

```

0 (* 001 fileof *)
1 (* 11 avr 87 *)
2
3 (*$r+*)
4 PROGRAM desassemble_file_of;
5   VAR fichier: FILE OF INTEGER;
6     fiche: INTEGER;
7     nom_fichier: STRING[$20];
8   BEGIN
9     ASSIGN(fichier, nom_fichier);
10    REWRITE(fichier);
11    READ(fichier, fiche);
12    WRITE(fichier, fiche);
13    CLOSE(fichier);
14  END.

```

Fig. 5.107 Le programme *desassemble_file_of*.

Voici le code généré:

```

2D4D E849DD      call    TU_PARAGRAPHS
2D50 0600        hex     0600 ; options
2D52 780B        hex     780B ; cs
2D54 3C15        hex     3C15 ; ds
2D56 DD02        hex     DD02 ; code
2D58 2E00        hex     2E00 ; globales
2D5A 0004        hex     0004 ; min piles
2D5C 00A0        hex     00A0 ; max piles
2D5E 1000        hex     1000 ; fichiers
2D60 0000        hex     0000 ; input
2D62 0000        hex     0000 ; output

2D64 8BEC        mov     bp, sp
2D66 E814DF      call    TU_TAILLE_CODE
2D69 5700        hex     5700 ; taille
2D6B 6D2D        adr     SAUT_PRINCIPAL

2D6D E90000      SAUT_PRINCIPAL jmp    PRINCIPAL

2D70 BF6202      PRINCIPAL  mov     di, $0262
2D73 1E          push    ds
2D74 57          push    di
2D75 BFB002      mov     di, $02B0
2D78 1E          push    ds
2D79 E806E4      call    TU_POUSSE_STRING
2D7C E80FFC      call    TU_ASSIGN_FILE

2D7F BF6202      mov     di, $0262
2D82 1E          push    ds
2D83 B90200      mov     cx, $0002
2D86 E80EFC      call    TU_REWRITE
2D89 E847E2      call    TU_IO_CHECK

```

```

2D8C BF6202      mov     di, $0262
2D8F 1E          push    ds
2D90 E857FC      call    TU_DEB_RW_FICH
2D93 BFAE02      mov     di, $02AE
2D96 1E          push    ds
2D97 E86DFC      call    TU_READ_FICHIE
2D9A E836E2      call    TU_IO_CHECK

2D9D BF6202      mov     di, $0262
2DA0 1E          push    ds
2DA1 E846FC      call    TU_DEB_RW_FICH
2DA4 BFAE02      mov     di, $02AE
2DA7 1E          push    ds
2DA8 E864FC      call    TU_WRITE_FICH
2DAB E825E2      call    TU_IO_CHECK

2DAE BF6202      mov     di, $0262
2DB1 1E          push    ds
2DB2 E823FC      call    TU_CLOSE
2DB5 E81BE2      call    TU_IO_CHECK

2DB8 E90000      jmp     FIN

2DBB 33C0        xor     ax, ax
2DBD E880DE      call    TU_HALT
2DC0 0000        hex     0000

```

Fig. 5.108 fichier de fiches

Notez que:

- toutes les primitives de traitement de fichier invoquent des routines de la librairie, en ayant auparavant poussé sur la pile le segment correspondant au fichier et placé dans DI son décalage
- si la procédure utilise d'autres paramètres (nom du fichier pour ASSIGN, taille de la fiche pour REWRITE, adresse de la fiche courante utilisée pour READ et WRITE...), ceux-ci sont soit placés dans des registres soit poussés sur la pile.

La gestion du fichier est en fait effectuée en utilisant en mémoire un bloc de contrôle (FCB ou FIB) qui contient, entre autres:

- le nom du fichier (avec le lecteur et la table des matières utilisée)
- la position de la prochaine lecture ou écriture
- la taille totale du fichier...

Le compilateur se contente donc d'indiquer aux routines de la librairie l'adresse en mémoire de ce FCB. Ces routines utiliseront les paramètres contenus dans ce bloc de contrôle pour invoquer les fonctions DOS adéquates.

5.5.3.2 LE BLOC DE CONTROLE D'UN FICHIER

Alors que la version 2 de TURBO gérait directement les fichiers (nombre d'octets, position du pointeur de la fiche courante etc.), la version 3 se fie entièrement à MS-DOS.

La zone réservée à la variable *fichier* contient donc un bloc de contrôle réduit de 76 octets qui contiennent:

- dans les deux premiers octets \$FFFF si le fichier est fermé ou le numéro de fichier s'il est ouvert
- dans les 2 octets suivants, la taille de la fiche

Le code généré fournit aux routines de la librairie:

- l'adresse du fcb TURBO (qui contient le numéro du fichier et la taille de la fiche)
- l'adresse de la fiche courante

Les routines de la librairie utilisent ces informations pour effectuer les appels à MS-DOS.

Voici un programme qui affiche le contenu du fcb actuel de TURBO:

```

0 (* 001 fcbfo *)
1 (* 28 avr 87 *)
2
3 (*$r+*)
4 PROGRAM teste_fcb_turbo_file_of;
5 TYPE t_array_16_caracteres= ARRAY[0..15] OF CHAR;
6 t_fiche= RECORD
7     nom: STRING[3];
8     age: INTEGER;
9 END;
10 VAR choix: CHAR;
11     avant: t_array_16_caracteres;
12     fichier: FILE OF t_fiche;
13     fiche_courante: t_fiche;
14     apres: t_array_16_caracteres;
15
16 PROCEDURE affiche_memoire(p_segment, p_decalage_debut,
...
76 END; (* affiche_memoire *)
77
78 PROCEDURE affiche_fcb_turbo_file_of;
79 VAR l_ligne: STRING[255];
80 BEGIN
81     WRITELN(' taille fcb: ', SIZEOF(fichier)); WRITELN;
82
83     affiche_memoire(SEG(fichier), OFS(fichier)- 16,
84                     OFS(fichier)+ SIZEOF(fichier)+ 16- 1);
85
86     ASSIGN(fichier, 'FILEOF.DTA');
87     REWRITE(fichier);
88
```

```

89 affiche_memoire(SEG(fichier), OFS(fichier)- 16,
90                 OFS(fichier)+ SIZEOF(fichier)+ 16- 1);
91
92 WITH fiche_courante DO
93 BEGIN
94     WRITE(' nom <JOE> ? '); READLN(nom);
95     WRITE(' age <41> ? '); READLN(age);
96 END;
97 WRITE(fichier, fiche_courante);
98
99 affiche_memoire(SEG(fichier), OFS(fichier)- 16,
100                OFS(fichier)+ SIZEOF(fichier)+ 16- 1);
101
102 CLOSE(fichier);
103
104 affiche_memoire(SEG(fichier), OFS(fichier)- 16,
105                OFS(fichier)+ SIZEOF(fichier)+ 16- 1);
106
107 END; (* affiche_entier *)
108
109 PROCEDURE initialise;
110 BEGIN
111     avant:= 'AAAAAAAAAAAAAAAA';
112     apres:= 'BBBBBBBBBBBBBBBB';
113 END; (* initialise *)
114
115 BEGIN (* principal *)
116     initialise;
117     REPEAT
118         WRITELN;
119         WRITE('fcb, q(uitte ? ');
120         READ(KBD, choix); WRITELN(choix);
121         CASE choix OF
122             'f': affiche_fcb_turbo_file_of;
123         END;
124     UNTIL choix= 'q';
125 END. (* principal *)

```

Fig. 5.109 Le programme *teste_fcb_turbo_file_of*.

et voici un exemple d'exécution:

```

fcb, q(uitte ? f
taille fcb: 76
$09B6:$0260          $41 $41 $41 $41 $41 : AAAAA
$09B6:$0268 $41 $41 $41 $41 $41 $41 $41 : AAAAAAA
$09B6:$0270 $41 $41 $41 : AAA

```



```

$09B6:$0270          $00 $00 $00 $00 $00 $00 : .....
$09B6:$0278 $00 $00 $00 $00 $00 $00 $00 : .....
$09B6:$0280 $00 $00 $00 $00 $00 $00 $00 : .....
$09B6:$0288 $00 $00 $00 $00 $00 $00 $00 : .....
$09B6:$0290 $00 $00 $00 $00 $00 $00 $00 : .....
$09B6:$0298 $00 $00 $00 $00 $00 $00 $00 : .....
$09B6:$02A0 $00 $00 $00 $00 $00 $00 $00 : .....
$09B6:$02A8 $00 $00 $00 $00 $00 $00 $00 : .....
$09B6:$02B0 $00 $00 $00 $00 $00 $00 $00 : .....
$09B6:$02B8 $00 $00 $00 $00 $00 $00 $00 : .....
$09B6:$02B8          $00 : .....
$09B6:$02C0 $00 $00 $00 $00 $00 : .....
$09B6:$02C0          $42 $42 $42 : BBB
$09B6:$02C8 $42 $42 $42 $42 $42 $42 : BBBBBB

nom <JOE> ? JOE
age <41> ? $77

$09B6:$0260          $41 $41 $41 $41 $41 : AAAAA
$09B6:$0268 $41 $41 $41 $41 $41 $41 : AAAAAA
$09B6:$0270 $41 $41 $41 : AAA
$09B6:$0270          $06 $00 $06 $00 $00 : .....
$09B6:$0278 $00 $00 $00 $00 $00 $00 $46 : .....F
$09B6:$0280 $49 $4C $45 $4F $46 $2E $44 $54 : ILEOF.DT
$09B6:$0288 $41 $00 $00 $00 $00 $00 $00 : A.....
$09B6:$0290 $00 $00 $00 $00 $00 $00 $00 : .....
$09B6:$0298 $00 $00 $00 $00 $00 $00 $00 : .....
$09B6:$02A0 $00 $00 $00 $00 $00 $00 $00 : .....
$09B6:$02A8 $00 $00 $00 $00 $00 $00 $00 : .....
$09B6:$02B0 $00 $00 $00 $00 $00 $00 $00 : .....
$09B6:$02B8 $00 $00 $00 $00 $00 $00 $00 : .....
$09B6:$02B8          $00 : .....
$09B6:$02C0 $00 $00 $00 $00 $00 : .....

$09B6:$02C0          $42 $42 $42 : BBB
$09B6:$02C8 $42 $42 $42 $42 $42 $42 : BBBBBB
$09B6:$0260          $41 $41 $41 $41 $41 : AAAAA
$09B6:$0268 $41 $41 $41 $41 $41 $41 : AAAAAA
$09B6:$0270 $41 $41 $41 : AAA
$09B6:$0270          $06 $00 $06 $00 $00 : .....
$09B6:$0278 $00 $00 $00 $00 $00 $00 $46 : .....F
$09B6:$0280 $49 $4C $45 $4F $46 $2E $44 $54 : ILEOF.DT
$09B6:$0288 $41 $00 $00 $00 $00 $00 $00 : A.....
$09B6:$0290 $00 $00 $00 $00 $00 $00 $00 : .....
$09B6:$0298 $00 $00 $00 $00 $00 $00 $00 : .....
$09B6:$02A0 $00 $00 $00 $00 $00 $00 $00 : .....
$09B6:$02A8 $00 $00 $00 $00 $00 $00 $00 : .....
$09B6:$02B0 $00 $00 $00 $00 $00 $00 $00 : .....
$09B6:$02B8 $00 $00 $00 $00 $00 $00 $00 : .....
$09B6:$02B8          $03 : .....
$09B6:$02C0 $4A $4F $45 $77 $00 : JOEw.
$09B6:$02C0          $42 $42 $42 : BBB
$09B6:$02C8 $42 $42 $42 $42 $42 $42 : BBBBBB

$09B6:$0260          $41 $41 $41 $41 $41 : AAAAA
$09B6:$0268 $41 $41 $41 $41 $41 $41 : AAAAAA
$09B6:$0270 $41 $41 $41 : AAA

```

```

$09B6:$0270          $FF $FF $00 $00 $00 : .....
$09B6:$0278 $00 $00 $00 $00 $00 $00 $46 : .....F
$09B6:$0280 $49 $4C $45 $4F $46 $2E $44 $54 : ILEOF.DT
$09B6:$0288 $41 $00 $00 $00 $00 $00 $00 : A.....
$09B6:$0290 $00 $00 $00 $00 $00 $00 $00 : .....
$09B6:$0298 $00 $00 $00 $00 $00 $00 $00 : .....
$09B6:$02A0 $00 $00 $00 $00 $00 $00 $00 : .....
$09B6:$02A8 $00 $00 $00 $00 $00 $00 $00 : .....
$09B6:$02B0 $00 $00 $00 $00 $00 $00 $00 : .....
$09B6:$02B8 $00 $00 $00 $00 $00 $00 $00 : .....
$09B6:$02B8          $03 : .....
$09B6:$02C0 $4A $4F $45 $77 $00 : JOEw.
$09B6:$02C0          $42 $42 $42 : BBB
$09B6:$02C8 $42 $42 $42 $42 $42 $42 : BBBBBB

f(cb, q(uitte ? q

```

Fig. 5.110 Le contenu du fcb TURBO

En plus de ce bloc de contrôle, la mémoire contient:

- le véritable FCB du DOS (ici situé en \$00C0:\$0242, donc dans les données globales du DOS, parce que notre fichier est parmi les 8 premiers ouverts)
- des tables gérées par le DOS, dont l'une est située dans le PSP de notre programme en train de s'exécuter
- une table de conversion entre nos variables et les numéros de fichiers, située dans la zone des données globales de TURBO (en DS:\$0242)

Compte tenu de cette floraison de tables gérées par MS-DOS et par TURBO, nous recommandons:

- soit de n'utiliser que les procédures standard (RESET, READ...)
- soit de gérer tout directement, en utilisant le DOS ou même le BIOS.

5.5.3.3 LES AUTRES PROCEDURES STANDARD

Les autres procédures utilisables avec les fichiers de fiches (RESET, SEEK, CHAIN, ERASE, EXECUTE, FLUSH, RENAME, LONGSEEK, EOF, FILEPOS, LONFILEPOS, FILESIZE, LONGFILESIZE) sont compilées de façon similaire aux routines que nous avons présentées.

Signalons que nous ne pouvons pas utiliser avec un fichier de fiches les fonctions EOLN, SEEKEOF et SEEKEOLN, qui sont réservées aux fichiers de type TEXT, ni BLOCKREAD et BLOCKWRITE réservées aux fichiers de blocs.

5.5.4 LES FICHIERS DE BLOCS

5.5.4.1 TRAITEMENT D'UN FICHIER DE BLOCS

Voici un programme utilisant un fichier de blocs (FILE;):

```

0  (* 001 FILE *)
1  (* 11 avr 87 *)
2
3  (*$r+*)
4  PROGRAM desassemble_fichier_de_blocs;
5  VAR fichier: FILE;
6      tampon: ARRAY[0..$FF] OF BYTE;
7      nom_fichier: STRING($20);
8      taille_bloc: INTEGER;
9      nombre_blocs, nombre_lus: INTEGER;
10 BEGIN
11     ASSIGN(fichier, nom_fichier);
12     REWRITE(fichier, taille_bloc);
13     BLOCKREAD(fichier, tampon, nombre_blocs, nombre_lus);
14     CLOSE(fichier);
15 END.
```

Fig. 5.111 Le programme *desassemble_file*.

Voici le code généré:

```

2D4D E849DD      call    TU_PARAGRAPHS
2D50 0600      hex      0600 ; options
2D52 780B      hex      780B ; cs
2D54 3C15      hex      3C15 ; ds
2D56 DC02      hex      DC02 ; code
2D58 3E00      hex      3E00 ; globales
2D5A 0004      hex      0004 ; min piles
2D5C 00A0      hex      00A0 ; max piles
2D5E 1000      hex      1000 ; fichiers
2D60 0000      hex      0000 ; input
2D62 0000      hex      0000 ; output

2D64 8BEC      mov      bp, sp
2D66 E814DF      call    TU_TAILLE_CODE
2D69 4E00      hex      4E00 ; taille
2D6B 6D2D      adr      SAUT_PRINCIPAL

2D6D E90000      SAUT_PRINCIPAL jmp    PRINCIPAL
```

```

2D70 BF6202      PRINCIPAL mov    di, $0262
2D73 1E          push   ds
2D74 57          push   di
2D75 BFAE03      mov     di, $03AE
2D78 1E          push   ds
2D79 E806E4      call   TU_POUSSE_STRING
2D7C E80FFC      call   TU_ASSIGN_FILE

2D7F BF6202      mov     di, $0262
2D82 1E          push   ds
2D83 57          push   di
2D84 A1CF03      mov     ax, [$03CF]
2D87 E8C3FD      call   TU_REWRITE_FILE
2D8A E846E2      call   TU_IO_CHECK

2D8D BF6202      mov     di, $0262
2D90 1E          push   ds
2D91 57          push   di
2D92 BFAE02      mov     di, $02AE
2D95 1E          push   ds
2D96 57          push   di
2D97 A1D103      mov     ax, [$03D1]
2D9A 50          push   ax
2D9B BFD303      mov     di, $03D3
2D9E 1E          push   ds
2D9F E8E2FD      call   TU_BLOCKREAD
2DA2 E82EE2      call   TU_IO_CHECK

2DA5 BF6202      mov     di, $0262
2DA8 1E          push   ds
2DA9 E82CFC      call   TU_CLOSE
2DAC E824E2      call   TU_IO_CHECK

2DAF E90000      jmp     FIN

2DB2 33C0      FIN      xor     ax, ax
2DB4 E889DE      call   TU_HALT
2DB7 0000      hex      0000
```

Fig. 5.112 fichier de blocs.

Notes que:

- toutes les primitives de traitement de fichier invoquent des routines de la librairie, en ayant auparavant poussé sur la pile le segment correspondant au FCB du fichier et placé dans DI son décalage
- si la procédure utilise d'autres paramètres (nom du fichier pour ASSIGN, taille du bloc pour REWRITE, adresse du tampon, nombre de blocs à lire, nombre de blocs effectivement lus pour BLOCKREAD...), ceux ci sont soit placés dans des registres soit poussés sur la pile.
- si vous ne précisez pas la taille du bloc, TURBO utilise 128 octets par défaut
- le nombre de blocs effectivement lus par BLOCKREAD est optionnel.

Le bloc de contrôle utilisé par les fichiers de blocs est identique à celui des fichiers de fiches.

5.5.4.2 LES AUTRES PROCEDURES STANDARD

Les autres procédures utilisables avec les fichiers de blocs (RESET, SEEK, CHAIN, ERASE, EXECUTE, FLUSH, RENAME, LONGSEEK, EOF, FILEPOS, LONFILEPOS, FILESIZE, LONGFILESIZE) sont compilées de façon similaire aux routines que nous avons présentées.

Signalons que nous ne pouvons pas utiliser avec un fichier de blocs les fonctions EOLN, SEEKEOF et SEEKEOLN, qui sont réservées aux fichiers de type TEXT, ni READ et WRITE réservées aux fichiers de fiches.

5.5.5 LES FICHIERS DE TYPE TEXT

5.5.5.1 TRAITEMENT D'UN FICHIER DE TYPE TEXT

Voici un programme utilisant un fichier de type TEXT:

```

0  (* 001 texte *)
1  (* 11 avr 87 *)
2
3  (*$r+*)
4  PROGRAM desassemble_texte;
5  VAR texte: TEXT;
6      caractere: CHAR;
7      nom_texte: STRING[$20];
8  BEGIN
9      ASSIGN(texte, nom_texte);
10     REWRITE(texte);
11     READ(texte, caractere);
12     WRITE(texte, caractere);
13     CLOSE(texte);
14 END.
```

Fig. 5.113 Le programme *desassemble_texte*.

Voici le code généré:

```

2D4D E849DD      call    TU_PARAGRAPHS
2D50 0600      hex     0600 ; options
2D52 780B      hex     780B ; cs
2D54 3C15      hex     3C15 ; ds
2D56 DD02      hex     DD02 ; code
2D58 3500      hex     3500 ; globales
2D5A 0004      hex     0004 ; min piles
2D5C 00A0      hex     00A0 ; max piles
2D5E 1000      hex     1000 ; fichiers
2D60 0000      hex     0000 ; input
2D62 0000      hex     0000 ; output
```

```

2D64 8BEC      mov     bp, sp
2D66 E814DF     call    TU_TAILLE_CODE
2D69 5F00      hex     5F00 ; taille
2D6B 6D2D      adr     SAUT_PRINCIPAL

2D6D E90000     jmp     PRINCIPAL

2D70 BF6202     mov     di, $0262
2D73 1E        push    ds
2D74 57        push    di
2D75 BF2F03     mov     di, $032F
2D78 1E        push    ds
2D79 E806E4     call    TU_POUSSE_STRING
2D7C E81EF5     call    TU_ASSIGN_TXT
2D7F E851E2     call    TU_IO_CHECK

2D82 BF6202     mov     di, $0262
2D85 1E        push    ds
2D86 B98000     mov     cx, $0080
2D89 E86AF5     call    TU_REWRITE_TXT
2D8C E844E2     call    TU_IO_CHECK

2D8F BF6202     mov     di, $0262
2D92 1E        push    ds
2D93 E86FF7     call    TU_DEB_READ_TE
2D96 BF2E03     mov     di, $032E
2D99 1E        push    ds
2D9A E88FF9     call    TU_READ_CAR
2D9D E833E2     call    TU_IO_CHECK

2DA0 BF6202     mov     di, $0262
2DA3 1E        push    ds
2DA4 E88DF7     call    TU_DEB_W_TEXT
2DA7 A02E03     mov     al, [$032E]
2DAA 32E4      xor     ah, ah
2DAC 50        push    ax
2DAD B80000     mov     ax, $0000
2DB0 E8B4FA     call    TU_WRITE_CAR
2DB3 E81DE2     call    TU_IO_CHECK

2DB6 BF6202     mov     di, $0262
2DB9 1E        push    ds
2DBA E860F6     call    TU_CLOSE_TEXT
2DBD E813E2     call    TU_IO_CHECK

2DC0 E90000     jmp     FIN

2DC3 33C0      xor     ax, ax
2DC5 E878DE     call    TU_HALT
2DC8 0000      hex     0000
```

Fig. 5.114 fichier de type TEXT.

Notez que:

- les primitives d'ouverture, fermeture, changement de nom... fonctionnent exactement comme pour les fichiers de fiches ou les fichiers de blocs: le compilateur ne génère que des appels à la librairie, en passant l'adresse du FCB. Les routines de la librairie sont chargées d'effectuer les appels à MS-DOS, en utilisant les numéros de fichiers ("handle")
- les procédures READ et WRITE fonctionnent comme les lecture clavier ou écriture écran, mais des routines d'initialisation spécifiques (TU_DEB_READ_TEXT et TU_DEB_WRITE_TEXT) sont invoquées
- la taille du tampon est par défaut de \$80. Cette valeur est utilisée par RESET et REWRITE. Nous pouvons la modifier au moment de la déclaration:

```
VAR fichier_texte: TEXT[$20];
```

Le problème est ici compliqué par:

- la structure par ligne (qui doit être testable par EOLN, avec concordance de EOLN et EOF à TRUE en fin de fichier)
- l'impossibilité de détecter à l'avance la fin de ligne ou de fichier lorsque le fichier est un périphérique autre que le disque (clavier, ligne série...)
- les propositions du PASCAL standard (remplacer le RETURN par un ESPACE, en positionnant EOLN, possibilité de lire plusieurs entiers par READ, en les séparant par des signes d'opération, par exemple)
- les choix de TURBO: utilisation de tampons par défaut, interdiction de lire des entiers séparés par des signes + ou -, possibilités d'édition temporaire...)
- la possibilité de gérer les périphériques de type caractère (le clavier, l'écran, le modem) comme des fichiers de type TEXT
- la possibilité de rediriger un fichier TEXT vers la disquette ou vers un fichier de caractère

5.5.5.2 LES PERIPHERIQUES COMME FICHIERS TEXT

Voici un programme qui émet directement des caractères vers des périphériques autres que le périphérique standard:

```
0 (* 001 write4 *)
1 (* 11 avr 87 *)
2
3 (*$r+*)
4 PROGRAM desassemble_write_standard;
5   VAR entier: INTEGER;
6   BEGIN
7     WRITE(CON, entier);
8     WRITE(TRM, entier);
9     WRITE(KBD, entier);
10    WRITE(LST, entier);
11  END.
```

Fig. 5.115 Le programme *desassemble_write_standard*.

Voici le code généré:

2D4D E849DD	call	TU_PARAGRAPHS
2D50 0600	hex	0600 ; options
2D52 780B	hex	780B ; cs
2D54 3C15	hex	3C15 ; ds
2D56 DD02	hex	DD02 ; code
2D58 2700	hex	2700 ; globales
2D5A 0004	hex	0004 ; min piles
2D5C 00A0	hex	00A0 ; max piles
2D5E 1000	hex	1000 ; fichiers
2D60 0000	hex	0000 ; input
2D62 0000	hex	0000 ; output
2D64 8BEC	mov	bp, sp
2D66 E814DF	call	TU_TAILLE_CODE
2D69 5F00	hex	5F00 ; taille
2D6B 6D2D	adr	SAUT_PRINCIPAL
2D6D E90000	SAUT_PRINCIPAL jmp	PRINCIPAL
2D70 BF4801	PRINCIPAL mov	di, \$0148
2D73 1E	push	ds
2D74 E8BDF7	call	TU_DEB_W_TEXT
2D77 A16202	mov	ax, [\$0262]
2D7A 50	push	ax
2D7B B80000	mov	ax, \$0000
2D7E E802FB	call	TU_WRITE_ENTIE
2D81 E84FE2	call	TU_IO_CHECK
2D84 BF4801	mov	di, \$0148
2D87 1E	push	ds
2D88 E8A9F7	call	TU_DEB_W_TEXT
2D8B A16202	mov	ax, [\$0262]
2D8E 50	push	ax
2D8F B80000	mov	ax, \$0000
2D92 E8EEFA	call	TU_WRITE_ENTIE
2D95 E83BE2	call	TU_IO_CHECK
2D98 BF4C01	mov	di, \$014C
2D9B 1E	push	ds
2D9C E895F7	call	TU_DEB_W_TEXT
2D9F A16202	mov	ax, [\$0262]
2DA2 50	push	ax
2DA3 B80000	mov	ax, \$0000
2DA6 E8DAFA	call	TU_WRITE_ENTIE
2DA9 E827E2	call	TU_IO_CHECK
2DAC BF5001	mov	di, \$0150
2DAF 1E	push	ds
2DB0 E881F7	call	TU_DEB_W_TEXT
2DB3 A16202	mov	ax, [\$0262]
2DB6 50	push	ax
2DB7 B80000	mov	ax, \$0000
2DBA E8C6FA	call	TU_WRITE_ENTIE
2DBD E813E2	call	TU_IO_CHECK


```

2DC0 E90000          jmp    FIN
2DC3 33C0            xor    ax, ax
2DC5 E878DE          call   TU_HALT
2DC8 0000            hex    0000

```

Fig. 5.116 vers des périphériques standard

Notez que:

- le code est identique au code d'affichage vers un fichier TEXT. Seule l'adresse du bloc de contrôle est modifiée
- ces fichiers TEXT sont ouverts par défaut. Il n'est pas nécessaire d'exécuter un ASSIGN et un RESET ou REWRITE
- les autres fichiers (AUX et USR) sont situés aux adresses \$0154 et \$0158.
- l'émission vers LST ne garantit pas que la transmission fonctionnera. Encore faut-il que la liaison série ait été initialisée correctement auparavant

De façon similaire pour READ, nous avons:

```

0  (* 001 read3 *)
1  (* 11 avr 87 *)
2
3  (*$r+*)
4  PROGRAM desassemble_read_standard;
5  VAR entier: INTEGER;
6  BEGIN
7    READ(CON, entier);
8    READ(trm, entier);
9    READ(KBD, entier);
10   READ(LST, entier);
11   END.

```

Fig. 5.117 Le programme desassemble_read_standard.

avec le code suivant:

```

2D4D E849DD          call   TU_PARAGRAPHS
2D50 0600            hex    0600 ; options
2D52 780B            hex    780B ; cs
2D54 3C15            hex    3C15 ; ds
2D56 DC02            hex    DC02 ; code
2D58 2700            hex    2700 ; globales
2D5A 0004            hex    0004 ; min piles
2D5C 00A0            hex    00A0 ; max piles
2D5E 1000            hex    1000 ; fichiers
2D60 0000            hex    0000 ; input
2D62 0000            hex    0000 ; output

```

```

2D64 8BEC            mov    bp, sp
2D66 E814DF          call   TU_TAILLE_CODE
2D69 5300            hex    5300 ; taille
2D6B 6D2D            adr     SAUT_PRINCIPAL
2D6D E90000          jmp     PRINCIPAL
2D70 BF4801          mov     di, $0148
2D73 1E              push    ds
2D74 E88EF7          call   TU_DEB_READ_TE
2D77 BF6202          mov     di, $0262
2D7A 1E              push    ds
2D7B E8C2F9          call   TU_READ_ENTIER
2D7E E852E2          call   TU_IO_CHECK
2D81 BF4801          mov     di, $0148
2D84 1E              push    ds
2D85 E87DF7          call   TU_DEB_READ_TE
2D88 BF6202          mov     di, $0262
2D8B 1E              push    ds
2D8C E8B1F9          call   TU_READ_ENTIER
2D8F E841E2          call   TU_IO_CHECK
2D92 BF4C01          mov     di, $014C
2D95 1E              push    ds
2D96 E86CF7          call   TU_DEB_READ_TE
2D99 BF6202          mov     di, $0262
2D9C 1E              push    ds
2D9D E8A0F9          call   TU_READ_ENTIER
2DA0 E830E2          call   TU_IO_CHECK
2DA3 BF5001          mov     di, $0150
2DA6 1E              push    ds
2DA7 E85BF7          call   TU_DEB_READ_TE
2DAA BF6202          mov     di, $0262
2DAD 1E              push    ds
2DAE E88FF9          call   TU_READ_ENTIER
2DB1 E81FE2          call   TU_IO_CHECK
2DB4 E90000          jmp     FIN
2DB7 33C0            xor     ax, ax
2DB9 E884DE          call   TU_HALT
2DBC 0000            hex    0000

```

Fig. 5.118 READ des périphériques TEXT

5.5.5.3 LE BLOC DE CONTROLE DES FICHIERS TEXT

Actuellement, pour les fichiers TEXT sur disque, TURBO utilise un bloc de contrôle limité, contenant:

- aux deux premiers octets le numéro de fichier
- aux deux octets suivant:
 - dans les bits 0 à 3, sur 4 bits le fichier logique
 - dans le bit 5, une étiquette indiquant qu'un entier a été lu par READ (le caractère suivant est déjà lu)
 - dans le bit 6, une étiquette indiquant que le fichier est un fichier qui peut être écrit
 - dans le bit 7, une étiquette indiquant que le fichier est un fichier qui peut être lu

Voici un programme qui affiche ces blocs de contrôle:

```

0 (* 021 fcbut *)
1 (* 28 avr 87 *)
2
3 (*$r+*)
4 PROGRAM teste_fcb_turbo_texte;
5   TYPE t_array_16_caracteres = ARRAY[0..15] OF CHAR;
6   VAR choix: CHAR;
7       avant: t_array_16_caracteres;
8       fichier_texte: TEXT[$20];
9       apres: t_array_16_caracteres;
10
11  PROCEDURE affiche_octet_hexadecimal(p_octet: BYTE);
12    CONST ks_hex: ARRAY[0..15] OF CHAR = '0123456789ABCDEF';
13    BEGIN
14      WRITE(ks_hex[(p_octet AND $F0) SHR 4], ks_hex[p_octet AND $0F]);
15    END; (* affiche_octet_hexadecimal *)
16
17  PROCEDURE affiche_entier_hexadecimal(p_entier: INTEGER);
18    BEGIN
19      affiche_octet_hexadecimal(HI(p_entier));
20      affiche_octet_hexadecimal(LO(p_entier));
21    END; (* affiche_entier_hexadecimal *)
22
23  PROCEDURE affiche_memoire(p_segment, p_decalage_debut,
...
71    END; (* affiche_memoire *)
72
73  PROCEDURE affiche_fcb;
74    VAR fcb_texte: RECORD
75      numero: INTEGER;
76      etiquette: INTEGER;
77      decalage_tampon: INTEGER;
78      taille_tampon: INTEGER;
79      decalage_prochain_caractere: INTEGER;
80      decalage_fin_tampon: INTEGER;
81    END ABSOLUTE fichier_texte;
```

```

82 BEGIN (* affiche_fcb *);
83 WITH fcb_texte DO
84 BEGIN
85   affiche_entier_hexadecimal(numero);
86   CASE etiquette AND $000F OF
87     0: WRITE(' disque ');
88     1: WRITE(' CON ');
89     2: WRITE(' KBD ');
90     3: WRITE(' LST ');
91     4: WRITE(' AUX ');
92     5: WRITE(' USR ');
93   END;
94   (*
95   WRITE((etiquette AND $0010) SHR 4: 2);
96   *)
97   WRITE((etiquette AND $0020) SHR 5: 2);
98   WRITE((etiquette AND $0040) SHR 6: 2);
99   WRITE((etiquette AND $0080) SHR 7: 2);
100  WRITE(' ');
101  affiche_entier_hexadecimal(decalage_tampon); WRITE(' ');
102  affiche_entier_hexadecimal(taille_tampon); WRITE(' ');
103  affiche_entier_hexadecimal(decalage_prochain_caractere); WRITE(' ');
104  affiche_entier_hexadecimal(decalage_fin_tampon);
105  WRITELN;
106  END;
107 END; (* affiche_le_fcb *)
108
109 PROCEDURE affiche_fcb_turbo_texte_disque;
110 VAR l_caractere: CHAR;
111     l_entier: INTEGER;
112     l_ligne: STRING[80];
113 BEGIN (* affiche_fcb_turbo_texte *)
114   ASSIGN(fichier_texte, 'TEXT.TXT');
115   REWRITE(fichier_texte);
116   (* interdit:
117   READ(fichier_texte, l_caractere);
118   *)
119   affiche_memoire(SEG(fichier_texte), OFS(fichier_texte)-16,
120                  OFS(fichier_texte)+SIZEOF(fichier_texte)+16-1);
121
122   WRITE(' ligne <5678> ? ');
123   READLN(l_ligne);
124
125   WRITELN(' num sortie f o i tamp taille proch fin ');
126   affiche_fcb;
127
128   WRITELN(fichier_texte, l_ligne);
129   affiche_fcb;
130   WRITELN(fichier_texte, l_ligne);
131   affiche_fcb;
132   WRITELN(fichier_texte, l_ligne);
133   affiche_fcb;
134
135   CLOSE(fichier_texte);
136   affiche_fcb; WRITELN;
137
```



```

138 ASSIGN(fichier_texte, 'TEXT.TXT');
139 RESET(fichier_texte);
140 (* interdit:
141 WRITE(fichier_texte, l_ligne);
142 *)
143
144 affiche_memoire(SEG(fichier_texte), OFS(fichier_texte)-16,
145                OFS(fichier_texte)+SIZEOF(fichier_texte)+16-1);
146 affiche_fcb;
147
148 READ(fichier_texte, l_entier); WRITELN(' l_entier= ', l_entier);
149 affiche_fcb;
150 READ(fichier_texte, l_caractere); WRITELN(' #, ORD(l_caractere));
151 affiche_fcb;
152
153 READLN(fichier_texte, l_entier); WRITELN(' l_entier= ', l_entier);
154 affiche_fcb;
155 READ(fichier_texte, l_caractere); WRITELN(' #, ORD(l_caractere));
156 affiche_fcb;
157 CLOSE(fichier_texte);
158 affiche_fcb;
159 END; (* affiche_fcb_turbo_texte_disque *)
160
161 PROCEDURE affiche_fcb_turbo_texte_con;
162 VAR l_caractere, l_deux: CHAR;
163     l_entier: INTEGER;
164 BEGIN (* affiche_fcb_turbo_texte_con *)
165     ASSIGN(fichier_texte, 'CON:');
166     RESET(fichier_texte);
167     affiche_fcb;
168
169     WRITE(' entier <5678> ? ');
170     READ(fichier_texte, l_entier); WRITELN(' l_entier= ', l_entier);
171     affiche_fcb;
172     WRITE(' caractere ? ');
173     READ(fichier_texte, l_caractere); WRITELN(' #, ORD(l_caractere));
174     affiche_fcb;
175     WRITE(' entier ? ');
176     READLN(fichier_texte, l_entier); WRITELN(' l_entier= ', l_entier);
177     affiche_fcb;
178     WRITE(' caractere ? ');
179     READ(fichier_texte, l_caractere); WRITELN(' #, ORD(l_caractere));
180     affiche_fcb;
181     CLOSE(fichier_texte);
182 END; (* affiche_fcb_turbo_texte_con *)
183
184 PROCEDURE initialise;
185 BEGIN
186     avant:= 'AAAAAAAAAAAAAAAA';
187     apres:= 'BBBBBBBBBBBBBBBB';
188 END; (* initialise *)
189
190 BEGIN (* principal *)
191     initialise;
192     REPEAT
193         WRITELN;

```

```

194 WRITE('fcb disque, fc(b CON:, q(uite ? ');
195 READ(KBD, choix); WRITELN(choix);
196 CASE choix OF
197     'c' : affiche_fcb_turbo_texte_con;
198     'f' : affiche_fcb_turbo_texte_disque;
199 END;
200 UNTIL choix= 'q';
201 END. (* principal *)

```

Fig. 5.119 Le programme teste_fcb_turbo_texte.

Voici un exemple d'exécution avec un fichier TEXT sur disque:

```

f(cb disque, fc(b con:, q(uite ? f
$1677:$0260          $41 $41 $41 $41 $41 : AAAAA
$1677:$0268 $41 $41 $41 $41 $41 $41 $41 : AAAAAAA
$1677:$0270 $41 $41 $41 : AAA
$1677:$0270          $06 $00 $40 $00 $BF : .@..
$1677:$0278 $02 $20 $00 $BF $02 $DF $02 $54 : . . . . .T
$1677:$0280 $45 $58 $54 $2E $54 $58 $54 $00 : EXT. TXT.
$1677:$0288 $00 $00 $00 $00 $00 $00 $00 : . . . . .
$1677:$0290 $00 $00 $00 $00 $00 $00 $00 : . . . . .
$1677:$0298 $00 $00 $00 $00 $00 $00 $00 : . . . . .
$1677:$02A0 $00 $00 $00 $00 $00 $00 $00 : . . . . .
$1677:$02A8 $00 $00 $00 $00 $00 $00 $00 : . . . . .
$1677:$02B0 $00 $00 $00 $00 $00 $00 $00 : . . . . .
$1677:$02B8 $00 $00 $00 $00 $00 $00 $00 : . . . . .
$1677:$02B8          $00 : .
$1677:$02C0 $00 $00 $00 $00 $00 $00 $00 : . . . . .
$1677:$02C8 $00 $00 $00 $00 $00 $00 $00 : . . . . .
$1677:$02D0 $00 $00 $00 $00 $00 $00 $00 : . . . . .
$1677:$02D8 $00 $00 $00 $00 $00 $00 $00 : . . . . .
$1677:$02D8          $42 : B
$1677:$02E0 $42 $42 $42 $42 $42 $42 $42 : BBBBBBBB
$1677:$02E8 $42 $42 $42 $42 $42 $42 $42 : BBBBBBB

ligne <5678> ? 5678
0006 disque 0 1 0 02BF 0020 02BF 02DF
0006 disque 0 1 0 02BF 0020 02C5 02DF
0006 disque 0 1 0 02BF 0020 02CB 02DF
0006 disque 0 1 0 02BF 0020 02D1 02DF
FFFF disque 0 0 0 02BF 0020 02BF 02DF

$1677:$0260          $41 $41 $41 $41 $41 : AAAAA
$1677:$0268 $41 $41 $41 $41 $41 $41 $41 : AAAAAAA
$1677:$0270 $41 $41 $41 : AAA

```



```

$1677:$0270          $06 $00 $80 $00 $BF : .....
$1677:$0278 $02 $20 $00 $BF $02 $BF $02 $54 : .....T
$1677:$0280 $45 $58 $54 $2E $54 $58 $54 $00 :EXT.TXT.
$1677:$0288 $00 $00 $00 $00 $00 $00 $00 : .....
$1677:$0290 $00 $00 $00 $00 $00 $00 $00 : .....
$1677:$0298 $00 $00 $00 $00 $00 $00 $00 : .....
$1677:$02A0 $00 $00 $00 $00 $00 $00 $00 : .....
$1677:$02A8 $00 $00 $00 $00 $00 $00 $00 : .....
$1677:$02B0 $00 $00 $00 $00 $00 $00 $00 : .....
$1677:$02B8 $00 $00 $00 $00 $00 $00 $00 : .....
$1677:$02B8          $35 : 5
$1677:$02C0 $36 $37 $38 $0D $0A $35 $36 $37 :678..567
$1677:$02C8 $38 $0D $0A $35 $36 $37 $38 $0D :8..5678.
$1677:$02D0 $0A $00 $00 $00 $00 $00 $00 : .....
$1677:$02D8 $00 $00 $00 $00 $00 $00 $00 : .....
$1677:$02D8          $42 : B
$1677:$02E0 $42 $42 $42 $42 $42 $42 $42 :BBBBBBBB
$1677:$02E8 $42 $42 $42 $42 $42 $42 $42 :BBBBBBBB

num sortie f o i tamp taille proch fin
0006 disque 0 0 1 02BF 0020 02BF 02BF
l_entier= 5678
0006 disque 1 0 1 02BF 0020 02C4 02D1
#13
0006 disque 0 0 1 02BF 0020 02C4 02D1
l_entier= 5678
0006 disque 0 0 1 02BF 0020 02CB 02D1
#53
0006 disque 0 0 1 02BF 0020 02CC 02D1
FFFF disque 0 0 0 02BF 0020 02CC 02D1

f(cb disque, fc(b con:, q(uite ? q

```

Fig. 5.120 Affichage du bloc de contrôle d'un fichier TEXT sur disque

Notez que:

- lorsqu'un fichier TEXT est ouvert à l'aide de REWRITE, vous n'avez pas le droit de lire ce fichier
- lorsqu'un fichier TEXT est ouvert à l'aide de RESET, vous n'avez pas le droit d'écrire dans ce fichier
- si vous lisez un entier par READ, le prochain caractère (le RETURN, qui est le seul caractère qui ne provoquera pas d'erreur) est conservé, et sera utilisé par le prochain READ. Dans notre exemple, c'est la lecture du caractère qui suit qui hérite du RETURN. Si vous lisez un entier en utilisant READLN, le RETURN et l'INTERLIGNE seront bien consommés

Voici ce que donne l'exécution de ce programme lorsque notre fichier est redirigé vers la console:

```

f(cb disque, fc(b con:, q(uite ? c
num sortie f o i tamp taille proch fin
FFFF CON 0 1 1 02BF 0020 02CC 02D1
entier <5678> ? 5678
l_entier= 5678
FFFF CON 1 1 1 02BF 0020 02CC 02D1
caractere ? #13
FFFF CON 0 1 1 02BF 0020 02CC 02D1
entier ? <8765> 8765
l_entier= 8765
FFFF CON 0 1 1 02BF 0020 02CC 02D1
caractere ? A
#65
FFFF CON 0 1 1 02BF 0020 02CC 02D1

f(cb disque, fc(b con:, q(uite ? q

```

Fig. 5.121 Le bloc de contrôle d'un fichier TEXT lié à la console

Notez que:

- le fichier associé à la console n'utilise pas un nouveau numéro de fichier (la console étant déjà ouverte)
- les zones concernant le tampon ne sont plus utilisées
- le même comportement se retrouve pour la lecture d'entiers par READ.

5.5.6 RESUME

Pour gérer les fichiers:

- le code effectue des appels à la librairie en respectant la mécanique du DOS (utilisation d'un numéro de fichier (appelé handle), qui permet au DOS de gérer le véritable FCB du fichier)
- TURBO PASCAL continue à utiliser un bloc de contrôle réduit

5.6 LES INSTRUCTIONS DE CONTROLE

Examinons à présent comment TURBO compile les instructions de test (IF et CASE) et les instructions de répétition (REPEAT, WHILE et FOR).

5.6.1 IF

Voici un programme utilisant IF:

```

0 (* 001 IF *)
1 (* 11 avr 87 *)
2
3 (*$r+ *)
4 PROGRAM desassemble_if;
5   VAR entier: INTEGER;
6   BEGIN
7     IF entier= $11
8     THEN entier:= $22;
9     IF entier= $33
10    THEN entier:= $44
11    ELSE entier:= $55;
12    IF TRUE
13    THEN ;
14  END.
```

Fig. 5.122 Le programme *desassemble_if*.

Voici le code compilé:

```

2D4D E849DD      call    TU_PARAGRAPHS
2D50 0600      hex     0600 ; options
2D52 7B06      hex     7B06 ; cs
2D54 3F10      hex     3F10 ; ds
2D56 DB02      hex     DB02 ; code
2D58 2700      hex     2700 ; globales
2D5A 0004      hex     0004 ; min piles
2D5C 00A0      hex     00A0 ; max piles
2D5E 1000      hex     1000 ; fichiers
2D60 0000      hex     0000 ; input
2D62 0000      hex     0000 ; output

2D64 8BEC      mov     bp, sp
2D66 E814DF     call    TU_TAILLE_CODE
2D69 4400      hex     4400 ; taille
2D6B 6D2D      adr     SAUT_PRINCIPAL

2D6D E90000     SAUT_PRINCIPAL jmp    PRINCIPAL
```

```

2D70 A16202     PRINCIPAL  mov     ax, [$0262]
2D73 3D1100     cmp     ax, $0011
2D76 7403       je      THEN1
2D78 E90600     jmp     FIN_IF1

2D7B B82200     THEN1     mov     ax, $0022
2D7E A36202     mov     [$0262], ax

2D81 A16202     FIN_IF1   mov     ax, [$0262]
2D84 3D3300     cmp     ax, $0033
2D87 7403       je      THEN2
2D89 E90900     jmp     ELSE2

2D8C B84400     THEN2     mov     ax, $0044
2D8F A36202     mov     [$0262], ax
2D92 E90600     jmp     FIN_IF2

2D95 B85500     ELSE2     mov     ax, $0055
2D98 A36202     mov     [$0262], ax

2D9B B80100     FIN_IF2   mov     ax, $0001
2D9E 0BC0       or      ax, ax
2DA0 7503       jnz     FIN_IF3
2DA2 E90000     jmp     FIN_IF3

2DA5 E90000     FIN_IF3   jmp     FIN

2DA8 33C0       FIN       xor     ax, ax
2DAA E893DE     call    TU_HALT
```

Fig. 5.123 Le code du programme *desassemble_if*.

Notez que:

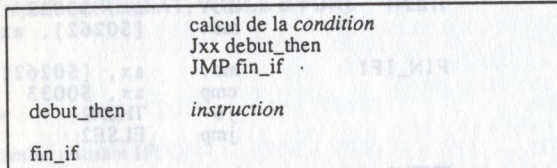
- une fois la condition de test évaluée, le code saute soit au THEN soit au ELSE ou à la fin du THEN
- comme la taille du code à exécuter lorsque la condition est vraie n'est pas connue au moment de la compilation de la condition, le code met en oeuvre un saut long:

calcul de la condition
saut court au THEN
saut long après le code exécuté lorsque la condition est vraie

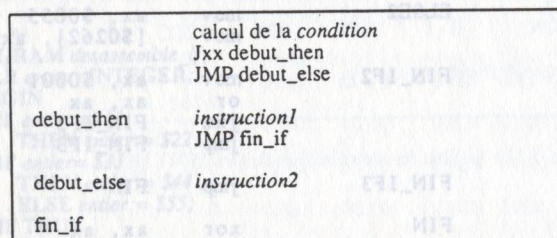
- aucune optimisation n'est effectuée pour accélérer l'évaluation de la condition (IF FALSE AND autre_condition calculera entièrement autre_condition, bien que la condition soit déjà fausse). Il est vrai que WIRTH n'avait pas demandé que les expressions booléennes soient optimisées.

RESUME

La structure de IF condition THEN instruction est la suivante:



et la structure de IF condition THEN instruction1 ELSE instruction2:



5.6.2 CASE

Voici un programme contenant un CASE:

```

0 (* 001 CASE *)
1 (* 11 avr 87 *)
2
3 (*$r+ *)
4 PROGRAM desassemble_case;
5 VAR selecteur, entier: INTEGER;
6 BEGIN
7   CASE selecteur OF
8     $11 : entier:= $22;
9     $22..$33 : entier:= $44;
10    $AAAA : entier:= $55;
11    $25 : entier:= $66;
12    $40, $50..$60, $70 : entier:= $77;
13  END;
14 END.
```

Fig. 5.124 Le programme desassemble_case.

Ce programme génère le code suivant:

2D4D E849DD	call	TU_PARAGRAPHS
2D50 0600	hex	0600 ; options
2D52 7B06	hex	7B06 ; cs
2D54 3F10	hex	3F10 ; ds
2D56 DF02	hex	DF02 ; code
2D58 2700	hex	2700 ; globales
2D5A 0004	hex	0004 ; min piles
2D5C 00A0	hex	00A0 ; max piles
2D5E 1000	hex	1000 ; fichiers
2D60 0000	hex	0000 ; input
2D62 0000	hex	0000 ; output
2D64 8BEC	mov	bp, sp
2D66 E814DF	call	TU_TAILLE_CODE
2D69 7800	hex	7800 ; taille
2D6B 6D2D	adr	SAUT_PRINCIPAL
2D6D E90000	SAUT_PRINCIPAL jmp	PRINCIPAL
2D70 A16202	PRINCIPAL mov	ax, [\$0262]
2D73 3D1100	cmp	ax, \$0011
2D76 7403	je	CAS_1
2D78 E90900	jmp	TEST_2
2D7B B82200	CAS_1 mov	ax, \$0022
2D7E A36402	mov	[\$0264], ax
2D81 E95500	jmp	FIN_CASE
2D84 3D2200	TEST_2 cmp	ax, \$0022
2D87 7C05	j1	TEST_21
2D89 3D3300	cmp	ax, \$0033
2D8C 7E03	jle	CAS_2
2D8E E90900	jmp	TEST_3
2D91 B84400	CAS_2 mov	ax, \$0044
2D94 A36402	mov	[\$0264], ax
2D97 E93F00	jmp	FIN_CASE
2D9A 3DAAAA	TEST_3 cmp	ax, \$AAAA
2D9D 7403	je	CAS_3
2D9F E90900	jmp	TEST_4
2DA2 B85500	CAS_3 mov	ax, \$0055
2DA5 A36402	mov	[\$0264], ax
2DA8 E92E00	jmp	FIN_CASE
2DAB 3D2500	TEST_4 cmp	ax, \$0025
2DAE 7403	je	CAS_4
2DB0 E90900	jmp	TEST_5
2DB3 B86600	CAS_4 mov	ax, \$0066
2DB6 A36402	mov	[\$0264], ax
2DB9 E91D00	jmp	FIN_CASE


```

2DBC 3D4000    TEST_5    cmp    ax, $0040
2DBF 7412      je        CAS_5
2DC1 3D5000    cmp    ax, $0050
2DC4 7C05      jl        TEST_51
2DC6 3D6000    cmp    ax, $0060
2DC9 7E08      jle       CAS_5
2DCB 3D7000    TEST_51   cmp    ax, $0070
2DCE 7403      je        CAS_5
2DD0 E90600    jmp       $FIN_CASE

2DD3 B87700    CAS_5      mov    ax, $0077
2DD6 A36402    mov    [$0264], ax

2DD9 E90000    FIN_CASE   jmp    FIN
2DDC 33C0      FIN        xor    ax, ax
2DDE E85FDE    call    TU_HALT

```

Fig. 5.125 Le code du programme *desassemble_case*.

Notez que:

- le CASE est implémenté strictement comme une cascade de IF
- il n'y a donc pas de limite maximale à la valeur scalaire que peuvent prendre les étiquettes (dans la limite de MAXINT)
- ceci explique aussi:
 - la possibilité d'utiliser des plages d'étiquettes
 - que lorsque plusieurs étiquettes ont la même valeur (ou qu'une valeur est située dans la plage des valeurs d'un autre cas) c'est la première valeur rencontrée qui est utilisée
- lorsque le nombre de valeurs distinctes dépasse 20, les sauts courts ne peuvent pas accéder au code du cas. Un JMP_S relai est alors placé dans le code (non présenté ici)

RESUME

La structure du CASE est la suivante:

```

évaluation de la valeur du sélecteur
  CMP AX, étiquette_11
  JE traite_cas

  CMP AX, borne_inférieure12
  JL étiquette_suivante_14
  CMP AX, borne_supérieure13
  JLE traite_cas

étiquette_suivante_14
  CMP AX, dernière_étiquette_1
  JE traite_cas
  JMP teste_liste_étiquettes_2 ou fin_case

traite_cas      instructions
                JMP fin_case

teste_liste_étiquettes_2
...
fin_case

```

5.6.3 REPEAT

Voici un programme avec des REPEAT:

```

0 (* 001 REPEAT *)
1 (* 11 avr 87 *)
2
3 (*$+ *)
4 PROGRAM desassemble_repeat;
5   VAR entier: INTEGER;
6   BEGIN
7     REPEAT
8       entier:= $11;
9     UNTIL entier> $22;
10    REPEAT
11      UNTIL FALSE;
12  END.

```

Fig. 5.126 Le programme *desassemble_repeat*.

Voici le code généré:

```

2D4D E849DD      call    TU_PARAGRAPHS
2D50 0600        hex     0600 ; options
2D52 7B06        hex     7B06 ; cs
2D54 3F10        hex     3F10 ; ds
2D56 DA02        hex     DA02 ; code
2D58 2700        hex     2700 ; globales
2D5A 0004        hex     0004 ; min piles
2D5C 00A0        hex     00A0 ; max piles
2D5E 1000        hex     1000 ; fichiers
2D60 0000        hex     0000 ; input
2D62 0000        hex     0000 ; output

2D64 8BEC        mov     bp, sp
2D66 E814DF      call    TU_TAILLE_CODE
2D69 2A00        hex     2A00 ; taille
2D6B 6D2D        adr     SAUT_PRINCIPAL

2D6D E90000      SAUT_PRINCIPAL jmp    PRINCIPAL

2D70             PRINCIPAL
2D70 B81100      DEBUT_REPEAT1 mov    ax, $0011
2D73 A36202      mov     [0262], ax

2D76 A16202      mov     ax, [0262]
2D79 3D2200      cmp     ax, $0022
2D7C 7F03        jnl     FIN_REPEAT1
2D7E E9EFFF      jmp     DEBUT_REPEAT1

2D81             FIN_REPEAT1
2D81 B80000      DEBUT_REPEAT2 mov    ax, $0000
2D84 0BC0        or      ax, ax
2D86 7503        jnz     FIN_REPEAT2
2D88 E9F6FF      jmp     DEBUT_REPEAT2

2D8B E90000      FIN_REPEAT2 jmp     FIN

2D8E 33C0        FIN xor     ax, ax
2D90 E8ADDE      call    TU_HALT

```

Fig. 5.127 Le code du programme *desassemble_repeat*.

Notez que:

- le saut arrière est effectué par un JMP, ce qui permet de traiter par une même mécanique les sauts longs et courts

RESUME

Voici la structure de REPEAT instruction UNTIL condition:

debut_repeat	instruction
	calcul de condition
	Jxx fin_repeat
fin_repeat	JMP debut_repeat

5.6.4 WHILE

Voici un programme utilisant des WHILE:

```

0 (* 001 WHILE *)
1 (* 11 avr 87 *)
2
3 (* $r+ *)
4 PROGRAM desassemble_while;
5   VAR entier: INTEGER;
6   BEGIN
7     WHILE entier > $11 DO
8       entier := $22;
9
10    WHILE TRUE DO;
11
12    WHILE FALSE AND (entier > $33) DO;
13  END.

```

Fig. 5.128 Le programme *desassemble_while*.

et le code correspondant:

```

2D4D E849DD      call    TU_PARAGRAPHS
2D50 0600        hex     0600 ; options
2D52 7B06        hex     7B06 ; cs
2D54 3F10        hex     3F10 ; ds
2D56 DC02        hex     DC02 ; code
2D58 2700        hex     2700 ; globales
2D5A 0004        hex     0004 ; min piles
2D5C 00A0        hex     00A0 ; max piles
2D5E 1000        hex     1000 ; fichiers
2D60 0000        hex     0000 ; input
2D62 0000        hex     0000 ; output

```



```

2D64 8BEC      mov     bp, sp
2D66 E814DF    call    TU_TAILLE_CODE
2D69 4D00      hex     4D00 ; programme
2D6B 6D2D      adr     SAUT_PRINCIPAL

2D6D E90000    SAUT_PRINCIPAL jmp     PRINCIPAL

2D70          PRINCIPAL
2D70 A16202    TESTE_WHILE1 mov     ax, [$0262]
2D73 3D1100    cmp     ax, $0011
2D76 7F03      jnle    DEBUT_WHILE1
2D78 E90900    jmp     FIN_WHILE1

2D7B B82200    DEBUT_WHILE1 mov     ax, $0022
2D7E A36202    mov     [$0262], ax
2D81 E9ECFF    jmp     TESTE_WHILE1

2D84          FIN_WHILE1
2D84 B80100    TESTE_WHILE2 mov     ax, $0001
2D87 0BC0      or      ax, ax
2D89 7503      jnz     DEBUT_WHILE2
2D8B E90300    jmp     FIN_WHILE2

2D8E E9F3FF    DEBUT_WHILE2 jmp     TESTE_WHILE2

2D91          FIN_WHILE2
2D91 B80000    TESTE_WHILE3 mov     ax, $0000
2D94 50        push    ax
2D95 A16202    mov     ax, [$0262]
2D98 3D3300    cmp     ax, $0033
2D9B B80100    mov     ax, $0001
2D9E 7F01      jnle    TESTE_WHILE31
2DA0 48        dec     ax
2DA1 59        pop     cx
2DA2 23C1      and     ax, cx
2DA4 0BC0      or      ax, ax
2DA6 7503      jnz     DEBUT_WHILE3
2DA8 E90300    jmp     FIN_WHILE3

2DAB E9E3FF    DEBUT_WHILE3 jmp     TESTE_WHILE3

2DAE E90000    FIN_WHILE3 jmp     FIN

2DB1 33C0      FIN      xor     ax, ax
2DB3 E88ADE    call    TU_HALT

```

Fig. 5.129 Le code programme *desassemble_while*.

Notez:

- l'utilisation des JMP longs pour contourner les instructions du WHILE vers l'avant ou vers l'arrière
- l'absence d'optimisation de test des expressions booléennes

RESUME

Le code généré pour WHILE condition DO instruction est:

teste_while	calcul de la condition
	Jxx debut_while
	JMP fin_while
debut_while	instruction
	JMP teste_while
fin_while	

5.6.5 FOR

Et un programme muni de boucles FOR:

```

0 (* 001 FOR *)
1 (* 11 avr 87 *)
2
3 (* $r+ *)
4 PROGRAM desassemble_for;
5   VAR indice, borne1, borne2: INTEGER;
6       indice1, indice2: INTEGER;
7   BEGIN
8     FOR indice:= borne1 TO borne2 DO
9       WRITELN;
10
11     FOR indice1:= $10 TO $20 DO
12       FOR indice2:= $30 TO $40 DO
13         indice:= SUCC(indice);
14
15     FOR indice:= $22 TO $11 DO
16       indice:= SUCC(indice);
17   END.

```

Fig. 5.130 Le programme *desassemble_for*.

et voici son code:


```

2D4D E849DD      call    TU_PARAGRAPHS
2D50 0600        hex     0600 ; options
2D52 7B06        hex     7B06 ; cs
2D54 3F10        hex     3F10 ; ds
2D56 E202        hex     E202 ; code
2D58 2700        hex     2700 ; globales
2D5A 0004        hex     0004 ; min piles
2D5C 00A0        hex     00A0 ; max pile
2D5E 1000        hex     1000 ; fichiers
2D60 0000        hex     0000 ; input
2D62 0000        hex     0000 ; output

2D64 8BEC        mov     bp, sp
2D66 E814DF      call    TU_TAILLE_CODE
2D69 A600        hex     A600 ; taille
2D6B 6D2D        adr     SAUT_PRINCIPAL

2D6D E90000      SAUT_PRINCIPAL jmp     PRINCIPAL

2D70             PRINCIPAL
2D70 A16402      INIT_FOR1   mov     ax, [$0264]
2D73 50          push     ax
2D74 A16602      mov     ax, [$0266]
2D77 59          pop      cx
2D78 91          xchg     cx
2D79 2BC8        sub      cx, ax
2D7B 7D03        jnl     ENTRE_FOR1
2D7D E91900      jmp      FIN_FOR1

2D80 41          ENTRE_FOR1  inc     cx
2D81 A36202      mov     [$0262], ax
2D84 51          BOUCLE_FOR1 push     cx
2D85 E89AF7      call    TU_DEB_RW_DEFAU
2D88 E8B4FB      call    TU_WRITELN
2D8B E845E2      call    TU_IO_CHECK
2D8E 59          pop      cx
2D8F 49          dec      cx
2D90 7407        je      FIN_FOR1
2D92 FF066202    inc     w [$0262]
2D96 E9EBFF      jmp      BOUCLE_FOR1

2D99             FIN_FOR1
2D99 B81000      INIT_FOR2   mov     ax, $0010
2D9C 50          push     ax
2D9D B82000      mov     ax, $0020
2DA0 59          pop      cx
2DA1 91          xchg     cx
2DA2 2BC8        sub      cx, ax
2DA4 7D03        jnl     ENTRE_FOR2
2DA6 E93700      jmp      FIN_FOR2

```

```

2DA9 41          ENTRE_FOR2  inc     cx
2DAA A36802      mov     [$0268], ax
2DAD 51          BOUCLE_FOR2 push     cx
2DAE B83000      INIT_FOR3   mov     ax, $0030
2DB1 50          push     ax
2DB2 B84000      mov     ax, $0040
2DB5 59          pop      cx
2DB6 91          xchg     cx
2DB7 2BC8        sub      cx, ax
2DB9 7D03        jnl     ENTRE_FOR3
2DBB E91700      jmp      FIN_FOR3

2DBE 41          ENTRE_FOR3  inc     cx
2DBF A36A02      mov     [$026A], ax
2DC2 51          BOUCLE_FOR3 push     cx
2DC3 A16202      mov     ax, [$0262]
2DC6 40          inc     ax
2DC7 A36202      mov     [$0262], ax
2DCA 59          pop      cx
2DCB 49          dec      cx
2DCC 7407        je      FIN_FOR3
2DCE FF066A02    inc     w [$026A]
2DD2 E9EDFF      jmp      BOUCLE_FOR3

2DD5 59          FIN_FOR3    pop      cx
2DD6 49          dec      cx
2DD7 7407        je      FIN_FOR2
2DD9 FF066802    inc     w [$0268]
2DDD E9CFFF      jmp      BOUCLE_FOR2

2DE0             FIN_FOR2
2DE0 B82200      INIT_FOR4   mov     ax, $0022
2DE3 50          push     ax
2DE4 B81100      mov     ax, $0011
2DE7 59          pop      cx
2DE8 91          xchg     cx
2DE9 2BC8        sub      cx, ax
2DEB 7D03        jnl     ENTRE_FOR4
2DED E91700      jmp      FIN_FOR4

2DF0 41          ENTRE_FOR4  inc     cx
2DF1 A36202      mov     [$0262], ax
2DF4 51          BOUCLE_FOR4 push     cx
2DF5 A16202      mov     ax, [$0262]
2DF8 40          inc     ax
2DF9 A36202      mov     [$0262], ax
2DFC 59          pop      cx
2DFD 49          dec      cx
2DFE 7407        je      FIN_FOR4
2E00 FF066202    inc     w [$0262]
2E04 E9EDFF      jmp      BOUCLE_FOR4

2E07 E90000      FIN_FOR4    jmp      FIN

2E0A 33C0        FIN         xor     ax, ax
2E0C E831DE      call    TU_HALT

```

Fig. 5.131 Le code du programme *desassemble_for*.

Notez que:

- le code comporte une partie initialisation, les instructions du FOR et l'incrémentation du compteur
- pendant l'initialisation, la valeur de la première borne est sauvee sur la pile (pour ne pas être modifiée par l'initialisation de la seconde borne, au cas où celle-ci était réalisée par une fonction ayant des effets de bord)
- la valeur du compteur est indéterminée après la boucle:
 - si à l'entrée dans le FOR, la valeur de la borne de départ est supérieure à celle de la borne d'arrivée, nous ne rentrons jamais dans la boucle, et la valeur du compteur n'est pas modifiée par le FOR
 - si nous rentrons dans la boucle, sa valeur est égale à la borne finale
- TURBO n'empêche pas de placer dans la boucle FOR des instructions qui modifient la valeur du compteur (et par conséquent modifient le nombre de répétitions)
- il est interdit de sauter par un GOTO dans le corps du FOR (parce que CX n'est pas sur la pile). Le GOTO en dehors du FOR nécessite de même des instructions spéciales pour dépiler CX. Ce cas sera examiné plus bas.
- le cas de DOWNTO est similaire au cas de TO, en remplaçant les INC par des DEC.

Des trois instructions de répétition, FOR est, comme d'habitude, la plus complexe.

RESUME

Le code généré pour FOR *compteur* := *debut* TO *fin* DO instruction est:

```

calcul des bornes
stockage du nombre de répétitions dans CX
JGE entre_for
JMP fin_for

entre_for    INC CX
             MOV [compteur], debut

boucle_for   PUSH CX
             instruction
             POP CX
             DEC CX
             JE fin_for
             INC W [compteur]
             JMP boucle_for

fin_for

```

5.6.6 EXIT

Voici un programme comportant une instruction EXIT:

```

0 (* 001 EXIT *)
1 (* 10 avr 87 *)
2
3 PROGRAM desassemble_exit;
4 BEGIN
5   EXIT;
6   WRITELN;
7 END.

```

Fig. 5.132 Le programme *desassemble_exit*.

Voici son code:

```

2D4D E849DD      call    TU_PARAGRAPHS
2D50 0600        hex     0600 ; options
2D52 780B        hex     780B ; cs
2D54 3C15        hex     3C15 ; ds
2D56 D902        hex     D902 ; code
2D58 2700        hex     2700 ; globales
2D5A 0004        hex     0004 ; min piles
2D5C 00A0        hex     00A0 ; max piles
2D5E 1000        hex     1000 ; fichiers
2D60 0000        hex     0000 ; input
2D62 0000        hex     0000 ; output

2D64 8BEC        mov     bp, sp
2D66 E814DF      call    TU_TAILLE_CODE
2D69 1B00        hex     1B00 ; taille
2D6B 6D2D        adr     SAUT_PRINCIPAL

2D6D E90000      SAUT_PRINCIPAL jmp    PRINCIPAL
2D70 E90900      PRINCIPAL     jmp     $2D7C

2D73 E8ACF7      call    TU_WRITE_DEFAULT
2D76 E8C6FB      call    TU_WRITELN
2D79 E857E2      call    TU_IO_CHECK

2D7C E90000      LR_2D7C      jmp     FIN
2D7F 33C0        FIN         xor     ax, ax
2D81 E8BCDE      call    TU_HALT
2D84 0000        hex     0000

```

Fig. 5.133 Le code de *desassemble_exit*.

Notez que:

- le code généré effectue simplement un saut à la fin du programme. Ceci est aussi le cas pour les EXIT de procédures

5.6.7 L'INSTRUCTION GOTO

Voici un programme avec des GOTO:

```
0 (* 001 GOTO *)
1 (* 10 avr 87 *)
2
3 (*$r+*)
4 PROGRAM desassemble_goto;
5 LABEL avant, apres;
6 VAR entier: INTEGER;
7 BEGIN
8   avant:
9     entier:= $11;
10    GOTO avant;
11    GOTO apres;
12    entier:= $22;
13   apres:
14   END.
```

Fig. 5.134 Le programme *desassemble_goto*.

Le code de ce programme est:

```
2D4D E849DD      call    TU_PARAGRAPHS
2D50 0600        hex     0600 ; options
2D52 780B        hex     780B ; cs
2D54 3C15        hex     3C15 ; ds
2D56 D902        hex     D902 ; code
2D58 2700        hex     2700 ; globales
2D5A 0004        hex     0004 ; min piles
2D5C 00A0        hex     00A0 ; max piles
2D5E 1000        hex     1000 ; fichiers
2D60 0000        hex     0000 ; input
2D62 0000        hex     0000 ; output

2D64 8BEC        mov     bp, sp
2D66 E814DF      call    TU_TAILLE_CODE
2D69 2100        hex     2100 ; taille
2D6B 6D2D        adr     SAUT_PRINCIPAL

2D6D E90000      SAUT_PRINCIPAL jmp    PRINCIPAL
```

```
2D70                                PRINCIPAL
2D70 B81100      AVANT      mov     ax, $0011
2D73 A36202      mov     [$0262], ax

2D76 E9F7FF      jmp     AVANT

2D79 E90600      jmp     APRES

2D7C B82200      mov     ax, $0022
2D7F A36202      mov     [$0262], ax

2D82 E90000      APRES      jmp     FIN

2D85 33C0        FIN      xor     ax, ax
2D87 E8B6DE      call    TU_HALT
2D8A 0000        hex     0000
```

Fig. 5.135 Un programme avec des GOTO.

Notez que:

- le GOTO est mis en oeuvre par de simples sauts.

Par contre lorsque nous souhaitons sortir d'une boucle FOR, le code est un peu plus compliqué:

```
0 (* 001 gotof *)
1 (* 10 avr 87 *)
2
3 (*$r+*)
4 PROGRAM desassemble_goto_for;
5 LABEL avant, apres;
6 VAR compteur: INTEGER;
7 BEGIN
8   avant:
9     FOR compteur:= $11 TO $22 DO
10    BEGIN
11      GOTO avant;
12      GOTO apres;
13    END;
14   apres:
15   END.
```

Fig. 5.136 Le programme *desassemble_goto_for*.

dont voici le code:


```

2D4D E849DD      call    TU_PARAGRAPHS
2D50 0600        hex     0600 ; options
2D52 780B        hex     780B ; cs
2D54 3C15        hex     3C15 ; ds
2D56 DB02        hex     DB02 ; code
2D58 2700        hex     2700 ; globales
2D5A 0004        hex     0004 ; min piles
2D5C 00A0        hex     00A0 ; max piles
2D5E 1000        hex     1000 ; fichiers
2D60 0000        hex     0000 ; input
2D62 0000        hex     0000 ; output

2D64 8BEC        mov     bp, sp
2D66 E814DF      call    TU_TAILLE_CODE
2D69 3D00        hex     3D00 ; taille
2D6B 6D2D        adr     SAUT_PRINCIPAL

2D6D E90000      SAUT_PRINCIPAL jmp    PRINCIPAL

2D70              PRINCIPAL
2D70 B81100      AVANT      mov     ax, $0011
2D73 50          push     ax
2D74 B82200      mov     ax, $0022
2D77 59          pop      cx
2D78 91          xchg    cx
2D79 2BC8        sub     cx, ax
2D7B 7D03        jnl     ENTRE_FOR
2D7D E91600      jmp     FIN_FOR

2D80 41          ENTRE_FOR  inc     cx
2D81 A36202      mov     [$0262], ax

2D84 51          BOUCLE_FOR  push    cx

2D85 E91100      jmp     RELAI_GOTO_1
2D88 E91200      jmp     RELAI_GOTO_2

2D8B 59          pop      cx
2D8C 49          dec     cx
2D8D 7407        je      FIN_FOR
2D8F FF066202    inc     w [$0262]
2D93 E9EEFF      jmp     BOUCLE_FOR

2D96              FIN_FOR
2D96 E90800      APRES      jmp     FIN

2D99 58          RELAI_GOTO_1 pop     ax
2D9A E9D3FF      jmp     AVANT

2D9D 58          RELAI_GOTO_2 pop     ax
2D9E E9F5FF      jmp     APRES

2DA1 33C0        FIN        xor     ax, ax
2DA3 E89ADE      call    TU_HALT
2DA6 0000        hex     0000

```

Fig. 5.137 Des GOTO hors d'une boucle FOR.

Notez que:

- lorsqu'un GOTO vers un label situé plus loin est rencontré, le compilateur ne sait pas encore si ce label sera situé en dehors du FOR ou non. Il ne peut donc pas savoir s'il faut dépiler CX ou non. D'où les instructions spéciales, coincées à la fin du bloc, et utilisées pour tous les GOTO en dehors d'une boucle FOR.

RESUME

Les instructions EXIT et GOTO sont exécutées au moyens de sauts. Lorsqu'un GOTO ou un EXIT saute hors d'une boucle FOR, des instructions placées en fin de bloc se chargent de dépiler le nombre de répétitions.

5.7 LES APPELS DE PROCEDURES

Nous allons ici examiner le code généré par les appels de procédures. Une bonne compréhension de ce paragraphe est essentielle pour la programmation en PASCAL en général, et surtout pour la mise en oeuvre d'extensions de TURBO PASCAL (dépasser 64K de code, rendre des procédures résidentes, effectuer des EXIT de plusieurs niveaux...)

Nous allons examiner:

- un appel de routine en 8088
- un appel de procédure simple
- la pile des variables locales
- la chaîne d'appel
- l'accès aux variables visibles
- les paramètres valeur
- les paramètres variable
- les appels de fonctions

5.7.1 UN APPEL DE ROUTINE 8088

Commençons par examiner en détail comment est effectué un appel de routine en 8088. Voici un petit programme comportant un tel appel:

```

; 001 AFFA.ASM
; 29 avr 87

org $0100

call sous_routine
ret

```



```

sous_routine    push  ds
                mov   ax, $B000
                mov   ds, ax

                mov   al, 'A'
                mov   [$0280], al

                pop   ds
                ret

```

Fig. 5.138 Un programme en assembleur avec appel de sous-routine

Voici le code de ce programme après assemblage:

```

0100:                ; 001 AFFA.ASM
0100:                ; 29 avr 87
0100:                org $0100
0100:E80100        call sous_routine
0103:C3           ret
0104:
0104:1E           sous_routine push ds
0105:B800B0        mov ax, $B000
0108:8ED8         mov ds, ax
010A:
010A:B041         mov al, 'A'
010C:A28002        mov [$0280], al
010F:
010F:1F          pop ds
0110:C3           ret

```

Fig. 5.139 Le code du programme

Nous chargeons le fichier .COM correspondant. En supposant que la première adresse disponible soit \$067B:\$0000, MS-DOS:

- construit une zone de données de \$067B:\$0000 à \$067B:\$00FF (le Program Segment Prefix)
- charge le programme à l'adresse \$067B:\$0100
- place les valeurs \$067B dans les registres de segment CS, DS, SS et ES
- place la valeur \$0100 dans le compteur programme (PC)
- place la valeur \$10000 dans le pointeur de pile 8088 (SP) et pousse \$0000 sur la pile (pour permettre le retour à MS-DOS)

Voici le déroulement de l'exécution à partir de cet instant:

- juste après le chargement, la valeur du compteur programme, PC, est \$0100 et la valeur du pointeur de pile, SP, est \$FFFE:

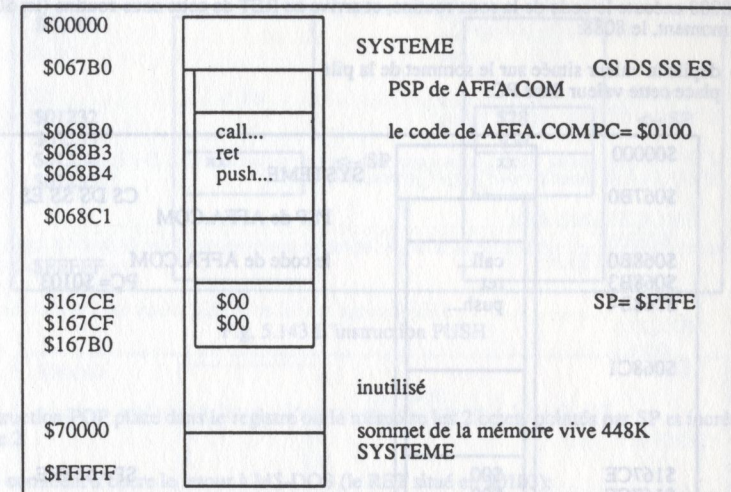


Fig. 5.140 La mémoire juste après le chargement

- le 8088 exécute l'appel de la sous-routine. Pour cela il pousse l'adresse de retour (ici \$0103) sur la pile et place dans PC l'adresse de la première instruction de la sous-routine (ici \$0104):

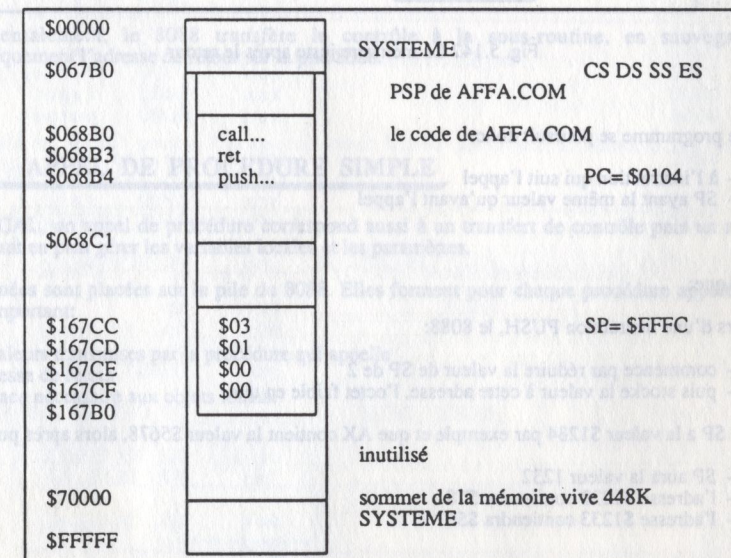


Fig. 5.141 La mémoire juste après l'appel

– le 8088 exécute le code de la sous-routine, et arrive au RET de cette sous-routine (ici \$0110). A ce moment, le 8088:

- dépile la valeur située sur le sommet de la pile
- place cette valeur dans PC:

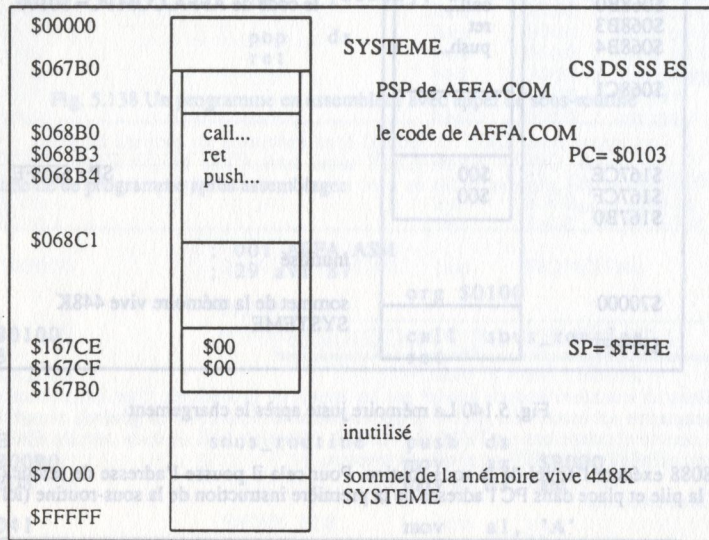


Fig. 5.142 La mémoire juste après le retour

Le programme se poursuit donc:

- à l'instruction qui suit l'appel
- SP ayant la même valeur qu'avant l'appel

Notez que:

- lors d'une instruction PUSH, le 8088:
 - commence par réduire la valeur de SP de 2
 - puis stocke la valeur à cette adresse, l'octet faible en tête.

Si SP a la valeur \$1234 par exemple et que AX contient la valeur \$5678, alors après push ax:

- SP aura la valeur 1232
- l'adresse \$1232 contiendra \$78
- l'adresse \$1233 contiendra \$56

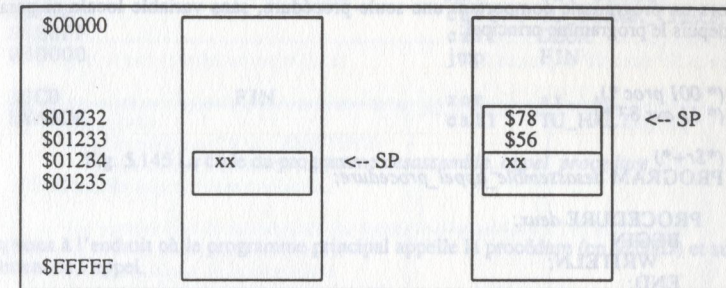


Fig. 5.143 L'instruction PUSH

– l'instruction POP place dans le registre ou la mémoire les 2 octets pointés par SP et incrémente SP de 2

– voici comment s'opère le retour à MS-DOS (le RET situé en \$0103):

- SP est égal à \$FFFF
- notre instruction ret (ici en \$01) retire du sommet de la pile l'adresse de retour. Ici elle a pour valeur \$0000 (elle a été placée par MS-DOS lors du lancement du programme). Cette adresse est la première du PSP. A cet endroit MS-DOS a placé une instruction pour redonner la main à MS-DOS (ici l'instruction INT \$20)

Fondamentalement, le 8088 transfère le contrôle à la sous-routine, en sauvegardant automatiquement l'adresse de retour sur la pile 8088.

5.7.2 APPEL DE PROCEDURE SIMPLE

En PASCAL, un appel de procédure correspond aussi à un transfert de contrôle puis un retour, mais il faut en plus gérer les variables locales et les paramètres.

Ces données sont placées sur la pile du 8088. Elles forment pour chaque procédure appelée une zone comportant:

- les valeurs transmises par la procédure qui appelle
- l'adresse de retour
- l'espace nécessaire aux objets locaux

Voici alors un programme comportant une seule procédure, sans variable locale ni paramètre, appelée depuis le programme principal:

```

0 (* 001 proc *)
1 (* 11 avr 87 *)
2
3 (* $r+*)
4 PROGRAM desassemble_appel_procedure;
5
6   PROCEDURE deux;
7     BEGIN
8       WRITELN;
9     END;
10
11   BEGIN
12     deux;
13   END.
```

Fig. 5.144 Le programme *desassemble_appel_procedure*.

Voici le code du programme:

```

2D4D E849DD      call    TU_PARAGRAPHS
2D50 0600      hex     0600 ; options
2D52 7B06      hex     7B06 ; cs
2D54 3F10      hex     3F10 ; ds
2D56 DA02      hex     DA02 ; code
2D58 2700      hex     2700 ; globales
2D5A 0004      hex     0004 ; min piles
2D5C 00A0      hex     00A0 ; max piles
2D5E 1000      hex     1000 ; fichiers
2D60 0000      hex     0000 ; input
2D62 0000      hex     0000 ; output

2D64 8BEC      mov     bp, sp
2D66 E814DF     call    TU_TAILLE_CODE
2D69 2F00      hex     2F00 ; taille
2D6B 6D2D      adr     SAUT_PRINCIPAL

2D6D E91700      SAUT_PRINCIPAL jmp    PRINCIPAL

2D70 55        DEUX      push   bp
2D71 8BEC      mov     bp, sp
2D73 55        push   bp
2D74 E90000      jmp     DEBUT_DEUX

2D77 E8A8F7     DEBUT_DEUX call    TU_DEB_RW_DEFAU
2D7A E8C2FB     call    TU_WRITELN
2D7D E853E2     call    TU_IO_CHECK
2D80 E90000      jmp     FIN_DEUX

2D83 8BE5      FIN_DEUX  mov     sp, bp
2D85 5D        pop     bp
2D86 C3        ret
```

```

2D87 B90600      PRINCIPAL  mov     cx, $0006
2D8A E8F5E1      call    TU_STACK_CHECK
2D8D E8E0FF      call    DEUX
2D90 E90000      jmp     FIN

2D93 33C0      FIN      xor     ax, ax
2D95 E8A8DE      call    TU_HALT
```

Fig. 5.145 Le code du programme *desassemble_appel_procedure*.

Plaçons nous à l'endroit où le programme principal appelle la procédure (en \$2D8D) et suivons le déroulement de l'appel.

– initialement PC est égal à \$2D8D et SP est égal à \$FFFF

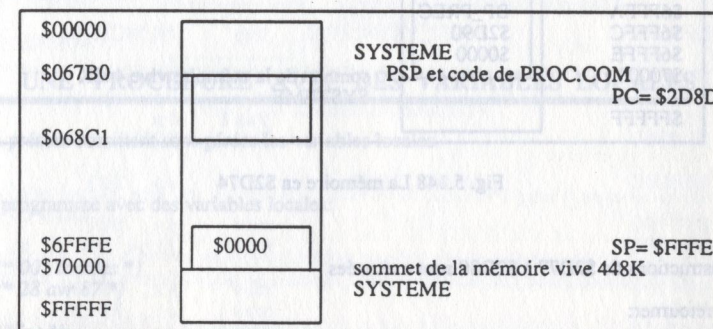


Fig. 5.146 La mémoire avant l'appel

– l'appel en \$2D8D pousse l'adresse de retour \$2D90 sur la pile et transfère le contrôle à l'adresse \$2D70

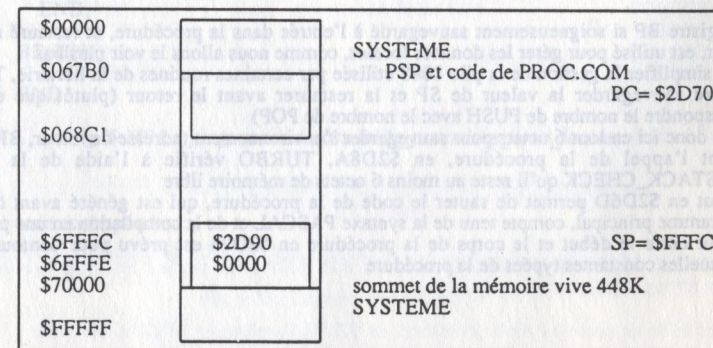


Fig. 5.147 La mémoire avant l'appel

- en \$2D70 la procédure:
- pousse BP
- place SP dans BP et pousse BP:

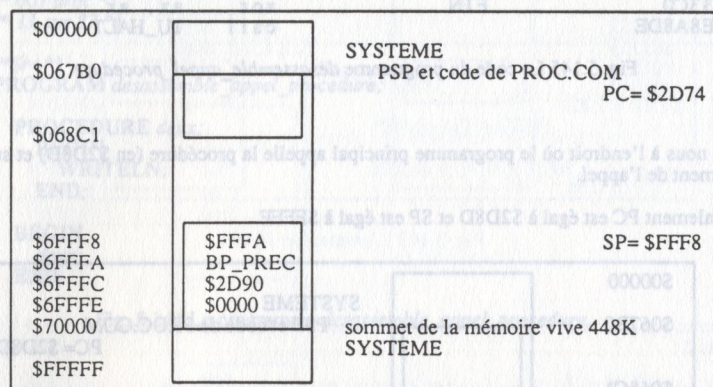


Fig. 5.148 La mémoire en \$2D74

- les instructions de \$2D77 à \$2D80 sont exécutées
- pour retourner:
 - en \$2D83 SP prend la valeur contenue dans BP, ici \$6FFA
 - en \$2D85, BP retrouve la valeur qu'il avait avant l'entrée dans cette procédure
 - en \$2D86 l'instruction RET dépile \$2D90 et place cette valeur dans PC, ce qui constitue le retour dans le programme principal

Notes que:

- le registre BP si soigneusement sauvegardé à l'entrée dans la procédure, et restauré avant le retour, est utilisé pour gérer les données locales, comme nous allons le voir plus bas
- pour simplifier la gestion de la pile 8088 utilisée par certaines routines de la librairie, TURBO préfère sauvegarder la valeur de SP et la restaurer avant le retour (plutôt que de faire correspondre le nombre de PUSH avec le nombre de POP)
- il y a donc ici en tout 6 octets pour sauvegarder l'environnement (adresse de retour, BP et SP). Avant l'appel de la procédure, en \$2D8A, TURBO vérifie à l'aide de la routine TU_STACK_CHECK qu'il reste au moins 6 octets de mémoire libre
- le saut en \$2D6D permet de sauter le code de la procédure, qui est généré avant celui du programme principal, compte tenu de la syntaxe PASCAL et de la compilation en une passe
- le saut entre le début et le corps de la procédure en \$2D74 est prévu pour contourner les éventuelles constantes typées de la procédure

RESUME

L'appel d'une procédure est réalisé:

- en vérifiant que la pile contient assez de place pour accommoder la procédure
- en effectuant un CALL

Le code de la procédure comporte:

- les instructions de sauvegarde du contexte d'appel (ici PC et SP)
- la mise en place de la pile locale
- les instructions de la procédure
- la restauration du contexte de l'appel

5.7.3 UNE PROCEDURE AVEC DES VARIABLES LOCALES

Voyons à présent comment sont gérées les variables locales.

Voici un programme avec des variables locales:

```

0 (* 001 locales *)
1 (* 28 avr 87 *)
2
3 (*$+*)
4 PROGRAM desassemble_procedure_avec_locales;
5
6 PROCEDURE deux;
7   VAR l_entier: INTEGER;
8       l_caractere: CHAR;
9 BEGIN
10   l_entier:= $77;
11   l_caractere:= 'A';
12 END;
13
14 BEGIN
15   deux;
16 END.
```

Fig. 5.149 Le programme *desassemble_procedure_avec_locales*.

Et voici le code correspondant:


```

2D4D E849DD      call    TU_PARAGRAPHS
2D50 0600      hex     0600 ; options
2D52 7B06      hex     7B06 ; cs
2D54 3F10      hex     3F10 ; ds
2D56 DB02      hex     DB02 ; code
2D58 2700      hex     2700 ; globales
2D5A 0004      hex     0004 ; min piles
2D5C 00A0      hex     00A0 ; max
2D5E 1000      hex     1000 ; fichiers
2D60 0000      hex     0000 ; input
2D62 0000      hex     0000 ; output

2D64 8BEC      mov     bp, sp
2D66 E814DF    call    TU_TAILLE_CODE
2D69 3E00      hex     3E00 ; taille
2D6B 6D2D      adr     SAUT_PRINCIPAL

2D6D E92600    SAUT_PRINCIPAL jmp    PRINCIPAL

2D70 55        DEUX      push   bp
2D71 8BEC      mov     bp, sp
2D73 55        push   bp
2D74 E90000    DEBUT_DEUX jmp    DEBUT_DEUX

2D77 83EC03    DEBUT_DEUX sub     sp, $03

2D7A B87700    mov     ax, $0077
2D7D 8946FC    mov     [bp+ $FC], ax

2D80 B84100    mov     ax, $0041
2D83 B90000    mov     cx, $0000
2D86 BAFF00    mov     dx, $00FF
2D89 E8E8E1    call    TU_RANGE_CHECK
2D8C 8846FB    mov     [bp+ $FB], al
2D8F E90000    jmp     FIN_DEUX

2D92 8BE5      FIN_DEUX   mov     sp, bp
2D94 5D        pop     bp
2D95 C3        ret

2D96 B90900    PRINCIPAL   mov     cx, $0009
2D99 E8E6E1    call    TU_STACK_CHECK
2D9C E8D1FF    call    LC_2D70
2D9F E90000    jmp     $2DA2

2DA2 33C0      LR_2DA2    xor     ax, ax
2DA4 E899DE    call    TU_HALT

```

Fig. 5.150 Le code de *desassemble_procedure_avec_locales*.

Nous voyons ici comment TURBO utilise la pile 8088 pour stocker les variables locales, en utilisant BP pour accéder à ces variables:

- après la fin du prologue de la procédure, en \$2D74, la mémoire a l'allure suivante:

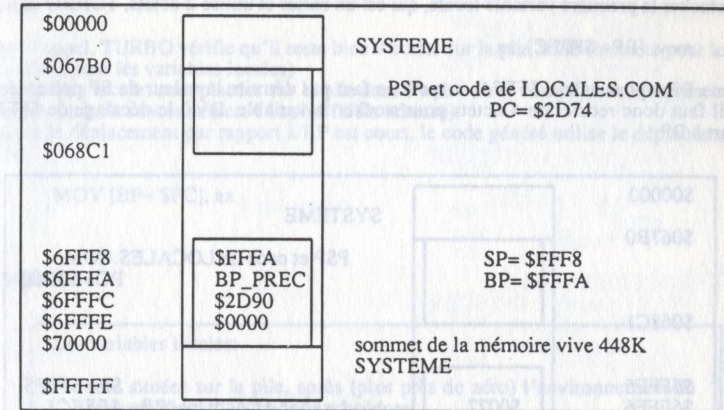


Fig. 5.151 La mémoire en \$2D74

- les variables locales vont être placées en SS:\$FFF6 et SS:\$FFF7 pour *l_entier* et en SS:\$FFF5 pour *l_caractère*.

Pour éviter que ces données ne soient détruites par des appels subséquents de routines, le code déplace alors le sommet de la pile de 3 octets. C'est le résultat de l'instruction en \$2D77.

Voilà alors la situation:

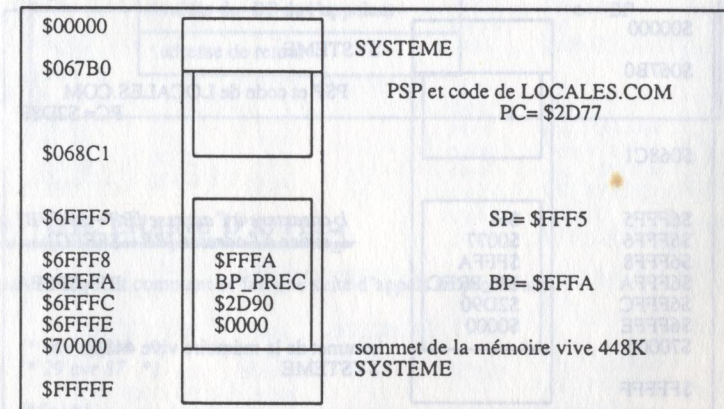


Fig. 5.152 La mémoire en \$2D77

- pour stocker la première variable locale, qui est un entier et utilise 2 octets, TURBO utilise:

```
mov [BP+ $FFFC], ax
```

Comme BP a pour valeur \$FFFA, et qu'il ne faut pas détruire la valeur de SP préservée sur la pile, il faut donc reculer de 4 octets pour stocker la variable. D'où le décalage de \$FFFC par rapport à BP:

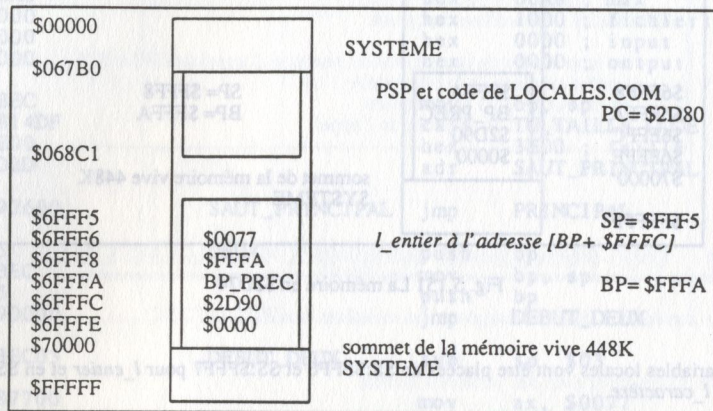


Fig. 5.153 La mémoire après $l_entier := \$77$

- de façon similaire, la variable suivante sera placée à SS:\$FFF5, soit $[BP+ \$FFFB]$:

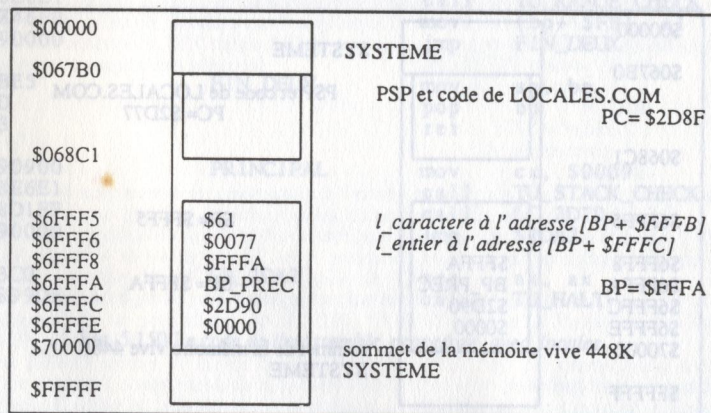


Fig. 5.154 La mémoire après $l_caractere := 'A'$

Notez que:

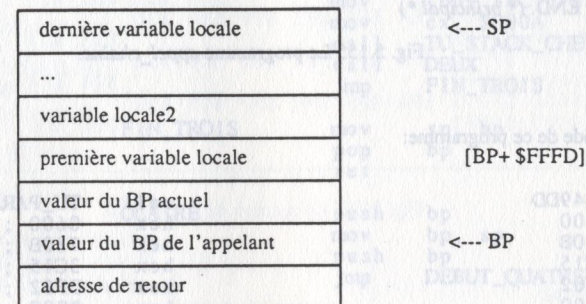
- avant l'appel, TURBO vérifie qu'il reste bien 9 octets sur la pile 8088 (6 octets pour le contexte et 3 octets pour les variables locales)
- les variables locales sont placées sur la pile dans l'ordre où le compilateur les a rencontrées (la dernière variable rencontrée est à la plus faible adresse)
- lorsque le déplacement par rapport à BP est court, le code généré utilise le déplacement sur un octet:

```
MOV [BP+ $FC], ax
```

RESUME

Les variables locales:

- sont situées sur la pile, après (plus près de zéro) l'environnement de sauvegarde et d'accès de la procédure
- sont accédées en utilisant BP avec des décalages négatifs



5.7.4 UNE CHAÎNE D'APPELS

Voici par conséquent comment se fait une suite d'appels de procédures:

```
0 (* 001 appch *)
1 (* 29 avr 87 *)
2
3 (* $r+*)
4 PROGRAM appel_chaine;
5
6 PROCEDURE deux;
7   VAR l_locale_21: INTEGER;
8     l_locale_22: INTEGER;
```



```

9      BEGIN (* deux *)
10         l_locale_21 := $1122;
11         l_locale_22 := $2222;
12         WRITELN;
13     END; (* deux *)
14
15     PROCEDURE trois;
16     VAR l_locale_3: INTEGER;
17     BEGIN (* trois *)
18         l_locale_3 := $3333;
19         deux;
20     END; (* trois *)
21
22     PROCEDURE quatre;
23     VAR l_locale_4: INTEGER;
24     BEGIN (* quatre *)
25         l_locale_4 := $4444;
26         trois;
27     END; (* quatre *)
28
29     BEGIN (* principal *)
30         quatre;
31     END. (* principal *)

```

Fig. 5.155 Le programme *appel_chaine*.

Voici le code de ce programme:

```

2D4D E849DD      call    TU_PARAGRAPHS
2D50 0600        hex     0600 ; options
2D52 780B        hex     780B ; cs
2D54 3C15        hex     3C15 ; ds
2D56 DF02        hex     DF02 ; code
2D58 2700        hex     2700 ; globales
2D5A 0004        hex     0004 ; min piles
2D5C 00A0        hex     00A0 ; max piles
2D5E 1000        hex     1000 ; fichiers
2D60 0000        hex     0000 ; input
2D62 0000        hex     0000 ; output

2D64 8BEC        mov     bp, sp
2D66 E814DF      call    TU_TAILLE_CODE
2D69 7C00        hex     7C00 ; taille
2D6B 6D2D        adr     SAUT_PRINCIPAL

2D6D E90000      SAUT_PRINCIPAL jmp    PRINCIPAL

2D70 55          DEUX      push    bp
2D71 8BEC        mov     bp, sp
2D73 55          push    bp
2D74 E90000      jmp     DEBUT_DEUX

```

```

2D77 83EC04      DEBUT_DEUX sub     sp, $04
2D7A B82211      mov     ax, $1122
2D7D 8946FC      mov     [bp+ $FC], ax
2D80 B82222      mov     ax, $2222
2D83 8946FA      mov     [bp+ $FA], ax
2D86 E899F7      call    TU_WRITE_DEFAULT
2D89 E8B3FB      call    TU_WRITELN
2D8C E844E2      call    TU_IO_CHECK
2D8F E90000      jmp     FIN_DEUX

2D92 8BE5        FIN_DEUX  mov     sp, bp
2D94 5D          pop     bp
2D95 C3          ret

2D96 55          TROIS     push    bp
2D97 8BEC        mov     bp, sp
2D99 55          push    bp
2D9A E90000      jmp     DEBUT_TROIS

2D9D 4C          DEBUT_TROIS dec     sp
2D9E 4C          dec     sp
2D9F B83333      mov     ax, $3333
2DA2 8946FC      mov     [bp+ $FC], ax
2DA5 B90A00      mov     cx, $000A
2DA8 E8D7E1      call    TU_STACK_CHECK
2DAB E8C2FF      call    DEUX
2DAE E90000      jmp     FIN_TROIS

2DB1 8BE5        FIN_TROIS mov     sp, bp
2DB3 5D          pop     bp
2DB4 C3          ret

2DB5 55          QUATRE    push    bp
2DB6 8BEC        mov     bp, sp
2DB8 55          push    bp
2DB9 E90000      jmp     DEBUT_QUATRE

2DBC 4C          DEBUT_QUATRE dec     sp
2DBD 4C          dec     sp
2DBE B84444      mov     ax, $4444
2DC1 8946FC      mov     [bp+ $FC], ax
2DC4 B90800      mov     cx, $0008
2DC7 E8B8E1      call    TU_STACK_CHECK
2DCA E8C9FF      call    TROIS
2DCD E90000      jmp     FIN_QUATRE

2DD0 8BE5        FIN_QUATRE mov     sp, bp
2DD2 5D          pop     bp
2DD3 C3          ret

2DD4 B90800      PRINCIPAL mov     cx, $0008
2DD7 E8A8E1      call    TU_STACK_CHECK
2DDA E8D8FF      call    QUATRE
2DDD E90000      jmp     FIN

2DE0 33C0        FIN        xor     ax, ax
2DE2 E85BDE      call    TU_HALT
2DE5 0000        hex     0000

```

Fig. 5.156 Le code du programme *appel_chaine*.

Ce programme ne fait que généraliser ce que nous avons vu pour l'appel d'une procédure avec des variables locales.

Notez en particulier que :

- avant l'appel, TURBO vérifie que la pile 8088 dispose d'assez de place, en tenant compte de la place occupée par les variables locales
- après l'appel, TURBO protège les locales en déplaçant au-delà d'elles le sommet de la pile 8088. Ceci a lieu même dans *deux*, car *deux* peut appeler des routines de la librairie (qui appelleront éventuellement des routines de MS-DOS ou du BIOS), et risqueraient par conséquent d'écraser les locales de *deux*.

Si nous incorporons dans ce programme la procédure d'affichage du contenu de la mémoire, et que nous remplaçons le WRITELN de la procédure *deux* par :

```
affiche_memoire(SSEG, ofs(l_locale_22), $FFFF);
```

alors nous pourrions examiner le contenu de la pile 8088 :

```
$9000:$FFE0          $22 $22 $22 $11 :  ....
$9000:$FFE8 $EA $FF $F2 $FF $B1 $30 :  ....0
$9000:$FFE8          $33 $33 :  ....33
$9000:$FFF0 $F2 $FF $FA $FF $D0 $30 :  ....0
$9000:$FFF0          $44 $44 :  ....DD
$9000:$FFF8 $FA $FF $FE $FF $15 $32 :  ....2
$9000:$FFF8          $00 $00 :  ....
```

Fig. 5.157 Le contenu de la pile 8088

Nous apercevons :

- de SS:\$FFFE à SS:\$FFFF l'adresse de retour à MS-DOS (via l'instruction INT \$20 placée par MS-DOS dans le PSP)
- de SS:\$FFF6 à SS:\$FFFD la zone utilisée par *quatre*
- de SS:\$FFE4 à SS:\$FFD5 la zone utilisée par *trois*
- de SS:\$FFE4 à SS:\$FFD5 la zone utilisée par *deux*

Chaque procédure utilise bien une zone comportant :

```
dernière variable locale
...
deuxième variable locale
première variable locale

sauvegarde du BP actuel

sauvegarde du BP de l'appelant
adresse de retour
```

Notez que :

- la figure est incomplète car nous n'avons pas représenté les zones de la procédure *affiche_memoire*, *affiche_entier_hexadécimal* et *affiche_octet_hexadécimal*, qui sont situées au-dessus des zones de *deux*
- les adresses de retour dans le programme principal ou dans *quatre* ou *trois* ne correspondent pas à celle du désassemblage, car nous avons inséré les procédures d'affichage de la pile 8088

5.7.5 APPELS EMBOITES

Lorsque des procédures PASCAL sont emboîtées les unes dans les autres, la procédure la plus profondément nichée peut utiliser les variables locales de toutes les procédures qui l'englobent.

Voici un exemple :

```
0 (* 001 visib *)
1 (* 22 jan 87 *)
2
3 PROGRAM visibilite;
4
5   PROCEDURE deux;
6   BEGIN
7     END; (* deux *)
8
9   PROCEDURE trois;
10
11   PROCEDURE quatre;
12
13   PROCEDURE cinq;
14   BEGIN (* cinq *)
15     END; (* cinq *)
16
17   BEGIN (* quatre *)
18     END; (* quatre *)
19
20   PROCEDURE six;
21   BEGIN (* six *)
22     END; (* six *)
23
24   BEGIN (* trois *)
25     END; (* trois *)
26
27   PROCEDURE sept;
28   BEGIN (* sept *)
29     END; (* sept *)
30
31   BEGIN (* principal *)
32     END; (* principal *)
```

Fig. 5.158 Le programme visibilite.

Il ne faut donc pas confondre:

- la chaîne des appels (qui est un phénomène lié à l'exécution)
- l'emboîtement (qui est lié à la structure statique du programme)
- la possibilité d'appeler une procédure (qui est lié à l'emboîtement ET à la position, à cause de la compilation en une passe)

Dans notre exemple:

- la chaîne d'appel est *principal* -> *sept* -> *trois* -> *quatre* -> *cinq*
- *six* et *deux* ne figurent pas dans cette chaîne
- *cinq* est emboîtée dans *quatre*, dans *trois* et dans *principal*
- *sept* figure dans la chaîne d'appel mais *cinq* ne peut accéder à ses données
- *sept* peut appeler *deux*, mais *deux* ne peut appeler *sept* car le compilateur n'a pas encore rencontré la déclaration de *sept* lorsqu'il compile *deux* (nous pourrions bien sûr utiliser une déclaration FORWARD qui permet de traiter le cas des procédures qui s'appellent l'une l'autre)

Les données auxquelles a accès une procédure appartiennent nécessairement à des procédures figurant dans la chaîne d'appel (*quatre*, *trois* et *principal* figurent bien dans la chaîne d'appel de *cinq*). L'inverse n'est pas vrai: une procédure n'a pas accès aux données des procédures figurant dans la chaîne d'appel qui ne l'englobent pas (*cinq* ne peut pas accéder aux données de *sept*).

A chaque appel de procédure correspond une zone locale sur la pile 8088. Pour accéder à des données englobantes il aurait été possible de remonter la suite des zones locales, en paramétrant soigneusement ces remontées.

TURBO a préféré utiliser une autre technique qui place dans chaque zone locale une table contenant l'adresse des zones englobantes.

Voici un programme avec des procédures emboîtées:

```

0  (* 001 appemb *)
1  (* 29 avr 87 *)
2
3  (*$r+*)
4  PROGRAM appel_emboite;
5    VAR globale: INTEGER;
6
7    PROCEDURE deux;
8      VAR l_locale_2: INTEGER;
9
10   PROCEDURE trois;
11     VAR l_locale_3: INTEGER;
12
```

```

13  PROCEDURE quatre;
14    VAR l_locale_4: INTEGER;
15    BEGIN (* quatre *)
16      l_locale_4 := $4444;
17      l_locale_3 := $3333;
18      l_locale_2 := $2222;
19      globale := $1111;
20    END; (* quatre *)
21
22  BEGIN (* trois *)
23    quatre;
24  END; (* trois *)
25
26  BEGIN (* deux *)
27    trois;
28  END; (* deux *)
29
30  BEGIN (* principal *)
31    deux;
32  END; (* principal *)
```

Fig. 5.159 Le programme *appel_emboite*.

Et voici le code généré:

2D4D E849DD	call	TU_PARAGRAPHS
2D50 0600	hex	0600 ; options
2D52 7B06	hex	7B06 ; cs
2D54 3F10	hex	3F10 ; ds
2D56 E002	hex	E002 ; code
2D58 2700	hex	2700 ; globales
2D5A 0004	hex	0004 ; min piles
2D5C 00A0	hex	00A0 ; max piles
2D5E 1000	hex	1000 ; fichiers
2D60 0000	hex	0000 ; input
2D62 0000	hex	0000 ; output
2D64 8BEC	mov	bp, sp
2D66 E814DF	call	TU_TAILLE_CODE
2D69 8700	hex	8700 ; taille
2D6B 6D2D	adr	SAUT_PRINCIPAL
2D6D E96F00	SAUT_PRINCIPAL jmp	PRINCIPAL
2D70 55	DEUX push	bp
2D71 8BEC	mov	bp, sp
2D73 55	push	bp
2D74 E95600	jmp	DEBUT_DEUX
2D77 55	TROIS push	bp
2D78 8BC4	mov	ax, sp
2D7A FF76FE	push	[bp+ \$FE]
2D7D 8BE8	mov	bp, ax
2D7F 55	push	bp
2D80 E93800	jmp	DEBUT_TROIS


```

2D83 55          QUATRE      push    bp
2D84 8BC4        mov     ax, sp
2D86 FF76FE      push    [bp+ $FE]
2D89 FF76FC      push    [bp+ $FC]
2D8C 8BE8        mov     bp, ax
2D8E 55          push    bp
2D8F E90000      jmp     DEBUT_QUATRE

2D92 4C          DEBUT_QUATRE dec     sp
2D93 4C          dec     sp

2D94 B84444      mov     ax, $4444
2D97 8946F8      mov     [bp+ $F8], ax

2D9A B83333      mov     ax, $3333
2D9D 8B5EFC      mov     bx, [bp+ $FC]
2DA0 36          seg     ss
2DA1 8947FA      mov     [bx+ $FA], ax

2DA4 B82222      mov     ax, $2222
2DA7 8B5EFE      mov     bx, [bp+ $FE]
2DAA 36          seg     ss
2DAB 8947FC      mov     [bx+ $FC], ax

2DAE B81111      mov     ax, $1111
2DB1 A36202      mov     [$0262], ax

2DB4 E90000      jmp     FIN_QUATRE

2DB7 8BE5        FIN_QUATRE  mov     sp, bp
2DB9 5D          pop     bp
2DBA C3          ret

2DBB 4C          DEBUT_TROIS  dec     sp
2DBC 4C          dec     sp
2DBD B90C00      mov     cx, $000C
2DC0 E8BFE1      call    TU_STACK_CHECK
2DC3 E8BDFE      call    QUATRE
2DC6 E90000      jmp     FIN_TROIS

2DC9 8BE5        FIN_TROIS   mov     sp, bp
2DCB 5D          pop     bp
2DCC C3          ret

2DCD 4C          DEBUT_DEUX   dec     sp
2DCE 4C          dec     sp
2DCF B90A00      mov     cx, $000A
2DD2 E8ADE1      call    TU_STACK_CHECK
2DD5 E89FFF      call    TROIS
2DD8 E90000      jmp     FIN_DEUX

2DDB 8BE5        FIN_DEUX    mov     sp, bp
2DDD 5D          pop     bp
2DDE C3          ret

```

```

2DDF B90800      PRINCIPAL  mov     cx, $0008
2DE2 E89DE1      call    TU_STACK_CHECK
2DE5 E888FF      call    DEUX
2DE8 E90000      jmp     FIN

2DEB 33C0        FIN         xor     ax, ax
2DED E850DE      call    TU_HALT

```

Fig. 5.160 Le code du programme *appel_emboite*.

Voici comment se déroule l'appel de *quatre*:

- en \$2DC3, *trois* appelle *quatre*
- en \$2D83, *quatre* prépare le changement de contexte:
 - en \$2D83 la valeur de BP utilisée par *trois* est sauvée sur la pile
 - en \$2D84 SP est temporairement sauvé dans AX
 - en \$2D86 et \$2D89 la table des accès de *trois* est recopiée dans la zone de *quatre*
 - en \$2D8C BP pointe vers le début de la table des accès
 - en \$2D8E cette valeur de BP est ajoutée à la table, au cas où une procédure *cinq* aurait à recopier la table de *quatre*
 - en \$2D8F le contournement des constantes typées éventuelles
 - en \$2D92 et \$2D93 l'allocation des 2 octets nécessaires à la variable locale de *quatre*

Chaque zone contient donc la table des accès aux procédures qui l'englobent. Au début de chaque procédure la table de la procédure immédiatement englobante est recopiée, et complétée par le BP de la procédure, pour préparer la copie éventuelle par une procédure emboîtée.

Schématiquement nous avons donc:

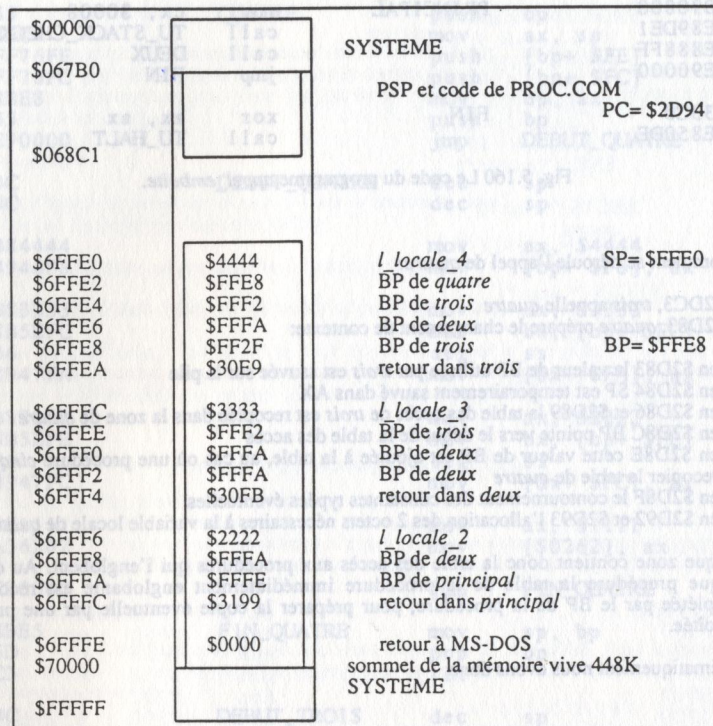


Fig. 5.161 La mémoire en \$2D94

Ces tables d'accès sont alors utilisées pour accéder aux données des procédures englobantes.

Dans le cas de *quatre*, la zone locale comporte donc \$0C octets (l'adresse de retour, les 4 valeurs de BP et la variable locale). Par rapport à la valeur de BP, la première variable locale est donc décalée de \$08 octets (par conséquent elle sera accédée par [BP+ \$FFF8])

- en \$2D94 et \$2D97 nous affectons \$4444 à la variable locale de *quatre*. Cette affectation se fait donc par rapport à BP:

```
mov [BP+ $F8], ax
```

- de \$2D9A à \$2DA1 nous traitons la variable englobante *l_locale_3*:

- le BP de *trois* est placé dans BX (la valeur de ce BP est située dans la table des accès de *quatre*, 4 octets plus haut, donc à [BP+ \$FC])
- *l_locale_3* est située dans la zone de *trois*, à 6 octets du BP de *trois* (donc accessible par [BX+ \$FA])

- même chose pour *l_locale_2*:

- le BP de *deux* est placé dans BX (la valeur de ce BP est situé dans la table des accès de *quatre*, 2 octets plus haut, donc à [BP+ \$FE])
- *l_locale_2* est située dans la zone de *deux*, à 4 octets du BP de *deux* (donc accédée par [BX+ \$FC])
- en \$2DAE et \$2DB1 nous affectons une valeur à une variable globale, en utilisant DS comme d'habitude

Notez que:

- avant chaque appel, le code teste si la pile a assez de place pour la zone locale de la procédure appelée.

Ainsi en \$2DBD, *trois* vérifie que la pile 8088 a au moins \$0C octets pour la zone locale de *quatre*.

Pour le retour, le code reste le même:

- SP pointe le BP de la procédure qui a appelé
- POP BP permet de restaurer BP vers la pile de l'appelant
- RET récupère le PC de l'appel

Si nous ajoutons la procédure d'affichage mémoire, et plaçons dans *quatre* l'appel:

```
affiche_memoire(SSEG, OFS(l_locale_4), $FFFF);
```

nous obtiendrons:

```

$6000:$FFE0 $44 $44 $E8 $FF $F2 $FF $FA $FF :DD.....
$6000:$FFE8 $F2 $FF $E9 $30                      :...0
$6000:$FFE8                      $33 $33 $F2 $FF : 33..
$6000:$FFF0 $FA $FF $FA $FF $FB $30                :....0
$6000:$FFF0                      $22 $22 : ""
$6000:$FFF8 $FA $FF $FE $FF $99 $31                :....1
$6000:$FFF8                      $00 $00 : ..

```

Fig. 5.162 Les zones locales sur la pile 8088

RESUME

Dans chaque pile locale nous avons:

la dernière variable locale	<--- SP
...	
la deuxième variable locale	
la première variable locale	
la valeur de BP actuelle	
la valeur de BP de la procédure englobante	
...	
la valeur de BP de la première procédure	
la valeur de BP de l'appelant	<--- BP
l'adresse de retour	

Lorsque la procédure est la dernière de la chaîne de l'appel, SP pointe la dernière variable locale, et BP pointe la valeur de BP de l'appelant.

5.7.6 LES PARAMETRES VALEUR ET VARIABLE

Voici à présent une procédure comportant des paramètres valeur et variable:

```

0  (* 001 varvall *)
1  (* 22 jan 87 *)
2
3  (*$r+*)
4  PROGRAM desassemble_parametres;
5    VAR globale1, globale2, globale3, globale4: INTEGER;
6
7  PROCEDURE deux(p1, p2: INTEGER; VAR pv3, pv4: INTEGER);
8    BEGIN (* deux *)
9      p1:= $1111; p2:= $2222;
10     pv3:= $3333; pv4:= $4444;
11     END; (* deux *)
12
13 BEGIN
14   deux($AAAA, $BBBB, globale2, globale4);
15 END.
```

Fig. 5.163 Le programme *desassemble_parametres*.

Voici le code généré:

```

2D4D E849DD      call    TU_PARAGRAPHS
2D50 0600        hex     0600 ; options
2D52 780B        hex     780B ; cs
2D54 3C15        hex     3C15 ; ds
2D56 DD02        hex     DD02 ; code
2D58 2700        hex     2700 ; globales
2D5A 0004        hex     0004 ; min piles
2D5C 00A0        hex     00A0 ; max piles
2D5E 1000        hex     1000 ; fichiers
2D60 0000        hex     0000 ; input
2D62 0000        hex     0000 ; output

2D64 8BEC        mov     bp, sp
2D66 E814DF      call    TU_TAILLE_CODE
2D69 5800        hex     5800 ; taille
2D6B 6D2D        adr     SAUT_PRINCIPAL

2D6D E90000      SAUT_PRINCIPAL jmp    PRINCIPAL

2D70 55          DEUX      push   bp
2D71 8BEC        mov     bp, sp
2D73 55          push   bp
2D74 E90000      jmp     DEBUT_DEUX

2D77 B81111      DEBUT_DEUX mov    ax, $1111
2D7A 89460E      mov     [bp+ $0E], ax

2D7D B82222      mov     ax, $2222
2D80 89460C      mov     [bp+ $0C], ax

2D83 C47E08      les     di, [bp+ $08]
2D86 B83333      mov     ax, $3333
2D89 26          seg     es
2D8A 8905        mov     [di], ax

2D8C C47E04      les     di, [bp+ $04]
2D8F B84444      mov     ax, $4444
2D92 26          seg     es
2D93 8905        mov     [di], ax

2D95 E90000      jmp     FIN_DEUX

2D98 8BE5        FIN_DEUX  mov     sp, bp
2D9A 5D          pop     bp
2D9B C20C00      ret     000C ; intra
```


2D9E B91200	PRINCIPAL	mov	cx, \$0012
2DA1 E8DEE1		call	TU_STACK_CHECK
2DA4 B8AAAA		mov	ax, \$AAAA
2DA7 50		push	ax
2DA8 B8BBBB		mov	ax, \$BBBB
2DAB 50		push	ax
2DAC BF6402		mov	di, \$0264
2DAF 1E		push	ds
2DB0 57		push	di
2DB1 BF6802		mov	di, \$0268
2DB4 1E		push	ds
2DB5 57		push	di
2DB6 E8B7FF		call	DEUX
2DB9 E90000		jmp	FIN
2DBC 33C0	FIN	xor	ax, ax
2DBE E87FDE		call	TU_HALT
2DC1 0000		hex	0000

Fig. 5.164 Le code du programme *desassemble_parametres*.

Notez que:

- au moment de l'appel:
 - le code vérifie si la pile dispose bien de \$12 octets (6 pour l'environnement, 2*2 pour les paramètres valeur, 2*4 pour les paramètres variable)
 - pousse la valeur des paramètres valeur sur la pile
 - pousse l'adresse des paramètres variable sur la pile
- la procédure appelée:
 - met en place son environnement comme d'habitude (ici sauvegarde du BP de l'appelant et du BP actuel). La situation est alors la suivante:

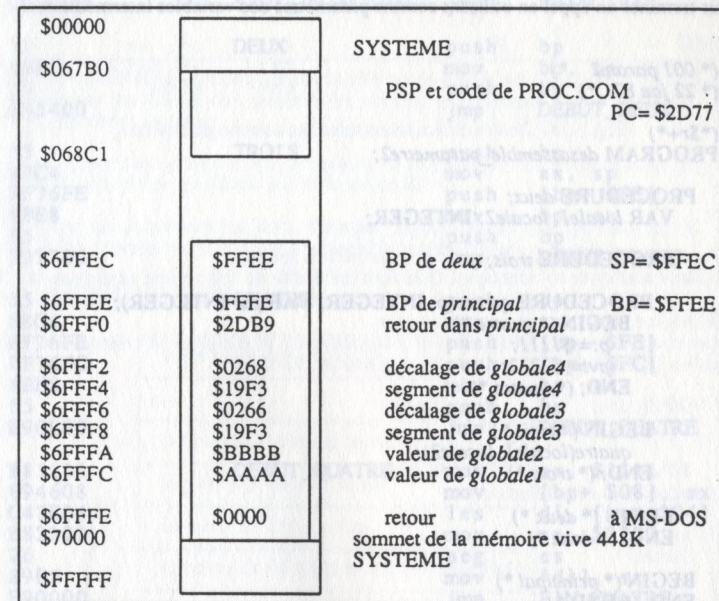


Fig. 5.165 La mémoire en \$2D77

- Les paramètres se trouvent donc sous l'environnement local. Le premier octet des paramètres se trouve après le BP du programme appelant, et après l'adresse de retour, donc à BP+ 4.

Par conséquent, pour accéder au premier paramètre valeur, le code utilisera:

MOV AX, [BP+ \$0E]

En effet ce paramètre se trouve après les 2 paramètres variable (8 octets) et après le second paramètre valeur (2 octets). En tout 4+ 8+ 2.

- pour accéder au paramètre valeur, il suffit de charger son adresse dans ES et DI. Ainsi pour le premier paramètre variable (situé avant le second paramètre variable), nous utilisons:

LES DI, [BP+ \$08]

Voici pour terminer un appel en utilisant comme paramètres des variables intermédiaires:

```

0  (* 001 param2 *)
1  (* 22 jan 87 *)
2
3  (* $r+*)
4  PROGRAM desassemble_parametre2;
5
6  PROCEDURE deux;
7    VAR locale1, locale2: INTEGER;
8
9  PROCEDURE trois;
10
11  PROCEDURE quatre(p: INTEGER; VAR pv: INTEGER);
12    BEGIN (* quatre *)
13      p:= $1111;
14      pv:= $2222;
15    END; (* quatre *)
16
17  BEGIN (* trois *)
18    quatre(locale1, locale2);
19  END; (* trois *)
20
21 BEGIN (* deux *)
22 END; (* deux *)
23
24 BEGIN (* principal *)
25 END. (* END *)

```

Fig. 5.166 Le programme *desassemble_parametre2*.

Voici le code généré:

```

2D4D E849DD      call    TU_PARAGRAPHS
2D50 0600      hex     0600 ; options
2D52 780B      hex     780B ; cs
2D54 3C15      hex     3C15 ; ds
2D56 DE02      hex     DE02 ; code
2D58 2700      hex     2700 ; globales
2D5A 0004      hex     0004 ; min piles
2D5C 00A0      hex     00A0 ; max piles
2D5E 1000      hex     1000 ; fichiers
2D60 0000      hex     0000 ; input
2D62 0000      hex     0000 ; output

2D64 8BEC      mov     bp, sp
2D66 E814DF    call    TU_TAILLE_CODE
2D69 7400      hex     7400 ; taille
2D6B 6D2D      adr     SAUT_PRINCIPAL

```

```

2D6D E90000      SAUT_PRINCIPAL jmp    PRINCIPAL

2D70 55          DEUX      push   bp
2D71 8BEC        mov     bp, sp
2D73 55          push   bp
2D74 E95400      jmp     DEBUT_DEUX

2D77 55          TROIS     push   bp
2D78 8BC4        mov     ax, sp
2D7A FF76FE      push   [bp+ $FE]
2D7D 8BE8        mov     bp, ax
2D7F 55          push   bp
2D80 E92700      jmp     DEBUT_TROIS

2D83 55          QUATRE    push   bp
2D84 8BC4        mov     ax, sp
2D86 FF76FE      push   [bp+ $FE]
2D89 FF76FC      push   [bp+ $FC]
2D8C 8BE8        mov     bp, ax
2D8E 55          push   bp
2D8F E90000      jmp     DEBUT_QUATRE

2D92 B81111      DEBUT_QUATRE mov    ax, $1111
2D95 894608      mov     [bp+ $08], ax
2D98 C47E04      les     di, [bp+ $04]
2D9B B82222      mov     ax, $2222
2D9E 26          seg     es
2D9F 8905        mov     [di], ax
2DA1 E90000      jmp     FIN_QUATRE

2DA4 8BE5        FIN_QUATRE mov    sp, bp
2DA6 5D          pop     bp
2DA7 C20600      ret     0006 ; intra

2DAA B91000      DEBUT_TROIS  mov    cx, $0010
2DAD E8D2E1      call   TU_STACK_CHECK
2DB0 8B5EFE      mov     bx, [bp+ $FE]
2DB3 36          seg     ss
2DB4 8B47FC      mov     ax, [bx+ $FC]
2DB7 50          push   ax

2DB8 8B5EFE      mov     bx, [bp+ $FE]
2DBB 36          seg     ss
2DBC 8D7FFA      lea     di, [bx+ $FA]
2DBF 16          push   ss
2DC0 57          push   di
2DC1 E8BFFF      call   LC_2D83
2DC4 E90000      jmp     FIN_TROIS

2DC7 8BE5        FIN_TROIS  mov    sp, bp
2DC9 5D          pop     bp
2DCA C3          ret

2DCB 83EC04      DEBUT_DEUX  sub     sp, $04
2DCE E90000      jmp     FIN_DEUX

2DD1 8BE5        FIN_DEUX   mov     sp, bp
2DD3 5D          pop     bp
2DD4 C3          ret

```



```

2DD5 E90000    PRINCIPAL    jmp    FIN
2DD8 33C0      FIN          xor    ax, ax
2DDA E863DE    call         TU_HALT
2DDD 0000      hex          0000

```

Fig. 5.167 Des variables intermédiaires comme paramètres.

Notez que:

- pour simplifier le code nous n'avons pas effectué tous les appels
- l'accès aux paramètres est identique. Cela était prévisible du fait qu'une procédure doit pouvoir accéder à plusieurs piles locales (d'où la table des BP englobants), mais n'a qu'un seul jeu de paramètres, tous situés avant l'environnement
- voici par contre comment *trois* place les paramètres sur la pile avant d'effectuer l'appel. Voici la situation juste avant de pousser ces paramètres (en \$2DB0):

BP de *trois* <-- SP actuel
BP de *deux*

BP de *deux* <-- BP actuel
adresse de retour dans *deux*

locale2
locale1

BP de *deux*

BP de *principal* (BP de *deux*)
adresse de retour dans *principal*

Pour récupérer la valeur d'un paramètre valeur, l'accès se fait comme d'habitude pour les variables intermédiaires:

- [BP+ \$FE] nous fournit le BP de *deux*
- cette valeur est stockée dans BX
- *locale2* se trouve alors à [BX+ \$FC] (2 octets pour sauter la copie de BP de *deux* et 2 pour accéder au premier octet de *locale1*)

Pour récupérer l'adresse de *locale2*:

- [BP+ \$FE] nous fournit le BP de *deux*
- cette valeur est stockée dans BX
- BX+ \$FA contient alors le décalage de *locale2*. Ce décalage est alors placé dans DI par l'instruction

```
LEA DI, [BX+ $FA]
```

- puis SS et DI sont poussés sur la pile

RESUME

Les paramètres sont poussés sur la pile par la procédure appelante. Celle ci pousse la valeur des paramètres valeur, et l'adresse segmentée des paramètres variable.

La procédure accède à ces paramètres en utilisant BP. Le premier octet de ces paramètres est situé à une distance de BP+ 4.

Pour les paramètres variable, l'adresse segmentée est chargée dans ES et DI qui permettent d'accéder à la variable passée comme paramètre.

La pile comporte donc:

la dernière variable locale	<--- SP
...	
la deuxième variable locale	
la première variable locale	
la valeur de BP actuelle	
la valeur de BP de la procédure englobante	
...	
la valeur de BP de la première procédure	
la valeur de BP de l'appelant	<--- BP
l'adresse de retour	
dernier paramètre	
...	
premier paramètre	

5.7.7 LES APPELS DE FONCTIONS

Voici à présent un appel de fonction:


```

0 (* 001 sfct *)
1 (* 29 aou 87 *)
2
3 PROGRAM desassemble_appel_fonction;
4   VAR entier: INTEGER;
5
6   FUNCTION f_entier: INTEGER;
7     BEGIN
8       f_entier := $77;
9     END; (* f_entier *)
10
11   BEGIN (* principal *)
12     entier := f_entier;
13   END.

```

Fig. 5.168 Le programme *desassemble_fonction*.

Et voici le code généré:

```

2D4D E849DD      call    TU_PARAGRAPHS
2D50 0600        hex     0600 ; options
2D52 780B        hex     780B ; cs
2D54 3C15        hex     3C15 ; ds
2D56 DB02        hex     DB02 ; code
2D58 2700        hex     2700 ; globales
2D5A 0004        hex     0004 ; min piles
2D5C 00A0        hex     00A0 ; max piles
2D5E 1000        hex     1000 ; fichiers
2D60 0000        hex     0000 ; input
2D62 0000        hex     0000 ; output

2D64 8BEC        mov     bp, sp
2D66 E814DF      call    TU_TAILLE_CODE
2D69 3600        hex     3600 ; taille
2D6B 6D2D        adr     SAUT_PRINCIPAL

2D6D E90000      SAUT_PRINCIPAL jmp    PRINCIPAL

2D70 55          F_ENTIER   push   bp
2D71 8BEC        mov     bp, sp
2D73 55          push   bp
2D74 E90000      jmp     DEBUT_F_ENTIER

2D77 B87700      DEBUT_F_ENTIER mov   ax, $0077
2D7A 894604      mov     [bp+$04], ax
2D7D E90000      jmp     FIN_F_ENTIER

2D80 8B4604      FIN_F_ENTIER mov   ax, [bp+$04]
2D83 8BE5        mov     sp, bp
2D85 5D          pop     bp
2D86 C20200      ret     0002 ; intra

```

```

2D89 B90800      PRINCIPAL   mov     cx, $0008
2D8C E8F3E1      call    TU_STACK_CHECK
2D8F 4C          dec     sp
2D90 4C          dec     sp
2D91 E8DCFF      call    F_FONCTION
2D94 A36202      mov     [0262], ax
2D97 E90000      jmp     FIN

2D9A 33C0        FIN         xor     ax, ax
2D9C E8A1DE      call    TU_HALT
2D9F 0000        hex     0000

```

Fig. 5.169 Le code de *desassemble_fonction*.

Notez que:

- au moment de l'appel, en 2D89, le code vérifie que la pile pourra accommoder la fonction (6 octets pour l'environnement traditionnel plus 2 octets pour le résultat entier de la fonction). La pile est alors décréémentée de 2 octets pour laisser la place au résultat de la fonction.
- dans le corps de la fonction:
 - l'affectation de la valeur à la fonction se fait en stockant directement cette valeur en [BP+\$04], où se trouve l'emplacement réservé pour le résultat de la fonction
 - avant de quitter la fonction, la valeur de celle-ci est chargée dans AX. L'expression utilisant ce résultat entier est donc assurée de trouver cette valeur dans AX.

Lorsque le résultat de la fonction est un réel ou une chaîne, le mécanisme est similaire:

```

0 (* 001 sfctst *)
1 (* 29 aou 87 *)
2
3 PROGRAM desassemble_appel_fonction_string;
4   TYPE string11= STRING[11];
5   VAR chaine: string11;
6
7   FUNCTION f_chaine: string11;
8     BEGIN
9       f_chaine := 'ABC';
10    END; (* f_chaine *)
11
12   BEGIN (* principal *)
13     chaine := f_chaine;
14   END.

```

Fig. 5.170 Le programme *desassemble_appel_fonction_chaine*.

Voici le code:

```

2D4D E849DD      call    TU_PARAGRAPHS
2D50 0600        hex     0600 ; options
2D52 780B        hex     780B ; cs
2D54 3C15        hex     3C15 ; ds
2D56 DC02        hex     DC02 ; code
2D58 2700        hex     2700 ; globales
2D5A 0004        hex     0004 ; min piles
2D5C 00A0        hex     00A0 ; max piles
2D5E 1000        hex     1000 ; fichiers
2D60 0000        hex     0000 ; input
2D62 0000        hex     0000 ; output

2D64 8BEC        mov     bp, sp
2D66 E814DF      call    TU_TAILLE_CODE
2D69 4B00        hex     4B00 ; taille
2D6B 6D2D        adr     SAUT_PRINCIPAL

2D6D E90000      SAUT_PRINCIPAL jmp    PRINCIPAL

2D70 55          F_CHAINE    push    bp
2D71 8BEC        mov     bp, sp
2D73 55          push    bp
2D74 E90000      jmp     DEBUT_F_CHAINE

2D77 8D7E04      DEBUT_F_CHAINE lea     di, [bp+ $04]
2D7A 16          push    ss
2D7B 57          push    di
2D7C E81CE4      call    TU_POUSSE_K_ST
2D7F 03          hex     03
2D80 414243      asc     'ABC'
2D83 B10B        mov     cl, $0B
2D85 E829E4      call    TU_POP_STRING
2D88 E90000      jmp     FIN_F_CHAINE

2D8B BA0000      FIN_F_CHAINE  mov     dx, $0000
2D8E B10B        mov     cl, $0B
2D90 8BE5        mov     sp, bp
2D92 5D          pop     bp
2D93 E9C5E6      jmp     TU_RESU_FCT_ST

2D96 BF6202      PRINCIPAL    mov     di, $0262
2D99 1E          push    ds
2D9A 57          push    di
2D9B B91200      mov     cx, $0012
2D9E E8E1E1      call    TU_STACK_CHECK
2DA1 83EC0C      sub     sp, $0C
2DA4 E8C9FF      call    F_CHAINE
2DA7 B10B        mov     cl, $0B
2DA9 E805E4      call    TU_POP_STRING
2DAC E90000      jmp     FIN

2DAF 33C0        FIN          xor     ax, ax
2DB1 E88CDE      call    TU_HALT
2DB4 0000        hex     0000

```

Fig. 5.171 Le code de `desassemble_appel_fonction_chaine`.

Notez que:

- la fonction pousse la constante sur la pile, dépile cette valeur dans la zone réservée sur la pile à la chaîne. L'expression utilisant peut alors utiliser cette valeur qui reste sur la pile après l'appel de la fonction.

RESUME

Le résultat d'une fonction est placé sur la pile. Dans le cas d'une fonction entière, la fonction place la valeur du résultat dans AX avant de retourner. Dans le cas de réels ou de chaînes, la valeur du résultat est laissée sur la pile.

Pendant l'exécution de la fonction, la pile comporte donc:

la dernière variable locale	<--- SP
...	
la deuxième variable locale	
la première variable locale	
la valeur de BP actuelle	
la valeur de BP de la procédure englobante	
...	
la valeur de BP de la première procédure	
la valeur de BP de l'appelant	<--- BP
l'adresse de retour	
dernier paramètre	
...	
premier paramètre	
résultat de la fonction	

5.8 LES CONSTANTES TYPEES**5.8.1 DEFINITION DES CONSTANTES TYPEES**

TURBO PASCAL a inventé une nouvelle race de données, appelées "constantes typées".

Ces objets:

- sont définis après CONST
- utilisent une syntaxe qui est une combinaison de la syntaxe des déclarations de variables et des définitions de types
- sont modifiables comme n'importe quelle variable
- sont stockés au milieu du code du programme
- sont accédés à l'aide du registre CS

Voici quelques exemples de déclarations:

```
CONST kt_entier: INTEGER= $88;
      kt_caractere: CHAR= 'A';
      kt_string: STRING[5]= 'ABC';
      kt_tableau: ARRAY[0..2] OF BYTE= ($11, $22, $33);
      kt_ensemble: SET OF CHAR= ['a'..'z', 'A'];
```

Ce qui distingue donc une constante ordinaire d'une constante typée est la présence du type de la constante entre le nom de la constante et la valeur de cette constante. La syntaxe de la définition est donc:

```
CONST nom_constante : type_constante = valeur_constante ;
```

qui est une combinaison de la syntaxe de définition des constantes:

```
CONST nom_constante = valeur_constante ;
```

et de la syntaxe de déclaration de variables:

```
VAR nom_variable : type_variable ;
```

La valeur des constantes typées est placée au milieu du code du programme. Ceci entraîne que:

- la valeur est unique. Si vous utilisez la constante en plusieurs endroits, la copie unique sera utilisée. Le code contient en effet l'adresse où se trouve la valeur (et non pas une copie de la valeur, comme pour les constantes usuelles)
- les octets utilisés pour les constantes typées diminuent d'autant la place utilisable pour les instructions, et augmentent d'autant la place pour les variables globales ou locales
- ces octets sont accessibles à partir du segment CS du 8088 (et non pas à partir de DS ou SS comme pour les variables globales ou locales). Ceci est surtout intéressant lorsqu'un programme TURBO PASCAL est rendu résident et que nous y accédons par interruption: au moment de l'interruption, seul CS et le compteur programme sont connus.

Finalement nous pouvons modifier la valeur d'une constante typée, en utilisant l'affectation ou la lecture:

```
READLN(kt_entier);
kt_tableau[3]:= 5;
```

Elles se comportent donc de ce point de vue:

- comme des variables initialisées par le compilateur (au lieu d'être initialisées par nos instructions).
- comme des variables statiques: les constantes typées d'une procédure existent en dehors de la procédure, et gardent leur valeur d'un appel à l'autre. Le fait que la valeur ne soit pas modifiée provient de ce qu'elle est stockée, en un seul exemplaire, et dans le code (et non pas créée puis détruite sur la pile des variables locales). Si une procédure récursive modifie cette valeur, c'est la dernière modification qui sera retenue (alors que pour les variables locales, les valeurs empilées seront dépilées dans l'ordre inverse, et la dernière détruite à la sortie du dernier appel).

Concrètement, ces constantes typées sont utilisées:

- pour disposer des constantes structurées
- pour économiser de la place lorsque la constante est utilisée de nombreuses fois et prend beaucoup de place (chaînes de caractères)
- pour éviter d'écrire des instructions d'initialisation de variables et pour éviter de perdre du temps à les initialiser au moment de l'exécution
- pour disposer de variables statiques
- pour disposer de variables accessibles par CS

Il aurait été plus heureux de séparer ces propriétés fort différentes:

- en permettant l'initialisation de variables sous VAR (au lieu de CONST)
- en créant l'attribut STATIQUE pour séparer l'existence d'une variable locale de l'exécution d'une procédure
- en créant l'attribut CODE pour imposer le stockage de la variable dans le code du programme

Quoiqu'il en soit, examinons comment sont traitées ces données.

5.8.2 LES CONSTANTES TYPEES SIMPLES

Voici un programme utilisant des constantes typées simples:

```
0 (* 001 const1l *)
1 (* 11 avr 87 *)
2
3 (*$r+*)
4 PROGRAM desassemble constantes typees;
5   CONST kt_entier: INTEGER= $1111;
6         kt_booleen: BOOLEAN= TRUE;
7         kt_caractere: CHAR= 'A';
8         kt_reel: REAL= 1.75;
```



```

9  VAR entier: INTEGER;
10     reel: REAL;
11     booleen: BOOLEAN;
12     caractere: CHAR;
13  BEGIN
14     kt_entier:= entier;
15     kt_booleen:= booleen;
16     caractere:= kt_caractere;
17     kt_reel:= reel;
18  END.

```

Fig. 5.172 Le programme *desassemble_constantes_typées*

Voici le code correspondant:

```

2D4D E849DD      call    TU_PARAGRAPHS
2D50 0600        hex     0600 ; options
2D52 780B        hex     780B ; cs
2D54 3C15        hex     3C15 ; ds
2D56 DC02        hex     DC02 ; code
2D58 2700        hex     2700 ; globales
2D5A 0004        hex     0004 ; min piles
2D5C 00A0        hex     00A0 ; max piles
2D5E 1000        hex     1000 ; fichiers
2D60 0000        hex     0000 ; input
2D62 0000        hex     0000 ; output

2D64 8BEC        mov     bp, sp
2D66 E814DF      call    TU_TAILLE_CODE
2D69 5300        hex     5300 ; taille
2D6B 6D2D        adr     SAUT_PRINCIPAL

2D6D E90A00      SAUT_PRINCIPAL jmp    PRINCIPAL

2D70 1111        adr     1111
2D72 01          hex     01
2D73 41          asc     'A'
2D74 810000000060 hex     810000000060

2D7A A16202      PRINCIPAL mov     ax, [$0262]
2D7D 2E          seg     cs
2D7E A3702D      mov     [$2D70], ax

2D81 A06A02      mov     al, [$026A]
2D84 32E4        xor     ah, ah
2D86 B90000      mov     cx, $0000
2D89 BA0100      mov     dx, $0001
2D8C E8E5E1      call    TU_RANGE_CHECK
2D8F 2E          seg     cs
2D90 A2722D      mov     [$2D72], al

```

```

2D93 2E          seg     cs
2D94 A0732D      mov     al, [$2D73]
2D97 32E4        xor     ah, ah
2D99 B90000      mov     cx, $0000
2D9C BAFF00      mov     dx, $00FF
2D9F E8D2E1      call    TU_RANGE_CHECK
2DA2 A26B02      mov     [$026B], al

2DA5 BF742D      mov     di, $2D74
2DA8 0E          push    cs
2DA9 57          push    di
2DAA BF6402      mov     di, $0264
2DAD 1E          push    ds
2DAE E885EB      call    TU_POUSSE_REEL
2DB1 E8A2EB      call    TU_POP_REEL_DI

2DB4 E90000      jmp     FIN

2DB7 33C0        xor     ax, ax
2DB9 E884DE      call    TU_HALT
2DBC 0000        hex     0000

```

Fig. 5.173 Traitement des constantes typées simples

Notez que:

- le code contient bien les valeurs des constantes telles que nous les avons initialisées
- le programme saute par dessus la zone de ces constantes, pour débiter à la première instruction 8088. Cette technique est aussi appliquée dans le cas des procédures
- l'accès aux constantes typées se fait de la façon suivante:
 - pour toutes les instructions qui chargeaient des valeurs simples dans AX au moyen de MOV, les traitements sont simplement préfixés par SEG CS
 - pour toutes les instructions qui poussaient le segment de la donnée sur la pile, c'est simplement CS qui est poussé (au lieu de DS)

5.8.3 LES CONSTANTES TYPEES COMPLEXES

Voici un exemple de constante typée plus complexe:

```

0  (* 001 const2 *)
1  (* 11 avr 87 *)
2
3  (*$r+*)
4  PROGRAM desassbmle constante_typee2;
5  TYPE t_record= RECORD
6      nom: STRING[$3];
7      age: INTEGER;
8  END;
9      t_tableau_record= ARRAY[$10..$11] OF t_record;
10  CONST kt_tableau_record: t_tableau_record=
11      (
12          (nom: 'ABC'; age: $1F),
13          (nom: 'DEF'; age: $2F));

```



```

14  VAR indice: INTEGER;
15  BEGIN
16      kt tableau_record[indice].age:= $FF;
17      WITH kt tableau_record[indice] DO
18          age:= $77;
19  END.

```

Fig. 5.174 Le programme *desassemble_constantetype2*.

Voici son code:

```

2D4D E849DD      call    TU_PARAGRAPHS
2D50 0600      hex     0600 ; options
2D52 780B      hex     780B ; cs
2D54 3C15      hex     3C15 ; ds
2D56 DD02      hex     DD02 ; code
2D58 2700      hex     2700 ; globales
2D5A 0004      hex     0004 ; min piles
2D5C 00A0      hex     00A0 ; max piles
2D5E 1000      hex     1000 ; fichiers
2D60 0000      hex     0000 ; input
2D62 0000      hex     0000 ; output

2D64 8BEC      mov     bp, sp
2D66 E814DF      call   TU_TAILLE_CODE
2D69 6000      hex     6000 ; taille
2D6B 6D2D      adr     SAUT_PRINCIPAL

2D6D E90C00      jmp     PRINCIPAL

2D70 03      hex     03
2D71 414243      asc     'ABC'
2D74 1F00      hex     1F00
2D76 03      hex     03
2D77 444546      asc     'DEF'
2D7A 2F00      hex     0000

2D7C A16202      PRINCIPAL mov    ax, [$0262]
2D7F 2D1000      sub     ax, $0010
2D82 B90200      mov     cx, $0002
2D85 E8E2E1      call   TU_INDEX_CHECK
2D88 D1E0      shl     ax, 1
2D8A 8BC8      mov     cx, ax
2D8C D1E0      shl     ax, 1
2D8E 03C1      add     ax, cx
2D90 97      xchg    di, $2D70
2D91 B8FF00      mov     ax, $00FF
2D94 2E      seg     cs
2D95 8985742D      mov     [di+ $2D74], ax

```

```

2D99 A16202      mov     ax, [$0262]
2D9C 2D1000      sub     ax, $0010
2D9F B90200      mov     cx, $0002
2DA2 E8C5E1      call   TU_INDEX_CHECK
2DA5 D1E0      shl     ax, 1
2DA7 8BC8      mov     cx, ax
2DA9 D1E0      shl     ax, 1
2DAB 03C1      add     ax, cx
2DAD 97      xchg    di, $2D70
2DAE 81C7702D      add     di, $2D70
2DB2 0E      push    cs
2DB3 57      push    di
2DB4 C47EFC      les     di, [bp+ $FC]
2DB7 B87700      mov     ax, $0077
2DBA 26      seg     es
2DBB 894504      mov     [di+ $04], ax
2DBE 83C404      add     sp, $04

2DC1 E90000      jmp     FIN

2DC4 33C0      FIN      xor     ax, ax
2DC6 E877DE      call   TU_HALT
2DC9 0000      hex     0000

```

Fig. 5.175 Constantes typées structurées.

Notez que:

- pour accéder à la zone, l'instruction MOV est simplement préfixée par CS
- dans le cas du WITH, c'est le segment CS qui est poussé sur la pile. Les autres instructions sont alors similaires à celles du WITH normal.

5.8.4 RESUME

Les constantes typées:

- sont des variables initialisées stockées dans le code du programme et accessible par CS
- pour traiter ces variables, le code:
 - préfixe les instructions de transfert par le segment CS
 - pousse CS sur la pile pour les instructions ou les routines de la librairie qui attendent le segment de la donnée sur la pile

5.9 LES VARIABLES ABSOLUES**5.9.1 DEFINITION DES VARIABLES ABSOLUES**

TURBO PASCAL permet au programmeur de spécifier à quel endroit est stockée une variable. Pour cela il suffit d'utiliser l'attribut ABSOLUTE. Par exemple:

```
VAR entier: INTEGER ABSOLUTE $1234:$5678;
```

Il existe 4 types de variables absolues:

- les variables absolues situées dans le segment CS:

```
VAR options: INTEGER ABSOLUTE CSEG: $2D50;
```

- les variables absolues situées dans le segment DS:

```
VAR handle1: INTEGER ABSOLUTE DSEG: $0242;
```

- les variables absolues situées à la même adresse qu'une autre donnée (variable globale, locale, paramètre). Par exemple:

```
VAR chaine: STRING[15];  
    taille: BYTE ABSOLUTE chaine;
```

- des variables absolues situées à une adresse quelconque:

```
VAR equipement: BYTE ABSOLUTE $0040: $0010;
```

5.9.2 ACCES AUX VARIABLES ABSOLUES

Voici comment le compilateur traite les variables absolues:

- les variables absolues par rapport à DS sont traitées comme n'importe quelle variable globale. Le compilateur utilise simplement le décalage que vous lui fournissez, au lieu de le calculer lui-même
- les variables absolues par rapport à CS sont traitées comme les constantes typées
- les variables absolues greffées sur une autre variable sont traitées comme les variables du segment où se trouve la variable (DS ou SS)
- les variables situées à une adresse quelconque sont traitées comme des constantes typées.

Examinons le dernier cas à l'aide d'un petit exemple:

```
0 (* 001 abso *)  
1 (* 18 aou 87 *)  
2  
3 PROGRAM desassemble_absolue;  
4   VAR absolue: INTEGER ABSOLUTE $1234: $5678;  
5     entier: INTEGER;  
6   BEGIN  
7     entier:= absolue;  
8   END.
```

Fig. 5.176 Le programme *desassemble_absolue*.

Voici le code correspondant:

```
2D4D E849DD      call    TU_PARAGRAPHS  
2D50 0600        hex     0600 ; options  
2D52 780B        hex     780B ; cs  
2D54 3C15        hex     3C15 ; ds  
2D56 D902        hex     D902 ; code  
2D58 2700        hex     2700 ; globales  
2D5A 0004        hex     0004 ; min piles  
2D5C 00A0        hex     00A0 ; max piles  
2D5E 1000        hex     1000 ; fichiers  
2D60 0000        hex     0000 ; input  
2D62 0000        hex     0000 ; output  
  
2D64 8BEC        mov     bp, sp  
2D66 E814DF      call    TU_TAILLE_CODE  
2D69 1E00        hex     1E00 ; taille  
2D6B 6D2D        adr     SAUT_PRINCIPAL  
  
2D6D E90400      SAUT_PRINCIPAL jmp    PRINCIPAL  
2D70 7856        hex     7856  
2D72 3412        hex     3412  
  
2D74 2E          PRINCIPAL      seg    cs  
2D75 C43E702D    les     di, [$2D70]  
2D79 26          seg     es  
2D7A 8B05        mov     ax, [di]  
2D7C A36202      mov     [$0262], ax  
  
2D7F E90000      jmp     FIN  
  
2D82 33C0        FIN            xor     ax, ax  
2D84 E8B9DE      call    TU_HALT  
2D87 0000        hex     0000
```

Fig. 5.177 Une variable absolue.

Notez que:

- l'utilisation du code pour stocker l'adresse de la variable facilite son accès en utilisant LES DI. Elle permet aussi de bénéficier de la mécanique des constantes typées, en évitant d'avoir à créer une nouvelle technique d'accès.

5.9.3 RESUME

Les variables absolues sont traitées:

- comme des variables globales (absolues DSEG ou greffées sur une globale)
- comme des constantes typées (absolues CSEG ou à une adresse absolue)
- comme des variables locales (absolues greffées sur des variables locales ou des paramètres)

5.A SYNTHÈSE

Le code TURBO PASCAL distingue 2 types de données:

- les données manipulables dans les registres
- les données dont la taille peut dépasser la capacité des registres disponibles.

Pour les données trop volumineuses, le code utilise systématiquement la pile

- pour stocker les valeurs à traiter
- pour placer le segment de ces données (ce segment pouvant être DS, CS, SS ou ES). DI est alors utilisé pour connaître le déplacement.

Les données traitées ainsi comprennent les réels, les chaînes, les tableaux et les enregistrements lorsqu'ils sont manipulés globalement et les ensembles.

Pour les données manipulables dans les registres:

- toute l'arithmétique est effectuée dans AX, en sauvant les valeurs intermédiaires sur la pile et en les récupérant dans CX
- les accès par le biais de sélecteurs (tableaux, enregistrements, pointeurs)
 - sont calculés directement chaque fois que le compilateur connaît les valeurs à utiliser pour le calcul
 - sont effectuées en utilisant DI pour calculer le décalage, plus un décalage relatif constant dépendant des indices de départ ou des zones sélectionnées. ES est alors utilisé comme segment.
 - dans le cas de WITH, l'adresse de base est poussée sur la pile et récupérée dans ES et DI à l'aide de BP

- l'accès aux variables locales et aux paramètres est effectué à l'aide de BP
- l'accès aux variables intermédiaires est effectué en utilisant la table des pointeurs des piles englobantes stockés dans chaque pile locale.

Dans le cas de l'affectation, le calcul de l'adresse peut être séparé des instructions de stockage. La sauvegarde sur la pile des adresses calculées est alors effectuée.

Le traitement des fichiers utilise systématiquement la librairie, qui fait appel au DOS.

5.B CONCLUSION

Dans ce chapitre nous avons examiné:

- par quelles instructions le code manipulait:
 - les données simples et structurées
 - globales, locales, sur le tas, dans le code
 - en mémoire ou vers un périphérique

LES OPTIONS DE COMPILEUR

4.1.1. DEFINITION

CHAPITRE 6

LES OPTIONS DE COMPILEUR

4.1.1. SYNTAXE

Ces options sont tapées dans le source, dans un commentaire :



ATTENTION

Il est à noter que l'option doit être précédée d'un '+' et suivie d'un '*'.
 La présence d'espaces entre le '+' et l'option et la fin de l'option et la fin du commentaire pour compiler le programme ne doit pas être oubliée.
 Dans ce chapitre nous allons :

- décrire les options de compilation
- indiquer leur effet sur le code généré

A l'issue de ce chapitre vous serez en mesure :

- d'éviter l'arrêt d'exécution lorsqu'une erreur d'entrée / sortie se produit
- d'utiliser des lectures clavier non tamponnées
- de modifier le nombre de fichiers ou de dimensionner les tampons d'entrée