est très simple et permet d'obtenir des résultats très impressionnants. Ce logiciel est livré avec Julia, un générateur de

fractales en niveaux de gris très rapide. L'ensemble fait un peu moins de 5 Ko. Ce logiciel existe aussi sur s/sx et g/gx.

6) Divers

Lib maker par BOUHP : un logiciel INCONTOURNABLE. Il permet la création directe de Libraries sur la HP48 avec tous les avantages que cela apporte : Possibilité de cacher des sous-programmes, accès plus rapide aux données... Ce logiciel d'environ 6 Ko est disponible sur s/sx et g/gx.

7) Conclusion

Nous vous avons présenté ici une liste des logiciels indispensables sur HP48. cette liste n'est évidemment pas exhaustive, la logithèque de la HP48 comportant plusieurs milliers de programmes. On peut par exemple citer de nombreux excellents jeux comme Skwik, Boulder DasHP, Columns II, Pac Man, HPtris, Arkanoid, Arkanoiid... On peut aussi parler de nombreux utilitaires aux fonctions variées comme IREm qui émule les télécommandes infra-rouges ou AscLoader II qui permet la restitution de fichiers mal transférés...

Ces logiciels peuvent êtres obtenus de différentes façons :

Directement auprès de leur auteur, notamment pour String-Writer qui est un Share-Ware en envoyant votre commande à :

Jean-Yves AVENARD
16 rue de Savigny
91390 Morsang sur Orge

Par téléchargement, sur les serveurs de passionnés comme :

JEF : 47-95-28-10

Auprès de la société MAUBERT ELECTRONIC (Paris) pour les logiciels les plus récents

Il existe enfin de nombreux clubs HP comme par exemple :

CLUB HAUTE PERFORMANCE

Châteaugaillard, 1 av Foch

94700 Maisons Alfort

ou bien

CLUB HP ESME SUDRIA

17-19 Rue Mayet

75006 PARIS

ou ce club, publiant une revue en argot estudiantin d'excellente qualité :

HP'GANG

52, rue du Bois de Balizy

91360 Epinay sur Orge

Pour conclure, n'oubliez pas, pour les freewares, l'échange de la plupart de ces logiciels sont écrit pour être distribués de main en main.

COMMENT REPRENDRE LES LOGICIELS DE LA DISQUETTE D'ACCOMPAGNEMENT

Structure générale de la disquette

Elle est séparée en deux premiers sous répertoires. L'un pour les calculatrices de série S nommé "SX" et l'autre pour celle de série G nommé "GX".

Chacun de ces deux répertoires sont divisés en trois sous répertoires qui sont :

SOURCES : Répertoire qui contient les sources contenues dans la bibliothèque des programmes et prêts à être assemblés et modifiés.

EXE : Contient les programmes exécutables de la bibliothèque des programmes.

UTILS : Contient des utilitaires pour la HP 48 comme l'assembleur ou un éditeur de texte. Ce répertoire contient de même certaines documentations de programmes.

Sur un PC

Vous devez posséder le logiciel KERMIT pour DOS qui est un logiciel du domaine publique. En premier lieu copiez sur votre disque dur la disquette. En suite lancez KERMIT, allez dans le répertoire qui contient le programme et tapez "SEND" et le nom du

fichier à envoyer. Connectez votre HP avec votre PC et mettez votre HP en mode réception par câble (WIRE).

Sur un macintosh

Pour utiliser la disquette fournie avec le livre, vous devez, en un premier temps, la convertir avec l'utilitaire Apple File Exchange qui est fourni en standard avec votre Mac. Vous pouvez néanmoins utiliser tout autre convertisseur PC / Mac.

Quand vous avez fini de convertir la disquette, allez dans Kermit qui est, en général, fourni quand vous achetez le câble de liaison HP. Mettez vous en mode binaire aussi bien sur votre Mac que sur votre HP. Cependant sur votre HP vous devez aussi vous mettre en mode "wire". Il ne vous reste plus qu'à envoyer vos fichiers.

Remarque : Si lors de votre réception vous recevez un fichier qui commence par "HPHP48" utilisez le programme "ASCBIN", dont les codes hexas vous sont donnés ci-dessous.

```
CCD20  C8000  8FB97  60808  F1471  35143  34C2A  208A2
01342  02008  F08F0  11741  47818  FA5D5  17434  CF820
1438A  66063  10CD5  1F808  08F9B  83117  41338  F2D76
01418  F2D76  08080  14216  4808C
```

CRC : # 304Eh (S et G)

Glossaire

Adresse — Entier de 5 quartets permettant de repérer une zone particulière de la mémoire.

Algorithme — Processus permettant d'arriver à un résultat donné.

And — Opération logique "et".

Assemblage — Action d'assembler.

Assembler — Traduire un programme-assembleur en langage-machine afin de le rendre compréhensible par l'ordinateur.

Assembleur — Langage symbolique représentant le langage-machine. Désigne aussi le logiciel responsable de l'assemblage du programme-assembleur.

Base — Ensemble de symboles permettant de dénombrer.

Binaire — Base de calcul constituée des deux seuls chiffres 0 et 1.

Bit — Entier binaire, la plus petite unité gérable par la machine.

Bit de poids faible Le bit 0 d'un entier hexadécimal.

Bit de poids fort Le bit de numéro le plus élevé dans un entier hexadécimal..

Bug De l'anglais "petite bête qui gratte", le bug est la hantise du programmeur. Il s'agit en effet d'une erreur de programmation, nom pas dans la syntaxe, mais dans le déroulement du programme. Il s'agit donc d'une erreur que l'assembleur ne peut détecter.

Byte Nom anglo-saxon de l'octet.

Champ Sous-division d'un registre. De taille diverse (un quartet, cinq quartet...). Deux champs peuvent se chevaucher (A et X par exemple.).

Compactage Action de compacter.

Compacter Transformer un objet en un objet équivalent, mais de taille inférieur. Pour ce faire, on emploie des algorithmes souvent compliqués.

C.P.U. Central Process Unit Nom anglo-saxon du microprocesseur.

Décaler Effectuer un décalage.

Décalage Permuter des éléments d'un ensemble de façon ordonnée. Chaque éléments prenant la place de son "voisin".

Décimal Base de calcul "courante". Elle est constituée des dix chiffres 0, 1, 2, 3, 4, 5, 6, 7, 8 et 9

Décompactage Action de décompacter.

Décompacter Restituer l'objet initial à partir de l'objet compacté.

Désassemblage Action de désassembler.

Désassembler Transformer un programme en langage-machine en un programme assembleur.

Désassembleur Programme réalisant le désassemblage.

Drapeau Registre réduit à un bit, et ne pouvant donc prendre que deux valeurs 0 ou 1.

Flag Nom anglais du drapeau.

Garbage Collector Libération de mémoire par destruction d'objets temporaires devenus inutiles.

Hexadécimal Base de calcul introduite afin de faciliter les calculs sur les entiers binaires. Elles est constituée des seize chiffres 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E et F.

Kilo- Préfixe multiplicateur. En informatique, le kilo ne vaut pas 1000 mais 1024.

Ko Abréviation de kilo-octet. Un Ko vaut 1024 octets.

Mémoire Zone d'un ordinateur où sont stockées les informations qu'il utilise.

Mémoire de masse Type de mémoire pouvant être séparée de la machine. Disquettes, cassettes.. dans le cas de l'ordinateur. Cartes Ram dans le cas de la HP 48.

Mémoire morte Zone de mémoire non modifiable. C'est ici que sont stockées les commandes de la machine, son système...

Mémoire vive Par opposition à la mémoire morte, zone de mémoire modifiable. C'est ici que sont stockées les informations temporaires, les variables, les programmes...

Méga- Préfixe multiplicateur. En informatique, le méga ne vaut pas 1000000 mais 1024*1024=1048576.

Mégahertz Unité permettant d'apprécier la vitesse de calcul d'un microprocesseur. Un mégahertz correspond à un million de cycles par seconde. Le SATURN des HP 48s/sx tourne à environ deux mégahertz. Celui des HP 48g/gx à environ quatre mégahertz.

Microprocesseur Le coeur de votre machine l'entité qui exécute les programmes en langage machine. Celui de la HP 48 est un SATURN 4 BITS.

MIPS Million Instructions Per Second, soit Million d'instructions par seconde.

Mo Abréviation de méga-octet. 1 Mo vaut 1024 K.O. ou 1048576 octets.

Mot Entier codé sur 16 bits. Peut prendre toutes les valeurs entre # 0h et # FFFFh.

Not Opération logique "non".

Objet Entité gérable par la machine. Voir "Voyage au centre de la HP 48" pour de plus amples détails.

Octet Entier codé sur 8 bits. Peut prendre toutes les valeurs entre # 0h et # FFh.

Or Opération logique "ou"

Pile Zone de mémoire où les objets sont stockés les uns à la suite des autres. Le dernier objet stocké étant le seul accessible .

RAM Random Acces Memory. Nom anglo-saxon de la mémoire vive.

Registre Mémoire du microprocesseur. Un registre est sous-divisé en champs.

ROM Read Only Memory. Nom anglo-saxon de la mémoire morte.

RPL Reverse Polish Lisp. Langage externe de la HP 48.

RPN Reverse Polish Notation. Notation polonaise inversée. Système de calcul où les objets sont empilés avant opération.

SATURN Nom du microprocesseur équipant votre machine. Il y a eu plusieurs générations de Saturns. Le premier équipait le HP 71, le dernier en date équipe les HP 48g/gx.

Source Nom donné au texte représentant le programme avant son assemblage.

Sprite Objet graphique de petite taille.

Xor Opération logique "ou exclusif".

INDEX

$ 49
->MACRO 139
->OBJECT 134
@ 48

A

addition 31
affichage d'un écran 84
affichage HARD 90
affichage SOFT 90
AND 38
ASCII 49
ASM NEWOB 177
ASM REPL 165
assembleur 47

B

base 11
BEBETE 187
binaire 14
Bit de poids faible 26
Bit de poids fort 26
Bits de poids faible 26
Bits de poids forts 26
boucle 52; 60
Byte 24

C

caractère 24
Carry 106
carry 60
champ 52; 57
clavier 67
Code 47
commentaires 53
complémentation à deux 41
constante 53
couple OUT / IN 70
Couples OUT / IN des touches 67
CRNAME 174

D

décalage 32
DECOMP 221
Décrémentation 54
désassemblage 47
Dev Library 240
DIGITCONV 155
DIGITPLAY 159
DIGITSOUND 154
division 32
DOUBLE BUFFER 200
DROP 79
DUMPER 143

F

FIRe 2 241
FTL Dessas 240

G

GASS 132
gestion circulaire 30
Gestion d'une table 201
giga-octet 24
GRAPHICS VIEWER (HARD) 152
GRAPHICS VIEWER (SOFT) 149
GRIS 171

H

hexadécimale 11; 14
HP BBT 241
HPTRON 179
HPUISSANCE 4 206

I

IFF Viewer 241
instruction 52
intelligence artificielle 201; 217
interruption 69

K

KAsm 240
kilo-octe 24

L

label 54
LABY 163
langage machine 47
LASM 48
Lib maker 242

M

Macro 49
méga-octet 24
mémoire vive 48
microprocesseur 47
Mined 239
mnémonique 47
mode de calcul 108
multiplication 32

N

Nibble 23
niveaux de gris 200
NOT 38
Nybble 23

O

OBJECT-> 137
octet 24

P

PCT 240
point d'entrée 69
pointeur 76

Q

quartet 23

R

Redundance Fighter 241
registre 52
registres de sauvegarde 97
registres-drapeaux 103
réservation de mémoire 175
Retenue 60
RPL 48
RSTK 98

S

SATURN 54
Saut inconditionnel 62
sauts conditionnels 116
Sortie 56
source 49
soustraction 32
sprite 90
String Writer 239
SWAP 78
syncro VBL 173

T

Test 55
tester une touche 66
True Color 241

V

variable 29

W

W 57; 92; 96

X

XOR 39

NOTES PERSONELLES

Impression : EUROPE MEDIA DUPLICATION S.A.
F 53110 Lassay-les-Châteaux
N° 3547 - Dépôt légal : Novembre 1994

Voulez-vous apprendre un langage des centaines de fois plus rapide que le langage classique de la HP 48, un langage où tout est possible ? Voulez-vous enfin connaître les réelles capacités de votre machine ? Oui ? Alors L'INVITATION AU VOYAGE est fait pour vous.

Véritable manuel d'initiation, cet ouvrage vous conduira par l'exemple, des bases à une haute maîtrise de l'assembleur Saturn.

Mais attention ! Ce livre n'est pas seulement un recueil de cours illustrés, c'est aussi une formidable bibliothèque de programmes largement commentés pour vous apprendre de multiples techniques de programmation (gestion de plusieurs niveaux de gris sur l'écran de la HP 48 à travers un jeu d'arcade-réflexion, gestion de sons digitalisés, gestion de la mémoire, des objets ...)

L'INVITATION AU VOYAGE, c'est enfin une disquette regroupant, bien entendu tous les programmes du livre, mais aussi tous les outils utiles au programmeur sur HP 48 (assembleur, éditeur de texte ...) ainsi qu'une sélection des meilleurs logiciels du domaine publique.



ISBN 2-87892-008-2