

VALEUR DECIMALE SIGNÉE	REPRESENTATION						DECIMALE NON SIGNÉE
	HEXA	BINAIRE					
-32768	\$8000	1	000	0000	0000	0000	32768 (2 ¹⁵)
-32767	\$8001	1	000	0000	0000	0001	32769
.	.	.	.	.	.	.	.
-2	\$FFFE	1	111	1111	1111	1110	65534
-1	\$FFFF	1	111	1111	1111	1111	65565 (2 ¹⁶ -1)
0	\$0000	0	000	0000	0000	0000	0
1	\$0001	0	000	0000	0000	0001	1
2	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
32766	\$7FFE	0	111	1111	1111	1110	32766
32767	\$7FFF	0	111	1111	1111	1111	32767 (2 ¹⁵ -1)

Fig. 4.1 La représentation des entiers

Cette représentation explique l'apparition de nombres négatifs lorsque nous dépassons la valeur +MAXINT. Par exemple, si vous exécutez:

```
READLN(valeur);
deborde:= 5* valeur;
WRITELN(5* valeur);
```

et que vous tapez 10000, vous verrez apparaître -15536 (50000 est en effet égal à 65536- 15536). Par contre le compilateur rejettera des valeurs dépassant MAXINT:

```
entier:= 40000;
```

et au moment de l'exécution:

```
READLN(valeur);
```

provoquera une erreur 10 si vous tapez 40000.

4.1.1.3 IMPLANTATION DES ENTIERES

Au point de vue implantation mémoire ou disquette, l'octet de plus faible poids est placé d'abord, l'octet de poids le plus fort vient ensuite. Cette inversion des deux octets d'un entier provient essentiellement du langage machine du 8088 qui range les adresses en plaçant toujours l'octet de plus faible poids avant l'octet de plus fort poids.

Ainsi, les déclarations et affectations suivantes:

```
PROGRAM entiers;
VAR entier1: INTEGER;
    entier2: INTEGER;
    entier3: INTEGER;
```

```
BEGIN
    entier1:= 1;
    entier2:= 2;
    entier3:= 3;
END.
```

correspondront pour les données à l'occupation mémoire que voici:

```
adresse 0000 0001  entier1
adresse+1 0000 0000
adresse+2 0000 0010  entier2
adresse+3 0000 0000
adresse+4 0000 0011  entier3
adresse+5 0000 0000
```

Cette inversion de l'ordre de stockage des octets d'un mot de 2 octets se rencontre fréquemment pour les données stockées dans un seul mot.

4.1.1.4 EXAMEN MEMOIRE

Pour examiner un entier en mémoire, nous utilisons le programme d'affichage mémoire présenté en début de chapitre, avec une procédure spécifique à l'entrée d'un entier:

```
0 (* 001 affentie *)
1 (* 08 avr 87 *)
2
3 (* $r+*)
4 PROGRAM examen_entier;
5 TYPE t_array 16 caracteres= ARRAY[0..15] OF CHAR;
6 VAR choix: CHAR;
7     avant: t_array 16 caracteres;
8     entier: INTEGER;
9     apres: t_array 16 caracteres;
10
11 PROCEDURE affiche_memoire(p_segment, p_decalage_debut,
...
71 END; (* affiche_memoire *)
72
73 PROCEDURE affiche_entier;
74 BEGIN
75     WRITE('entier <35> ? ');
76     READLN(entier);
77     affiche_memoire(SEG(entier), OFS(entier)- 16,
78                     OFS(entier)+ SIZEOF(entier)+ 16- 1);
79 END; (* affiche_entier *)
```



```
79  
80 PROCEDURE initialise;  
81 BEGIN  
82   avant:= 'AAAAAAAAAAAAA';  
83   apres:= 'BBBBBBBBBBBBBBB';  
84 END; (* initialise *)  
85  
86 BEGIN (* principal *)  
87   initialise;  
88   REPEAT  
89     WRITELN;  
90     WRITE('e(ntier, q(uite ? ');  
91     READ(KBD, choix); WRITELN(choix);  
92     CASE choix OF  
93       'e' : affiche_entier;  
94     END;  
95     UNTIL choix= 'q';  
96 END. (* principal *)
```

Fig. 4.2 L'affichage d'une variable entière

Voici un exemple d'exécution de ce programme:

```
e(ntier, q(uite ? e  
entier <35> ? $1234  
  
$1E83:$0260          $41 $41 $41 $41 $41 : AAAAA  
$1E83:$0268 $41 $41 $41 $41 $41 $41 $41 : AAAAA  
$1E83:$0270 $41 $41 $41 : AAA  
$1E83:$0270          $34 $12 : 4.  
$1E83:$0270          $42 $42 $42 : BBB  
$1E83:$0278 $42 $42 $42 $42 $42 $42 $42 : BBBBBBBB  
$1E83:$0280 $42 $42 $42 $42 $42 : BBBBB  
  
e(ntier, q(uite ? q
```

Fig. 4.3 Un entier en mémoire

4.1.1.5 RESUME

Un entier:

- peut prendre les valeurs non fractionnaires entre -32767 et 32768
- est représenté en utilisant 16 bits avec une représentation de complément à 2
- occupe 2 octets en mémoire, l'octet de poids le plus faible précédant l'octet de poids le plus fort

4.1.2 LE TYPE CARACTERE

4.1.2.1 DEFINITION DU TYPE CARACTERE

Le type caractère comprend les caractères utilisés par PASCAL.

Pour les besoins syntactiques de PASCAL, il faut pouvoir disposer:

- des lettres de l'alphabet en majuscule
- des 10 chiffres
- des caractères ' () * = - : + ; / . > , < # ^
- d'un caractère marquant la fin de la ligne

Nous souhaitons, en plus, pouvoir utiliser dans les variables les caractères tels que ? ou %, ou écrire nos programmes en utilisant des minuscules.

4.1.2.2 REPRESENTATION DU TYPE CARACTERE

En tout il nous faut donc environ 70 caractères.

Par conséquent 7 bits, permettant de représenter 2^7 , soit 128 valeurs distinctes, suffisent amplement.

TURBO PASCAL utilise la représentation ASCII étendue (American Standard Code for Information Interchange). Dans cette représentation, le caractère A, par exemple, correspond aux 7 bits suivants: 100 0001. Cette valeur s'appelle le CODE ASCII du caractère A. Sa valeur décimale est 65, sa valeur hexadécimale est \$41.

Le code ASCII normal comporte en tout 128 caractères, et le tableau des correspondances entre la valeur décimale, hexadécimale et le caractère est donné ci-dessous:

Déc	Hex	Car	Déc	Hex	Car	Déc	Hex	Car	Déc	Hex	Car
0	\$00	NUL	32	\$20	SP	64	\$40	@	96	\$60	'
1	\$01	SOH	33	\$21	!	65	\$41	A	97	\$61	a
2	\$02	STX	34	\$22	"	66	\$42	B	98	\$62	b
3	\$03	ETX	35	\$23	#	67	\$43	C	99	\$63	c
4	\$04	EOT	36	\$24	\$	68	\$44	D	100	\$64	d
5	\$05	ENQ	37	\$25	%	69	\$45	E	101	\$65	e
6	\$06	ACK	38	\$26	&	70	\$46	F	102	\$66	f
7	\$07	BEL	39	\$27	'	71	\$47	G	103	\$67	g

8 \$08 BS	40 \$28 (	72 \$48 H	104 \$68 h
9 \$09 HT	41 \$29)	73 \$49 I	105 \$69 i
10 \$0A LF	42 \$2A *	74 \$4A J	106 \$6A j
11 \$0B VT	43 \$2B +	75 \$4B K	107 \$6B k
12 \$0C FF	44 \$2C ,	77 \$4C L	108 \$6C l
13 \$0D CR	45 \$2D -	77 \$4D M	109 \$6D m
14 \$0E SO	46 \$2E .	78 \$4E N	110 \$6E n
15 \$0F DI	47 \$2F /	89 \$4F O	111 \$6F o
16 \$10 DLE	48 \$30 0	80 \$50 P	112 \$70 p
17 \$11 DC1	49 \$31 1	81 \$51 Q	113 \$71 q
18 \$12 DC2	50 \$32 2	82 \$52 R	114 \$72 r
19 \$13 DC3	51 \$33 3	83 \$53 S	115 \$73 s
20 \$14 DC4	52 \$34 4	84 \$54 T	116 \$74 t
21 \$15 NAK	53 \$35 5	85 \$55 U	117 \$75 u
22 \$16 SYN	54 \$36 6	86 \$56 V	118 \$76 v
23 \$17 ETB	55 \$37 7	87 \$57 W	119 \$77 w
24 \$18 CAN	56 \$38 8	88 \$58 X	120 \$78 x
25 \$19 EM	57 \$39 9	89 \$59 Y	120 \$78 x
26 \$1A SUB	58 \$3A :	90 \$5A Z	122 \$7A z
27 \$1B ESC	59 \$3B ;	91 \$5B [	123 \$7B {
28 \$1C FS	60 \$3C <	92 \$5C \	124 \$7C]
29 \$1D GS	61 \$3D =	93 \$5D ^	125 \$7D _
30 \$1E RS	62 \$3E >	94 \$5E ^	126 \$7E `
31 \$1F US	63 \$3F ?	95 \$5F _	127 \$7F DEL

Fig. 4.4 Le code ASCII

Quelques remarques sur ce code:

- les 32 premiers caractères ne correspondent pas tous à des touches du clavier. Il s'agit là de caractères utilisés au moment de la création du code pour les protocoles de transmissions. Par exemple ACK (ACKnowledge: accuser réception) indiquait à l'émetteur que son message était bien arrivé. Certains sont encore utilisés, comme le CR (RETURN), BS (flèche arrière) ou ESC (ESCape).
- du fait de la monopolisation des premiers numéros par ces antiques codes de transmissions, les codes des chiffres n'ont pas pour valeur le chiffre. Heureusement que les codes des chiffres se suivent, dans l'ordre et sans codes intermédiaires, ce qui était la contrainte que WIRTH avait posée pour pouvoir utiliser un code pour PASCAL. Ces conditions facilitent en effet les conversions caractère / valeurs numériques (à défaut d'avoir pour code d'un chiffre le chiffre même).

Le code ASCII étendu, défini par IBM au moment du lancement du PC, utilise les codes 128 à 255 pour coder des caractères tels que des lettres accentuées, des symboles (intégration), des lettres étrangères (la lettre alpha grecque) ou des dessins graphiques (un angle ou un "T"). Tous les codes entre 0 et 255 sont donc associés à une image, bien que certains codes ne puissent pas être saisis au clavier par la frappe d'une seule touche.

Voici les dessins prévus par IBM:

DECIMAL VALUE	HEXA DECIMAL VALUE	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
0	0	BLANK (NULL)	BLANK (SPACE)	0	@	P	'	p	¢	£	ä	å	⌘	⌘	⌘	⌘	⌘
1	1	☺	☹	!	1	A	Q	a	q	ü	Æ	í	⌘	⌘	⌘	⌘	β ±
2	2	☹	↑	"	2	B	R	b	r	é	FE	ó	⌘	⌘	⌘	⌘	γ ≥
3	3	♥	!!	#	3	C	S	c	s	â	ô	ú	⌘	⌘	⌘	⌘	π ≤
4	4	♦	¶	\$	4	D	T	d	t	ä	ö	ñ	⌘	⌘	⌘	⌘	Σ ∫
5	5	♣	§	%	5	E	U	e	u	à	ò	Ñ	⌘	⌘	⌘	⌘	σ ∫
6	6	♠	■	&	6	F	V	f	v	â	û	ä	⌘	⌘	⌘	⌘	μ ÷
7	7	•	↓	'	7	G	W	g	w	ç	ù	ø	⌘	⌘	⌘	⌘	τ ≈
8	8	•	↑	(	8	H	X	h	x	ê	ÿ	ï	⌘	⌘	⌘	⌘	Φ °
9	9	○	↓	)	9	I	Y	i	y	ë	Ö	⌘	⌘	⌘	⌘	⌘	Θ •
10	A	○	→	*	:	J	Z	j	z	è	Ü	⌘	⌘	⌘	⌘	⌘	Ω •
11	B	♂	←	+	;	K	I	k	{	ï	ç	½	⌘	⌘	⌘	⌘	δ √
12	C	♀	⌞	,	<	L	\	l		â	ℓ	¼	⌘	⌘	⌘	⌘	∞ η
13	D	♪	↔	=	M	J	m	}		ÿ	ï	⌘	⌘	⌘	⌘	⌘	∅ ²
14	E	♫	▲	.	>	N	^	n	~	À	Pt	«	⌘	⌘	⌘	⌘	∈ ■
15	F	☼	▼	/	?	O	_	o	Δ	Å	f	»	⌘	⌘	⌘	⌘	∩ BLANK FF

Fig. 4.5 Les dessins correspondant au code ASCII étendu

Pour représenter les 256 caractères du code ASCII étendu, TURBO PASCAL utilise un octet, dont la valeur est le code ASCII étendu du caractère.

4.1.2.3 EXAMEN DE LA MEMOIRE

Pour examiner un caractère en mémoire, nous utilisons le programme suivant:

```

0 (* 001 affcarac *)
1 (* 08 avr 87 *)
2
3 (* $r+*)
4 PROGRAM examen_caractere;
5   TYPE t_array_16_caracteres= ARRAY[0..15] OF CHAR;
```



```

6  VAR choix: CHAR;
7  avant: t_array 16_caracteres;
8  caractere: CHAR;
9  apres: t_array 16_caracteres;
10
11 PROCEDURE affiche_memoire(p_segment, p_decalage_debut,
...
71   END; (* affiche_memoire *)
72
73 PROCEDURE affiche_caractere;
74 BEGIN
75   WRITE('caractere <a> ? ');
76   READLN(caractere);
77   affiche_memoire(SEG(caractere), OFS(caractere)- 16,
78   OFS(caractere)+ SIZEOF(caractere)+ 16- 1);
79   END; (* affiche_caractere *)
80
81 PROCEDURE initialise;
82 BEGIN
83   avant:= 'AAAAAAAAAAAAAAAA';
84   apres:= 'BBBBBBBBBBBBBBBB';
85   END; (* initialise *)
...

```

Fig. 4.6 Le programme *examen_caractere*.

Et voici un exemple d'utilisation:

```

c(aractere, q(uite ? c
caractere <a> ? a

$152A:$0260          $41 $41 $41 $41 $41 : AAAAA
$152A:$0268 $41 $41 $41 $41 $41 $41 $41 :AAAAAAA
$152A:$0270 $41 $41 $41 $41 $41 $41 $41 :AAA
$152A:$0270          $61 : a
$152A:$0270          $42 $42 $42 $42 $42 :BBBB
$152A:$0278 $42 $42 $42 $42 $42 $42 $42 :BBBBBBBB
$152A:$0280 $42 $42 $42 $42 $42 $42 $42 :BBBB

c(aractere, q(uite ? q

```

Fig. 4.7 Les caractères en mémoire

4.1.2.4 LES CARACTERES ET LES PERIPHERIQUES

Les périphériques qui manipulent des caractères n'utilisent pas nécessairement le code ASCII étendu. Dans le cas du clavier, par exemple, l'enfoncement de la touche 'e' correspond à l'émission du code de balayage 18 vers le 8088. De même pour afficher un caractère à l'écran il faut non seulement envoyer le code du caractère mais encore son attribut (inverse, clignotant...).

Les transformations entre les différentes représentations et le code ASCII étendu sont effectuées par les couches du BIOS, de l'Operating Système ou de la librairie TURBO PASCAL. Ces représentations internes sont totalement transparentes (invisibles) pour le programmeur PASCAL. Les données de type CHAR dans les piles globales ou locales, sur le tas des variables dynamiques et sur les disquettes sont bien conformes au code ASCII étendu.

Pour utiliser une variable de type caractère, nous déclarons:

```
VAR caractere: CHAR;
```

et:

- pour lui affecter un caractère que nous pouvons taper au clavier, nous pouvons employer:
 - l'instruction d'affectation:


```
caractere:= '+';
```
 - l'instruction de lecture clavier:


```
READ(caractere);
```
- pour lui affecter un caractère que nous ne pouvons pas taper au clavier, nous pouvons employer:
 - l'instruction d'affectation et CHR, # ou le caractère ^:


```
caractere:= CHR(Nombre);
caractere:= #Nombre;
caractere:= ^M;
```
 - éventuellement, pour les caractères non filtrés par le système, certaines combinaisons de touches. Ainsi, actuellement:

```
READ(caractere);
```

et la frappe de CTRL J affecteront à *caractere* le caractère dont le code ASCII est 10 (parce que J est le 10 ième caractère de l'alphabet: cf la table ci-dessus). De même la frappe de ALT 219 placerait dans *caractere* le caractère de code ASCII 219.

4.1.2.5 RESUME

Le type caractère:

- peut prendre les 256 valeurs correspondant au code ASCII étendu
- est représenté sur 8 bits (un octet) en utilisant le code ASCII étendu
- occupe en mémoire 1 octet

4.1.3 LE TYPE BOOLEEN

4.1.3.1 DEFINITION DU TYPE BOOLEEN

Une variable booléenne peut prendre deux valeurs uniquement: **FALSE** et **TRUE**.

4.1.3.2 REPRESENTATION DU TYPE BOOLEEN

Pour représenter une variable booléenne, il suffit d'un bit.

La valeur 0 représente **FALSE**, et la valeur 1 représente **TRUE**.

TURBO PASCAL utilise un octet pour placer une valeur booléenne, 8 bits pour représenter une variable booléenne:

- le bit numéro 0 est celui qui est utilisé pour indiquer l'état **TRUE** ou **FALSE**:
 - si celui-ci a pour valeur 1, PASCAL considérera la valeur booléenne comme vraie
 - si ce bit a pour valeur 0, la valeur booléenne sera considérée comme fausse.
- les 7 autres bits ont en général la valeur zéro

Les instructions d'affectation telles que:

```
ok:= TRUE;
```

affectent aux 8 bits du booléen les valeurs suivantes:

```
Bit Numéro      7654 3210
```

```
Valeur du bit: 0000 0001 (FAIBLE)
```

alors que

```
ok:= FALSE;
```

lui affecte les valeurs:

```
Bit Numéro:      7654 3210
```

```
Valeur du bit: 0000 0000
```

Cela est encore exact pour les affectations résultant d'une évaluation:

```
fini:= adresse> 2537;
```

par exemple.

Notons toutefois que TURBO PASCAL considère que toute valeur non nulle comme **TRUE**. De telles valeurs peuvent résulter de surtypage provenant de l'utilisation d'enregistrements variants, de l'instruction **ABSOLUTE** ou de l'instruction **MOVE**. Ainsi:

```
VAR boolean: BOOLEAN;
    octet: BYTE ABSOLUTE boolean;

octet:= 4;
WRITELN(boolean);
```

affichera **TRUE**, bien que le bit 0 soit égal à zéro.

Les instructions **OR** et **NOT** ne manipulent que le bit 0 du booléen, alors que l'instruction **AND** remet systématiquement les bits 1 à 7 à 0. Ce comportement provient du code 8088 utilisé pour mettre en oeuvre **AND OR** et **NOT** (voir chapitre 3).

Notez que la valeur **TRUE** attribuée à toute valeur non nulle peut donc provoquer des comportements aberrants (dans le cas de l'exemple précédent où *booléen* était **TRUE**, **NOT booléen** serait aussi **TRUE**). Rappelons toutefois que ces phénomènes proviennent de surtypages, et que si vous vous absteniez d'employer de telles techniques, les booléens se comporteront de façon normale.

4.1.3.3 EXAMEN DE BOOLEENS

Voici le programme d'examen des booléens:

```
0 (* 001 affbool *)
1 (* 08 avr 87 *)
2
3 (*$r+*)
4 PROGRAM examen_booléen;
5   TYPE t_array_16_caracteres= ARRAY[0..15] OF CHAR;
6   VAR choix,choix1: CHAR;
7       avant: t_array_16_caracteres;
8       boolean: BOOLEAN;
9       apres: t_array_16_caracteres;
10
11  PROCEDURE affiche_memoire(p_segment, p_decalage_debut,
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71  END; (* affiche_memoire *)
72
```



```
73 PROCEDURE affiche_booléen;  
74 BEGIN  
75   WRITE(' booléen: t(rue, f(alse ? ');  
76   READ(KBD, choix1); WRITELN(choix1);  
77   booléen:= (choix1= 'r');  
78   affiche_memoire(SEG(booléen), OFS(booléen)- 16,  
79     OFS(booléen)+ SIZEOF(booléen)+ 16- 1);  
80 END; (* affiche_booléen *)  
81  
82 PROCEDURE initialise;  
83 BEGIN  
84   avant:= 'AAAAAAAAAAAAAAAA';  
85   apres:= 'BBBBBBBBBBBBBBBB';  
86 END; (* initialise *)  
...
```

Fig. 4.8 Le programme *examen_booléen*.

et voici un exemple d'exécution:

```
b(booléen, q(uitte ? b  
booléen: t(rue, f(alse ? t  
$14D6: $0260          $41 $41 $41 $41 : AAAA  
$14D6: $0268 $41 $41 $41 $41 $41 $41 $41 $41 : AAAAAAAA  
$14D6: $0270 $41 $41 $41 $41 : AAAA  
$14D6: $0270          $01 :  
$14D6: $0270          $42 $42 $42 : BBB  
$14D6: $0278 $42 $42 $42 $42 $42 $42 $42 $42 : BBBBBBBB  
$14D6: $0280 $42 $42 $42 $42 $42 : BBBBBB  
  
b(booléen, q(uitte ? b  
booléen: t(rue, f(alse ? f  
$14D6: $0260          $41 $41 $41 $41 : AAAA  
$14D6: $0268 $41 $41 $41 $41 $41 $41 $41 $41 : AAAAAAAA  
$14D6: $0270 $41 $41 $41 $41 : AAAA  
$14D6: $0270          $00 :  
$14D6: $0270          $42 $42 $42 : BBB  
$14D6: $0278 $42 $42 $42 $42 $42 $42 $42 $42 : BBBBBBBB  
$14D6: $0280 $42 $42 $42 $42 $42 : BBBBBB  
  
b(booléen, q(uitte ? q
```

Fig. 4.9 Un booléen vrai en mémoire

4.1.3.4 RESUME

Le type **BOOLEAN**:

- prend les seules valeurs **TRUE** et **FALSE**
- est représentée par un octet:
 - la valeur **FALSE** correspondant à un octet de valeur 0
 - la valeur **TRUE** correspondant à un octet de valeur non nulle. Si le booléen a été manipulé par des instructions logiques normales (pas par des surtypages), c'est le bit 0 qui est égal à 1
- **OR** et **NOT** ne modifient que le bit 0, alors que **AND** remet les bits 1 à 7 à zéro systématiquement

4.1.4 LE TYPE ENUMERE

4.1.4.1 DEFINITION DU TYPE ENUMERE

Le type utilisateur énuméré permet de définir des types dont les différentes valeurs portent des noms que nous avons choisis.

Ce type est essentiellement là pour améliorer la lisibilité des programmes, en effectuant le "codage" à la place du programmeur. Du fait de l'impossibilité dans de nombreuses versions PASCAL de les lire par **READLN** ou de les afficher par **WRITELN**, leur utilisation est limitée à des programmes communiquant peu avec l'utilisateur (compilateurs) ou à des tâches internes de programmes (suivi des erreurs dans un Operating System).

Le nombre maximal de valeurs distinctes est égale à **MAXINT** (32768).

4.1.4.2 REPRESENTATION DU TYPE ENUMERE

Le système utilise pour représenter le type énuméré:

- un nombre de bits suffisant pour représenter le nombre des valeurs définies ou déclarées
- chaque valeur est représentée par un nombre correspondant au numéro d'ordre de la valeur dans la définition ou la déclaration.

Prenons par exemple les jours de la semaine:

```
TYPE Type_jour= (Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi, Dimanche);  
VAR Jour: Type_jour;
```


Alors:

- il y a 7 valeurs à coder
- la première puissance de 2 supérieure à 7 est 8
- il faudra 3 bits pour représenter ce type énuméré ($2^3 = 8$)
- *Lundi* correspondra à une valeur décimale 0, *Mardi* à une valeur 1, *Mercredi* à une valeur 2 etc

Cette conversion entre les identificateurs choisis par le programmeur (*Lundi*) et les valeurs numériques (0) explique l'impossibilité de lecture par READLN ou d'affichage par WRITELN (encore qu'il serait assez simple de placer dans le code une table de conversion !).

4.1.4.3 IMPLANTATION DU TYPE ENUMERE

En mémoire ou sur disquette, une variable de type énuméré utilisera un nombre d'octets suffisant pour contenir les bits nécessaires au codage. Dans l'exemple précédent, un octet suffirait.

Si deux octets sont nécessaires, l'octet de poids faible est stocké en premier, l'octet de poids fort en second.

4.1.4.4 EXAMEN DES TYPES ENUMERES

Voici le programme qui nous permet d'examiner des types énumérés:

```
0 (* 001 affenum *)
1 (* 08 avr 87 *)
2
3 (* $r+*)
4 PROGRAM examen_enumere;
5 TYPE t_array_16_caracteres= ARRAY[0..15] OF CHAR;
6 VAR choix, choix1: CHAR;
7     avant: t_array_16_caracteres;
8     enumere: (lundi, mardi, mercredi, jeudi, vendredi);
9     apres: t_array_16_caracteres;
10
11 PROCEDURE affiche_memoire(p_segment, p_decalage_debut,
12 ...
71 END; (* affiche_memoire *)
72
73 PROCEDURE affiche_enumere;
74 BEGIN
75     WRITE(' enumere: l(undi, m(ardi, me(rcredi, j(eudi, v(endredi ? ');
76     READ(KBD, choix1); WRITELN(choix1);
77     CASE choix1 OF
78         'l' : enumere:= lundi;
79         'm' : enumere:= mardi;
80         'e' : enumere:= mercredi;
81         'j' : enumere:= jeudi;
82         'v' : enumere:= vendredi;
83     END;
```

```
84 affiche_memoire(SEG(enumere), OFS(enumere)- 16,
85     OFS(enumere)+ SIZEOF(enumere)+ 16- 1);
86 END; (* affiche_enumere *)
87
88 PROCEDURE initialise;
89 BEGIN
90     avant:= 'AAAAAAAAAAAAAAAA';
91     apres:= 'BBBBBBBBBBBBBBBB';
92 END; (* initialise *)
...
```

Fig. 4.10 Le programme examen_enumere.

et voici un exemple d'exécution:

```
e(enumere, q(uitte ? e
enumere: l(undi, m(ardi, me(rcredi, j(eudi, v(endredi ? l
$1548:$0260                                $41 $41 $41 $41 : AAAA
$1548:$0268 $41 $41 $41 $41 $41 $41 $41 $41 : AAAAAAAAAA
$1548:$0270 $41 $41 $41 $41 $41 $41 $41 $41 : AAAA
$1548:$0270                                $00
$1548:$0270                                $42 $42 $42 $42 : BBB
$1548:$0278 $42 $42 $42 $42 $42 $42 $42 $42 : BBBBBBBB
$1548:$0280 $42 $42 $42 $42 $42 $42 $42 $42 : BBBBBB

e(enumere, q(uitte ? e
enumere: l(undi, m(ardi, me(rcredi, j(eudi, v(endredi ? m
$1548:$0260                                $41 $41 $41 $41 : AAAA
$1548:$0268 $41 $41 $41 $41 $41 $41 $41 $41 : AAAAAAAAAA
$1548:$0270 $41 $41 $41 $41 $41 $41 $41 $41 : AAAA
$1548:$0270                                $01
$1548:$0270                                $42 $42 $42 $42 : BBB
$1548:$0278 $42 $42 $42 $42 $42 $42 $42 $42 : BBBBBBBB
$1548:$0280 $42 $42 $42 $42 $42 $42 $42 $42 : BBBBBB

e(enumere, q(uitte ? e
enumere: l(undi, m(ardi, me(rcredi, j(eudi, v(endredi ? v
$1548:$0260                                $41 $41 $41 $41 : AAAA
$1548:$0268 $41 $41 $41 $41 $41 $41 $41 $41 : AAAAAAAAAA
$1548:$0270 $41 $41 $41 $41 $41 $41 $41 $41 : AAAA
$1548:$0270                                $04
$1548:$0270                                $42 $42 $42 $42 : BBB
$1548:$0278 $42 $42 $42 $42 $42 $42 $42 $42 : BBBBBBBB
$1548:$0280 $42 $42 $42 $42 $42 $42 $42 $42 : BBBBBB

e(enumere, q(uitte ? q
```

Fig. 4.11 Le type énuméré en mémoire

4.1.4.5 RESUME

Le type énuméré:

- a pour valeur les valeurs définies ou déclarées par l'utilisateur
- utilise un nombre de bits suffisant pour représenter le nombre de valeurs du type. Chaque valeur est représentée par son rang.
- en mémoire:
 - si le type comporte moins de 256 valeurs, 1 seul octet est nécessaire
 - si le type comporte plus de 256 valeurs, 2 octets sont utilisés, l'octet de poids faible étant stocké en premier

4.1.5 LE TYPE INTERVALLE NUMERIQUE

4.1.5.1 DEFINITION DU TYPE INTERVALLE

Le type utilisateur intervalle définit une partie de l'intervalle -32568..32767

Il est ainsi possible de définir, par exemple, 0..45 ou -32..423.

Ce type est surtout utilisé:

- comme indice de tableaux
- pour limiter la taille en mémoire ou sur disque de valeurs numériques comprises entre 0 et 255

4.1.5.2 REPRESENTATION DU TYPE INTERVALLE

La représentation du type intervalle est la même que celle des entiers: 16 bits maximum, en complément à 2.

4.1.5.3 IMPLANTATION EN MEMOIRE

L'implantation est fondamentalement celle des entiers, avec utilisation d'un seul octet lorsque c'est possible:

- si les deux bornes sont positives et toutes les deux inférieures à 256 (par exemple 2..250), un seul octet est utilisé
- si l'une des bornes est négative (par exemple -1..3) ou si l'une des bornes dépasse 256 (par exemple 250..260), l'intervalle utilisera 2 octets, l'octet de poids faible étant stocké en premier

Ce n'est donc pas l'étendue de l'intervalle qui détermine le nombre de bits utilisés, mais le fait que l'une des bornes est négative ou supérieure à 256. Ceci évite des calculs de décalage lorsque ces valeurs sont utilisées par la suite.

Notons aussi que TURBO PASCAL offre le type pré-déclaré BYTE qui n'est en fait que l'intervalle 0..255.

4.1.5.4 EXAMEN DU TYPE INTERVALLE

Voici un programme d'examen d'un intervalle:

```

0  (* 001 affinter *)
1  (* 08 avr 87 *)
2
3  (*$r+*)
4  PROGRAM examen_intervalle;
5  TYPE t_array_16_caracteres= ARRAY[0..15] OF CHAR;
6  VAR choix: CHAR;
7      avant: t_array_16_caracteres;
8      intervalle1: 7..74;
9      separe1: t_array_16_caracteres;
10     intervalle2: 510..514;
11     separe2: t_array_16_caracteres;
12     intervalle3: -15..18;
13     apres: t_array_16_caracteres;
14
15  PROCEDURE affiche_memoire(p_segment, p_decalage_debut,
16  ...
17  75  END; (* affiche_memoire *)
18
19  76
20  77  PROCEDURE affiche_intervalle1;
21  78  BEGIN
22  79      WRITE('intervalle1 [7..74] ? ');
23  80      READLN(intervalle1);
24  81      affiche_memoire(SEG(intervalle1), OFS(intervalle1)- 16,
25  82      OFS(intervalle1)+ SIZEOF(intervalle1)+ 16- 1);
26  83      END; (* affiche_intervalle1 *)
27
28  84  PROCEDURE affiche_intervalle2;
29  85  BEGIN
30  86      WRITE('intervalle2 [510..514] ? ');
31  87      READLN(intervalle2);
32  88      affiche_memoire(SEG(intervalle2), OFS(intervalle2)- 16,
33  89      OFS(intervalle2)+ SIZEOF(intervalle2)+ 16- 1);
34  90      END; (* affiche_intervalle2 *)
35
36  91  PROCEDURE affiche_intervalle3;
37  92  BEGIN
38  93      WRITE('intervalle3 [-15..18] ? ');
39  94      READLN(intervalle3);
40  95      affiche_memoire(SEG(intervalle3), OFS(intervalle3)- 16,
41  96      OFS(intervalle3)+ SIZEOF(intervalle3)+ 16- 1);
42  97      END; (* affiche_intervalle3 *)

```



```

97
98  PROCEDURE initialise;
99  BEGIN
100    avant:= 'AAAAAAAAAAAAAAAA';
101    separe1:= 'BBBBBBBBBBBBBBBB';
102    separe2:= 'CCCCCCCCCCCCCCCC';
103    apres:= 'DDDDDDDDDDDDDDDD';
104  END; (* initialise *)

```

Fig. 4.12 Le programme *examen_intervalle*.

Et voici un exemple d'utilisation:

```

intervalle 1(, 2(, 3, q(uite ? 1
intervalle1 [7..74] ? 31

$1519:$0260          $41 $41 $41 $41 $41 : AAAAA
$1519:$0268 $41 $41 $41 $41 $41 $41 $41 : AAAAAAA
$1519:$0270 $41 $41 $41 : AAA
$1519:$0270          $1F : 
$1519:$0270          $42 $42 $42 $42 : BBBB
$1519:$0278 $42 $42 $42 $42 $42 $42 $42 : BBBBBBBB
$1519:$0280 $42 $42 $42 $42 : BBBB

intervalle 1(, 2(, 3, q(uite ? 2
intervalle2 [510..514] ? 512

$1519:$0270          $42 $42 $42 $42 : BBBB
$1519:$0278 $42 $42 $42 $42 $42 $42 $42 : BBBBBBBB
$1519:$0280 $42 $42 $42 $42 : BBBB
$1519:$0280          $00 $02 : 
$1519:$0280          $43 $43 : .. CC
$1519:$0288 $43 $43 $43 $43 $43 $43 $43 : CCCCCCCC
$1519:$0290 $43 $43 $43 $43 $43 $43 : CCCCCC

intervalle 1(, 2(, 3, q(uite ? 3
intervalle3 [-15..18] ? -1

$1519:$0280          $43 $43 : .. CC
$1519:$0288 $43 $43 $43 $43 $43 $43 $43 : CCCCCCCC
$1519:$0290 $43 $43 $43 $43 $43 $43 : CCCCCC
$1519:$0290          $FF $FF : ..
$1519:$0298 $44 $44 $44 $44 $44 $44 $44 : DDDDDDDD
$1519:$02A0 $44 $44 $44 $44 $44 $44 $44 : DDDDDDDD

intervalle 1(, 2(, 3, q(uite ? q

```

Fig. 4.13 Un intervalle en mémoire

Attention, READLN ne signale aucune erreur lorsque vous introduisez des valeurs hors de l'intervalle au clavier (bien que nous soyons en mode R+):

- pour les intervalles utilisant 2 octets, aucune action ni modification n'ont lieu
- pour les intervalles n'utilisant qu'un seul octet, TURBO PASCAL tronque la valeur entrée à 1 octet. Cette valeur sur un octet peut très bien ne pas être dans l'intervalle. Elle est essentiellement destinée à éviter l'écrasement de zones situées après l'intervalle

```

intervalle 1(, 2(, 3, q(uite ? 1
intervalle1 [7..74] ? $0345

$1519:$0260          $41 $41 $41 $41 $41 : AAAAA
$1519:$0268 $41 $41 $41 $41 $41 $41 $41 : AAAAAAA
$1519:$0270 $41 $41 $41 : AAA
$1519:$0270          $03 : 
$1519:$0270          $42 $42 $42 $42 : BBBB
$1519:$0278 $42 $42 $42 $42 $42 $42 $42 : BBBBBBBB
$1519:$0280 $42 $42 $42 $42 : BBBB

intervalle 1(, 2(, 3, q(uite ? 2
intervalle2 [510..514] ? 31

$1519:$0270          $42 $42 $42 $42 : BBBB
$1519:$0278 $42 $42 $42 $42 $42 $42 $42 : BBBBBBBB
$1519:$0280 $42 $42 $42 $42 : BBBB
$1519:$0280          $1F $00 : 
$1519:$0280          $43 $43 : .. CC
$1519:$0288 $43 $43 $43 $43 $43 $43 $43 : CCCCCCCC
$1519:$0290 $43 $43 $43 $43 $43 $43 : CCCCCC

intervalle 1(, 2(, 3, q(uite ? q

```

Fig. 4.14 Un intervalle en mémoire

Par contre toute expressions arithmétique provoquera une erreur de débordement d'intervalle:

```

VAR octet: BYTE;
intervalle: 3..9;

```

```

READLN(octet);
intervalle:= octet;

```

déclenchera une erreur si vous tapez, par exemple, 10.

4.1.5.5 RESUME

Le type intervalle numérique:

- peut prendre des valeurs dans la plage -32768..32767
- utilise une représentation sur 16 bits, par complément à 2
- occupe normalement en mémoire:
 - 1 octet si l'intervalle est compris dans 0..255
 - 2 octets si l'intervalle débord 0..255
- READ ne provoque pas d'erreur en cas de saisie en dehors de l'intervalle. Pour les intervalles codés sur 1 octet, READ tronque à un octet toute valeur lue.

4.2 LE TYPE REEL

4.2.1 DEFINITION

Le type réel permet de représenter des valeurs en séparant la partie significative (appelée mantisse) de l'échelle (l'exposant).

Pour le réel:

9.1513 E 4

9.1513 est la mantisse et E4 l'exposant.

Voici quelques exemples de valeurs réelles:

3.51
-42.5E3
0.012E-26

Les valeurs d'un réel sont les valeurs numériques décimales comprises entre 1 E-38 et 1 E+38.

4.2.2 REPRESENTATION DES REELS

Pour représenter un réel, nous devons pouvoir représenter:

- le signe de la mantisse
- la valeur de la mantisse
- le signe de l'exposant
- la valeur de l'exposant

En TURBO PASCAL, chaque nombre réel utilise 6 octets, soit 48 bits répartis ainsi:

- 8 bits pour l'exposant et son signe
- un bit de signe de la mantisse
- une mantisse de 31 bits

Nous allons illustrer cela par un exemple.

Considérons la valeur 48.9375

- la valeur 48 est égale à $2^4 + 2^4$ et sera représentée en binaire par:

0011 0000

- 0.9375 est égal à $2^{-1} + 2^{-2} + 2^{-3} + 2^{-4}$ et sera représenté en binaire par les puissances négatives de 2 (1/2, 1/4 etc.) comme:

1111000000000000 etc

soit: $1/2 + 1/4 + 1/8 + 1/16 + 0/32 + 0/64$ etc.

- 48.9375 est donc représentable par:

0011 0000 . 1111 0000 0000 etc.

- pour ne pas gâcher de la place correspondant aux zéros de tête, nous décalons le point décimal de 6 places vers la gauche:

0.1100 0011 1100 0000 etc (6 décalages)

Ce procédé s'appelle la normalisation

- à la suite de la normalisation, toutes les valeurs comporteront un 1 comme premier chiffre décimal binaire. Il est donc inutile de le stocker dans la représentation du réel.

Par contre cette place est récupérée pour placer le signe du réel.

- le nombre de décalages est placé dans l'exposant. Un décalage vers la gauche signifie qu'il faudra multiplier la valeur de la mantisse par 2^1 , deux décalages signifient qu'il faudra multiplier la valeur de la mantisse par 2^2 etc.

Pour pouvoir définir des valeurs plus grandes ou plus petites que 1 (décalages du point vers la gauche ou la droite), nous centrons l'exposant autour de la valeur 128 (\$80):

- pas de décalage est représenté par un exposant égal à 128,
- un décalage est représenté par un exposant égal à 129
- 6 décalages vers la gauche comme dans notre exemple seront représentés par 134 (\$86).

De plus, la valeur 0 de l'exposant est utilisée pour indiquer que le réel a pour valeur 0.

L'exposant peut donc prendre des valeurs entre 0 et 255. Il a donc besoin de 8 bits.

Notre nombre prend alors l'allure suivante:

Bit numéro:	4444 4444 3 333 3333 3322 2222 2222 1111 ...
	7654 3210 9 876 5432 1098 7654 3210 9876 ...
Valeur du bit:	1000 0110 0 100 0011 1100 0000 0000 0000 ...
	EXPOSANT S -----MANTISSE-----

En résumé, l'utilisation des 48 bits est donc la suivante:

- bits 40 à 47: 8 bits pour l'exposant comme complément à 128. 0 correspond à un réel nul. La plage d'exposants est -128..127
- bit 39: signe du réel (0 positif, 1 négatif)
- bits 0 à 38: 39 bits pour les chiffres significatifs

4.2.3 IMPLANTATION MEMOIRE

En mémoire, les octets seront placés dans l'ordre suivant:

- l'octet de l'exposant
- les 5 octets de la mantisse en commençant par l'octet de poids le plus faible

Nous aurons donc:

adresse	1000 0110	\$86	exposant
adresse+1	0000 0000	\$00	octet 0 de la mantisse
adresse+2	0000 0000	\$00	octet 1 de la mantisse
adresse+3	0000 0000	\$00	octet 2 de la mantisse
adresse+4	1100 0000	\$C0	octet 3 de la mantisse
adresse+5	0100 0011	\$43	signe et octet 4 de la mantisse

4.2.4 EXAMEN DES REELS

Voici le programme nous permettant d'examiner un réel en mémoire:

```

0 (* 001 affree *)
1 (* 08 avr 87 *)
2
3 (* $r+*)
4 PROGRAM examen_reel;
5 TYPE t_array 16 caracteres= ARRAY[0..15] OF CHAR;
6 VAR choix: CHAR;
7     avant: t_array 16 caracteres;
8     reel: REAL;
9     apres: t_array 16 caracteres;
10
11 PROCEDURE affiche_memoire(p_segment, p_decalage_debut,
12 ...
13 END; (* affiche_memoire *)
14
15 PROCEDURE affiche_reel;
16 BEGIN
17     WRITE('reel <48.375> ? ');
18     READLN(reel);
19     affiche_memoire(SEG(reel), OFS(reel)- 16, OFS(reel)+ SIZEOF(reel)+ 16- 1);
20 END; (* affiche_reel *)
21
22 PROCEDURE initialise;
23 VAR l_puissance: INTEGER;
24 BEGIN
25     avant:= 'AAAAAAAAAAAAAAAA';
26     apres:= 'BBBBBBBBBBBBBBBB';
27 END; (* initialise *)

```

Fig. 4.15 Le programme *examen_reel*.

et voici un exemple d'exécution:


```
r(eel, q(uitte ? r
reel <48.375> ? 48.9375
```

```
$1528:$0260          $41 $41 $41 $41 $41 : AAAAA
$1528:$0268 $41 $41 $41 $41 $41 $41 $41 : AAAAAAAA
$1528:$0270 $41 $41 $41 : AAA
$1528:$0270          $86 $00 $00 $00 $C0 : .....
$1528:$0278 $43 : C
$1528:$0278          $42 $42 $42 $42 $42 $42 $42 : BBBBBBBB
$1528:$0280 $42 $42 $42 $42 $42 $42 $42 $42 : BBBBBBBB
$1528:$0288 $42 : B
```

```
r(eel, q(uitte ? q
```

Fig. 4.16 Exemple de réel en mémoire

4.2.5 DECODAGE D'UN REEL

Voici à présent un programme permettant:

- d'afficher les bits de l'exposant, du signe et de la mantisse
- de recalculer la valeur décimale du réel

```
0 (* 001 interpre *)
1 (* 08 avr 87 *)
2
3 (* $r+*)
4 PROGRAM interprete_reel;
5   VAR choix: CHAR;
6       reel: REAL;
7       table_puissances_de_deux: ARRAY[-50..50] OF REAL;
8
9   PROCEDURE decode_reel;
10    VAR l_exposant: INTEGER;
11        l_numero_octet: 0..4;
12        l_numero_bit: 0..7;
13        l_signe, l_bit: 0..1;
14        l_octet: byte;
15        l_valeur_exposant: REAL;
16
17    PROCEDURE affiche_octet_hexadecimal(p_octet: byte);
18    CONST ks_hex: ARRAY[0..15] OF CHAR = '0123456789ABCDEF';
19    BEGIN
20      WRITE(ks_hex[(p_octet AND $F0) SHR 4], ks_hex[(p_octet AND $0F)]);
21    END; (* affiche_octet_hexadecimal *)
22
23    BEGIN (* decode_reel *)
24      WRITE(' octet exposant: ');
25      affiche_octet_hexadecimal(mem[SEG(reel): OFS(reel)]);
26      WRITELN;
```

```
27 WRITE(' octets de la mantisse: ');
28 FOR l_numero_octet:= 5 DOWNT0 1 DO
29 BEGIN
30   affiche_octet_hexadecimal(mem[SEG(reel): OFS(reel)+l_numero_octet]);
31   WRITE(' ');
32 END;
33 WRITELN; WRITELN;
34
35 l_exposant:= mem[SEG(reel): OFS(reel)];
36 IF l_exposant= 0
37 THEN WRITELN(' reel égal a zéro')
38 ELSE BEGIN
39   l_valeur_exposant:= 1* table_puissances_de_deux[l_exposant- $80];
40   WRITELN(' exposant: ', l_valeur_exposant);
41
42   l_signe:= (mem[SEG(reel): OFS(reel)+ 5] AND $80) SHR 7;
43   WRITELN(' bit de signe: ', l_signe);
44
45   WRITE(' octets de la mantisse: ');
46   l_numero_bit:= 6;
47   FOR l_numero_octet:= 5 DOWNT0 1 DO
48   BEGIN
49     l_octet:= mem[SEG(reel): OFS(reel)+ l_numero_octet];
50
51     (* rajoute la valeur de l normalisee *)
52     IF l_numero_octet= 4
53     THEN l_octet:= l_octet OR $80;
54
55     FOR l_numero_bit:= 7 DOWNT0 0 DO
56     BEGIN
57       l_bit:= (l_octet AND $80) SHR 7;
58       WRITE(l_bit);
59       l_octet:= (l_octet AND $7F) SHL 1;
60     END;
61     WRITE(' ');
62   END; (* l_numero_octet *)
63   WRITELN;
64 END; (* exposant <> 0 *)
65 END; (* decode_reel *)
66
67 PROCEDURE recalcule_reel;
68 VAR l_exposant: byte;
69     l_valeur_exposant: REAL;
70     l_signe: 0..1;
71     l_numero_octet: 0..4;
72     l_numero_bit: 0..7;
73     l_bit: 0..1;
74     l_octet: byte;
75     l_puissance: INTEGER;
76     reel_recalcule: REAL;
77 BEGIN
78   l_exposant:= mem[SEG(reel): OFS(reel)];
79   IF l_exposant= 0
80   THEN WRITELN(' reel égal a zéro')
81   ELSE BEGIN
82     l_valeur_exposant:= 1* table_puissances_de_deux[l_exposant- $80];
83
84     l_signe:= (mem[SEG(reel): OFS(reel)+ 5] AND $80) SHR 7;
```



```

85
86      (* recompose le reel *)
87      WRITELN(' puissance bit valeur_cumulee');
88      l_reel_recalcule:= 0.0;
89      l_puissance:= -1;
90      FOR l_numero_octet:= 5 DOWNT0 1 DO
91      BEGIN
92          l_octet:= mem[SEG(reel): OFS(reel)+ l_numero_octet];
93
94          (* rajoute le bit du l normalisé *)
95          IF l_numero_octet= 5
96          THEN l_octet:= l_octet OR $80;
97
98          FOR l_numero_bit:= 7 DOWNT0 0 DO
99          BEGIN
100              l_bit:= (l_octet AND $80) SHR 7;
101              l_reel_recalcule:= l_reel_recalcule+ l_bit*
102              table_puissances_de_deux[l_puissance];
103              WRITELN(l_puissance: 7, l_bit: 7, l_reel_recalcule: 18);
104
105              l_octet:= (l_octet AND $7F) SHL 1;
106              l_puissance:= l_puissance- 1;
107          END; (* l_numero_bit *)
108          WRITELN;
109          END; (* l_numero_octet *)
110          l_reel_recalcule:= l_reel_recalcule* l_valeur_exposant;
111          WRITELN(l_reel_recalcule);
112          END; (* exposant <= 0 *)
113      END; (* recalcule_reel *)
114
115  PROCEDURE initialise;
116  VAR l_puissance: INTEGER;
117  BEGIN
118      table_puissances_de_deux[0]:= 1.0;
119      FOR l_puissance:= 1 TO 50 DO
120      BEGIN
121          table_puissances_de_deux[l_puissance]:=
122          table_puissances_de_deux[l_puissance- 1]/ 2.0;
123          table_puissances_de_deux[l_puissance]:=
124          table_puissances_de_deux[l_puissance- 1]* 2.0;
125      END;
126      (*
127      WRITE(l_puissance: 3, ' ', table_puissances_de_deux[l_puissance]: 16);
128      WRITELN(table_puissances_de_deux[l_puissance]: 16);
129      *)
130      END;
131      reel:= 48.9375;
132      END; (* initialise *)
133
134  BEGIN (* principal *)
135      initialise;
136      REPEAT
137          WRITELN;
138          WRITE('e(ntre reel, d(ecode, r(ecalcule, q(uite ? ');
139          READ(KBD, choix); WRITELN(choix);
140          CASE choix OF
141              'd' : decode_reel;

```

```

140      'e' : BEGIN
141          WRITE('reel <38.9375 ? > ? ');
142          READLN(reel);
143          END;
144      'r' : recalcule_reel;
145      END;
146      UNTIL choix= 'q';
147      END. (* principal *)

```

Fig. 4.17 Le programme interprete_reel.

et voici ce programme en action:

```

e(ntre reel, d(ecode, r(ecalcule, q(uite ? e
reel <48.9375> ? 48.9375

e(ntre reel, d(ecode, r(ecalcule, q(uite ? d
octet exposant: 86
octets de la mantisse: 43 C0 00 00 00

exposant: 6.4000000000E+01
bit de signe: 0
octets de la mantisse: 01000011 11000000 00000000 00000000

e(ntre reel, d(ecode, r(ecalcule, q(uite ? r
puissance bit valeur_cumulee

-1      1      5.0000000000E-01
-2      1      7.5000000000E-01
-3      0      7.5000000000E-01
-4      0      7.5000000000E-01
-5      0      7.5000000000E-01
-6      0      7.5000000000E-01
-7      1      7.5781250000E-01
-8      1      7.6171875000E-01

-9      1      7.6367187500E-01
-10     1      7.6464843750E-01
-11     0      7.6464843750E-01
-12     0      7.6464843750E-01
...
-40     0      7.6464843750E-01

4.8937500000E+01

e(ntre reel, d(ecode, r(ecalcule, q(uite ? q

```

Fig. 4.18 Exemple de réel en mémoire

Notez que:

- pour comprendre la représentation des réels nous sommes obligés de travailler avec les puissances positives et négatives de 2
- pour effectuer des saisies contrôlées ou des affichages particuliers de réels, nous devons travailler avec des puissances de 10 (au clavier nous tapons des valeurs décimales, et souhaitons voir des résultats décimaux). De telles procédures ont été présentées en UCSD dans TOPIQUES PASCAL.

4.2.6 RESUME

Le type réel

- a pour valeur les valeurs décimales comprises entre -1 E 38 et 1 E 38
- est représenté sur 48 bits (6 octets), en utilisant:
 - bits 40 à 47: 8 bits pour l'exposant comme complément à 128. 0 correspond à un réel nul. La plage d'exposants est -128..127
 - bit 39: signe du réel (0 positif, 1 négatif)
 - bits 0 à 38: 39 bits pour les chiffres significatifs
- En mémoire, un réel occupe 6 octets, (exposant, mantisse0, mantisse1, mantisse2, mantisse3, mantisse4, mantisse5 avec signe)

4.3 LE TYPE BCD

4.3.1 DEFINITION

Le type réel BCD (Binary Coded Decimal) permet de représenter des valeurs en séparant la partie significative (appelée mantisse) de l'échelle (l'exposant).

Pour le réel BCD:

9.1513 E 4

9.1513 est la mantisse et E4 l'exposant.

Voici quelques exemples de valeurs réelles BCD:

3.51
-42.5E3
0.012E-26

Les valeurs d'un réel BCD sont les valeurs numériques décimales comprises entre 1 E-63 à 1 E+63, avec 18 chiffres significatifs (alors que les réels usuels offrent une plage de 1 E-38 et 1 E+38 avec 11 chiffres significatifs).

4.3.2 UTILITE DES REELS BCD

Ce type de réel BCD a été créé spécialement pour les applications de gestion. Il offre une plage de puissances plus étendue et une précision plus grande.

ATTENTION: ce type de réel BCD nécessite l'utilisation du compilateur TURBOBCD.COM. Le format de ces réels BCD, que nous allons présenter, n'est pas compatible avec les réels du compilateur normal. Il faut passer par des programmes de transcodages pour échanger des réels entre fichiers.

4.3.3 REPRESENTATION DES REELS BCD

Pour représenter un réel BCD, nous devons pouvoir représenter:

- le signe de la mantisse
- la valeur de la mantisse
- le signe de l'exposant
- la valeur de l'exposant

De plus, le codage BCD utilise fondamentalement les puissances de 10 (alors que le codage des réels usuels utilise les puissances de 2).

En TURBO PASCAL, chaque nombre réel BCD utilise 10 octets, soit 80 bits répartis ainsi:

- un bit de signe de la mantisse
- 7 bits pour l'exposant et son signe
- une mantisse de 72 bits

Nous allons illustrer cela par un exemple.

Considérons la valeur 48.9375

- nous commençons par normaliser le nombre en multipliant ou divisant par les puissances de 10 de façon que le premier chiffre soit compris entre 0 et 1. Dans notre cas, 48.9375 est égal à:

0.489375 10²

- le bit de signe est 0
- la valeur de l'exposant est égale à 63 (base)+ 2, soit \$41
- la mantisse est codée par groupe de 2 chiffres, un chiffre par quartet. Donc dans notre cas:

\$48 \$93 \$75 \$00 \$00 \$00 \$00 \$00 \$00

En résumé, l'utilisation des 80 bits est donc la suivante:

- bit 79: signe du réel BCD (0 positif, 1 négatif)
- bits 78 à 72: 7 bits pour l'exposant comme complément à 63. 0 correspond à un réel nul. Les puissances sont des puissances de 10
- bits 72 à 0: 72 bits pour les chiffres significatifs

4.3.4 IMPLANTATION MEMOIRE

En mémoire, les octets seront placés dans l'ordre suivant:

- l'octet du signe et de l'exposant
- les 9 octets de la mantisse en commençant par l'octet de poids le plus faible (rangement dans l'ordre inverse)

Nous aurons donc:

adresse	0100 0001	\$41	signe et exposant
adresse+1	0000 0000	\$00	octet 0 de la mantisse
adresse+2	0000 0000	\$00	octet 1 de la mantisse
adresse+3	0000 0000	\$00	octet 2 de la mantisse
adresse+4	0000 0000	\$00	octet 3 de la mantisse
adresse+5	0000 0011	\$00	octet 4 de la mantisse
adresse+6	0000 0011	\$00	octet 5 de la mantisse
adresse+7	0000 0011	\$00	octet 6 de la mantisse
adresse+8	0111 0101	\$75	octet 7 de la mantisse
adresse+9	1001 0011	\$93	octet 8 de la mantisse
adresse+10	0100 1000	\$48	octet 9 de la mantisse

4.3.5 EXAMEN DES REELS BCD

Voici le programme nous permettant d'examiner un réel BCD en mémoire:

```

0 (* 001 affbcd *)
1 (* 15 aou 87 *)
2
3 (* $r+*)
4 PROGRAM affiche_reel_bcd;
5 TYPE t_array_16_caracteres= ARRAY[0..15] OF CHAR;
6 VAR choix: CHAR;
7     avant: t_array_16_caracteres;
8     reel: REAL;
9     apres: t_array_16_caracteres;
10
11 PROCEDURE affiche_memoire(p_segment, p_decalage_debut,
...
71 END; (* affiche_memoire *)
72
```

```

73 PROCEDURE affiche_reel;
74 BEGIN
75     WRITE('reel <48.375> ? ');
76     READLN(reel);
77     affiche_memoire(SEG(reel), OFS(reel)-16, OFS(reel)+SIZEOF(reel)+16-1);
78 END; (* affiche_reel *)
79
80 PROCEDURE initialise;
81 VAR l_puissance: INTEGER;
82 BEGIN
83     avant:= 'AAAAAAAAAAAAAAAA';
84     apres:= 'BBBBBBBBBBBBBBBB';
85 END; (* initialise *)
86
87 BEGIN (* principal *)
88     initialise;
89     REPEAT
90         WRITELN;
91         WRITE('r(eel, q(uite ? ');
92         READ(KBD, choix); WRITELN(choix);
93         CASE choix OF
94             'r': affiche_reel;
95             'q':
96                 UNTIL choix='q';
97         END; (* principal *)

```

Fig. 4.19 Le programme examen_reel.

et voici un exemple d'exécution:

```

r(eel, q(uite ? r
reel <48.375> ? 48.3750

$1511:$0290                                $41 : A
$1511:$0298 $41 $41 $41 $41 $41 $41 $41 : AAAAAAAAAA
$1511:$02A0 $41 $41 $41 $41 $41 $41 $41 : AAAAAAAAAA
$1511:$02A8 $00 $00 $00 $00 $00 $00 $50 $37 : .....P7
$1511:$02B0 $48                                :H
$1511:$02B8 $42 $42 $42 $42 $42 $42 $42 : BBBBBBBB
$1511:$02C0 $42                                :B

```

Fig. 4.20 Exemple de réel BCD en mémoire

4.3.6 DECODAGE D'UN REEL BCD

Voici à présent un programme permettant:

- d'afficher les bits de l'exposant, du signe et de la mantisse
- de recalculer la valeur décimale du réel BCD

```

0  (* 001 interpre *)
1  (* 08 avr 87 *)
2
3  (* $r+*)
4  PROGRAM interprete_reel;
5  VAR choix: CHAR;
6  reel: REAL;
7  table_puissances_de_deux: ARRAY[-50..50] OF REAL;
8
9  PROCEDURE decode_reel;
10 VAR l_exposant: INTEGER;
11     l_numero_octet: 0..9;
12     l_signe, l_bit: 0..1;
13     l_octet: BYTE;
14
15 PROCEDURE affiche_octet_hexadecimal(p_octet: BYTE);
16 CONST ks_hex: ARRAY[0..15] OF CHAR = '0123456789ABCDEF';
17 BEGIN
18   WRITE(ks_hex[(p_octet AND $F0) SHR 4], ks_hex[p_octet AND $0F]);
19 END; (* affiche_octet_hexadecimal *)
20
21 BEGIN (* decode_reel *)
22   WRITE(' octet exposant: ');
23   affiche_octet_hexadecimal(MEM[SEG(reel): OFS(reel)]);
24   WRITELN;
25   WRITE(' octets de la mantisse: ');
26   FOR l_numero_octet := 9 DOWNTO 1 DO
27     BEGIN
28       affiche_octet_hexadecimal(MEM[SEG(reel): OFS(reel) + l_numero_octet]);
29       WRITE(' ');
30     END;
31   WRITELN; WRITELN;
32
33   l_exposant := MEM[SEG(reel): OFS(reel)];
34   IF l_exposant = 0
35     THEN WRITELN(' reel BCD égal a zéro')
36     ELSE BEGIN
37       l_exposant := (l_exposant AND $7F) - $3F;
38       WRITELN(' exposant: 10^', l_exposant);
39
40       l_signe := (MEM[SEG(reel): OFS(reel)] AND $80) SHR 7;
41       WRITELN(' bit de signe: ', l_signe);
42
43       WRITE(' chiffres de la mantisse: ');
44       FOR l_numero_octet := 9 DOWNTO 1 DO
45         BEGIN
46           l_octet := MEM[SEG(reel): OFS(reel) + l_numero_octet];

```

```

48   WRITE(CHR(l_octet SHR 4 + 48));
49   WRITE(CHR(l_octet AND $0F + 48));
50   END; (* l_numero_octet *)
51   WRITELN;
52   END; (* exposant <> 0 *)
53   END; (* decode_reel *)
54
55 PROCEDURE recalcule_reel;
56 VAR l_exposant: BYTE;
57     l_valeur_exposant: REAL;
58     l_signe: 0..1;
59     l_numero_octet: 0..9;
60     l_octet: BYTE;
61     l_puissance: INTEGER;
62     l_reel_recalcule: REAL;
63 BEGIN
64   l_exposant := MEM[SEG(reel): OFS(reel)] AND $7F;
65   IF l_exposant = 0
66     THEN WRITELN(' reel bcd égal a zéro')
67     ELSE BEGIN
68       l_valeur_exposant := 1 * table_puissances_de_deux[l_exposant - $3F];
69
70       l_signe := (MEM[SEG(reel): OFS(reel)] AND $80) SHR 7;
71
72       (* recompose le reel *)
73       WRITELN(' puissance bit valeur_cumulee');
74       l_reel_recalcule := 0.0;
75       l_puissance := -1;
76       FOR l_numero_octet := 9 DOWNTO 1 DO
77         BEGIN
78           l_octet := MEM[SEG(reel): OFS(reel) + l_numero_octet];
79
80           l_reel_recalcule := l_reel_recalcule +
81             (l_octet SHR 4) * table_puissances_de_deux[l_puissance];
82           WRITELN(l_puissance: 7, ' ', l_reel_recalcule: 18);
83           l_puissance := l_puissance - 1;
84
85           l_reel_recalcule := l_reel_recalcule +
86             (l_octet AND $0F) * table_puissances_de_deux[l_puissance];
87           WRITELN(l_puissance: 7, ' ', l_reel_recalcule: 18);
88           l_puissance := l_puissance - 1;
89
90         WRITELN;
91       END; (* l_numero_octet *)
92
93       l_reel_recalcule := l_reel_recalcule * l_valeur_exposant;
94       WRITELN(l_reel_recalcule);
95       END; (* exposant <> 0 *)
96   END; (* recalcule_reel *)
97
98 PROCEDURE initialise;
99 VAR l_puissance: INTEGER;
100 BEGIN
101   table_puissances_de_deux[0] := 1.0;

```



```

102  FOR l_puissance:= 1 TO 50 DO
103  BEGIN
104      table_puissances_de_deux[l_puissance]:=
105      table_puissances_de_deux[l_puissance+1]/ 10.0;
106      table_puissances_de_deux[l_puissance]:=
107      table_puissances_de_deux[l_puissance-1]* 10.0;
108      (*
109      WRITE(l_puissance: 3, ' ', table_puissances_de_deux[l_puissance]: 16);
110      WRITELN(table_puissances_de_deux[l_puissance]: 16);
111      *)
112  END;
113  reel:= 48.9375;
114  END; (* initialise *)
115
116  BEGIN (* principal *)
117      initialise;
118      REPEAT
119          WRITELN;
120          WRITE('e(ntre reel, d(ecode, r(ecalcule, q(uitte ? ');
121          READ(KBD, choix); WRITELN(choix);
122          CASE choix OF
123              'd' : decode reel;
124              'e' : BEGIN
125                  WRITE('reel <38.9375 ? > ');
126                  READLN(reel);
127              END;
128              'r' : recalcule reel;
129          END;
130          UNTIL choix= 'q';
131      END. (* principal *)

```

Fig. 4.21 Le programme *interprete_reel*.

et voici ce programme en action:

```

e(ntre reel, d(ecode, r(ecalcule, q(uitte ? e
reel <38.9375 ? > ?
e(ntre reel, d(ecode, r(ecalcule, q(uitte ? d
octet exposant: 41
octets de la mantisse: 38 93 75 00 00 00 00 00

exposant: 10^2
bit de signe: 0
chiffres de la mantisse: 389375000000000000

e(ntre reel, d(ecode, r(ecalcule, q(uitte ? r
puissance bit valeur_cumulee

```

-1	3.000000000000000E-01
-2	3.800000000000000E-01
-3	3.890000000000000E-01
-4	3.893000000000000E-01
-5	3.893700000000000E-01
-6	3.893750000000000E-01
-7	3.893750000000000E-01
-8	3.893750000000000E-01
...	
-18	3.893750000000000E-01
	3.8937500000000000000E+01

e(ntre reel, d(ecode, r(ecalcule, q(uitte ? q

Fig. 4.22 Exemple de réel BCD en mémoire

4.3.7 RESUME

Le type réel BCD

- a pour valeur les valeurs décimales comprises entre -1 E 63 et 1 E 63
- a une précision de 18 chiffres significatifs
- est représenté sur 80 bits (10 octets), en utilisant:
 - bit 79: signe du réel BCD (0 positif, 1 négatif)
 - bits 78 à 72: 7 bits pour l'exposant comme complément à 63. 0 correspond à un réel nul. Les puissances sont des puissances de 10
 - bits 72 à 0: 72 bits pour les chiffres significatifs
- En mémoire, un réel BCD occupe 10 octets, (signe et exposant, mantisse0, mantisse1, mantisse2, mantisse3, mantisse4, mantisse5, mantisse6, mantisse7, mantisse8 et mantisse9)

4.4 LE TYPE REEL 8087

4.4.1 DEFINITION

Le type réel 8087 permet de représenter des valeurs en séparant la partie significative (appelée mantisse) de l'échelle (l'exposant).

Pour le réel 8087:

9.1513 E 4

9.1513 est la mantisse et E4 l'exposant.

Voici quelques exemples de valeurs réelles 8087:

3.51
-42.5E3
0.012E-26

Les valeurs d'un réel 8087 sont les valeurs numériques décimales comprises entre 4.19 E-307 et 1.67 E+308 , avec une précision de 16 chiffres (contre une plage de 1 E-38 à 1 E+38 et une précision de 11 chiffre pour les réels usuels).

4.4.2 UTILITE DES REELS 8087

Les réels 8087 ont été créés pour pouvoir bénéficier de la vitesse de calcul et la meilleure précision offerte par le co-processeur 87.

ATTENTION:

- il faut utiliser le compilateur TURBO-8087 pour pouvoir utiliser ces réels
- ce compilateur se bloque si vous traitez des réels et que le co-processeur n'est pas présent (il utilise le co-processeur pour lire les constantes réelles)
- Les réels 8087 ont un autre format que les réels usuels ou avec les réels BCD. Pour échanger des données entre ces formats, il faut utiliser des programmes de transcodage

Le format des réels 8087 est celui des réels longs de la spécification Intel.

4.4.3 REPRESENTATION DES REELS 8087

Pour représenter un réel 8087, nous devons pouvoir représenter:

- le signe de la mantisse
- la valeur de la mantisse
- le signe de l'exposant
- la valeur de l'exposant

En TURBO PASCAL, chaque nombre réel 8087 utilise 8 octets, soit 64 bits répartis ainsi:

- un bit de signe de la mantisse
- 11 bits pour l'exposant et son signe
- une mantisse de 52 bits

Nous allons illustrer cela par un exemple.

Considérons la valeur 48.9375

- 48 est égal à $2^5 + 2^4$ et sera représenté en binaire par

0011 0000

- 0.9375 est égal à $2^{-1} + 2^{-2} + 2^{-3} + 2^{-4}$ et sera représenté en binaire par les fractions négatives de 2 ($1/2, 1/4$ etc.) comme:

1111000000000000 etc

soit: $1/2 + 1/4 + 1/8 + 1/16 + 0/32 + 0/64$ etc.

- 48.9375 est donc représentable par:

0011 0000 . 1111 0000 0000 etc.

- pour ne pas gâcher de la place correspondant aux zéros de tête, nous décalons le point décimal de 6 places vers la gauche:

0.1100 0011 1100 0000 etc (6 décalages)

Ce procédé s'appelle la normalisation

- à la suite de la normalisation, toutes les valeurs comporteront un 1 comme premier chiffre décimal binaire. Il est donc inutile de le stocker dans la représentation du réel.
- le nombre de décalages est placé dans l'exposant. Un décalage vers la gauche signifie qu'il faudra multiplier la valeur de la mantisse par 2^1 , deux décalages signifient qu'il faudra multiplier la valeur de la mantisse par 2^2 etc.

Pour pouvoir définir des valeurs plus grandes ou plus petites que 1 (décalages du point vers la gauche ou la droite), nous centrons l'exposant autour de la valeur 1023 (\$3FF):

- pas de décalage est représenté par un exposant égal à \$3FF,
- un décalage est représenté par un exposant égal à \$400
- 6 décalages vers la gauche comme dans notre exemple seront représentés par \$405.

De plus, la valeur 0 de l'exposant est utilisée pour indiquer que le réel a pour valeur 0.

L'exposant peut donc prendre des valeurs entre 0 et 1023. Il a donc besoin de 11 bits.

En résumé, l'utilisation des 64 bits est donc la suivante:

- bit 63: signe du réel (0 positif, 1 négatif)
- bits 52 à 62: 11 bits pour l'exposant comme complément à \$3FF. 0 correspond à un réel nul. La plage d'exposants est -1064..1063
- bits 0 à 51: 52 bits pour les chiffres significatifs

4.4.4 IMPLANTATION MEMOIRE

En mémoire, les octets seront placés dans l'ordre suivant:

- les 2 octets de l'exposant, avec dans le bit le plus significatif le signe du réel
- dans l'octet 7, les 4 premiers bits de la mantisse (quartet faible) et les 4 bits les plus faibles de l'exposant
- dans l'octet 7, les bits 4 à 10 de l'exposant (dans les bits 0 à 6 de l'octet) et le signe du réel (dans le bit 7 de l'octet)

Nous aurons donc:

adresse	0000 0000	\$00 mantisse
adresse+1	0000 0000	\$00 mantisse
adresse+2	0000 0000	\$00 mantisse
adresse+3	0000 0000	\$00 mantisse
adresse+4	0000 0000	\$00 mantisse
adresse+5	0011 0000	\$30 mantisse
adresse+6	0100 1000	\$48 exposant et mantisse
adresse+7	0100 0000	\$40 signe et exposant

4.4.5 EXAMEN DES REELS 8087

Voici le programme nous permettant d'examiner un réel 8087 en mémoire:

```

0 (* 001 affreel8 *)
1 (* 08 avr 87 *)
2
3 (*$r+*)
4 PROGRAM examen_reel_87;
5 TYPE t_array_16_caracteres= ARRAY[0..15] OF CHAR;
6 VAR choix: CHAR;
7     avant: t_array_16_caracteres;
8     reel: REAL;
9     apres: t_array_16_caracteres;
10
11 PROCEDURE affiche_memoire(p_segment, p_decalage_debut,
...
71 END; (* affiche_memoire *)
72
```

```

73 PROCEDURE affiche_reel;
74 BEGIN
75     WRITE('reel <48.375> ? ');
76     READLN(reel);
77     affiche_memoire(SEG(reel), OFS(reel)- 16, OFS(reel)+ SIZEOF(reel)+ 16- 1);
78 END; (* affiche_reel *)
79
80 PROCEDURE initialise;
81 VAR l_puissance: INTEGER;
82 BEGIN
83     avant:= 'AAAAAAAAAAAAAAAA';
84     apres:= 'BBBBBBBBBBBBBBBB';
85 END; (* initialise *)
...
```

Fig. 4.23 Le programme *examen_reel_8087*.

et voici un exemple d'exécution:

```

r(eel, q(uitte ? r
reel <48.375> ?
$2274:$0268          $41 $41 $41 $41 $41 : AAAAA
$2274:$0270 $41 $41 $41 $41 $41 $41 $41 : AAAAAAA
$2274:$0278 $41 $41 $41          : AAA
$2274:$0278          $00 $00 $00 $00 $00 : .....
$2274:$0280 $30 $48 $40          : 0H@
$2274:$0280          $42 $42 $42 $42 $42 : BBBB
$2274:$0288 $42 $42 $42 $42 $42 $42 : BBBBBBB
$2274:$0290 $42 $42 $42          : BBB

r(eel, q(uitte ? q
```

Fig. 4.24 Exemple de réel 8087 en mémoire

4.4.6 DECODAGE D'UN REEL 8087

Voici à présent un programme permettant:

- d'afficher les bits de l'exposant, du signe et de la mantisse
- de recalculer la valeur décimale du réel 8087

```

0 (* 001 intr87 *)
1 (* 08 avr 87 *)
2
3 (*$r+*)
4 PROGRAM interprete_reel_87;
5 VAR choix: CHAR;
6     reel: REAL;
7     table_puissances_de_deux: ARRAY[-50..50] OF REAL;
```



```

8
9  PROCEDURE decode_reel;
10  VAR l_exposant: INTEGER;
11      l_numero_octet: 0..4;
12      l_numero_bit: 0..7;
13      l_signe, l_bit: 0..1;
14      l_octet: BYTE;
15      l_valeur_exposant: REAL;
16
17  PROCEDURE affiche_octet_hexadecimal(p_octet: BYTE);
18  CONST ks_hex: ARRAY[0..15] OF CHAR= '0123456789ABCDEF';
19  BEGIN
20      WRITE(ks_hex[(p_octet AND $F0) SHR 4], ks_hex[(p_octet AND $0F)]);
21  END; (* affiche_octet_hexadecimal *)
22
23  BEGIN (* decode_reel *)
24      WRITE(' octets du reel: ');
25      FOR l_numero_octet:= 7 DOWNT0 0 DO
26          BEGIN
27              affiche_octet_hexadecimal(mem[SEG(reel):OFS(reel)+l_numero_octet]);
28              WRITE(' ');
29          END;
30      WRITELN; WRITELN;
31
32      l_exposant:= (memw[SEG(reel): OFS(reel)+ 6] AND $7FFF) SHR 4;
33      IF l_exposant= 0
34      THEN WRITELN(' reel egal a zero')
35      ELSE BEGIN
36          l_valeur_exposant:= 1*table_puissances_de_deux[l_exposant-$400];
37          WRITELN(' exposant: ', l_valeur_exposant);
38
39          l_signe:= (mem[SEG(reel): OFS(reel)+ 7] AND $80) SHR 7;
40          WRITELN(' bit de signe: ', l_signe);
41
42          WRITE(' bits de la mantisse: 1 ');
43          l_octet:= mem[SEG(reel): OFS(reel)+ 6] AND $0F;
44
45          FOR l_numero_bit:= 3 DOWNT0 0 DO
46              BEGIN
47                  l_bit:= (l_octet AND $08) SHR 3;
48                  WRITE(l_bit);
49                  l_octet:= (l_octet AND $07) SHL 1;
50              END;
51
52          FOR l_numero_octet:= 5 DOWNT0 0 DO
53              BEGIN
54                  l_octet:= mem[SEG(reel): OFS(reel)+ l_numero_octet];
55
56                  FOR l_numero_bit:= 7 DOWNT0 0 DO
57                      BEGIN
58                          l_bit:= (l_octet AND $80) SHR 7;
59                          WRITE(l_bit);
60                          l_octet:= (l_octet AND $7F) SHL 1;
61                      END;
62                  WRITE(' ');
63              END; (* l_numero_octet *)
64          WRITELN;
65          END; (* exposant <> 0 *)
66      END; (* decode_reel *)

```

```

67
68  PROCEDURE recalcule_reel;
69  VAR l_exposant: BYTE;
70      l_valeur_exposant: REAL;
71      l_signe: 0..1;
72      l_numero_octet: 0..4;
73      l_numero_bit: 0..7;
74      l_bit: 0..1;
75      l_octet: BYTE;
76      l_puissance: INTEGER;
77      l_reel_recalcule: REAL;
78  BEGIN
79      l_exposant:= (memw[SEG(reel): OFS(reel)+ 6] AND $7FFF) SHR 4;
80      IF l_exposant= 0
81      THEN WRITELN(' reel egal a zero')
82      ELSE BEGIN
83          l_valeur_exposant:= 1*table_puissances_de_deux[l_exposant-$400];
84
85          l_signe:= (mem[SEG(reel): OFS(reel)+ 7] AND $80) SHR 7;
86
87          (* recompose le reel *)
88          WRITELN(' puissance bit valeur_cumulee');
89          l_reel_recalcule:= 0.0;
90          l_puissance:= -1;
91          FOR l_numero_octet:= 6 DOWNT0 0 DO
92              BEGIN
93                  l_octet:= mem[SEG(reel): OFS(reel)+ l_numero_octet];
94
95                  (* rajoute le bit du 1 normalisé *)
96                  IF l_numero_octet= 6
97                  THEN BEGIN
98                      l_octet:= (l_octet AND $0F) OR $10;
99                      l_numero_bit:= 4;
100                  END
101                  ELSE l_numero_bit:= 7;
102
103              REPEAT
104                  l_bit:= (l_octet AND $80) SHR 7;
105                  l_reel_recalcule:= l_reel_recalcule+ l_bit*
106                      table_puissances_de_deux[l_puissance];
107                  WRITELN(l_puissance: 7, l_bit: 7, l_reel_recalcule: 18);
108
109                  l_octet:= (l_octet AND $7F) SHL 1;
110                  l_puissance:= l_puissance-1;
111                  l_numero_bit:= l_numero_bit-1;
112              UNTIL l_numero_bit< 0;
113              WRITELN;
114          END; (* l_numero_octet *)
115
116          l_reel_recalcule:= l_reel_recalcule* l_valeur_exposant;
117          WRITELN(l_reel_recalcule);
118          END; (* exposant <> 0 *)
119      END; (* recalcule_reel *)
120
121  PROCEDURE initialise;
122  VAR l_puissance: INTEGER;
123  BEGIN
124      table_puissances_de_deux[0]:= 1.0;

```



```

124 FOR l_puissance:= 1 TO 50 DO
125 BEGIN
126   table_puissances_de_deux[-l_puissance]:=
127     table_puissances_de_deux[-l_puissance+ 1]/ 2.0;
128   table_puissances_de_deux[l_puissance]:=
129     table_puissances_de_deux[l_puissance- 1]* 2.0;
130 END;
131 reel:= 48.9375;
132 END; (* initialise *)
133
134 BEGIN (* principal *)
135   initialise;
136   REPEAT
137     WRITELN;
138     WRITE('entre reel, d(encode, r(ecalcule, q(uite ?)');
139     READ(KBD, choix); WRITELN(choix);
140     CASE choix OF
141       'd' : decode reel;
142       'e' : BEGIN
143         WRITE('reel <38.9375 ? > ? ');
144         READLN(reel);
145         END;
146       'r' : recalcule reel;
147       END;
148     UNTIL choix= 'q';
149   END. (* principal *)

```

Fig. 4.25 Le programme `interprete_reel`.

Notez que:

- pour comprendre la représentation des réels 8087 nous sommes obligés de travailler avec les puissances positives et négatives de 2
- pour effectuer des saisies contrôlées ou des affichages particuliers de réels, nous devons travailler avec des puissances de 10 (au clavier nous tapons des valeurs décimales, et souhaitons voir des résultats décimaux).

4.4.7 RESUME

Le type réel 8087

- a pour valeur les valeurs décimales comprises entre $4.19 \text{ E-}307$ et $1.67 \text{ E+}308$, avec une précision de 16 chiffres
- est obtenu en utilisant le compilateur TURBO-87, et nécessite la présence du co-processeur 87 (tant à la compilation qu'à l'exécution)
- utilise le format des réels longs de la spécification Intel.

- est représenté sur 64 bits (8 octets), en utilisant:
 - bit 63: signe du réel (0 positif, 1 négatif)
 - bits 52 à 62: 11 bits pour l'exposant comme complément à 3FFF . 0 correspond à un réel nul. La plage d'exposants est $-1064..1063$
 - bits 0 à 51: 52 bits pour les chiffres significatifs
- En mémoire, un réel 8087 occupe 8 octets, (6 octets pour la mantisse, un octet pour la mantisse et l'exposant, l'octet du signe et de l'exposant)

4.5 LE TYPE STRUCTURE TABLEAU

4.5.1 DEFINITION

Un tableau est une structure composée:

- d'un nombre fixe
- de cases d'un même type

L'accès à une case se fait au moyen de l'indice de cette case.

4.5.2 IMPLEMENTATION DES TABLEAUX

Les tableaux sont formés de la suite des octets correspondant aux cases du tableau.

Dans certains langages, il est mis en place en mémoire une structure facilitant l'accès aux cases d'un tableau, surtout s'il a plusieurs dimensions. Ce n'est pas le cas en TURBO PASCAL:

- aucune place n'est réservée en mémoire pour l'indice d'un tableau
- dans le programme, pour manipuler une case il suffit d'indiquer son indice, et le code généré par le compilateur se chargera de la récupérer
- si vous souhaitez utiliser une variable comme indice, celle-ci pourra être n'importe quelle variable d'un type compatible avec celui de l'indice. Cette variable devra avoir été déclarée normalement, et sera implantée normalement.

L'accès aux cases se fait donc uniquement par des calculs de déplacement par rapport à la première adresse.

Si nous numérotions les cases du tableau à partir de 0, nous avons:

adresse début case i := adresse début tableau + i * taille case

Si la borne inférieure de l'indice n'est pas 0, il suffit de calculer le déplacement:

adresse début case i := adresse début tableau + (i - premier_indice) * taille case

par exemple, si nous déclarons:

```
VAR tableau: ARRAY[5..9] OF ...
```

alors l'adresse en mémoire de la case 7 s'obtiendra par:

adresse début case i := adresse début tableau + (i - 5) * taille case

Pour les tableaux à deux dimensions, les cellules seront regroupées par lignes. Ainsi, si nous déclarons:

```
VAR tableau: ARRAY[0..1, 0..2] OF INTEGER;
```

```
...
tableau[0, 0] := 1;
tableau[0, 1] := 2;
tableau[0, 2] := 3;
tableau[1, 0] := 40;
tableau[1, 1] := 50;
tableau[1, 2] := 60;
```

aura l'allure suivante (les adresses augmentent de 2 en 2 car il faut 2 octets par case):

adresse	1
adresse+ 2	2
adresse+ 4	3
adresse+ 6	40
adresse+ 8	50
adresse+10	60

Si nous considérons un tableau dont les bornes inférieures des indices de ligne et de colonne sont 0, la formule fournissant l'adresse d'une case est donc:

adresse début case [Ligne, Colonne] := adresse début tableau
+ Ligne * nombre colonnes * taille case
+ Colonne * taille case

Cette formule se généralise aisément pour les tableaux à n dimensions.

4.5.3 IMPLANTATION DES TABLEAUX EN MEMOIRE

En mémoire, la place occupée est égale à la somme des octets occupés par chaque case.

Si une case n'occupe pas tous les bits d'un octet, il y aura des bits inutilisés.

La disposition des octets est pour chaque case celle du type correspondant. Ainsi pour un tableau d'entiers, l'octet de poids le plus faible sera en tête pour chaque case.

4.5.4 EXAMEN D'UN TABLEAU

Voici un programme qui nous permet d'examiner un tableau en mémoire:

```
0 (* 001 afftabl *)
1 (* 08 avr 87 *)
2
3 (*$r+*)
4 PROGRAM examen_tableau;
5 TYPE t_array_16_caracteres = ARRAY[0..15] OF CHAR;
6 VAR choix: CHAR;
7     avant: t_array_16_caracteres;
8     tableau: ARRAY[2..5] OF INTEGER;
9     apres: t_array_16_caracteres;
10
11 PROCEDURE affiche_memoire(p_segment, p_decalage_debut,
...
71 END; (* affiche_memoire *)
72
73 PROCEDURE affiche_tableau;
74 VAR l_indice: INTEGER;
75 BEGIN
76     FOR l_indice := 2 TO 5 DO
77     BEGIN
78         WRITE('tableau <$77> ? ');
79         READLN(tableau[l_indice]);
80     END;
81     affiche_memoire(SEG(tableau), OFS(tableau) - 16,
82                     OFS(tableau) + SIZEOF(tableau) + 16 - 1);
83     END; (* affiche_tableau *)
84
85 PROCEDURE initialise;
86 BEGIN
87     avant := 'AAAAAAAAAAAAAAAA';
88     apres := 'BBBBBBBBBBBBBBBB';
89     END; (* initialise *)
90
91 ...
```

Fig. 4.26 Le programme `examen_tableau`.

et voici un exemple d'exécution:


```
t(ableau, q(uitte ? t
tableau <$77> ? $77
tableau <$77> ? $78
tableau <$77> ? $AA
tableau <$77> ? $BB
```

```
$1532:$0260          $41 $41 $41 $41 $41 : AAAAA
$1532:$0268 $41 $41 $41 $41 $41 $41 $41 : AAAAAAA
$1532:$0270 $41 $41 $41 $41 $41 $41 $41 : AAA
$1532:$0270          $77 $00 $78 $00 $AA : w.x..
$1532:$0278 $00 $BB $00 $00 $00 $00 $00 : ...
$1532:$0278          $42 $42 $42 $42 $42 : BBBB
$1532:$0280 $42 $42 $42 $42 $42 $42 $42 : BBBB
$1532:$0288 $42 $42 $42 $42 $42 $42 $42 : BBB
```

```
t(ableau, q(uitte ? q
```

Fig. 4.27 Un tableau à une dimension en mémoire

Voici à présent un programme pour examiner un tableau à 2 dimensions:

```
0 (* 001 afftbl2 *)
1 (* 08 avr 87 *)
2
3 (* $r+*)
4 PROGRAM examen_tableau_2_dimensions;
5 TYPE t_array 16 caracteres= ARRAY[0..15] OF CHAR;
6 VAR choix: CHAR;
7     avant: t_array 16 caracteres;
8     tableau: ARRAY[1..2] OF ARRAY[7..9] OF byte;
9     apres: t_array 16 caracteres;
10
11 PROCEDURE affiche_memoire(p_segment, p_decalage_debut,
...
71 END; (* affiche_memoire *)
72
73 PROCEDURE affiche_tableau;
74 VAR l_ligne, l_colonne: INTEGER;
75 BEGIN
76     FOR l_ligne:= 1 TO 2 DO
77         FOR l_colonne:= 7 TO 9 DO
78             BEGIN
79                 WRITE(' tableau <$77> ? ');
80                 READLN(tableau[l_ligne, l_colonne]);
81             END;
82             affiche_memoire(SEG(tableau), OFS(tableau)- 16,
83                             OFS(tableau)+ SIZEOF(tableau)+ 16- 1);
84         END; (* affiche_tableau *)
```

```
85 PROCEDURE initialise;
86 BEGIN
87     avant:= 'AAAAAAAAAAAAAAAA';
88     apres:= 'BBBBBBBBBBBBBBBB';
89 END; (* initialise *)
```

Fig. 4.28 Le programme examen_tableau_deux_dimensions.

et voici un exemple d'exécution:

```
t(ableau, q(uitte ? t
tableau <$77> ? $17
tableau <$77> ? $18
tableau <$77> ? $19
tableau <$77> ? $27
tableau <$77> ? $28
tableau <$77> ? $29
```

```
$153B:$0260          $41 $41 $41 $41 $41 : AAAAA
$153B:$0268 $41 $41 $41 $41 $41 $41 $41 : AAAAAAA
$153B:$0270 $41 $41 $41 $41 $41 $41 $41 : AAA
$153B:$0270          $17 $18 $19 $27 $28 : ...' (
$153B:$0278 $29 $00 $00 $00 $00 $00 $00 : )
$153B:$0278          $42 $42 $42 $42 $42 $42 $42 : BBBB
$153B:$0280 $42 $42 $42 $42 $42 $42 $42 : BBBB
$153B:$0288 $42 $42 $42 $42 $42 $42 $42 : B
```

```
t(ableau, q(uitte ? q
```

Fig. 4.29 Un tableau à plusieurs dimensions en mémoire

4.5.5 LES TABLEAUX COMPACTES

TURBO PASCAL ne sait pas compacter les cases d'un tableau. Un tableau de 1000 booléens occupera 1000 octets (alors que 1000 DIV 8 octets auraient suffi).

Par contre TURBO gère naturellement les données par octet (et non pas par mot de 2 octets, avec obligation de débiter à des adresses paires, comme dans certaines versions de PASCAL). De ce fait, si la case d'un tableau occupe 8 bits ou moins, chaque case occupera un octet.

Le programme qui suit montre que:

- une case de type 2.5 (n'ayant besoin que de 3 bits) utilise un octet par case
- une case de type 0.300 (n'ayant besoin que de 9 bits) utilise 2 octets par cellule


```

0 (* 001 afftabc *)
1 (* 10 avr 87 *)
2
3 (*$r+*)
4 PROGRAM examen_tableau_compacte;
5   TYPE t_array 16 caracteres= ARRAY[0..15] OF CHAR;
6   VAR choix: CHAR;
7       separateur A: t_array 16 caracteres;
8       tableau_1: ARRAY[2..5] OF 0.5;
9       separateur B: t_array 16 caracteres;
10      tableau_2: ARRAY[2..5] OF 0.300;
11      separateur C: t_array 16 caracteres;
12
13 PROCEDURE affiche_memoire(p_segment, p_decalage_debut,
...
73   END; (* affiche_memoire *)
74
75 PROCEDURE affiche_tableau;
76   VAR l_indice: INTEGER;
77   BEGIN
78     FOR l_indice:= 2 TO 5 DO
79       BEGIN
80         WRITE(' tableau 1, indice ', l_indice, ' [0.5] ? ');
81         READLN(tableau_1[l_indice]);
82       END;
83       affiche_memoire(SEG(tableau_1), OFS(tableau_1)- 16,
84         OFS(tableau_1)+ SIZEOF(tableau_1)+ 16- 1);
85     FOR l_indice:= 2 TO 5 DO
86       BEGIN
87         WRITE(' tableau 2, indice ', l_indice, ' [0..300] ? ');
88         READLN(tableau_2[l_indice]);
89       END;
90       affiche_memoire(SEG(tableau_2), OFS(tableau_2)- 16,
91         OFS(tableau_2)+ SIZEOF(tableau_2)+ 16- 1);
92     END; (* affiche_tableau *)
93
94 PROCEDURE initialise;
95   BEGIN
96     separateur_A:= 'AAAAAAAAAAAAAAAA';
97     separateur_B:= 'BBBBBBBBBBBBBBBB';
98     separateur_C:= 'CCCCCCCCCCCCCCCC';
99   END; (* initialise *)

```

Fig. 4.30 Le programme examen_tableau_compacte.

et voici un exemple d'exécution:

```

t(tableau, q(uitte ? t
tableau 1, indice 2 [0..5] ? 1
tableau 1, indice 3 [0..5] ? 2
tableau 1, indice 4 [0..5] ? 3
tableau 1, indice 5 [0..5] ? 4

$155E:$0260          $41 $41 $41 $41 $41 : AAAAA
$155E:$0268 $41 $41 $41 $41 $41 $41 : AAAAAA
$155E:$0270 $41 $41 $41 : AAA
$155E:$0270          $01 $02 $03 $04 : ....
$155E:$0270          $42 : B
$155E:$0278 $42 $42 $42 $42 $42 $42 $42 : BBBBBBBB
$155E:$0280 $42 $42 $42 $42 $42 $42 $42 : BBBBBBBB

tableau 2, indice 2 [0..300] ? 1
tableau 2, indice 3 [0..300] ? 2
tableau 2, indice 4 [0..300] ? 3
tableau 2, indice 5 [0..300] ? 4

$155E:$0270          $42 : B
$155E:$0278 $42 $42 $42 $42 $42 $42 $42 : BBBBBBBB
$155E:$0280 $42 $42 $42 $42 $42 $42 $42 : BBBBBBBB
$155E:$0280          $01 : .
$155E:$0288 $00 $02 $00 $03 $00 $04 $00 : .....
$155E:$0288          $43 : C
$155E:$0290 $43 $43 $43 $43 $43 $43 $43 : CCCCCCCC
$155E:$0298 $43 $43 $43 $43 $43 $43 $43 : CCCCCC

t(tableau, q(uitte ? q

```

Fig. 4.31 Exemple de compactation en TURBO

Le compilateur ne rejette pas le mot **PACKED**, mais n'effectue aucun traitement.

Si vous souhaitez compacter des données, il faut effectuer le travail vous-mêmes, en utilisant les instructions au niveau du bit (**AND**, **OR**, **XOR**, **NOT**, **SHR**, **SHL**) ou le type ensemble.

4.5.6 LES TABLEAUX DE CARACTERES

Compte tenu de l'importance des structures représentant du texte lisible par l'homme, les tableaux de caractères à une dimension ont un rôle privilégié en PASCAL.

Ces tableaux se déclarent comme n'importe quel tableau:

```
VAR tableau_caractere: ARRAY[5..7] OF CHAR;
```


Mais de plus, nous pouvons:

- initialiser ces tableaux par une seule instruction:

```
tableau_caractere:= 'AHA';
```

- utiliser **WRITE** ou **WRITELN** pour afficher tous les caractères du tableau:

```
WRITELN(tableau_caractere);
```

En revanche, notez que:

- nous ne pouvons pas lire ces tableaux par **READLN**. Le compilateur rejette:

```
READLN(tableau_caractere);
```

En mémoire, ces tableaux sont stockés comme les tableaux usuels. Nous les avons d'ailleurs utilisés pour délimiter le début et la fin des zones mémoire dans les paragraphes précédents.

Le PASCAL STANDARD ne comportant pas de type **STRING**, toutes les manipulations de chaînes de caractères se font à l'aide de tableaux compactés de caractères.

4.5.7 LES CHAINES DE CARACTERES

Les chaînes de caractères sont constituées par des tableaux de caractères, avec un caractère spécial en tête pour indiquer la longueur de la chaîne.

Chaque chaîne utilise un nombre d'octets égal à la longueur maximale de la chaîne, plus 1 caractère pour placer la longueur actuelle de la chaîne.

Prenons un exemple:

```
VAR chaine: STRING[4];
```

```
...
chaine:= 'ABCD';
...
```

Cela correspondra à la disposition en mémoire suivante:

adresse	\$04
adresse+ 1	\$41
adresse+ 2	\$42
adresse+ 3	\$43
adresse+ 4	\$45

Une instruction d'affectation postérieure, par exemple *chaine:= 'OH'*, ne modifiera pas la place totale occupée par la chaîne, mais uniquement la longueur, et les 2 premiers caractères.

Notez que:

- en TURBO PASCAL il est impératif de préciser la longueur maximale de la chaîne. Il n'existe pas de longueur par défaut
- si une chaîne est utilisée comme paramètre, il est impératif d'avoir défini son type à l'aide de **TYPE**:

```
PROCEDURE affiche(p_message: STRING[80]);
```

n'est PAS accepté. Utilisez:

```
TYPE string80= STRING[80];
```

```
PROCEDURE affiche(p_message: string80);
```

- la longueur actuelle de la chaîne est placée dans l'octet d'indice 0.
- la longueur maximale d'une chaîne est de 255 (c'est la plus grande valeur qu'il est possible de stocker dans l'octet de longueur)
- pour connaître la longueur actuelle d'une chaîne à un instant donné, nous utilisons en général la fonction **LENGTH**:

```
FOR indice:= 1 to LENGTH(chaine) DO
```

```
...
```

Nous pouvons cependant utiliser directement l'octet 0, qui est cependant considéré comme de type caractère:

```
FOR indice:= 1 to ORD(chaine[0]) DO
```

```
...
```

- il est plus fréquent de placer dans l'octet 0 une valeur pour fixer une valeur actuelle. Ceci est surtout utilisé pour des initialisations rapides:

```
chaine[0]:= chr(80);
```

- la chaîne '' est la chaîne de longueur nulle. Elle est initialisée par:

```
chaine:= '';
```

ou par **READLN**(Chaine); et la frappe de **RETURN**.

- aucune erreur d'exécution n'est provoquée si nous manipulons des octets situés entre la longueur actuelle et la longueur maximale:

```
VAR prenom: STRING[5];
```

```
(*$r+*)
```

```
prenom:= 'JOE';
```

```
WRITE(prenom[5]);
```

est accepté, même lorsque nous demandons des vérifications d'intervalle par **R+**. Nous pouvons de même accéder à l'octet 0. Seul un indice inférieur à 0 ou supérieur à la taille maximale déclarée provoquera une erreur

- l'option V (Verify) permet d'utiliser comme paramètre variable une chaîne de taille différente à celle du paramètre formel:

```
(*$V.*)
TYPE string255= STRING[255];
string3= STRING[3];

VAR prenom: string3;
    globale: INTEGER;

PROCEDURE saisie(VAR pv_chaine: string255);
...
    saisie(prenom);
```

Normalement c'est au programmeur de s'assurer que la procédure ne détruira pas les variables situées après la chaîne (dans notre cas, si des traitements dans *saisie* modifient des caractères au-delà du troisième, la variable *globale* devrait être détruite). En fait, en TURBO PASCAL, l'option V- a pour seul but de sauter les contrôles d'identité de type des chaînes. Comme l'affectation tronque systématiquement le nombre de caractères affectés à une chaîne, nous ne risquons pas d'écraser des données.

- finalement, les chaînes de longueur 1 sont compatibles avec les variables de type caractère.

```
VAR caractere: CHAR;
    chaine: STRING[255];

chaine:= 'A';
caractere:= chaine;
```

et il en est de même pour les procédures ou fonctions de traitement de chaînes:

```
chaine:= CONCAT(chaine, caractere);
```

Voici un programme qui permet l'examen de la mémoire au voisinage d'une chaîne de 4 caractères:

```
0 (* 001 affstring *)
1 (* 08 avr 87 *)
2
3 (*$r+*)
4 PROGRAM examen_string;
5 TYPE t_array_16_caracteres= ARRAY[0..15] OF CHAR;
6 VAR choix: CHAR;
7     avant: t_array_16_caracteres;
8     chaine: STRING[4];
9     apres: t_array_16_caracteres;
10
11 PROCEDURE affiche_memoire(p_segment, p_decalage_debut,
...
71 END; (* affiche_memoire *)
72
```

```
73 PROCEDURE affiche_chaine;
74 BEGIN
75     WRITE('chaine <JOE> ? ');
76     READLN(chaine);
77     affiche_memoire(SEG(chaine), OFS(chaine)- 16, OFS(chaine)+
        SIZEOF(chaine)+ 16- 1);
78 END; (* affiche_chaine *)
79
80 PROCEDURE initialise;
81 BEGIN
82     avant:= 'AAAAAAAAAAAAAAAA';
83     apres:= 'BBBBBBBBBBBBBBBB';
84 END; (* initialise *)
...
```

Fig. 4.32 Le programme *examen_string*.

et voici un exemple d'exécution:

```
c(haine, q(uitte ? c
chaine <JOE> ? jo

$1528:$0260          $41 $41 $41 $41 $41 : AAAAA
$1528:$0268 $41 $41 $41 $41 $41 $41 $41 : AAAAAAAA
$1528:$0270 $41 $41 $41 $41 $41 $41 $41 : AAA
$1528:$0270          $02 $6A $6F $26 $08 : .jo&.

$1528:$0278 $42 $42 $42 $42 $42 $42 $42 : BBBBBBBB
$1528:$0280 $42 $42 $42 $42 $42 $42 $42 : BBBBBBBB

c(haine, q(uitte ? c
chaine <JOE> ?

$1528:$0260          $41 $41 $41 $41 $41 : AAAAA
$1528:$0268 $41 $41 $41 $41 $41 $41 $41 : AAAAAAAA
$1528:$0270 $41 $41 $41 $41 $41 $41 $41 : AAA
$1528:$0270          $00 $6A $6F $26 $08 : .jo&.
$1528:$0278 $42 $42 $42 $42 $42 $42 $42 : BBBBBBBB
$1528:$0280 $42 $42 $42 $42 $42 $42 $42 : BBBBBBBB

c(haine, q(uitte ? q
```

Fig. 4.33 Une chaîne en mémoire

Notez que:

- la frappe de RETURN a simplement placé la valeur 0 dans l'octet de longueur