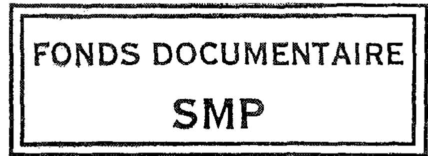
Réseaux
et systèmes
d'information

Bull

MANUEL DE RÉFÉRENCE


FONDS DOCUMENTAIRE
SMP

SOLAR

PL16

Langage de programmation système PL16

LOGICIEL

LOGICIEL

LOGICIEL

LOGICIEL

LOGICIEL

1 164 001 00 036

PL 16

MANUEL DE REFERENCE

A en haut de page indique le changement complet de la page par, rapport à l'IE précédent

I en marge indique la partie modifiée par rapport à l'IE précédent

JANVIER 1980

1 164 001 00 / -- 36 02

SOMMAIRE Pages

AVANT PROPOS 1

1 - PRESENTATION. DU LANGAGE PL16 3

1.1 - PROGRAMMATION SYSTEME 3
1.2 - LANGAGES EVOLUES 3
1.3 - LE LANGAGE PL16 4

2 - ELEMENTS DU LANGAGE 5

2.1 - FORMAT LIBRE 5
2.2 - LES CARACTERES UTILISES 5
2.3 - LES SYMBOLES DE BASE 5
2.4 - LES CONSTANTES 7
2.5 - LES IDENTIFICATEURS 7
2.6 - CONCLUSION 8

3 - INTRODUCTION A LA PROGRAMMATION EN PL16 9

3.1 - GENERALITES 9
3.2 - DECLARATIONS 11
3.3 - INSTRUCTIONS 13
3.4 - INSTRUCTIONS COMPOSEES 13
3.5 - PROCEDURE 16
3.6 - PROGRAMME PL16 17
3.7 - ALGORITHME ET PROGRAMMATION 23
3.8 - CONCLUSION 26

4 - DECLARATIONS ET ORGANISATION DES DONNEES 27

4.1 - PORTEE DES IDENTIFICATEURS 27
4.2 - SECTIONS DE DONNEES 28
4.2.1 - GENERALITES 28
4.2.2 - DECLARATION DE SECTION DE DONNEES 39
4.3 - DECLARATION DE CONSTANTES 33
4.4 - DECLARATION DE VARIABLES 35
4.4.1 - DECLARATION DE VARIABLE SIMPLE 35
4.4.2 - DECLARATION DE TABLEAU 36

4.4.3 - DECLARATION DE POINTEUR 36
4.4.4 - INITIALISATION 39
4.4.5 - SYNONYME 40
4.4.6 - ACCES AUX VARIABLES 43
4.5 - DECLARATION D'ECHANGE 45
4.6 - DECLARATION D'ETIQUETTE 47
4.7 - DECLARATION DE PROCEDURE 49
4.7.1 - MAIN PROCEDURE 49
4.7.2 - PROCEDURE 50
4.8 - DECLARATION DE KSTORE 53

5 - LES INSTRUCTIONS DU LANGAGE 54
5.1 - GENERALITES 54
5.2 - INSTRUCTIONS COMPOSEES ET STRUCTURE DE BLOC 57
5.2.1 - INSTRUCTIONS COMPOSEES 57
5.2.2 - STRUCTURE DE BLOC 57
5.2.3 - INSTRUCTION USING, CONTEXTE D'UN BLOC 58
5.3 - INSTRUCTIONS D'ASSIGNATION 82
5.3.1 - GENERALITES 62
5.3.2 - ASSIGNATION DES REGISTRES 66
5.3.3 - ASSIGNATION DE VARIABLES 71
5.3.4 - POSITIONNEMENT D'INDICATEURS, TRAITEMENT DU BIT 79
5.4 - INSTRUCTION CONDITIONNELLE : IF 80
5.4.1 - CONDITIONS SIMPLES 81
5.4.2 - CONDITIONS COMPOSEES AVEC OR ET AND 85
5.5 - INSTRUCTION DE CHOIX : CASE OF 86
5.6 - INSTRUCTIONS DE BRANCHEMENT 87
5.7 - APPEL DE PROCEDURE 91
5.8 - BRANCHEMENTS ET APPELS DE PROCEDURE DEFINIS DYNAMIQUEMENT 92
5.9 - INSTRUCTIONS DE BOUCLES 97
5.9.1 - CLAUSE VIDE 97
5.9.2 - CLAUSE WHILE 98
5.9.3 - CLAUSE TIMES 98
5.9.4 - CLAUSE FOR 99
5.9.5 - EXEMPLES 101
5.10 - INSTRUCTIONS INCR ET DECR 103
5.11 - INSTRUCTIONS SUR PILE GENERALE 104
5.11.1 - CHARGEMENT DU REGISTRE RK 104
5.11.2 - SAUVEGARDE ET RESTAURATION DE REGISTRE 105

5.12 - ENTREES-SORTIES ET FONCTIONS MONITEUR 106
5.12.1 - OPERATIONS D'ENTREES-SORTIES 106
5.12.2 - COMPTE RENDU D'ECHANGE 107
5.12.3 - MODIFICATION DES CARACTERISTIQUES D'ECHANGE 107
5.12.4 - FONCTIONS MONITEUR 108

6 - SEGMENTATION D'UN PROGRAMME PL16 109
6.1 - UNITE DE COMPILATION 109
6.2 - PROGRAMME PL16 110
6.2.1 - MAIN PROCEDURE 110
6.2.2 - SEGMENT PROCEDURE 110
6.3 - IDENTIFICATEUR COMMUNS A PLUSIEURS UNITES 112
6.4 - EDITION DE LIENS 114
6.5 - EXEMPLES 116
6.6 - CONCLUSION 120

7 - COMPLEMENTS POUR UNE PROGRAMMATION PLUS GENERALE 121
7.1 - OPERATIONS D'ENTREES-SORTIES : GENERALISATION 121
7.1.1 - DECLARATION D'IOCB 121
7.1.2 - UTILISATION 123
7.1.3 - COMPTE RENDU D'ECHANGE 125
7.1.4 - INSTRUCTIONS D'ENTREES-SORTIES SPECIALES 125
7.2 - SYMBOLES EXTERNES 127
7.2.1 - DEFINITION D'EXTERNE 127
7.2.2 - DECLARATION PAR REF 127
7.2.3 - DECLARATION PAR EXT 132
7.3 - GENERATIONS IMPLICITES 134
7.3.1 - DE DONNEES 134
7.3.2 - D'INSTRUCTIONS 134
7.4 - COMPLEMENTS SUR LES DECLARATIONS DE SECTIONS 135
7.4.1 - SECTIONS DUMMY 135
7.4.2 - SECTIONS SYNONYMES 136
7.4.3 - FORME GENERALE POUR UNE OUVERTURE DE SECTION 137
7.5 - INSTRUCTIONS MACHINE 140
7.5.1 - DECLARATION 140
7.5.2 - UTILISATION 142
7.5.3 - CODES INSTRUCTIONS SOLAR 16 144

8 - OPTION SCHEDULER ET INSTRUCTIONS DIVERSES 145

8.1 - DECLARATIONS DE TACHES 145

8.2 - MULTITRAITEMENT 147

8.2.1 - DECLARATION DE RESSOURCE 147

8.2.2 - INSTRUCTIONS PRIVILEGIEES 149

8.2.3 - DECLARATION DE RESSOURCES EXTERNES 152

8.2.4 - OPERATIONS D'ENTREES-SORTIES 154

8.3 - INSTRUCTION DE TRANSFERT MEMOIRE 155

8.4 - RECHERCHE D'UN OCTET DANS UNE ZONE 156

8.5 - INSTRUCTIONS SUR PILE UTILISATEUR 156

8.5.1 - DECLARATION DE PILE 156

8.5.2 - INSTRUCTIONS SUR PILE 157

9 - LE COMPILATEUR PL16 159

9.1 - DIRECTIVES AU COMPILATEUR 159

9.2 - LE COMPILATEUR PL16 162

9.2.1 - PRESENTATION 162

9.2.2 - LES SUPPORTS D'ENTREES-SORTIES 164

9.2.3 - PRODUCTION DE PROGRAMMES EXECUTABLES 165

9.2.4 - EDITION DE LIEN 166

9.2.5 - LE LISTING 168

9.2.6 - LES ERREURS 172

10 - AIDE A LA MISE AU POINT DES PROGRAMMES PL16 174

10.1 MISE AU POINT SOUS LE CONTROLE DE AID 174

10.1.1 - EXPLOITATION DU LISTING 174

10.1.2 - UTILISATION DE LA TABLE DES SYMBOLES 177

10.2 MODULE DRIP16 179

10.2.1 - DIRECTIVES AU COMPILATEUR 179

10.2.2 - INSTRUCTIONS 180

10.2.3 - COMMANDES AU MODULE DRIP16 181

10.3 TABLE DE CORRESPONDANCE DES REFÉRENCES 183

ANNEXE 183

AVANT PROPOS

Cette notice suppose connu le manuel de références SOLAR 16.

L'essentiel du langage se trouve présenté dans les chapitres 2, 3, 4, 5 du manuel dans le but de permettre une programmation simple.

Le reste du langage qui consiste en une généralisation de certains concepts de base est exposé dans les chapitres 6 et 7 et doit permettre une programmation plus évoluée, c'est-à-dire la résolution de cas particuliers de programmation ou d'organisation de données. Nous pensons que cette partie du langage n'est à utiliser qu'après une bonne pratique des principes essentiels de la programmation en PL 16

NOTATIONS UTILISEES

::=	indique une définition de syntaxe
[E]	les crochets signifient que la présence de E est facultative ex : [ARRAY n] WORD nom
E1 } E2 }	les accolades ou la barre verticale marquent un choix possible
E1/E2	ex : { WORD / BYTE } nom décalage ::= ≠ SLLS/SLLD/SCLS/.........
[{ E1 / E2 }]	Accolades et crochets peuvent se trouver combinés

1 - PRESENTATION DU LANGAGE PL16

PL16 est un langage permettant une programmation de haut niveau tout en donnant accès à toutes les possibilités des calculateurs SOLAR 16.

1.1 - PROGRAMMATION SYSTEME

Un programme système doit satisfaire à deux impératifs de performance :

- le temps
- l'occupation mémoire

Ceci implique, pour un tel programme, un accès possible à toutes les ressources du calculateur :

- utilisation de toutes les possibilités du code d'ordre en particulier des instructions spécialisées
- accès aux registres

Ces raisons expliquent le choix habituellement fait de l'assembleur pour écrire les programmes systèmes.

En conclusion, la programmation système ne bénéficie pas des progrès faits par les langages de programmation qui semblent réservés au domaine scientifique et à celui de la gestion.

1.2 - LANGAGES ÉVOLUES

Par rapport aux langages machine et assembleurs de tels langages ont apporté des notions indépendantes des calculateurs, mais seulement liées aux traitements qu'ils ont à exprimer.

Ce sont :

- des instructions évoluées :
  . boucles
  . branchements sur conditions complexes
  . calculs arithmétiques ou logiques
  . définition et appels de traitements partiels
  . Entrées - Sorties puissantes

. des définitions claires et complètes des types d'informations à traiter

. la possibilité d'organiser ces informations.

. la possibilité de donner à un programme une structure analogue à celle du traitement qu'il décrit grâce aux notions de traitements partiels, données locales...

L'apport de ces notions se traduit par :

- une programmation rapide
- une écriture des programmes dans un langage clair ce qui signifie :
  . facilité de lecture ou de relecture
  . corrections aisées
  . documentation claire des programmes

Mais l'indépendance de ces langages, vis à vis des calculateurs. suppose, par ailleurs, un sous emploi de toutes les possibilités offertes par ceux-ci.

1.3 - LE LANGAGE PL16

C'est un langage évolué de programmation système.

• Parce qu'il conserve les possibilités d'un assembleur :
- accès à tous les registres du calculateur
- accès à toutes les fonctions du code d'ordre
- accès à tous les niveaux d'informations (mot, octet, bit...)
- production d'un code machine optimisé

• Parce qu'il introduit cependant :
- un ensemble complet de type de données
- des instructions évoluées puissantes
- une structure de programme adaptée à la structure d'un traitement
- une organisation des données très souple et en accord avec l'organisation des traitements, ce qui permet notamment :
  . l'accès à des variables déclarées communes à plusieurs traitements
  . la définition de variables d'intérêt local pour un traitement
  . la définition de zones de travail pouvant être indépendantes et même se recouvrir d'une procédure de traitement à une autre.

Enfin parce qu'il atténue les risques d'erreurs en fournissant, à la compilation, un contrôle serré de la cohérence du programme.

Le PL16 permet une programmation utilisant ou non les registres suivant que l'on s'attache davantage aux performances du programme objet ou à l'indépendance vis à vis du calculateur.

Tout au long de ce manuel nous ferons la distinction entre une programmation avec ou une programmation sans registre.

2 - ÉLÉMENTS DU LANGAGE

2.1 - FORMAT LIBRE

Le langage PL16 est un langage à format libre. Un programme peut être écrit avec autant de mots que l'on souhaite par enregistrement physique (support du code source), la seule contrainte du langage est qu'aucun mot ne doit être coupé sur deux enregistrements physiques,

Compilation optionnelle.

par ailleurs, le premier caractère de la ligne est un caractère de contrôle pour une compilation optionnelle (cf . chap . 10) les caracteres valides par défaut étant : ␣ (espace) et !

Les commentaires, qui facilitent la documentation des programmes, peuvent apparaître n'importe où dans un enregistrement ; ils débutent par le symbole « et sont terminés par la fin de l'enregistrement.

2.2 - LES CARACTERES UTILISES

Les phrases d'un programme symbolique sont écrites en utilisant des caractères du code ASCII :

- les 26 lettres de l'alphabet :

  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- les chiffres :

  0 1 2 3 4 5 6 7 8 9

- des caractères spéciaux :

  (;) , (,) , (:) , (RC) ....

leur liste apparaîtra complètement dans le paragraphe suivant.

Les autres codes ne sont pas reconnus.

Chaque enregistrement se termine par le caractère retour chariot (RC) et ne peut comporter plus de 72 caractères. Dans le cas d'un code source lu sur des cartes, un enregistrement est une carte et les colonnes 73 à 80 sont ignorés

2.3 - LES SYMBOLES DE BASE

Certains caractères spéciaux ont un rôle dans le langage soit en eux mêmes, soit en étant combinés pour former d'autres symboles :

retour chariot	(RC)	**séparateurs**	
espace	( )		
virgule	(,)		
point virgule	(;)		
point	(.)		
deux points	(:)		
dollar	($)	**spécifieurs**	**de nombre binaire**
apostrophe	(')		**de nombre hexadécimal**
double apostrophe	(")		**de chaîne**
et	(&)		**d'indirection**

alpha	@	Spécifieurs	d'adresse
point d'exclamation	!		de directive au compilateur
guillemets ouverts	«		de commentaire (1)
parenthèse gauche	(	parenthèses	
parenthèse droite	)		
plus	+		opérateurs arithmétiques
moins	-		
multiplier	*		
diviser	/		
égal	=		opérateurs de relation
différent	/=		
supérieur	>		
supérieur ou égal	>=		
inférieur	<		
inférieur ou égal	<=		
deux points égal	:=		opérateurs d'assignation
échange	:=:		

De même, certains mots sont réservés à un emploi spécial dans le langage, et ne peuvent donc être utilisés dans un autre sens. Ils peuvent désigner des ordres, la nature d'une donnée etc...

Exemples :

WORD
BYTE
SECTION
CONTINUE
PROCÉDURE
ARRAY
POINTER
GOTO
END
. . . .

leur liste complète figure en annexe.

La signification et l'utilisation de tous ces symboles apparaîtront progressivement au cours de la lecture de ce manuel.

(1) Toute combinaison des caractères spéciaux <, >, /, : ne formant pas elle-même un symbole de base (par exemple : : < ou encore > < est équivalente au symbole commentaire ≪.

Remarques :

- les opérateurs sont aussi des séparateurs
- un séparateur peut toujours être accompagné d'espaces
- la suite de caractères qui constitue un commentaire peut contenir n'importe quel caractère vu précédemment.

2.4 - LES CONSTANTES

Elles représentent les valeurs sur lesquelles travaille un programme ; elles sont de différents types :

- les nombres entiers décimaux : ils sont composés d'une suite de chiffres
  Un nombre entier décimal court appartient à l'intervalle ± 32767
  Un nombre entier décimal long appartient à l'intervalle ± 2 147 483 647

  Exemples : 1, 4, 26567, 32765, 200000

- les nombres entiers hexadécimaux : ils sont composés d'une suite de caractères numériques et alphabétiques (A à F uniquement) et précédés du symbole (') . Un nombre hexadécimal représente une suite de bits. Chaque chiffre hexadécimal tient la place de 4 positions binaires.

  Exemple : '12F0, 'FFFF, '7F, '30, 'ABCD

  les nombres entiers binaires : ils sont composés d'une suite de chiffres 0 et 1 précédés du symbole ($)

  Exemple : $010010, $0111

- les nombres flottants : ils sont représentés par une suite de chiffres contenant un point décimal (.) autre que le premier caractère ou/et se terminant par un exposant composé de la lettre E immédiatement suivie d'un entier signé (+ ou -) ou non
  Un nombre flottant appartient aux intervalles ± $[0{,}15 10^{-38}, 1{,}7 10^{+38}]$

Exemples :	124.	nombre	124,0
	12E4	nombre	$12{,}0 \times 10^{4}$
	1.12E-5	nombre	$1{,}12 \times 10^{-5}$

- les chaînes de caractères : ces constantes sont composées d'une suite de caractères quelconque comprise entre doubles apostrophes (") . Si le caractère (") doit faire partie de la chaîne il est représenté par une double occurence. Seules ces constantes peuvent contenir des blancs.

  Exemples :

"A B C D E F G"	chaîne A B C D E F G
"A B - - F G"	chaîne A B F G
"+ - * / & @ "	chaîne + - * / & @
" AB " " CD " " "	chaîne A B " CD "

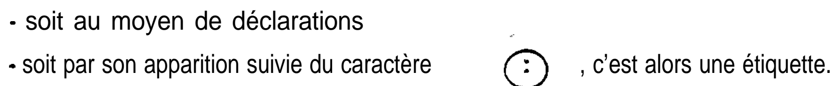
2.5 - LES IDENTIFICATEURS

Ils permettent de donner un nom aux différents éléments déclarés et utilisés par le programmeur, ces éléments pouvant être de toute nature : mémoires, registres, traitements, instructions...

Un identificateur est une suite de lettres et de chiffres commençant toujours par une lettre et d'un maximum de 15 caractères sauf pour les identificateurs d'externes, de sections de données d'unité de compilation limitées à 6 caractères. Les identificateurs doivent être différents des mots réservés.

Un identificateur est défini :

- soit au moyen de déclarations


- soit par son apparition suivie du caractère (:), c'est alors une étiquette.

Exemples :

TOTO, A124, JOJO
ETIQ, IDENTIFICATEUR 1
IDENTIFICATEUR 2

2.6 - CONCLUSION

Les phrases qui constituent un programme PL16 sont construites à l'aide des entités syntaxiques suivantes :

- symboles de base
- constantes
- identificateurs
- commentaires

Dans les définitions syntaxiques, on désignera par "Litteral",

un nombre entier court décimal,
hexadécimal,
ou binaire

3 - INTRODUCTION A LA PROGRAMMATION EN PL16

3.1 - GÉNERALITES

Un programme est constitué d'un ensemble structuré de déclarations et d'instructions :

- les déclarations définissent les éléments sur lesquels le programme va travailler et les associent à un identificateur.

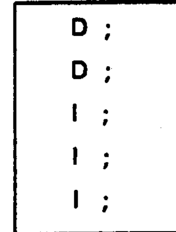
  Elles sont séparées par des point-virgules (;)

- les instructions définissent les opérations à effectuer sur ces éléments pour réaliser un traitement. Ce peut être des instructions de calcul, de contrôle, de test...

  Elles sont séparées par des point-virgules (;)

. un bloc est constitué d'une suite de déclarations et d'instructions, la partie déclarations étant facultative. Il permet de définir un traitement. Un bloc est repéré par des délimiteurs de début et de fin de bloc par exemple : BEGIN, END

On peut le schématiser ainsi :



D signifiant déclaration et I l'instruction.

- une procédure est la déclaration d'un traitement. Elle est définie par :

  . un en-tête qui définit les caractéristiques du traitement (son nom, ses conditions d'exécution)

  . un bloc qui définit le traitement

  . un délimiteur : fin de déclaration

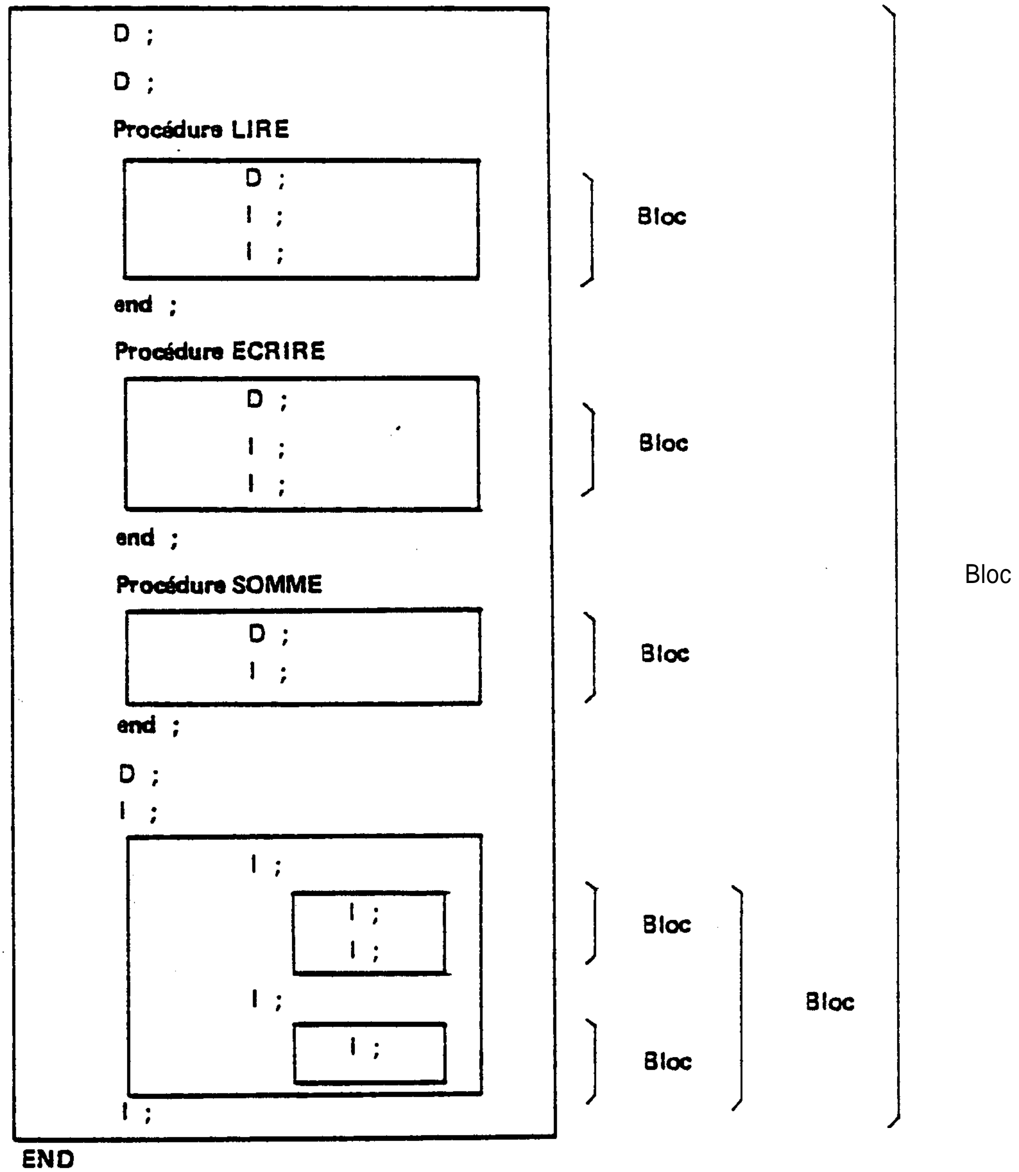
Le traitement déclaré peut être :

- global : la procédure est alors appelée "MAIN" procédure ; elle constitue le programme principal qui doit être lancé
- partiel : il peut être appelé alors en divers points du traitement global ou d'un autre traitement partiel

Un programme PL16 est la description d'un traitement global. Il consista en l'exécution contrôlée par le programme principal ou "MAIN" procédure, de traitements partiels déclarés ou non. Il sera donc constitué de blocs imbriqués définissant les traitements à exécuter dans un certain ordre.

Exemples :

MAIN PROCEDURE CALCUL



Dans cet exemple, la déclaration d'une "MAIN" procédure définit le traitement global. Le bloc qui le constitue contient six traitements partiels imbriqués. Les procédures LIRE, ECRIRE et SOMME sont trois traitements partiels déclarés susceptibles d'être appelés plusieurs fois par le programme principal et en des endroits différents. Les 3 autres traitements partiels ne peuvent être appelés mais sont exécutés dans l'ordre où ils sont écrits.

3.2 - DECLARATIONS

Elles servent à préciser certaines propriétés des identificateurs utilisés dans un programme :

1) - leur nature : variable (simple, tableau ou pointeur), constantes, étiquettes, procédures, échanges d'entrées sorties

2) - leur type : octet, mot, double mot, entier, réel

Elles permettent ainsi au compilateur de savoir quelle traduction machine adopter lors de l'utilisation de ces identificateurs.dans le programme et comment optimiser cette traduction.

Tous les identificateurs d'un programme, excepté certaines étiquettes, doivent être déclarés avant usage.

La notion de déclaration est très liée à celle de bloc pour 2 raisons :

- elles ont lieu en tête de bloc
- la portée d'un identificateur, c'est-à-dire la région du programme où il est utilisable est le bloc où il a été déclaré.

Exemples :

- WORD A ; — l'identificateur A symbolise une variable de type mot
- WORD DIX = (10) ; — Dix symbolise une variable de type mot initialisée à 10.

Dans une instruction

A désigne la valeur de la variable A
@ A désigne l'adresse de la variable A

- ARRAY 2 BYTE CHIFFRES; — CHIFFRES symbolise une variable tableau constituée de 2 octets
- ARRAY 8 BYTE SORTIE = ('8D, '0A, "S = 18", '8D, '0A) ;

SORTIE est un tableau de 8 octets initialisé avec les caractères ASCII suivants :

RC LF , S = 18 RC LF

L'accès à un élément d'un tableau se fait en donnant le nom du tableau suivi d'un indice :

CHIFFRES (0) — désigne le premier octet du tableau CHIFFRES

- POINTER WORD PTSUM =(@ A)

PTSUM est une variable pointeur, elle permet d'avoir accès indirectement au contenu de la variable "pointée"

PTSUM : accès à la variable PTSUM

& PTSUM : accès à la variable A

. LPFILE NOMBRES (MODE : INPUT, EM ; EU : SI ; DATA : CHIFFRES ; EOE : 2 ; CONTROL);

- INPUT → entrée
- EM → retour en fin d'échange
- SI → sur l'unité "symbolic input"
- CHIFFRES → buffer où sont stockés les caractères entrées
- 2 → il y a 2 caractères à échanger
- CONTROL → demande de compte rendu d'échange

. LPFILE RESULTAT (MODE : OUTPUT, EM ; EU : SO ; DATA : SORTIE ; EOE : 8 ; CONTROL);

- OUTPUT → sortie
- SO → sur "symbolic output"
- SORTIE → buffer où sont stockés les caractères à sortir
- 8 → il y a 8 caractères à sortir

Ces deux déclarations constituent des déclarations d'échange
NOMBRES est un échange d'entrée
RESULTAT est un échange de sortie

L'utilisation des échanges ainsi déclarés est faite par les instructions d'entrée-sortie.

```
. MAIN PROCEDURE CALCUL
    D ;
    I ;
    I ;
. END ;
```

L'identificateur CALCUL symbolise un traitement global

L'accès à ce traitement a lieu lors du lancement du programme car c'est le programme principal.

```
. PROCEDURE SOMME (X, Y, Z)
    D ;
    I ;
. END ;
```

L'identificateur SOMME représente un traitement partiel
X, Y, Z sont les paramètres de la procédure, ils caractérisent les conditions d'exécution de ce traitement.

L'accès à ce traitement est réalisé par l'instruction : appel de procédure

3.3 - INSTRUCTIONS

Les instructions d'un traitement peuvent être de différents types suivant les opérations qu'elles effectuent.

Exemples :

- instructions d'affectation

. A : = 10 ;	La variable A reçoit la valeur 10
. A : = CHIFFRES (0) ;	La variable A reçoit le 1er octet du tableau CHIFFRES
. A : = CHIFFRES (0) AND 'F ;	La valeur de A sera égale aux quatre bits de droite de l'octet CHIFFRES
. RA : = 0;	Le registre A est mis à zéro
. RB : = A;	Le registre B reçoit la valeur de la variable désignée par l'identificateur A
. RAB : = RAB/A;	L'accumulateur étendu est divisé par la valeur de A.
. &PTSUM : = RA ;	La variable pointée par PTSUM reçoit la valeur du registre RA.

Dans ce type d'instruction le symbole : = signifie que la valeur calculée au moyen de l'expression écrite à droite est affectée à la variable écrite à gauche.

- Instructions de contrôle :

Ces instructions, comme leur nom l'indique, permettent de contrôler l'ordre d'exécution d'un programme. Elles interrompent l'ordre normal d'exécution des instructions en permettant de se renvoyer à une autre séquence d'instructions.

Exemple :

. CALL SOMME (A, B, @ S) ;	Cette instruction est l'appel de la procédure SOMME avec les paramètres : valeur de A, valeur de B, adresse de S

- Instructions d'entrées-sorties :

Elles définissent des échanges entre le programme et l'environnement extérieur à la machine et permettent la communication d'informations entre les deux.

Exemple : (voir page 12 les déclarations d'échange correspondantes)

. READ NOMBRES ;	Cette instruction est une opération de lecture dont les caractéristiques sont décrites dans la déclaration de l'échange NOMBRES. Elle lit 2 caractères ASCII et les range dans CHIFFRES.
. WRITE RESULTAT ;	Cette instruction réalise l'écriture de 8 caractères ASCII pris dans le buffer SORTIE.

3.4 - INSTRUCTIONS COMPOSÉES

Les instructions que nous avons vues dans le paragraphe précédent sont des instructions simples car elles ne réalisent qu'une seule opération : une assignation, un branchement, une entrée, une sortie...

Une instruction composée réalise plusieurs opérations portant éventuellement sur des informations locales à cette instruction ; ces opérations définissent des sous traitements. Une instruction composée est donc constituée de :

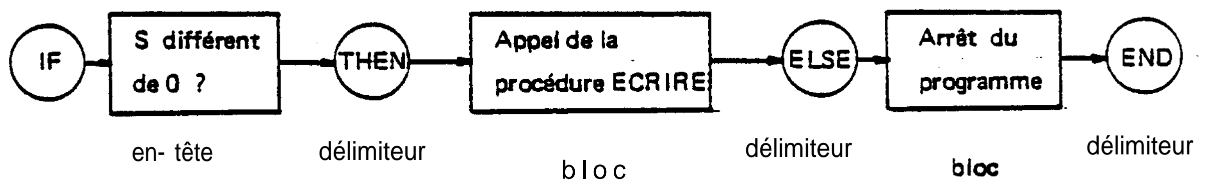
- un «en-tête»
- de délimiteurs
- de blocs repérés par ces délimiteurs

Exemples :

- instruction conditionnelle

```
IF (S/= 0)      THEN      CALL    ECRIRE (S) ;
                ELSE      STOP ;               END ;
```

Cette instruction permet le déroulement du programme sur l'instruction CALL ECRIRE (S) ou sur l'instruction STOP suivant le résultat du test de S.



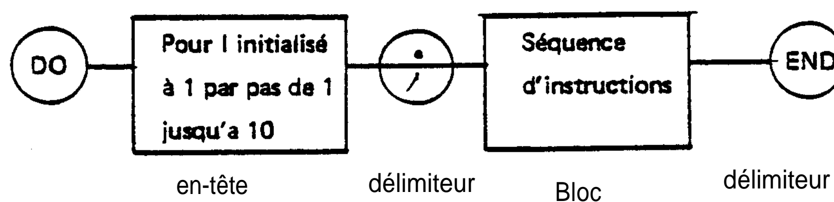
- instruction de boucle

```
DO      FOR I := 1     STEP + 1      UNTIL 10 ;
        CALL LIRE (@ A, @ B) ;
        CALL SOMME (A, B, @ S) ;
END ;
```

Cette instruction permet d'exécuter 10 fois la séquence d'instructions simples suivante :

CALL LIRE (@ A, @ B) appel procédure LIRE

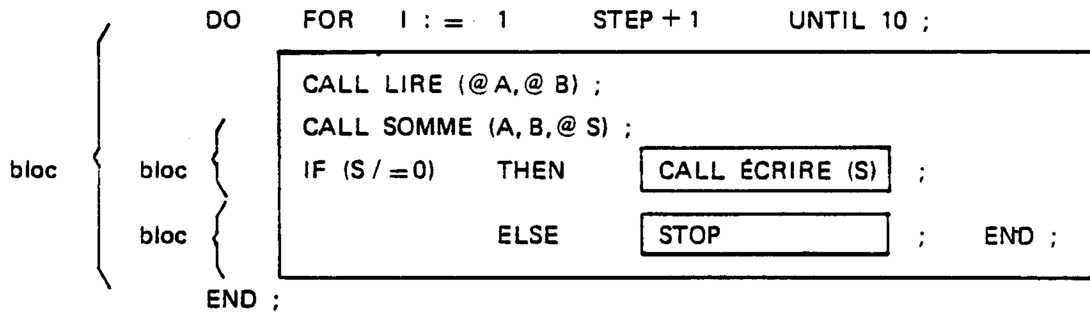
CALL SOMME (A, B, @ S) appel procédure SOMME



Une instruction composée est à nouveau considérée comme une instruction et peut donc apparaître n'importe où dans une séquence d'instructions appartenant à un traitement. Elle peut donc devenir composante d'une instruction composée plus vaste.

Exemple :

L'instruction conditionnelle précédente peut devenir une composante de l'instruction de boucle.



Le traitement définit dans cet exemple consiste en l'exécution de la séquence

CALL LIRE (@ A, @ 8) ;
CALL SOMME (A, B, @ S) ;
CALL ECRIRE (S) ;

dix fois à moins que S ne soit nul, dans ce cas le traitement est arrête.

Trois blocs imbriqués définissent le traitement ; les 2 blocs intérieurs sont 2 traitements partiels pour le bloc défini par l'instruction DO. On peut noter que ces blocs ne contiennent pas de déclaration.

3.5 - PROCEDURE

Lorsqu'un traitement doit être utilisé en des endroits différents d'un programme on a intérêt à l'écrire une fois pour toutes, c'est-a-dire à le déclarer, de façon à pouvoir s'en servir sans le réécrire chaque fois. Le traitement déclaré, ou procédure, est l'équivalent des sous-programmes dans tout langage de programmation.

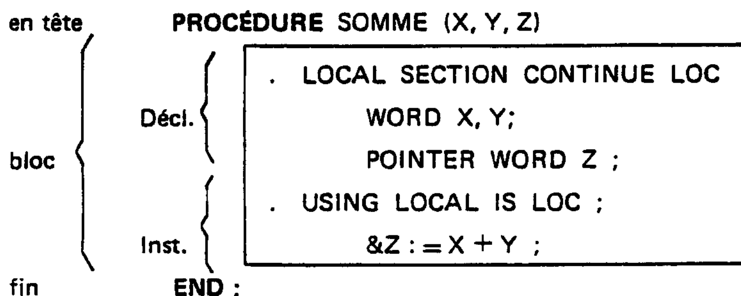
Remarque :

Une déclaration de procédure ne peut apparaître, que dans la partie "de déclaration" d'un bloc procédure.

Exemple :

Considérons l'instruction : Z : = X + Y ;

et supposons que l'on veuille l'utiliser en divers endroits d'un programme sans la réécrire. Il suffit de décrire ce traitement dans une déclaration qui permet de lui donner un nom et précise pour son utilisation future la place des paramètres sur lesquels doivent porter les opérations.



Nous retrouvons les 3 notions liées à une procédure :

a) L'en-tête définissant que :

- le nom de la procédure est l'identificateur SOMME
- la procédure travaille sur les valeurs de X, Y et range un résultat à l'adresse contenue dans un pointeur, Z

b) Le bloc définissant le traitement. comprend :

- des déclarations
  . WORD X, Y et POINTER WORD Z
    X, Y sont des identificateurs de variables mots
    Z est l'identificateur d'une variable adresse pointant sur un mot.
  . LOCAL SECTION CONTINUE LOC est une déclaration d'ouverture de section. En effet toute déclaration apparaissant dans un bloc doit se faire dans une section de données. Les variables X, Y, Z sont réservées dans une section appelée LOC accessible par la base L du calculateur (LOCAL SECTION).

- des instructions
  . &Z : = X + Y ;
    le résultat du calcul de l'expression X + Y est affecté par indirection à Z
  . USING LOCAL IS LOC ;
    Cette instruction spécifie au compilateur que la section LOC est accessible car la base L du calculateur est chargée correctement. Les instructions suivantes pourront donc avoir accès aux variables X, Y, Z rangées dans cette section.

c) Le délimiteur END

qui spécifie la fin de la déclaration de procédure, et qui provoque la génération d'une instruction de retour vers le point appelant, lors de l'exécution de cette procédure.

L'accès au traitement se fera par l'instruction :

- CALL SOMME (A, B, @ S) ;

A, B, étant des identificateurs de variables mots. Cet appel entraînera l'exécution du traitement suivant :

. addition des quantités apparaissant dans les 2 premières positions (A, B) et affectation indirect du résultat à l'identificateur S soit : &S : = A + B

- X, Y, Z sont les paramètres formels qui servent à définir la procédure
- A, B, S sont les paramètres effectifs sur lesquels on veut faire agir la procédure.

3.6 - PROGRAMME PL16

Un programme a pour but de décrire le traitement global d'un problème. Il comprendra :

- les déclarations des éléments nécessaires à l'exécution de ce traitement : variables, traitements partiels...
- les instructions utilisant les entités déclarées et réalisant le traitement :
  . calcul sur les variables, échanges, contrôles, appels de traitements partiels...

qui formeront le bloc d'une "MAIN" procédure ou programme principal.

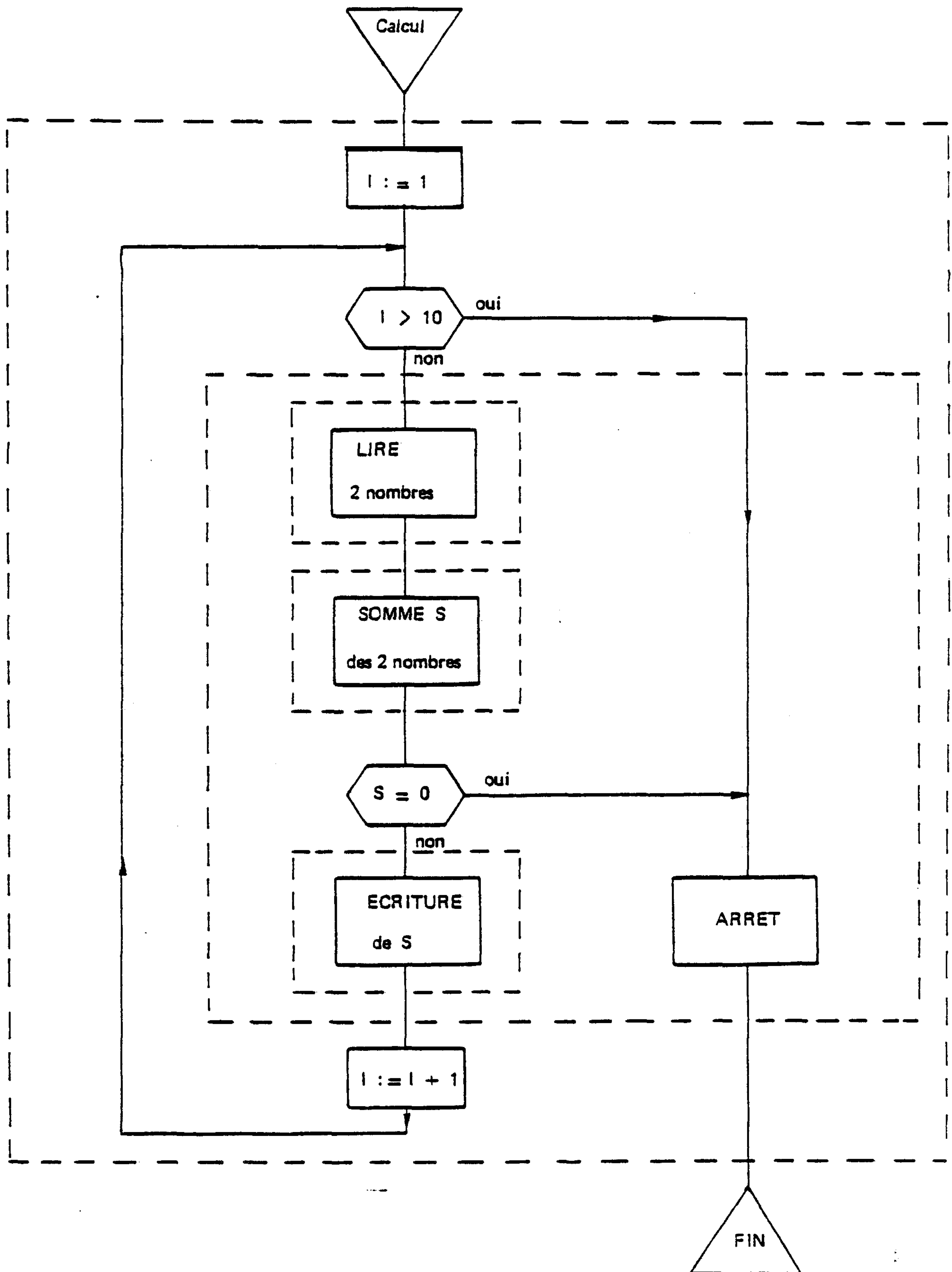
Exemple :

On veut écrire un programme qui réalise le traitement suivant :

- lecture de 2 nombres décimaux compris entre 0 et 9
- calcul de leur somme
- écriture du résultat s'il diffère de 0
- répétition du traitement précédent 10 fois à moins que la somme ne soit nulle, auquel cas le traitement est arrêté.

L'analyse de ce problème conduit à dégager :
- un traitement global qui est la répétition de l'enchaînement suivant :
  . lecture
  . calcul
  . écriture

Le Programme qui décrit ce traitement peut être décrit ainsi :



MAIN PROCÉDURE CALCUL

```
        « LE TRAITEMENT GLOBAL EFFECTUE PAR CETTE PROCEDURE
        « EST LA RÉPÉTITION DES TRAITEMENTS PARTIELS SUIVANTS :
        «           LECTURE DE 2 NOMBRES COMPRIS ENTRE 0 ET 9
        «           CALCUL DE LEUR SOMME S
        «           ECRITURE DU RESULTAT S'IL DIFFERE DE 0
        « LE TRAITEMENT S'ARRETE SI S = 0 OU SI S A ETE CALCULE 10 FOIS
  . KSTORE SECTION PILE                « PILE GENERALE POINTEE PAR K
        RES 50 ;
  . LOCAL SECTION LOC
        WORD A, B ;                    « VALEUR A LIRE
        WORD S ;                       « RESULTAT
        WORD I ;                       « MEMOIRE DE TRAVAIL
PROCEDURE LIRE (X, Y)
  . LOCAL SECTION CONTINUE LOC
        POINTER WORD X, Y ;            « PARAMETRES
        ARRAY 2 BYTE CHIFFRES ;        « 2 CARACTERES ASCII
        LPFILE NOMBRES = (MODE : INPUT, EM ; EU : SI ; DATA : CHIFFRES ; E0E : 2 ; CONTROL) ;
  . USING LOCAL IS LOC ;
        READ NOMBRES ;
        &X : = CHIFFRES (0) AND 'F ;   « X, Y = VALEURS DONNEES
        &Y : = CHIFFRES (1) AND 'F ;   « PAR LES CARACTERES ASCII
END ;
PROCEDURE ECRIRE (RESUL)
  . LOCAL SECTION CONTINUE LOC
        WORD RESUL ;                   « PARAMETRE
        WORD DIX = (10) ;              « POUR DIVISION PAR 10
        ARRAY 8 BYTE SORTIE = ('8D, '0A, "S = ␣␣", '8D, '0A) ;
        POINTER WORD PTSUM = (@ SORTIE + 2 AND '7FFF) ;
        LPFILE RESULTAT = (MODE : OUTPUT, EM ; EU : SO ; DATA : SORTIE ; E0E : 8 ; CONTROL) ;
  . USING LOCAL IS LOC ;
        RA : = 0 ; RB : = RESUL ; RAB : = RAP/DIX ;    « TRANSFORMATION
        RA : = RA OR'30 SLLS 8 OR RB OR'30 ;           « DU RESULTAT EN
        & PTSUM : = RA ;                               « CARACTERES ASCII
        WRITE RESULTAT ;
END ;
```

```
PROCEDURE SOMME (X, Y, Z)
     . LOCAL SECTION CONTINUE LOC
               WORD X, Y ;                          « PARAMETRES
               POINTER WORD Z ;
     . USING LOCAL IS LOC ;
               & Z : = X + Y ;                      « CALCUL DE LA SOMME
END ;
     . USING LOCAL = LOC, KSTORE = PILE ;                 « PROGRAMME PRINCIPAL
DO FOR I : = 1 STEP + 1 UNTIL 10 ;
               CALL LIRE (@ A, @ B) ;
               CALL SOMME (A, B, @ S) ;
               IF (S / = 0) THEN CALL ECRIRE (S) ; ELSE STOP ; END ;
END ;
END.
```

- la procédure LIRE lit 2 caractères ASCII
  . les range dans le tableau d'octets CHIFFRES
  . les transforme en 2 nombres binaires compris entre 0 et 9 et les range indirectement dans A et B.

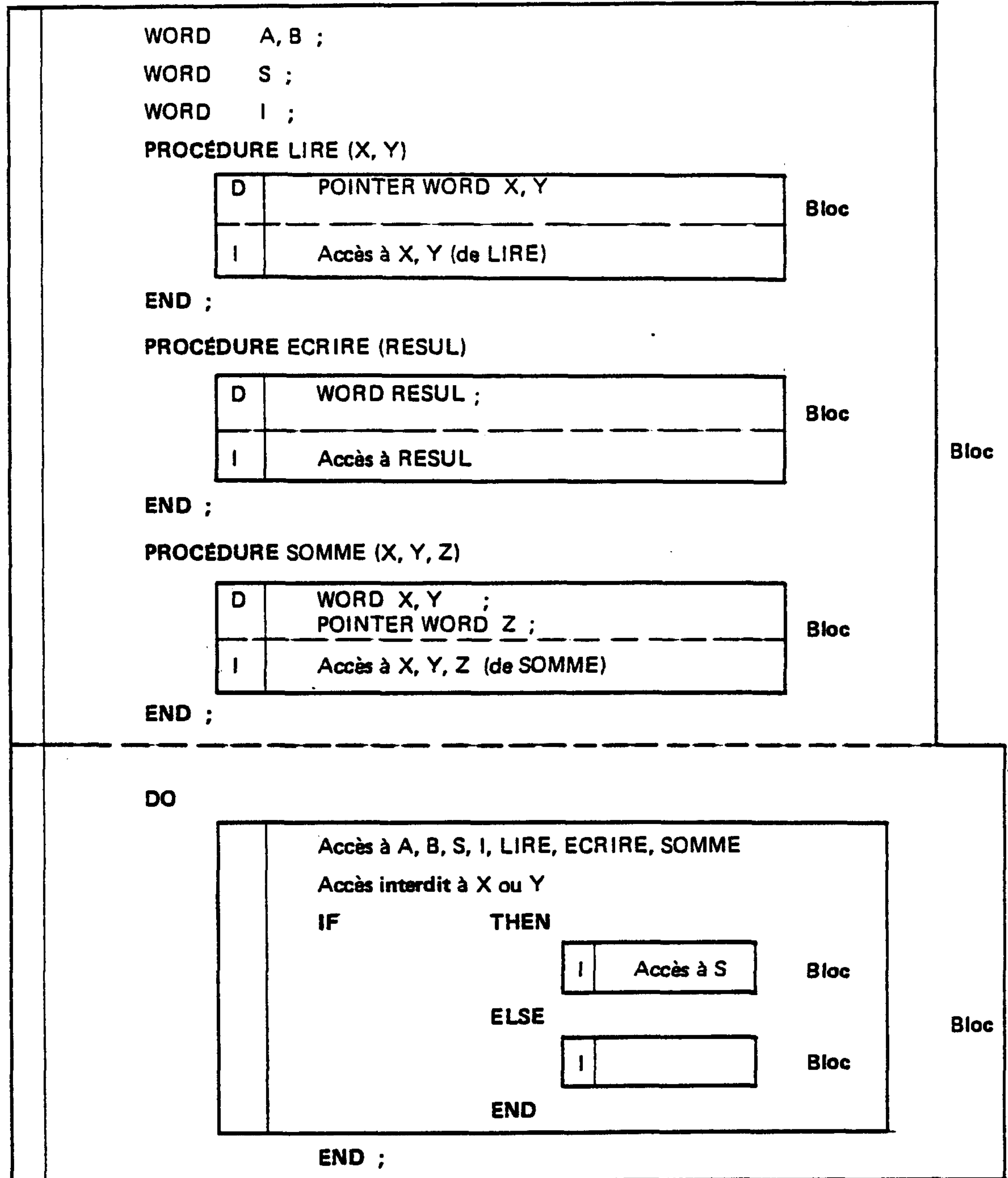
- la procédure SOMME calcule la somme des 2 nombres passés en paramètres
  . et range le résultat dans S.

- la procédure ECRIRE écrit un nombre décimal compris entre 1 et 18.
  . pour cela elle le divise par 10 pour récupérer le chiffre des dizaines dans le registre RA et celui des unités dans RB. Elle transforme ensuite ces 2 chiffres en 2 caractères ASCII qu'elle range dans le buffer de SORTIE.

- la boucle DO est l'instruction composée qui permet la répétition du traitement
- STOP est l'instruction par laquelle on arrête le traitement
- les déclarations existantes dans chaque bloc commencent par une déclaration d'ouverture de SECTION :
  . LOCAL SECTION LOC ouvre la section LOC dans laquelle se trouveront les variables A, B, S, I.
  . LOCAL SECTION CONTINUE LOC réouvre la section LOC pour la continuer par les déclarations qui suivent
  . KSTORE SECTION PILE est la déclaration de la pile gérée par le registre Kd SOLAR. La longueur réservée pour cette pile est ici de 50 mots.

- SCHEMA DE PRINCIPE DE LA STRUCTURE DU PROGRAMME PRECEDENT

MAIN PROCEDURE CALCUL



END.

D signifie déclarations, I instructions

- les instructions USING

. USING LOCAL = LOC, KSTORE = PILE réalise dans le programme principal :

- d'une part le chargement de la base L pour qu'elle pointe la section LOC afin que les variables appartenant à cette section soient accessibles par les instructions qui suivent.
- d'autre part le chargement du registre K par l'adresse de la pile.

Celui-ci doit en effet être chargé avant le premier appel de procédure.

. USING LOCAL IS LOC ; indique au compilateur que la base L est chargée correctement et donc que la section de donnée concernée est réellement accessible. Cette instruction suffit dans les 3 procédures de notre programme car; au moment de leur appel, la base L est déjà chargée.

- structure de bloc du programme

Les procédures LIRE, ECRIRE, SOMME et les instructions composées DO et IF définissent une structure de blocs arborescente pour le programme. L'espace des noms suit cette arborescence c'est-à-dire que chaque identificateur déclaré dans un bloc n'est connu que de l'intérieur de ce bloc. Un identificateur ne peut être déclaré deux fois dans un même bloc, mais peut être une fois dans un bloc et dans les sous blocs de ce dernier.

- accès aux identificateurs

Cet exemple nous a permis d'introduire 2 notions importantes à ne pas confondre, celle de bloc et celle de section :

. la première doit son existence à la nécessité pour l'utilisateur de structurer ses programmes de façon modulaire suivant les différents traitements qu'il doit effectuer pour traiter un problème global.

. la deuxième doit son existence au mode d'adressage SOLAR16 : adressage basé. L'exemple n'utilise ici que la base L mais il aurait pu utiliser aussi les bases C et W.

Pour écrire notre programme de façon modulaire nous l'avons organisé en sous-ensembles propres à chaque traitement, chaque sous-ensemble étant caractérisé par ses instructions et les données auxquelles elles ont accès C'est donc au niveau du traitement que les notions de blocs et sections se rejoignent car on ouvre une section en tête de bloc pour déclarer les données de ce bloc et par une instruction spéciale on la déclare accessible avant les instructions propres au bloc.

En résumé, une variable, pour être accessible à un instant donné, doit :

- être connue dans le bloc où on veut l'utiliser
- appartenir à une section accessible à cet instant.

Ces notions sont reprises et expliquées au chapitre 4.

3.7 - ALGORITHME ET PROGRAMMATION

Le langage PL16 par sa structure de blocs, permet et impose la programmation d'un traitement en allant du global au particulier, c'est-à-dire en conservant la même démarche qu'à l'analyse de ce traitement et à la rédaction de l'organigramme qui en résulte.

Exemple :

Soit le traitement suivant, où les Pi sont des procédures

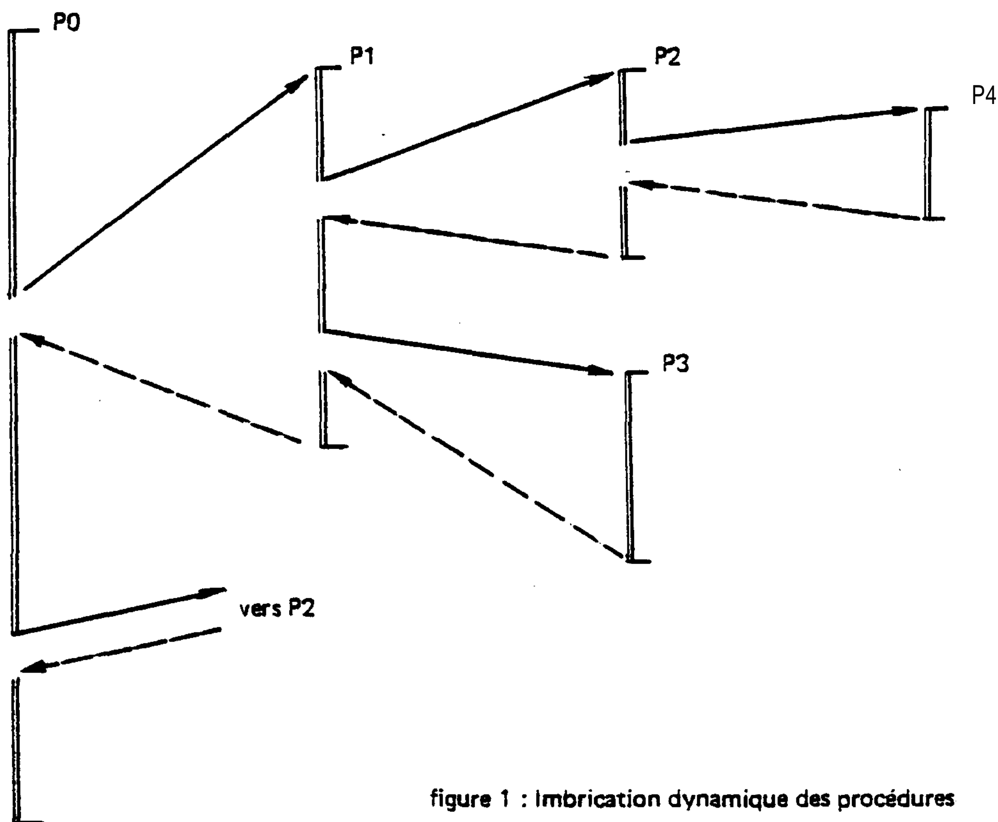


figure 1 : Imbrication dynamique des procédures

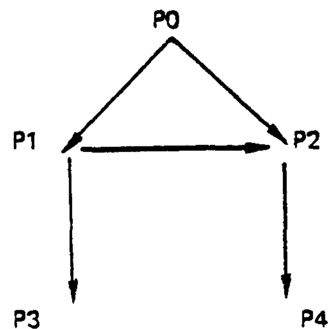
Les flèches schématisent les différents appels de procédure qui déterminent l'ordre de déroulement du programme.

Il faut traduire l'organigramme du traitement en partant chaque fois du plus haut niveau.

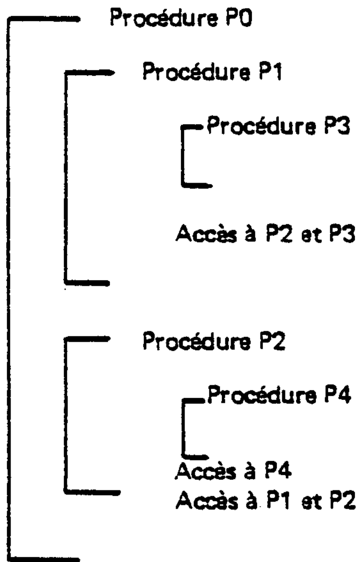
P0 appelle P1 et P2

P1 appelle P2 et P3

P2 appelle P4



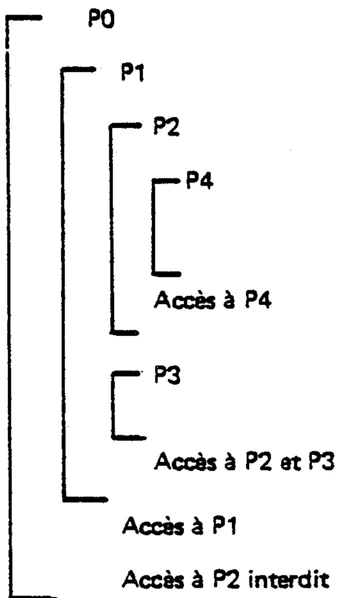
Les déclarations de procédure qui en résultent conduisent à l'imbrication suivante :



Un autre raisonnement dans la description du traitement qui serait :

P0 appelle P1 qui appelle P2 qui appelle P4

conduit, par manque de vue d'ensemble à l'imbrication suivante :



La procédure a été déclarée dans P1, ce qui rend P2 inaccessible depuis P0 car P2 n'est connue qu'à l'intérieur du bloc où elle a été déclarée c'est-à-dire P1.

figure 2 : Imbrication des déclarations de procédures.

L'exemple précédent nous montre qu'on peut traduire un traitement par deux sortes de schémas :

- un schéma statique (figure 2) qui correspond à l'ordre d'écriture du programme avec ses déclarations et ses instructions structurées en blocs. Cet ordre repose sur un organigramme.
- un schéma dynamique (figure 1) qui correspond à l'ordre d'exécution du programme ; il fait ressortir les ordres de contrôle contenus par ce dernier.

L'ordre dans lequel se déroule un programme peut être différent de celui dans lequel sont écrites les instructions par le jeu des appels de procédures.

Ainsi, pour un même programme le niveau d'imbrication atteint peut différer d'un schéma à l'autre.

La cohérence du programme au niveau des ordres de contrôle est vérifiée par le compilateur : une procédure ne peut en effet être appelée n'importe où dans un programme.

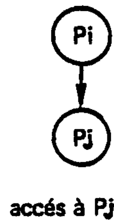
Principes généraux sur l'imbrication des procédures

1 - Accès à une procédure.

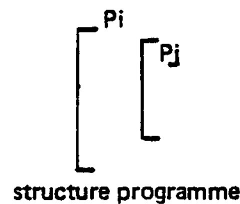
On appelle normalement une procédure depuis la procédure où elle est déclarée

Le principe est à respecter autant que possible car c'est lui qui donne au programme :

- sa clarté
- sa modularité
- sa logique

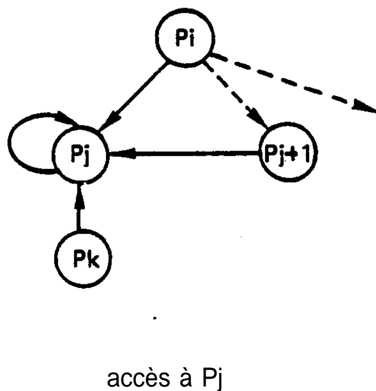


accés à Pj

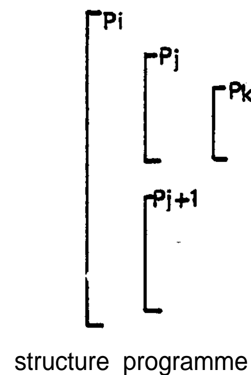


structure programme

Toutefois les possibilités d'accès générales, qui répondent à certains besoins particuliers sont traduites par les schémas ci-dessous.



accès à Pj



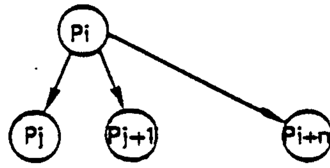
structure programme

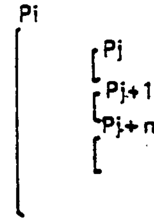
Une Procédure est accessible depuis :

- les procédures qu'elle renferme (appel récursif)
- les procédures parallèles
- les procédures qui la contiennent de tout niveau

2 – Accès depuis une procédure

On a accès normalement depuis une procédure, aux procédures qu'elle contient.



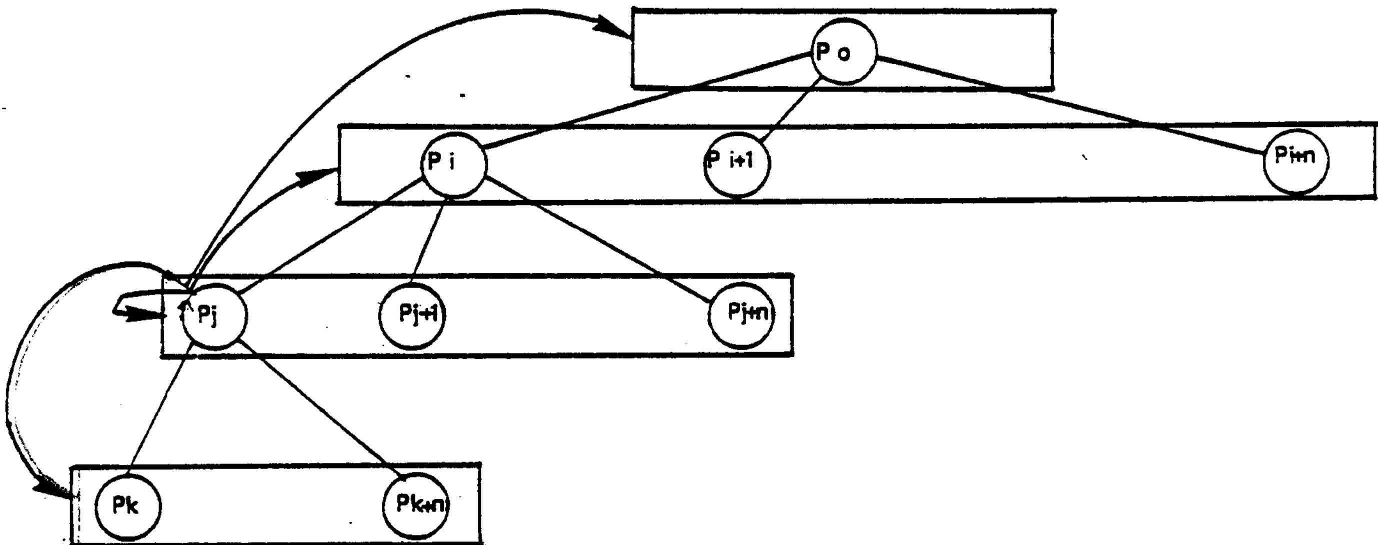


accès depuis Pi

structure programme

Plus généralement une procédure à accès à :

- toute procédure parallèle
- toute procédure qui l'englobe (accès récursif)
- toute procédure parallèle à une procédure englobante



accès depuis Pj

REMARQUE

- dans l'appel récursif syntaxiquement possible, les conditions de bon fonctionnement (sauvegarde) et restauration des données sont à réaliser par le programme.

3.8 - CONCLUSION

Un programme PL16 est un ensemble structuré de blocs qui constituent autant de traitements partiels dont l'enchaînement constituera le traitement global d'un problème. C'est l'imbrication des procédures et des instructions composées qui donne à un programme sa structure de blocs.

Les données sur lesquels travaillent les divers traitements sont cataloguées en sections. Ces sections sont valides au niveau d'un bloc.

4 - DÉCLARATIONS ET ORGANISATION DES DONNÉES

Les déclarations consistent à décrire les données d'un traitement et à leur donner des noms ou identificateurs.

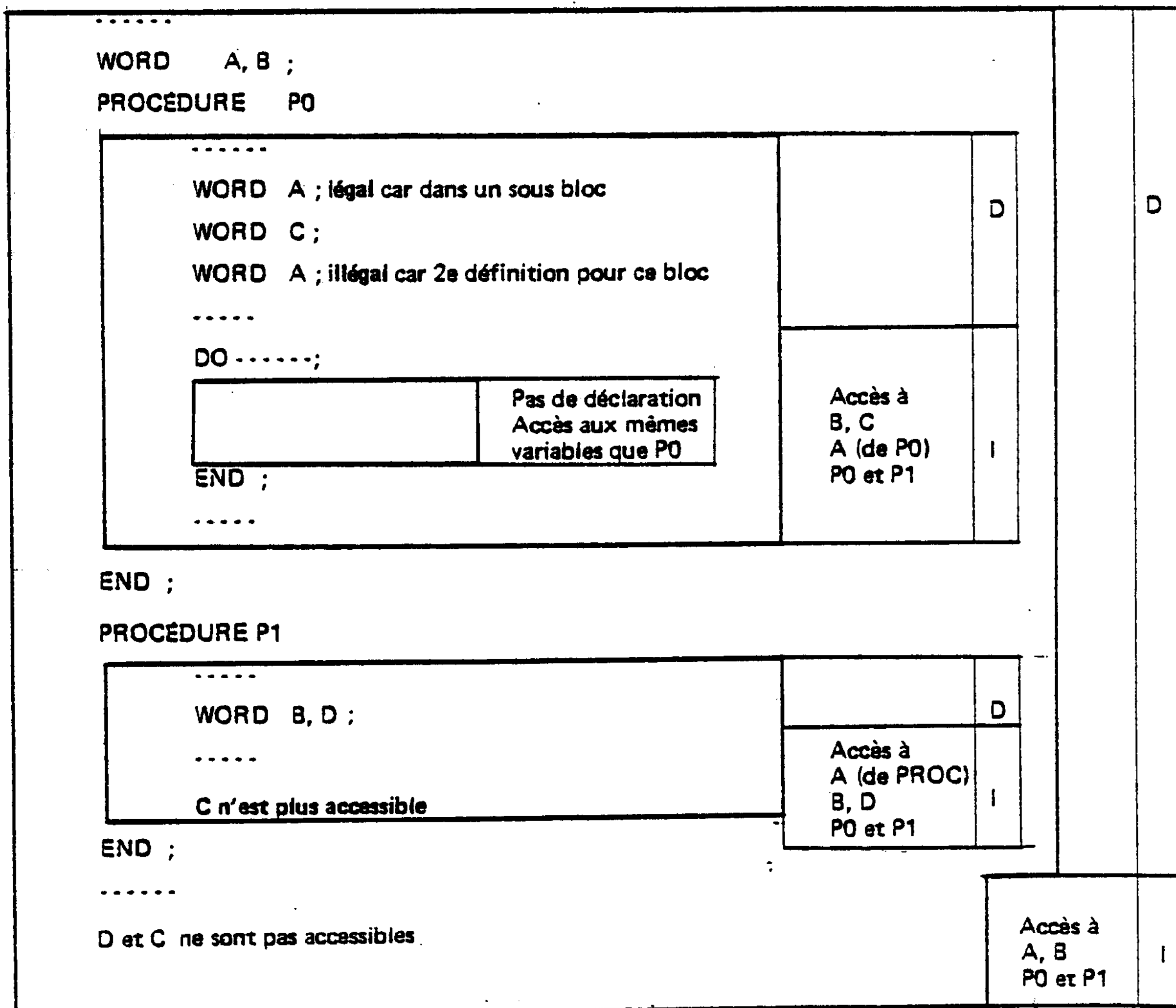
Les déclarations font partie du bloc qui définit ce traitement ; elles ne sont valides que pour ce bloc sauf cas particulier (cf chapitres 7 et 8 : définitions d'externes).

4.1 - PORTEE DES IDENTIFICATEURS

Un identificateur n'est connu que de l'intérieur du bloc où il est déclaré, on dit qu'il lui est local. Il ne peut être déclaré deux fois dans un même bloc, mais peut l'être une fois dans un bloc et dans les blocs contenus dans ce dernier ; la déclaration locale a toujours priorité sur une déclaration plus extérieure. Un identificateur déclaré dans un bloc est connu des blocs qu'il contient si ceux-ci ne le redéclarent pas : on dit qu'il leur est global. Dans un programme les blocs vont ainsi définir des niveaux dans la nomenclature des identificateurs.

Exemple :

MAIN PROCEDURE PROC



END.

4.2 - SECTIONS DE DONNEES

4.2.1 - GENERALITES

Indépendamment de l'arborescence, définie par les blocs, qui structure le programme et détermine la portée des identificateurs, l'ensemble des données est découpé en sections.

Sur la gamme SOLAR 16, l'adressage d'un opérande mémoire consiste en l'indication d'un déplacement relatif à une adresse de base contenue dans un registre spécialisé.

La zone mémoire ainsi accessible par un registre de base est de 256 mots (± 128 par rapport à la base) et constitue une SECTION de données pour le langage PL 16.

Il existe 3 registres de base : C, L, W (nommés RC, RL, RW en langage PL16) qui donnent donc accès, à un instant donné, à 3 sections de données.

Pour utiliser ce mode d'adressage, un certain nombre de règles sont à respecter :

- pour chaque élément déclaré, il faut spécifier la section où il se trouve
  . conséquence : les déclarations seront faites dans des sections
- au moment de l'accès à un élément il doit y avoir un registre de base qui pointe la section où est rangé cet élément.
  . conséquence : les registres de base doivent être chargés correctement pour permettre l'accès aux sections désirées.

Une section étant limitée à 256 mots, il faudra en ouvrir une nouvelle quand une section sera pleine. Aussi le programme devra-t-il utiliser au mieux ce fait qu'à un instant donné le calculateur permet d'accéder à 3 sections. De plus, les tables seront rangées hors sections de données, l'accès aux éléments qu'elles renferment étant possible par l'adressage indirect post-indexé.

Pour le programmeur, toutes ces conditions à respecter sont des causes d'erreur, aussi le langage PL16 rend t-il ces vérifications automatiques car la syntaxe à prévu le passage au compilateur des renseignements nécessaires à ces contrôles.

1) En tête des déclarations d'un bloc on trouve :

- une déclaration de section spécifiant un registre de base (C, L ou W). Les éléments déclarés à la suite seront rangés dans cette section jusqu'à la fermeture du bloc ou jusqu'à la déclaration d'une autre section.

2) En tête des instructions d'un bloc possédant des déclarations on trouve :

- une instruction de chargement des bases ou une indication donnée au compilateur lui disant que les éléments de telle et telle section sont désormais accessibles car les registres de base sont correctement chargés.

La découpe d'un problème en traitements, caractérisés par leurs données et leurs instructions, conduit à repartir les données en :

- données communes à plusieurs traitements
- données locales à un seul traitement
- variables de travail

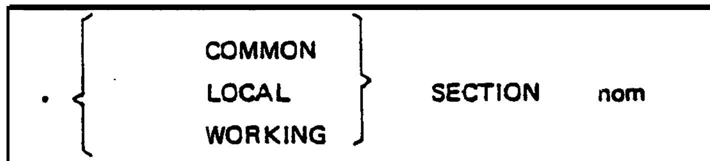
Pour une bonne gestion des bases, il est conseillé de ranger respectivement ces données dans des sections accessibles par les bases C, L et W. Ainsi à un instant donné un traitement pourra avoir accès à ces 3 types de données.

4.2.2 - DÉCLARATION DE SECTION DE DONNEES

Comme nous venons de le voir, aucune déclaration dans un bloc ne peut apparaître avant une déclaration de section. Par le type donné à la section l'utilisateur spécifie le registre de base qui servira à adresser ces données.

Ce type peut être soit COMMON
soit LOCAL
soit WORKING et le compilateur gère les accès aux variables de ces sections respectivement par rapport aux bases C, L ou W.

1) Ouverture de section



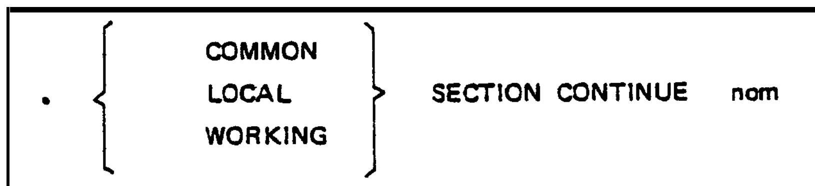
Le caractère ; n'est pas obligatoire derrière une déclaration de section

La section nommée par < nom > est alors ouverte. A un instant donné, au cours des déclarations, les données ne sont rangées que dans une seule section : la dernière ouverte.

Une section est définitivement fermée à la fermeture du bloc où elle a été déclarée. Son nom n'est alors plus connu et on ne peut plus y ranger de données. Le nom d'une section suit donc la même loi que les autres identificateurs en ce qui concerne sa connaissance en fonction de la structure de bloc d'un programme.

2) Réouverture de section

Lorsqu'on veut ranger de nouveau des données dans une section qui est connue mais qui n'est pas la dernière ouverte, on procède à la déclaration suivante :

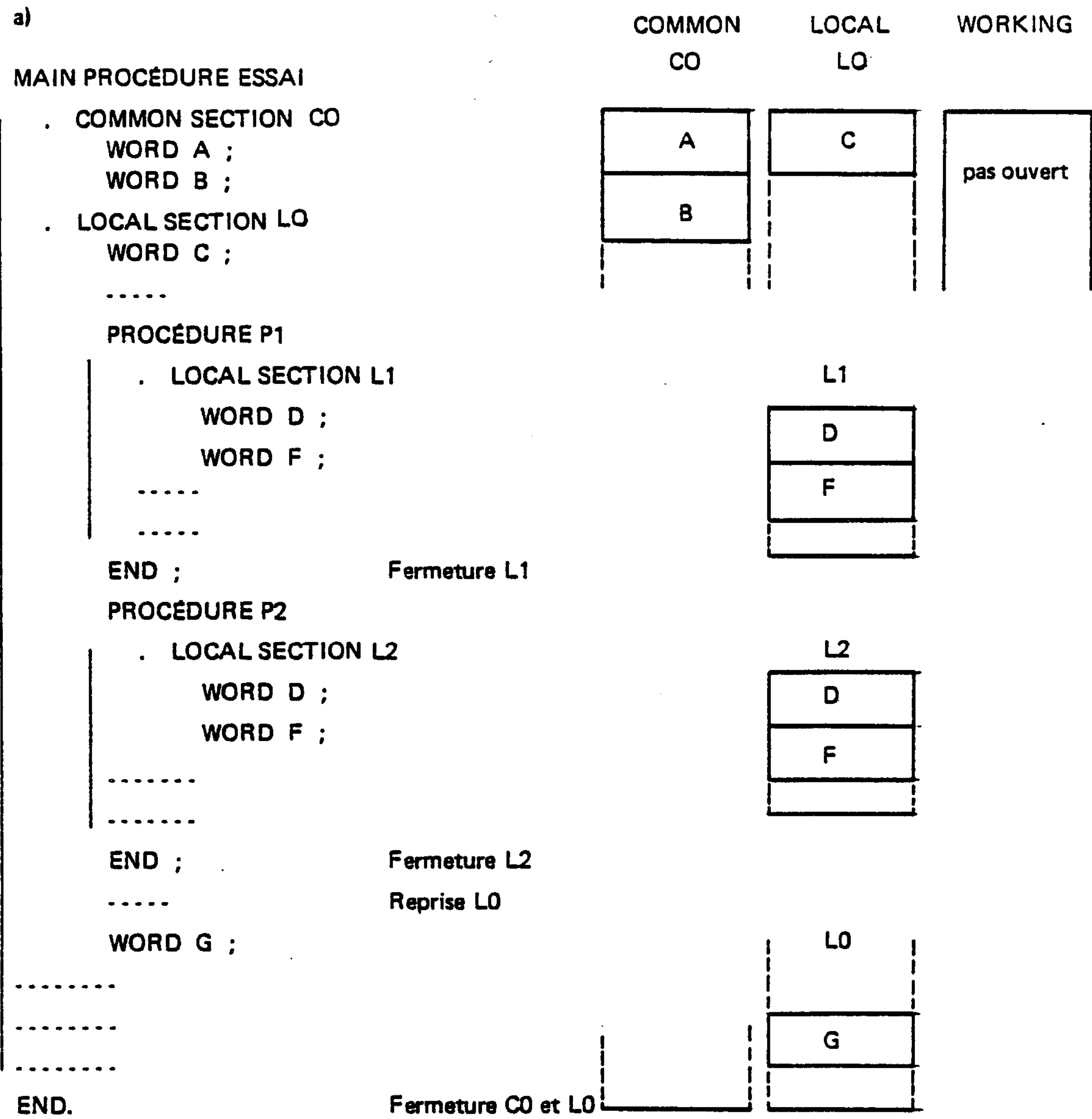


La section ouverte est alors celle dont le nom est indiqué dans la déclaration ; les déclarations qui suivront seront chargées en séquence à la suite des dernières variables de cette section.

Remarque :

La déclaration de "section KSTORE" est vu au paragraphe 4.8.

3) Exemples

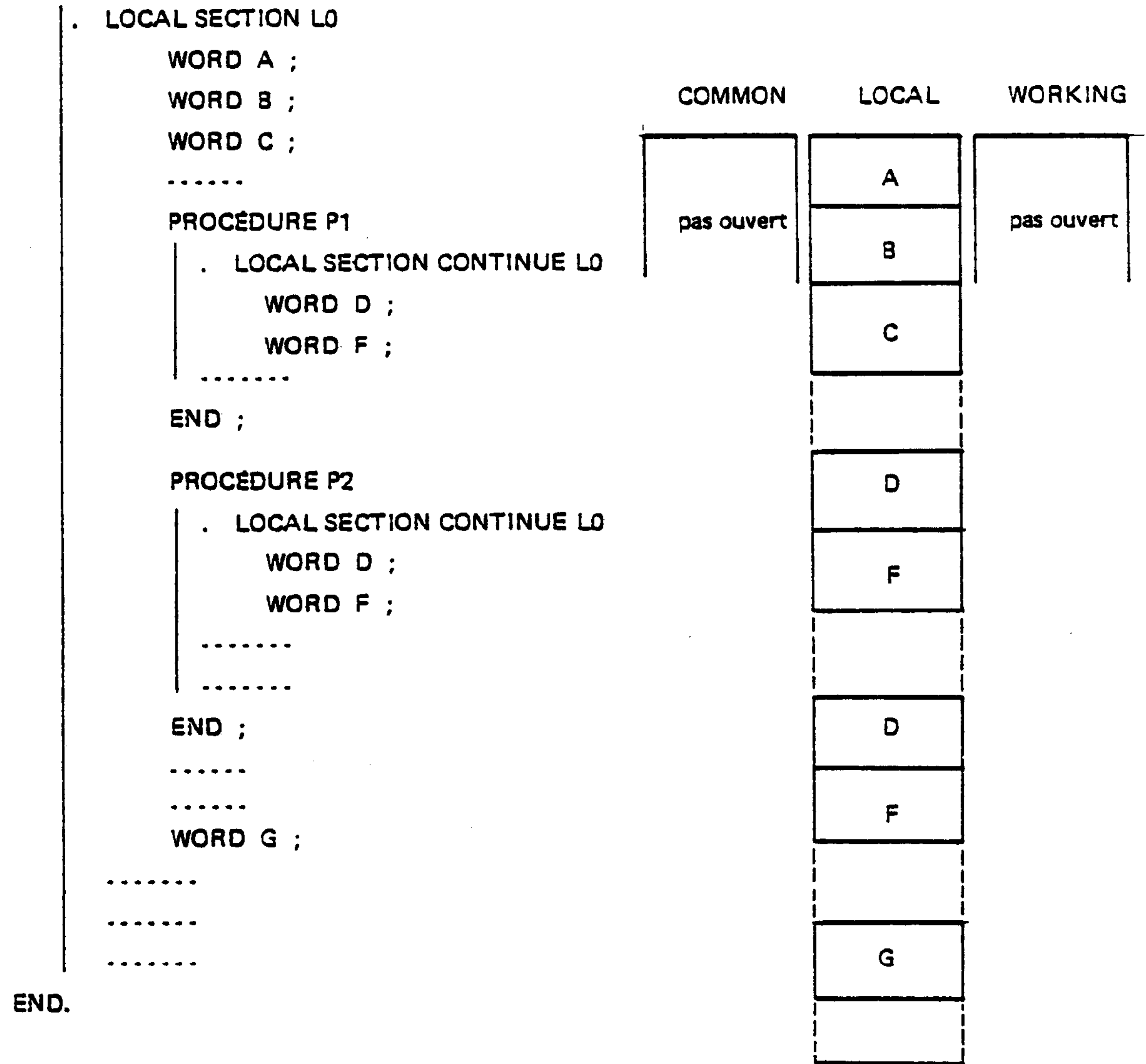


Dans chaque procédure il faudra charger la base L pour accéder successivement aux sections L0 (pour ESSAI), L1 (pour P1) et L2 (pour P2). Le chargement de la base C n'est utile que dans la procédure principale ESSAI.

b) Même programme que précédemment

On utilise dans cet exemple la même section pour toutes les déclarations

MAIN PROCEDURE ESSAI



La section L0 doit toujours être accessible, pour cela il suffit de charger la base L dans le programme principal. Ainsi pour un programme ayant peu de données, une seule section peut suffire ; on évite ainsi un chargement fréquent des bases.

4) Conclusion

En résumé une section de données est caractérisée par :

a) Son type

COMMON
LOCAL
WORKING

suivant que le registre de base utilisé est respectivement C, L ou W. Le programmeur précise aussi de cette façon l'utilisation qu'il fait de ses données.

b) Sa déclaration d'ouverture

Celle-ci est faite en tête de bloc et précède la déclaration des données qu'il renferme.

c) Sa taille

Elle est limitée à 256 mots

d) Sa fermeture

Elle est fermée à la fermeture du bloc dans laquelle elle a été ouverte. Une section peut être momentanément abandonnée par l'ouverture d'une nouvelle section et pourra ensuite être réouverte par CONTINUE.

e) Son accès par le programme

L'accès à une section de données est possible à partir du moment où un registre de base est positionné sur cette section. Ce positionnement se fera en tête des instructions d'un bloc.

4.3 - DECLARATION DE CONSTANTES

Les déclarations de constantes permettent d'attacher un nom à une valeur ; ce nom, pouvant alors par la suite apparaître partout où une telle valeur le pourrait, la valeur d'une constante n'est générée (éventuellement) par le compilateur, qu'au moment de son utilisation par une instruction ; et dans la section locale accessible à cet instant.

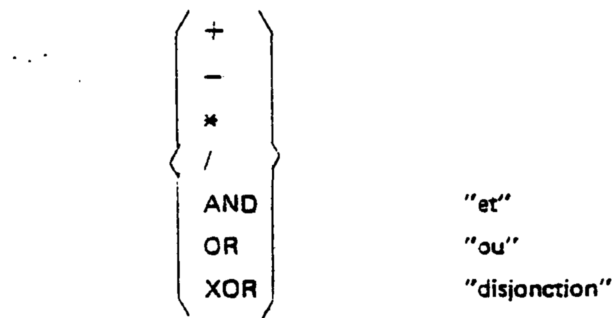
La forme générale d'une telle déclaration est :

```
CONSTANT nom = expression de constante ;
```

L'expression qui initialise cette valeur est évaluée de la gauche vers la droite sans priorité sur les opérateurs, mais avec priorité sur l'évaluation des termes.

Une expression de constante est de type :

ABSOLU c'est alors une expression formée avec des termes absolus et les opérateurs :



Exemples :

```
CONSTANT    N = 50 ;
CONSTANT    TAILLE = (@ PREMIER - @ DERNIER) * 2 ;
CONSTANT    M = N + 2 ;
CONSTANT    CIRCLE = 4 * M * N ;
CONSTANT    Y = 3 * (2 + 4) ;
```

ADRESSE c'est l'adresse d'une donnée ou d'une instruction à laquelle peut s'ajouter, se retrancher ou opérer logiquement une expression absolue.

Exemple :

```
CONSTANT    ADTAB = @ TAB AND '7FFF ;
CONSTANT    ADRSEG = @ LOC + 128
```

Remarque :

Une constante initialisée avec une adresse de tableau comportera le bit d'indexation, on peut l'éliminer par une intersection avec le masque '7FFF, comme dans l'exemple ci-dessus.

La notion "d'expression de constante" étant utilisée plus généralement lors de toute initialisation, nous donnons la syntaxe, ci-dessous hors de la notion de déclaration de constante.

Expression de constante	::=	{ expression absolue / expression d'adresse }
Expression d'adresse	::=	@ NOM (1) [opérateur - expression absolue [opérateur...]]
Expression absolue	::=	terme opérateur terme...
Terme	::=	[{ NOT }] { litteral / (expression absolue) / (expression d'adresse - expression d'adresse) }
Opérateur (2)	::=	+, −, *, /, AND, OR, XOR

(1) NOM peut être un nom de variable, section, ressource, PRIVATE, IOCB
Procédure, label
Identificateur externe
Identificateur non déclaré

dans ce dernier cas, l'identificateur est supposé défini ultérieurement comme label ou comme procédure si "NOM" est le nom d'un tableau, "@ NOM" comporte le bit d'indexation.

(2) Les opérateurs * et / sont interdits dans "expression d'adresse".

4.4 - DECLARATION DE VARIABLES

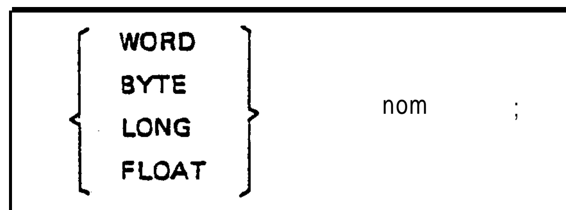
Par le nom qu'il donne à une variable le programme fait référence à une valeur contenue par un emplacement mémoire, cette valeur pouvant être de tous les types de constantes vus dans le chapitre 2. En même temps qu'elle lui associe un identificateur une déclaration de variable donne au compilateur certaines informations sur cette variable :

- la dimension de l'emplacement mémoire qu'il faut associer à cette variable. Le compilateur réserve alors dans la section où apparaît cette déclaration, la place mémoire nécessaire à la variable et établit la correspondance entre l'identificateur et son emplacement mémoire. La programmeur n'a donc pas à connaître l'emplacement de sa variable.
- la nature des valeurs que la variable sera amenée à prendre (entier. flottant...). Le compilateur peut alors contrôler que celles-ci sont bien en accord avec le type de la variable tout au long du programme.

De plus, le compilateur déduira de ces informations le codage, c'est-à-dire la traduction machine, à adopter lors de l'accès à cette variable par les instructions du programme.

4.4.1 - DECLARATION DE VARIABLE SIMPLE

Forme générale :



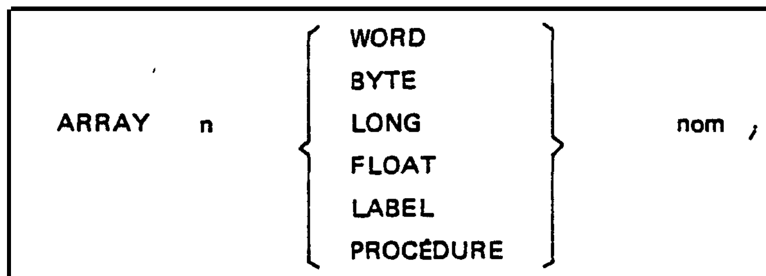
On définit ainsi des variables courtes (WORD et BYTE)
des variables longues (LONG)
des variables flottantes (FLOAT)

La place réservée pour chaque type de déclaration est de :

- un mot pour le WORD et BYTE
- deux mots pour le LONG et FLOAT

4.4.2 - DÉCLARATION DE TABLEAU

Forme générale :



où n est un littéral

On définit ainsi des tableaux à une dimension assimilés à une suite d'éléments. Ces éléments peuvent être des variables de type :

- courtes, longues flottantes
- adresses d'étiquettes
- adresses de procédures

Le nombre n est le nombre d'éléments du type indiqué après ce nombre.

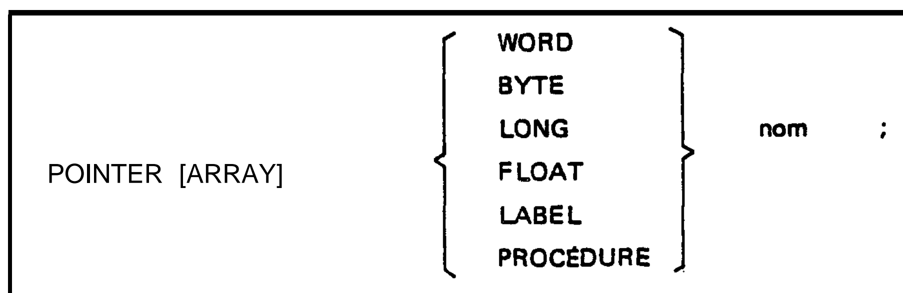
La place occupée par les éléments d'un tableau est de :

- un octet par élément pour les tableaux de BYTE
- un mot par élément pour les tableaux de WORD, de LABEL et de PROCEDURE
- deux mots par élément pour les tableaux de LONG et de FLOAT.

La déclaration d'un tableau "ARRAY n" réserve les n éléments du tableau proprement dit hors section de données (dans une zone appelée la section des tables) et réserve dans la section ouverte à cet instant un pointeur initialisé avec l'adresse de la table et contenant le bit d'indexation. C'est par ce pointeur implicite qu'on aura accès aux éléments du tableau.

4.4.3 - DÉCLARATION DE POINTEUR

Forme générale :



Les déclarations de pointeur d'entrée-sortie ou de ressource sont vus respectivement aux chapitres 7 et 9

On définit ainsi des variables destinées à contenir une adresse :

- de variable simple
- de tableau (dans ce cas le bit d'indexation est présent)
- d'étiquette
- de procédure
- de table d'échange ou de sémaphore (chapitre 7 et 8)

La place réservée par un pointeur à la taille d'un mot.

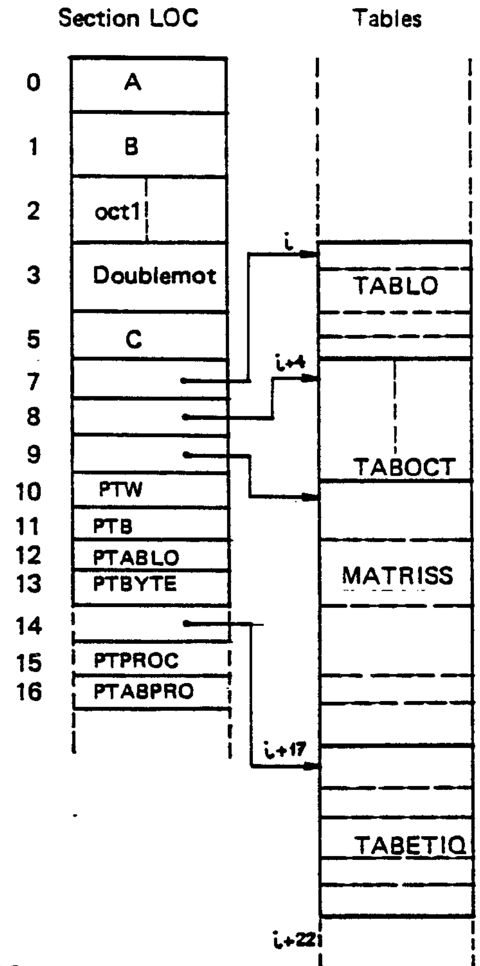
Cette “variable pointeur” est utilisée soit :

- comme variable mot comme une autre variable simple
- comme pointeur pour accéder à la variable pointée, on l'utilise alors précédée du symbole d'indirection &. Elle a alors le type indiqué dans la déclaration.
- comme pointeur pour accéder à une étiquette ou à une procédure, on l'utilise alors dans une instruction appropriée (GOTO ou CALL), non procédé du symbole d'indirection.

Remarque : l'accès aux variables longues par indirection (cas des tableaux ou des pointeurs explicites) est réalisé par des instructions double longueur qui nécessitent lors de l'exécution des programmes la présence de l'opérateur flottant câblé ou de l'arithmétique flottante programmée.

Exemples :

```
------

------

. LOCAL SECTION LOC

     WORD A, B ;

     BYTE OCT1 ;

     LONG DOUBLEMOT ;

     FLOAT C ;

     ARRAY 4 WORD TABLO ;

     ARRAY 5 BYTE TABOCT ;

     ARRAY 5 FLOAT MATRISS ;

     POINTER WORD PTW

     POINTER BYTE PTB

     POINTER ARRAY WORD PTABLO ;

     POINTER ARRAY BYTE PTBYTE

     ARRAY 5 LABEL TABETIQ ;

     POINTER PROCÉDURE PTPROC ;

     POINTER ARRAY PROCÉDURE PTABPRO ;
```



Ces déclarations réservant successivement :

A, B	:	deux mots
OCT1	:	un octet
Doublemot	:	une variable longue
C	:	une variable flottante
TABLO	:	un tableau de 4 mots
TABOCT	:	un tableau de 5 octets
MATRISS	:	un tableau de 5 flottants
PTW	:	un mot pouvant pointer sur un mot
PTB	:	un mot pouvant pointer sur un octet

PTABLO	:	un mot pouvant pointer sur un tableau ou une partie de tableau de mots
PTBYTE	:	un mot pouvant pointer sur un tableau ou une partie de tableau d'octets
TABETIQ	:	un tableau de 5 adresses d'étiquettes
PTPROC	:	un mot pouvant pointer sur une procédure
PTABPRO	:	un mot pouvant pointer sur un tableau ou une partie de tableau d'adresses de procédures

4.4.4 - INITIALISATION

L'initialisation d'une variable peut être fait à sa déclaration (1). Elle complète donc les déclarations précédentes.

Initialisation	: : =	déclaration	= (liste de valeurs)
Liste de valeur	: : =	{ **chaîne** / **valeur** / ***n (valeur)** }	[, liste de valeur]
Valeur	: : =	{ expression de constante / nombre flottant / nombre décimal long }	
n	: : =	littéral	

La définition : expression de constante a été détaillée au paragraphe : "déclaration de constante" (4.3). Une expression de constante initialise un élément ; les chaînes sont traitées de gauche à droite, il y a un caractère par octet.

La spécification : x n permet d'initialiser avec une même valeur n éléments consécutifs. Elie précède la valeur d'initialisation qui est à répéter. Si n n'est pas spécifié il est pris égal à 1.

Lorsque l'initialisation introduit un identificateur non encore déclaré, ce dernier est supposé être un identificateur de procédure ou d'étiquette et devra être déclaré ainsi par la suite avant la fermeture du bloc contenant cette initialisation.

L'initialisation d'une variable "flottante" nécessite la présence de l'option opérateur flottant câblé, OU celle, en mémoire de l'arithmétique flottante programmée (chapitre 9).

Exemple :

```
        WORD A = (4) ; BYTE OCT = (8) ; WORD [illegible] = ("* *") ;
ARRAY 10 WORD X = (1,2)
ARRAY 5 WORD Y = (1, 2, * 2 (6), 12) ;
ARRAY 14 BYTE MESER = ("ERREUR␣ NO ␣:␣0 0") ;
        FLOAT C = (1.25 E_6) ;
POINTER WORD NUMERO = (@ MESER + 6 AND '7FFF) ;
POINTER ARRAY WORD PX = (@ X + 5) ; «PX pointe un tableau de mots dont le premier
                                    «élément est X (5).
```

(1) en l'absence d'initialisation, le contenu d'une variable est indéfini.

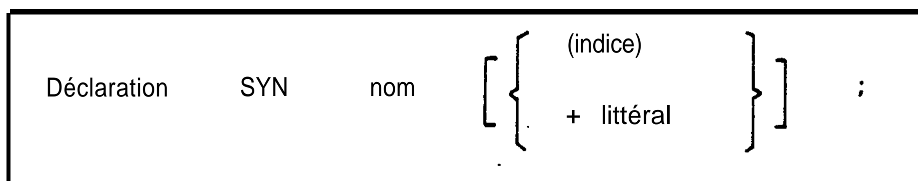
Lorsque l'initialisation est suivie d'un identificateur non encore déclaré, ce dernier est supposé être un identificateur de procédure ou d'étiquette et devra être déclaré ainsi par la suite avant la fermeture du bloc contenant cette initialisation.

Exemples :

```
MAIN PROCÉDURE ESSAI
    | LOCAL SECTION L0
    |
    | POINTER PROCÉDURE PTPROC = (@ PROC1) ;
    |
    | ARRAY 2 PROCÉDURE ADPROC = (@PROC1,@ PROC2) ;
    |
    | POINTER LABEL PETIQ = (@L1) ;
    |
    | --------
    |
    | PROCÉDURE PROC1
    |     | ------
    |     | ------
    | END ;
    | PROCÉDURE PROC2
    |     | ------
    |     | ------
    | END ;
    | -----
    |
    | L1 :-----
    | -----
END.
```

4.4.5 - SYNONYMIE

Une déclaration "synonyme" complète la déclaration générale de variable (1). Elle permet de donner des noms différents à un même élément de mémoire, de modifier le type ou de restructurer une variable déjà déclarée. Grâce à elle, on pourra donc définir à des adresses connues, de nouvelles variables de type différent ou non. Une variable "synonyme" n'est donc pas rangée à la suite dans une section mais s'y trouve déjà à l'adresse d'une autre variable rangée précédemment.



(1) Remarque : Une variable déclarée synonyme ne peut être une variable définie ou référencée comme externe (voir chapitres 6 et 7).

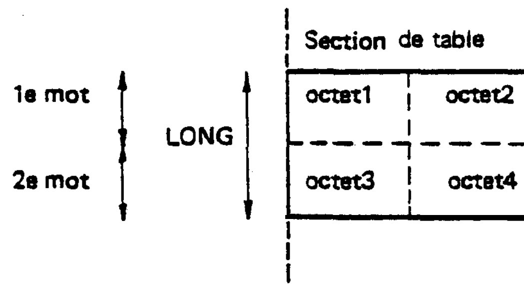
- le littéral un déplacement en nombre de mots
- indice indique un déplacement en nombre d'éléments. Il donnera donc pour la nouvelle variable un déplacement dans la section, égal à celui de l'ancienne, augmenté du nombre d'éléments égal au produit de l'indice par la longueur d'un élément (ce nombre doit obligatoirement être un nombre pair d'octets).

Exemples :

1)
```
WORD TOTO ;
WORD JOJO SYN TOTO ;
```
TOTO et JOJO sont 2 noms pour le même mot mémoire

```
BYTE OCT1 ;
BYTE OCT2 SYN OCT1 ;
```
OCT1 et OCT2 sont 2 noms pour le même octet

2)
```
ARRAY 1 LONG DOUBLEMOT ;
ARRAY 2 WORD TABM SYN DOUBLEMOT ;
ARRAY 4 BYTE TABO SYN DOUBLEMOT ;
```



adresse i : 1

adresse i : adresse Pointée par DOUBLEMOT

adresse i + 1

Ces 3 déclarations changent le type et restructurent une variable d'adresse connue i.

3)
```
ARRAY 10 WORD TABLO ;
ARRAY 5 WORD TABW SYN TABLO (5) ;
ARRAY 20 BYTE TABOCT SYN TABLO ;
ARRAY 5 LONG TABLON SYN TABLO ;
```

Cas déclarations restructurent la variable tableau TABLO. De plus TABW est un tableau de 5 mots correspondant aux 5 derniers mots de TABLO.

4)
```
ARRAY 50 WORD T ;
POINTER WORD LAST = (@T + 49 AND '7FFF)
```

Le dernier élément du tableau T est accessible par T (49) ou & LAST

L'écriture WORD LAST SYN T(49) ; est INTERDITE car on ne peut avoir synonyme entre une variable simple et un élément de tableau.

5) POINTER WORD PW ;

POINTER BYTE PB SYN PW ;

Le pointeur PW permet de pointer sur un mot par l'adresse qu'il contient. Le pointer PB permet de pointer sur le premier octet de ce même mot.

6) FLOAT F1 ;

WORD W1 SYN F1 ;

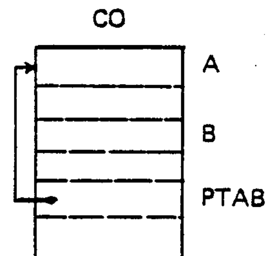
WORD W2 SYN F1 + 1 ;;

W1 représente le 1er mot mémoire du flottant F1, W2 le deuxième mot.

Remarque :
Pour structurer une zone de données, il est possible de réserver de la place sans déclaration, cela à l'aide de RES n, où n indique le nombre de mots à réserver.
Dans la version PL 1 164 001 01/, ces n mots sont mis à zéro.

Exemples :

```
MAIN PROCEDURE ESSAI
    . COMMON SECTION CO
            WORD A ;
            RES3 ;
            POINTER ARRAY WORD PTAB = (@ A OR '8000) ;
            WORD B SYN A + 2 ;
            ------
    . USING COMMON = CO ;
            RA : = B ;
            RA : = & PTAB (2) ;
END.
```



Les 2 instructions permettent le chargement, dans le registre RA, du 3e mot de la section CO.

Remarque :

Le terme '8000 dans l'initialisation de PTAB permet de positionner le bit d'indexation. Le terme serait inutile dans une initialisation par une adresse de tableau car dans ce cas le bit index est déjà présent.

4.4.6 - ACCES ET UTILISATION DES VARIABLES

Variable simple

\- - - - - -

déclaration	WORD A, BYTE B ;	
accès à l'adresse	@ A, @ B	
accès à la valeur	A, B	

Variable tableau

déclaration	ARRAY n WORD TAB ;	
accès à l'adresse	@ TAB	(terme d'expression entière)
	@ TAB (I)	(1er terme expression de registre)
	TAB	(constante adresse sans bit index désignant le 1er mot du tableau dans la déclaration d'échange ou les instructions de visualisation (SNAP)
accès à la valeur (d'un élément)	TAB (I)	

Pointeur de variable

déclaration	POINTER WORD PTA ;	
accès à la valeur	PTA	
accès à la valeur pointée	& PTA	

Pointeur de tableau de variable

déclaration	POINTER ARRAY WORD PTAB ;	
accès à la valeur	PTAB	
accès à un élément	& PTAB (I)	

Pointeur d'étiquette (de procédure)

déclaration	POINTER { LABEL / PROCEDURE } E ;	
accès à la valeur	E	(dans une expression)
accès à l'étiquette (la procédure)	E	(par une instruction de branchement)

Pointeur de tableau d'étiquette (de procédure)

déclaration	POINTER ARRAY N { LABEL / PROCEDURE } TE	
accès à la valeur	NON	
accès à l'étiquette (la procédure)	TE (I)	(dans l'instruction de branchement)

Exemples :

```
MAIN PROCEDURE ESSAI

   . LOCAL SECTION LO
          WORD A ; BYTE OCT

          ARRAY 10 WORD TABLO ;

          ARRAY 5 WORD TABW SYN TABLO (5) ;

          POINTER ARRAY WORD PTABLO = (@ TABLO + 5) ;

          ARRAY 20 BYTE TABOCT SYN TABLO ;

          POINTER ARRAY BYTE PTAB 1 = (@ TABLO + 5) ;

          POINTER BYTE LAST = (@ TABLO + 9 AND '7FFF) ;

          POINTER LABEL PLAB ;

          POINTER ARRAY PROCEDURE PTABPRO ;

          ........

          .......
```

LO
A
OCT
implicite
implicite
PTABO
implicite
PTAB1
LAST
PLAB
PTABPRO

A	accès au mot A
OCT	accès à l'octet OCT
TABLO (5)	accès au 6e élément de TABLO
TABW (0)	accès au premier élément de TABW ou au 6e élément de TABLO
& PTABO (0)	accès au 6e élément de TABLO par le pointeur tableau explicite PTABO
PLAB	accès à la variable adresse d'étiquette
GOTO PLAB	branchement à l'étiquette contenue dans PLAB
PTABPRO	accès au pointeur de tableau de procédure
TABOCT (10)	accès au 11e octet (octet 10) de TABLO
& PTAB1 (0)	accès au 11e octet de TABLO
& LAST	accès au 19e octet de TABLO (octet 18)
PTABPRO (4)	accès à la 5e adresse de procédure d'un tableau de procédures pointé par PTABPRO
CALL PTABPRO (4)	accès à la procédure pointée par cette 5e adresse

Tables

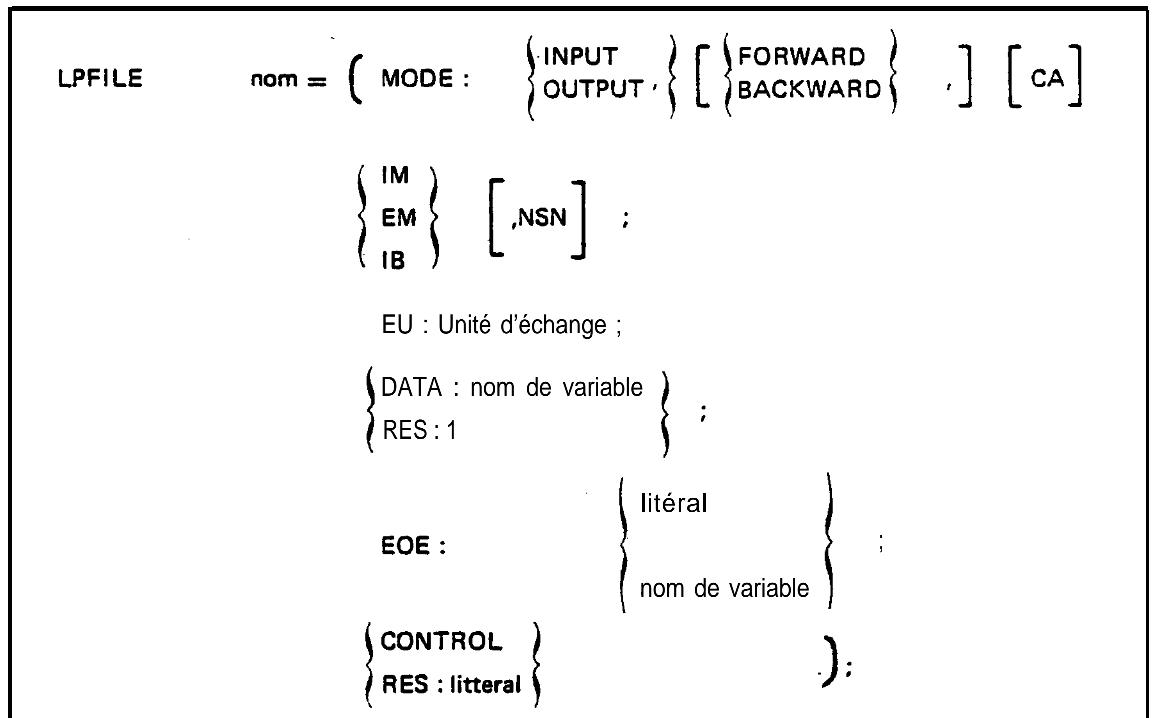
TABLO	0	1	0
TABOCT	2	3	
	4	5	
	6	7	
	8	9	
TABW	10	11	
	12	13	
	14	15	
	16	17	
	18	19	9

```
       .......

END.
```

4.5 - DECLARATION D'ECHANGE

Cette déclaration permet de réserver et d'initialiser, sous forme symbolique, la table d'échange nécessaire à l'I.O.C.S. pour réaliser les ordres d'entrées-sorties PL16.

Forme générale :



MODE	représente l'octet de fonction
INPUT	entrée
OUTPUT	sortie
FORWARD	entrée ou sortie en avant présente par défaut
BACKWARD	entrée en arrière
CA	fin d'enregistrement donnée par une table de Codes d'Arrêt
EM	retour en fin d'échange
IM	retour immédiat après initialisation
IB	retour immédiat et utilisation du "pool buffer"
NSN	par de suppression des caractères NULL en lecture
EU	nom d'unité d'échange (cf. liste cidessous) ou numéro
DATA	adresse des données : zone où s'effectuera l'échange
EOE	fin d'enregistrement {compte d'octets ou ,@ tableau d'octets si CA}
CONTROL	génère les 3 mots supplémentaires nécessités par les options EM et IM. Le troisième n'étant pas obligatoire, RES : 2 permet, par exemple, de ne réserver que deux mots pour le compte rendu d'échange.

Une instruction d'entrée-sortie PL16 fait référence à l'échange ainsi défini.

Le compilateur contrôle la définition correcte du LPFILE ainsi que l'utilisation qui en est faite.

Par exemple on ne pourra lancer un ordre de lecture que sur un LPFILE déclaré INPUT.

Exemples :

LPFILE COMMAND = (MODE : INPUT, EM ; EU : PC ;
DATA : BUFFER ; EOE : 4 ; CONTROL) ;

Entrée sur l'unité symbolique PC de 4 caractères qui seront rangés dans BUFFER. Retour en fin d'échange

LPFILE BINAIRE = (MODE : INPUT, EM, NSN,; EU : BI ;
DATA : BUFFER ; EOE : 80 ; CONTROL) ;

Entrée sur l'unité symbolique "binary input" de 80 caractères à ranger dans BUFFER sans suppression des "null". Retour en fin d'échange.

LPFILE OBJET = (MODE : OUTPUT, IB ; EU : BO ;
DATA : BUFFER ; EOE : 80) ;

Sortie sur "binary Output" de 80 caractères à partir de BUFFER. Retour immédiat et utilisation du "pool buffer".

LPFILE ENTREE = (MODE : INPUT, CA, EM, EU : SO ;
DATA : BUFFER ; EOE : TABCODE ; RES : 2) ;

Entrée sur l'unité 50 de caractères à mettre dans BUFFER avec arrêt sur code. Les codes d'arrêt se trouvent dans le tableau d'octets TABCODE.

Remarque :

Les options EU et DATA pourront être modifiées par programme en utilisant l'instruction ASSIGN (cf. 5.12.3).

Liste des noms d'unités d'échange acceptés (voir aussi, manuel de référence des systèmes d'exploitation SOLAR 16.

Unités symboliques	Unités fonctionnelles
SO	ME
SI	ZE
BO	TR
BI	TS
LL	TK
CC	TP
LO	HR
EC	HP
EL	CR
PC	LP
U1 à U6	TU1
	TU2
	DU1
	¦
	DU8

4.6 - DECLARATION D'ETIQUETTE

Rappel

Dans un programme les instructions sont exécutées séquentiellement dans l'ordre ou elles sont écrites ; cependant certaines instructions permettent d'interrompre ce déroulement séquentiel et d'effectuer des "sauts" vers d'autres instructions du programme. Une étiquette permet de repérer une instruction simple ou composée pour pouvoir y accéder directement.

Une étiquette est définie par l'apparition de son nom, suivit du symbole (:) en tête d'une instruction.

```
exemple : EI : BEGIN
                 E2 : RA := RB ;
               END ;
```

Une étiquette est référencée par l'apparition de son nom dans une instruction de rupture de séquence

exemple : GØTØ E3 ,

en dehors de ces cas une étiquette doit parfois être déclarée.

Le rôle d'une déclaration d'étiquette est de fournir au compilateur, les renseignements qui lui manquent au moment d'une référence "en avant" (étiquette non encore définie), à savoir :

- étiquette accessible par branchement direct (déplacement < 128) ou nécessité d'un relais (1)
- étiquette appartenant au bloc courant ou non (restauration ou non, du contexte externe lors d'un branchement)

Le premier point est résolu par la déclaration elle-même qui comporte ou non l'indicateur "indirect". Le deuxième point est résolu par le fait qu'une déclaration d'étiquette apparaît dans la partie INSTRUCTION du bloc ou elle sera définie ; le compilateur connaît alors le contexte de travail de la séquence étiquetée.

Forme générale d'une déclaration

```
[INDIRECT]      LABEL           nom ;
```

Une déclaration d'étiquette n'est obligatoire que s'il existe des références avant provoquant une sortie du bloc courant et/ou nécessitant un relais.

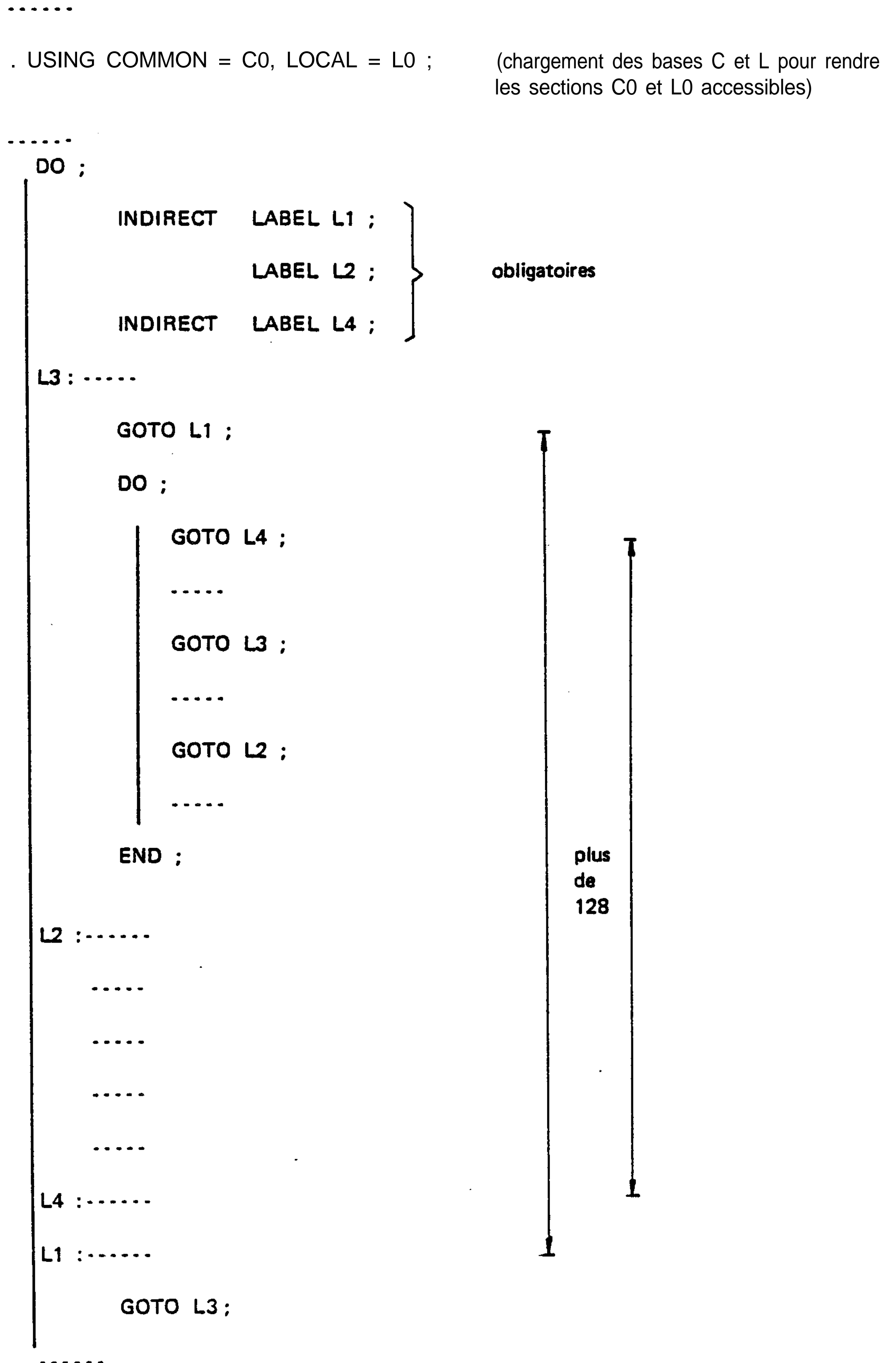
REMARQUE

Les étiquettes étant peu nombreuses, en PL16, on aura intérêt à déclarer systématiquement en tête de bloc les étiquettes définies dans ce bloc.

(1) par souci d'optimisation, le compilateur génère par défaut un branchement relatif (cf. instructions SOLAR 16).

Les déclarations INDIRECT LABEL génèrent, dans la section locale accessible par le bloc, un mot qui sert de relais pour le branchement vers l'étiquette nommée.

Exemple :



4.7 - DECLARATION DE PROCEDURE

4.7.1 - MAIN PROCEDURE

La "MAIN" procédure est la déclaration du traitement global décrit par le programme :

```
MAIN PROCEDURE    nom

        bloc

END.
```

L'écriture d'un programme PL16 sera donc la suite des déclarations et des instructions composant le bloc d'une MAIN procédure.

Cette procédure constitue le programme principal, c'est-à-dire la séquence de programme qui est lancée après le chargement et qui a donc le contrôle.

Exemple :

MAIN PROCEDURE CALCUL

```
D ;
D ;
D ;

I ;
I ;
I ;
I ;
```

Programme PL16

Nota :

Sur le délimiteur de fin de déclaration, END, le compilateur génère un retour au superviseur.

4.7.2 - PROCEDURE

Parmi les traitements partiels à effectuer pour réaliser une fonction donnée, il en est qui doivent être exécutés plusieurs fois et en des endroits différents. Ces traitements sont déclarés, une fois pour toutes en tête de ce bloc comme les autres entités utilisées dans ca bloc. Ce sont des déclarations de procédure.

Remarque : toutefois une déclaration de procédure ne peut apparaître que dans la partie déclaration d'un bloc de procédure.

```
PROCEDURE    < nom >   [ (paramètres [ , idem. . . ] ) ]

        < bloc >

END ;
```

Cette déclaration associe un nom au traitement décrit dans le bloc et en définit les paramètres ou conditions d'exécution.

Elle réserve un mot dans la section en cours initialisé avec l'adresse de début des instructions c'est-à-dire le point d'entrée de la procédure.

1) En tête

- le nom donné à une procédure est un identificateur
- les paramètres qui peuvent apparaître derrière le nom de la procédure sont appelés, paramètres formels

Un paramètre doit être déclaré dans le bloc de la procédure, c'est donc une variable locale pour la procédure. Au moment de l'appel de la procédure, les paramètres formels seront initialisés avec la valeur des paramètres effectifs correspondants, le compilateur ayant généré les instructions nécessaires à cette initialisation. L'association des paramètres effectifs aux paramètres formels se fait de gauche à droite en suivant la liste ; il doit y avoir le même nombre de paramètres à la définition de la procédure qu'à son appel.

Grâce aux paramètres une procédure peut avoir accès à des variables inconnues pour le bloc qu'elle constitue ceci permet aussi d'utiliser une procédure dans un contexte différent d'un appel à un autre.

2) Bloc

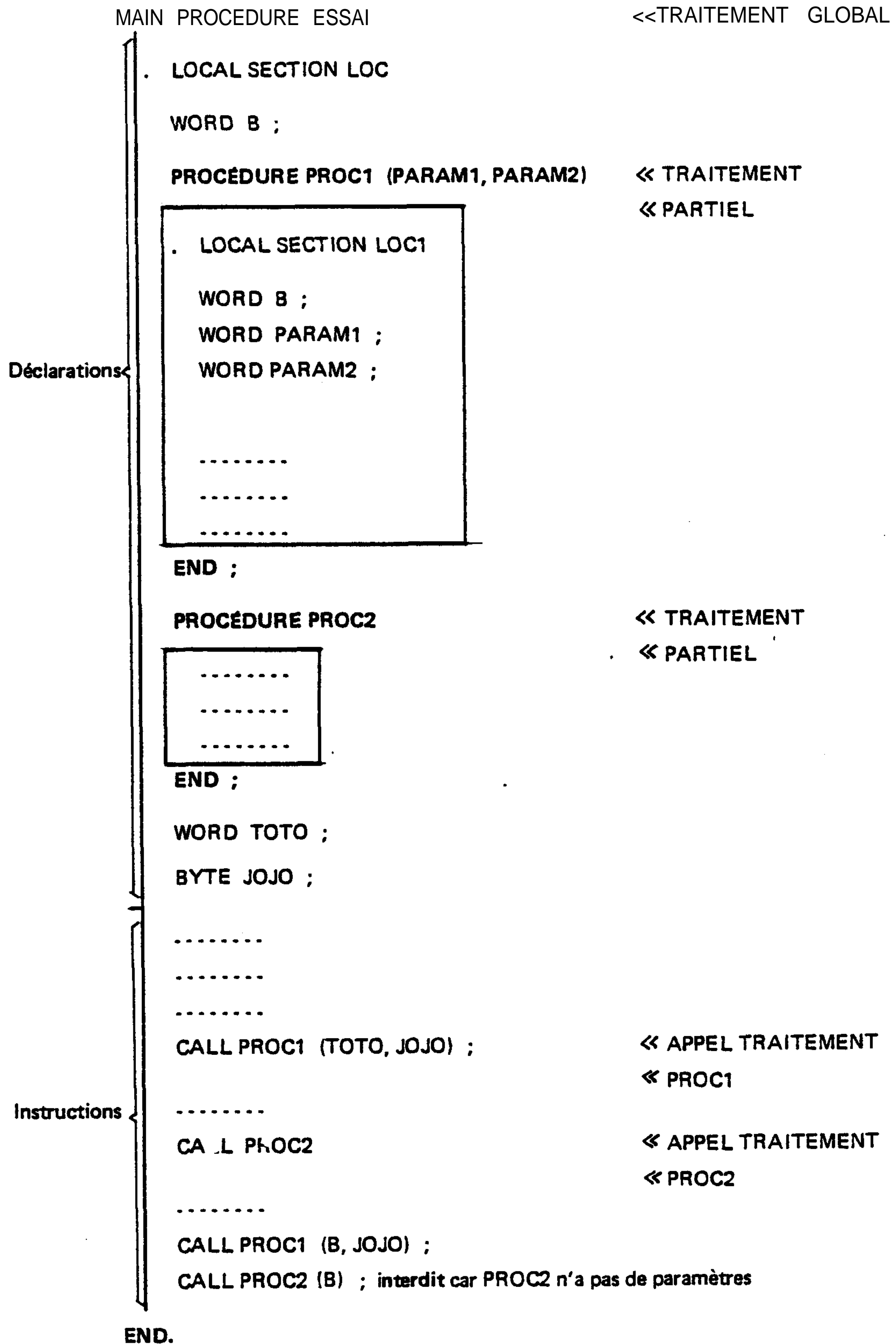
Les déclarations (facultatives) qu'il contient définissent les variables propres à la procédure : ce sont des paramètres, des données ou des variables de travail.

Les instructions travaillent sur ces variables et éventuellement sur des variables extérieures à la procédure mais accessibles en fonction de la structure de bloc.

3) Délimiteur END

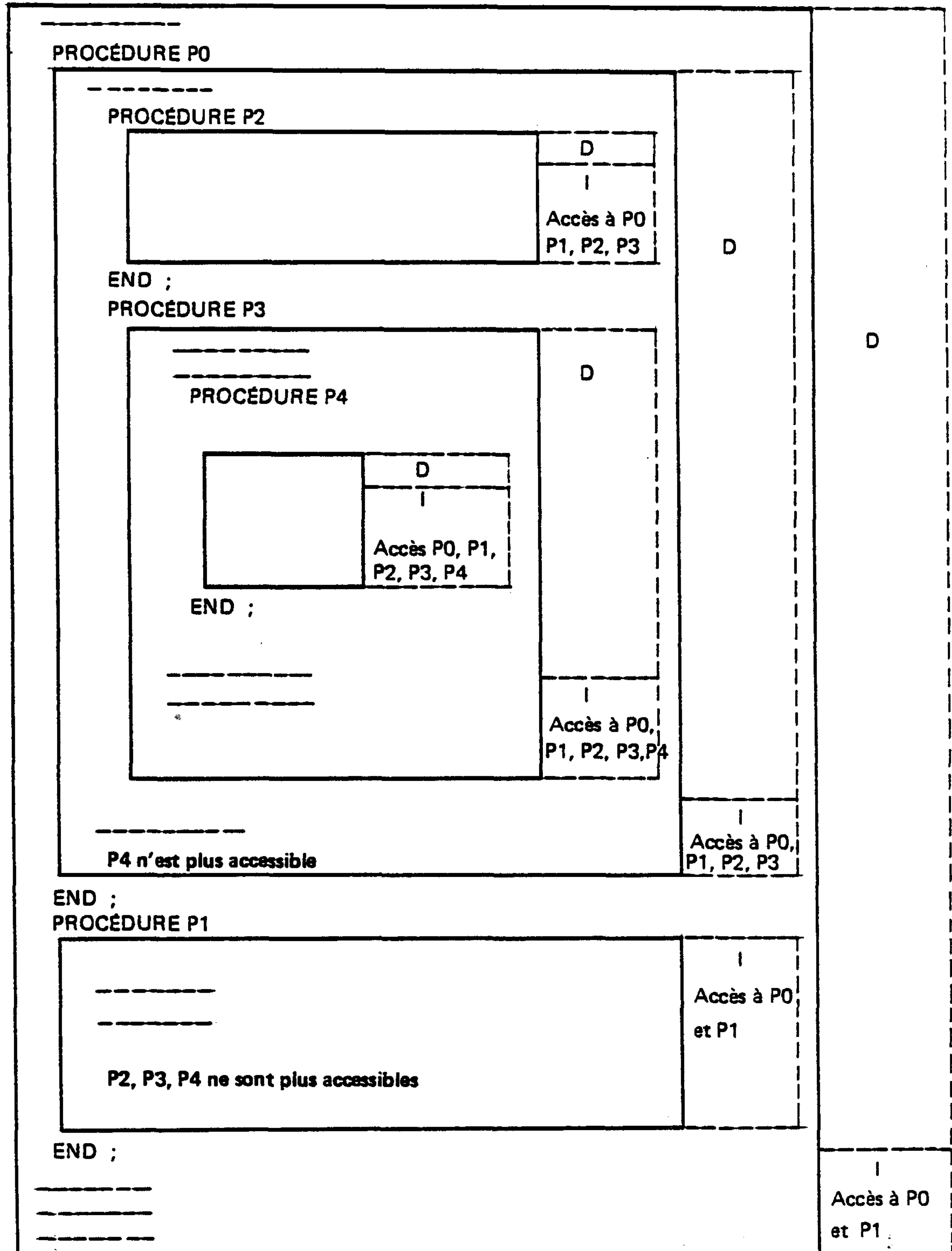
Il marque la fin de déclaration du traitement. Le compilateur génère un retour de procédure.

Exemple :



Comme nous l'avons déjà vu pour les variables simples, le nom d'une procédure est connu de tout le bloc où elle a été déclarée. Cette procédure est donc accessible en tout point de ce bloc.

MAIN PROCÉDURE ESSAI



END.

4.8 - DECLARATION DE KSTORE SECTION

Cette déclaration permet de réserver la pile associée au registre pointeur RK. Cette pile est nécessaire à l'exécution de tout programme et consiste en une zone réservée, dont l'adresse doit être chargée dans le registre RK avant lancement du programme.

En dehors de la déclaration de tâches (chapitre 8) cette initialisation est faite, au niveau d'une MAIN PROCEDURE, par une instruction "USING" (chapitre 5)

La pile KSTORE est utilisée par les instructions

- d'appel de procédure : sauvegarde adresse retour
- fin de procédure : retour vers l'adresse sommet de pile
- sauvegarde de registres
- restauration de registres.

Le compilateur PL16 génère le passage de paramètres de procédures par l'intermédiaire de la KSTORE

Déclaration :

Exemple :

```
MAIN PROCEDURE ESSAI
    • KSTORE SECTION PILE
      RES 100 ;
    • LOCAL SECTION LOC
      -
      -
      -
END.
```

Remarque :

Dans les déclarations la KSTORE section est considérée comme une section de donnée. Il y a donc lieu de veiller à l'ouverture ou la réouverture d'une véritable section de donnée, derrière cette déclaration.

5 - LES INSTRUCTIONS DU LANGAGE

5.1 - GÉNÉRALITÉS

Les instructions PL 16 apparaissent à la suite des déclarations éventuelles d'un bloc.

L'ensemble des instructions d'un programme constitue la "section instruction" de ce programme.

Les instructions PL 16 portent sur des registres, des variables, des constantes.

5.1.1 - LES REGISTRES

Tous les registres de SOLAR 16 sont prédéclarés, ils ont pour noms :

```
registre : : = RA / RB / RX / RY

registre long : : = RAB

registre flottant : : = RFL

             { RC     }   { RL    }   { RW      }
base : : =   { COMMON } / { LOCAL } / { WORKING }

                     { RK     }
pointeur pile : : =  { KSTORE }

registre début zone esclave : : = RSLO

registre fin zone esclave : : = RSLE
```

L'utilisateur peut cependant pour une meilleure compréhension du programme les redéclarer à tout moment par :

```
déclaration de registre : : = REGISTER nom SYN registre ;
```

Exemple :

```
REGISTER ACCUMULATEUR SYN RA ;
..........
ACCUMULATEUR : = 10 ;
```

Accès et affectation de registres :

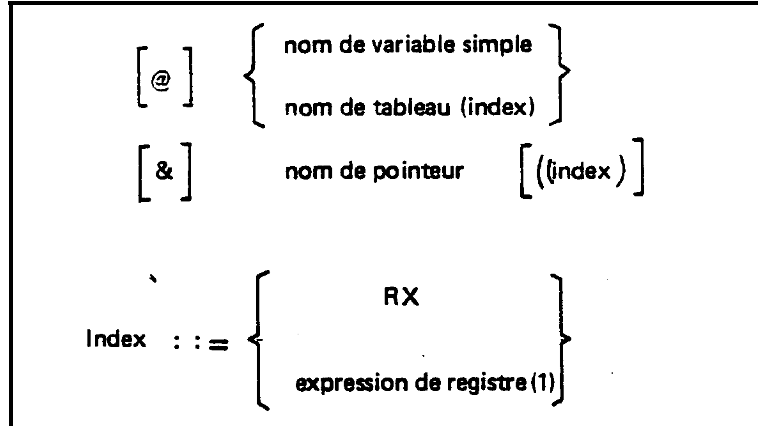
Toute apparition de registre dans toute instruction peut être remplacée par une instruction d'assignation de ce dernier mise entre parenthèses. Ceci ne sera toutefois vrai qu'à la condition que le registre n'apparaisse pas dans cette instruction ni comme index ni comme devant être assigné, mais y figure comme opérande.

5.1.2 - LES VARIABLES

Une variable peut être de l'un des types suivants :

- octet,
- mot,
- flottant,
- double mot (long).

. Forme générale de l'accès à une variable (cf. chap. 4.4.6 p. 43) :



Pointeurs :

Précédé du symbole d'indirection "&" une variable pointeur est de l'un des types cités plus haut. Non précédé de ce symbole, un nom de pointeur désigne une variable de type mot dont la valeur est une adresse.

Adresse :

Sauf cas particulier d'expression de registres, ou il s'agit de l'adresse effective, l'adresse d'une variable est considérée comme une constante comportant ou non le bit d'indexation (bit zéro) suivant qu'il s'agit d'un tableau ou non.

- @ A
- @ TABLO.

Eléments de tableaux :

Dans le cas de tableaux d'octets ou de mots, la valeur de l'index coïncide avec l'indice de l'élément (accès post-indexé SOLAR 16 aux mots et aux octets) pour les tableaux de variables LONG ou FLOAT il y a lieu de calculer l'index permettant l'accès au ième élément :

Exemples :

```
A
@B
TABLO (4)
& PTW
PTW
& PTABLO (4)
& PTABLO (RX + 2)
TABFLT ((RX : = 1 - 1 + RX)) ;     «ième élément flottant
TABFLT (RX + 2) ;                  «élément flottant suivant
```

(1) expression calculable dans l'un des registres RB, RX, RY (page 68).
L'utilisation de RA comme registre index est dangereuse. Elle est signalée par l'erreur '6F dans la version PL 1 164 001 01/.

5.1.3 - LES CONSTANTES

Les constantes sont représentées par un littéral (cf. chapitre 2.4) ou un nom de constante (cf. chapitre 4.3). L'utilisation d'une constante peut donner lieu à la génération, par le compilateur, d'un mot dans la section LOCAL accessible au moment de cette utilisation, si ce littéral ne s'écrit pas sur un octet. En l'absence de section LOCAL accessible il y a diagnostic d'une erreur.

- sauf cas particulier d'expression de registres, où il s'agit de l'adresse effective, l'adresse d'une variable est considérée comme une constante comportant ou non le bit d'indexation (bit zéro) suivant qu'il s'agit d'un tableau ou non.

5.1.4 - ACCES AUX INSTRUCTIONS

On distingue deux catégories d'instructions :

- les instructions simples réalisant une seule opération.
- les instructions composées réalisant plusieurs opérations.

Toutes les instructions d'un programme sont étiquettables par un identificateur. Celui-ci doit être en tête de l'instruction suivi du caractère deux points (:) ; il sera utilisé dans les instructions de branchement.

Exemples :

```
L1 : RA : = 10 ;                              «instruction simple
L2 :    DO FOR 1 : = 1 STEP + 1 UNTIL 10 ;    «instruction composée
        RA : = RA + 1 ;
        END ;
```

5.2 - INSTRUCTIONS COMPOSEES ET STRUCTURE DE BLOC

5.2.1 - INSTRUCTIONS COMPOSEES

Une instruction composée est délimitée par les symboles.:

BEGIN	--------	END ;
DO	--------	END ;
IF	--------	END ;

- BEGIN permet de grouper des instructions qui réalisent un traitement spécifique.
- DO est l'instruction de boucle
- IF est l'instruction conditionnelle.

Ces instructions définissent des blocs

BEGIN	bloc	END ;				
DO	bloc	END ;				
IF------	THEN	bloc	[	ELSE	bloc]	END ;

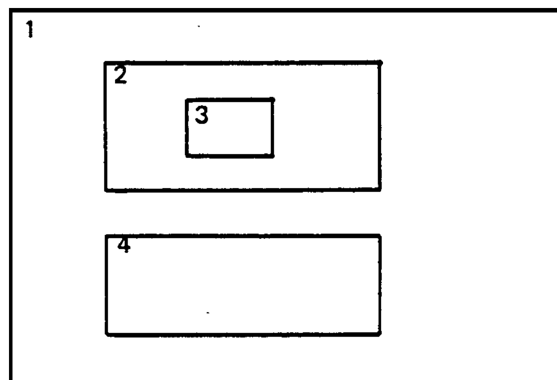
Une instruction composée est équivalente à une instruction simple en ce sens qu'elle peut être placée partout où l'on peut trouver une instruction simple. On le trouvera donc dans la partie instruction d'un bloc, ce bloc pouvant être défini par une MAIN procédure ou une autre instruction composée. C'est donc l'imbrication des procédures et des instructions composées qui donne à un programme PL16 sa structure de blocs.

5.2.2 - STRUCTURE DE BLOC

1) Blocs imbriqués

Deux blocs seront dits imbriqués si l'un contient l'autre ; deux blocs qui n'ont aucun point commun sont dits disjoints.

Exemple :



2) Ouverture et fermeture de bloc

Un bloc est ouvert après

- l'en-tête d'une procédure
- l'en-tète d'une instruction composée
- le symbole ELSE

Il est "ferme" par END ou ELSE.

Deux blocs ne peuvent donc pas se recouvrir car le premier END rencontré ferme le bloc précédemment ouvert :

5.2.3 - INSTRUCTION USING, CONTEXTE D'UN BLOC

Le contexte de travail d'un bloc est constitué par les noms des sections de données accessibles par ce bloc, c'est-à-dire par la valeur des registres base, dans ce bloc. Un contexte est défini par l'instruction USING

1) Instruction USING

L'instruction USING ne peut apparaître que comme première instruction d'un bloc : c'est la seule qui utilise les registres "base".

a) Elle précise pour le compilateur, les sections de données accessibles par ce bloc et en définit ainsi le contexte de travail.

b) Elle permet, éventuellement, de charger les registres de base avec les adresses des sections désirées.

Using clause : : = USING base	{ IS / = }	nom section	[. . . . . . .] ;

- IS ne charge pas la base, mais suppose que la base est déjà correctement chargée pour que la section de données concernée soit réellement accessible.
- = charge la base indiquée, de manière à rendre la section accessible (adresse de la section + 128).

La première instruction d'une procédure est toujours l'instruction USING, car c'est elle qui définit, pour le compilateur, le contexte de travail de cette procédure.

Pour les blocs plus internes définis par les instructions composées l'instruction USING n'est utilisée que pour changer de section de données accessibles, ou certains cas pour la clarté du texte. Si la clause est absente les registres de base sont considérés comme étant ceux du bloc extérieur.

Lorsque la clause nécessite le chargement des bases :

- leur anciennes valeurs sont sauvegardées dans la pile générale (pointée par le registre K) et restituées en fin des instructions du bloc à moins que ce bloc ne soit une MAIN procédure auquel cas ce n'est pas fait. Ainsi l'accès à une section est possible tant que le bloc, qui l'a rendue accessible par une instruction USING, n'est pas fermé.

C'est également par l'instruction USING que l'on peut initialiser le registre pile RK, mais seulement dans l'instruction USING d'une MAIN PROCEDURE, où une telle initialisation est d'ailleurs obligatoire pour le bon fonctionnement du programme.

2) Entrée dans un bloc

L'entrée dans un nouveau bloc est réalisée soit par l'appel d'une procédure, soit par l'exécution d'une instruction composée :

Remarque :

Les possibilités d'accéder à un bloc autrement que par sa première instruction sont évoqués aux paragraphes 5.6 et 5.8.

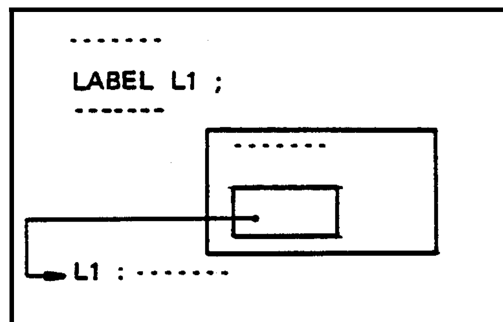
3) Sortie de bloc

Elle a lieu normalement lorsque l'exécution atteint l'instruction END. Le contexte de travail du bloc appelant est alors restauré si nécessaire.

Le contrôle passe alors :

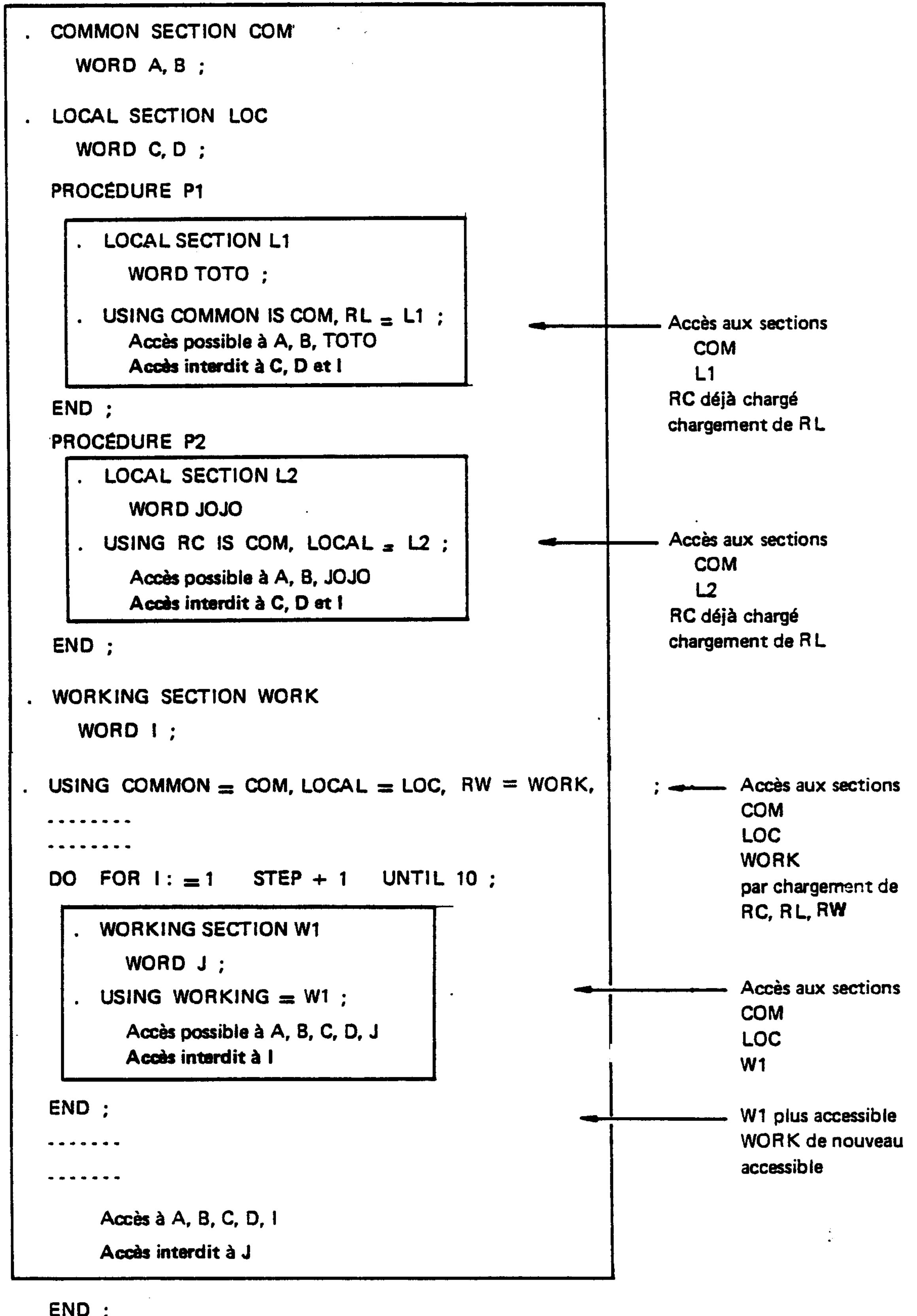
- à l'instruction suivante dans le cas d'une instruction composée
- à l'instruction suivant l'instruction d'appel pour une procédure
- au superviseur dans le cas d'une MAIN procédure.

Cette sortie peut aussi se faire par une instruction de transfert de contrôle intérieure au bloc vers une instruction extérieur au bloc. Une telle instruction peut alors faire sortir plusieurs blocs à la fois (cf. chapitre 5.6)



Exemple :

MAIN PROCEDURE ESSAI



Remarque : section de données et section d'instructions

Rappels :

- les données d'un programme PL16 sont organisées dans des sections de données
- les instructions du programme constituent pour le compilateur la section instruction.

Pour le compilateur la génération dans l'une ou l'autre de ces types de sections est commandée par l'apparition de déclaration ou d'instructions.

A l'ouverture d'un bloc, la section de travail est définie comme étant la section instruction, tout changement de section de travail doit être précédée du caractère (.)

Il en résulte les règles suivantes :

- toute ouverture de section de données par l'utilisateur doit être précédée du caractère (.)
- le point devant la première instruction d'un bloc n'est obligatoire que si cette instruction suit des déclarations.

Exemples :

cf. les exemples vus précédemment

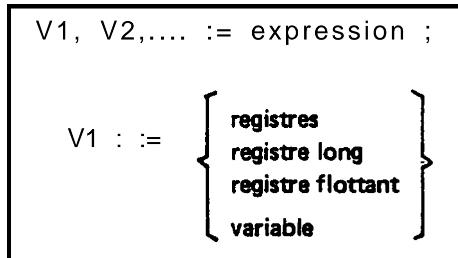
```
--------
  BEGIN
    • WORKING SECTION WORK
        WORD A, B  ;
    • USING WORKING =WORK  ;            «définition de contexte obligatoire
        -----                           «pour une procédure
            BEGIN
              • WORKING SECTION CONTINUE WORK
                  WORD C, D  ;
              • USING RW is WORK  ;     «USING facultatif ici
                  C := D * D  ;         «mais la première instruction
                  A := B - C  ;         «doit être précèdée de "."
                  -----
            END  ;
  END  ;
```

5.3 - INSTRUCTIONS D'ASSIGNATION

5.3.1 - GÉNÉRALITES

Une instruction d'assignation est définie par la présence du symbole := Elle consiste à affecter à la variable ou au registre apparaissant à gauche de ce symbole le résultat de l'évaluation de l'expression qui le suit

Forme générale :



Une expression est formée de termes et d'opérateurs :

Exemples :

a) A + B * (C + D OR MEM) NOT

b) - (A + B)

c) A + B NEG

d) Z + X * Y SARS 4

e) RA + RB

f) RAB/DIX

On trouve ici :

dans a) les opérateurs binaires + * OR
les termes A, B, MEM, (C + D OR MEM), C,D

dans b) la fonction " - "
les termes A, B, (A + B)

dans c) l'opérateur unaire NEG

dans d) l'opérateur "shift" SARS

dans e) les termes RA, RB

dans f) les termes RAB, DIX
l'opérateur binaire /

Une expression est évaluée de la gauche vers la droite sans priorité sur les opérations. Les priorités éventuelles sont alors imposées par les parenthèses. Ainsi l'exemple d) est différent de :

Z + (X * Y SARS 4)

Rappel : 1) par littéral, nous désignerons un nombre entier court et qui est représenté par sa valeur, ou par son nom s'il a été déclaré par CONSTANT
2) utilisée comme terme d'une expression, une chaîne ne peut excéder 2 caractères.

1) Types d'assignation

On distingue deux types d'assignation :

- assignation des registres
- assignation des variables
  . registre : = expression calculable dans le registre
  . variable : = expression parenthèse

L'expression de registre est caractérisée par :

- la notion de "premier terme" qui est l'information directement chargeable dans le registre.
- le fait qu'elle est calculable directement dans le registre affecté, ce qui implique des restrictions tant sur les termes suivants que sur les opérateurs utilisés.

L'expression parenthésée est caractérisée par :

- son type : entier ou flottant
- le fait que son écriture est indépendante des possibilités directes du calculateur, le compilateur générant le calcul de l'expression par l'utilisation de plusieurs registres.

2) Expressions

On définit ici les notions communes aux diverses formes d'expression utilisées. Les restrictions ou extensions à apporter suivant le type de l'instruction d'assignation réalisée sont détaillées dans les paragraphes suivants.

Une expression est écrite à l'aide :

- de termes
- d'opérateurs binaires
- d'opérateurs unaires

a) opérateurs binaires

+ - * /	opérateurs arithmétiques
AND OR XOR	opérateurs logiques "et" , "ou" , "disjonction"

b) opérateurs unaires

opérateurs de décalages

NEG	complément arithmétique
NOT	complément logique
SWAP	échange d'octets

C) forme générale d'une expression

Quel que soit le type de l'assignation une expression est ainsi définie :

expression ::= opérande OB [opérande OB opérande...]
opérande ::= terme [OU]
OB ::= opérateur binaire
OU ::= opérateur unaire

d) termes

Un terme d'une expression peut être :

- une variable (de tout type, indicée ou non)
- un littéral (nombre ou chaîne de caractère)
- une adresse
- un registre
- une expression elle même entre parenthèses (pour les expressions parenthésées)

e) évaluation d'une expression

L'expression étant évaluée de la gauche vers la droite sans priorité sur les opérateurs, il en résulta que :

- l'opération réalisée par un opérateur binaire porte sur :
  . le résultat de l'évaluation de l'expression qui le précède, d'une part
  . le terme suivant, d'autre part
- l'opération réalisée par un opérateur unaire porte sur :
  . le résultat de l'évaluation de l'expression qui le précède.

Exemple :

A + B NEG * D

L'opérateur unaire "NEG" réalise le complément à deux (négation) du résultat A + B

L'opérateur binaire "*" réalise le produit du résultat A + B NEG par le terme D.

Remarque sur l'opérateur : «—»
— Précédé d'un terme, le symbole «—» est :
L opérateur de SOUSTRACTION ex : A — B
— Précédé d'un opérateur ou du symbole : = , le symbole «—» est considéré comme une fonction portant sur le terme suivant : la nature de ce terme suivant étant fonction du type de l'expression.

Cette «priorité» accordée à l'opérateur «—», malgré la régle générale, a pour but de faciliter grandement l'écriture des expressions courantes.

Rappel sur l'emploi des opérateurs* et /

En programmation "machine" ces opérations introduisent les notions de :

- cadrage du premier opérande, sur RA ou sur RAB
- résultat, sur RA ou sur RAB

En PL16, ces notions doivent être connues et prises en charge par l'utilisateur dans le cas d'une affectation du registre RA pour laquelle la suite d'opérations à effectuer est entièrement dictée par l'utilisateur (cf. manuel de présentation SOLAR 16 multiplications et divisions).

Dans le cas d'une expression entière (affectation à un entier court) ou d'une expression de variable longue (affectation à RAB, ou variable LONG) l'utilisateur est déchargé de ces problèmes mais doit se souvenir que le résultat d'une multiplication est :

. un entier court (16 bits) pour une expression entière.

. un entier long (32 bits) pour une expression de variable longue.

Affecté à une variable longue, le résultat d'une division est considéré comme la suite de 2 entiers courts : quotient et reste.

Ces remarques sont résumées dans le tableau suivant où l'on trouve dans chaque cas, les opérations effectivement réalisées :

Type d'assignation	Opération à enchaîner: multiplication	Opération à enchaîner: division
du registre RA	multiplication	division
du registre RAB ou de variable "LONG"	1) cadrage sur RA si résultat précédent de type "LONG" 2) multiplication le résultat est sur RAB	1) cadrage sur RAB, à droite si résultat précèdent de type "COURT" 2) division résultat dans RA, reste RB
de variable entière	1) multiplication 2) cadrage sur RA du résultat	1) cadrage sur RAB, à droite 2) division le résultat est sur RA.

5.3.2 - ASSIGNATION DES REGISTRES

On traite de façon distincte le cas du registre RA d'une part et celui des autres registres assignables d'autre part (RB, RX, RY) pour tenir compte des possibilités propres à l'accumulateur.

Par la suite on parlera donc "d'expression de registre" pour désigner une expression assignable aux registres RB, RX, RY et "d'expression de RA" pour l'accumulateur.

D'autre part le registre double RAB (accumulateur étendu) de même qu'une variable de type LONG peuvent être affectés par des expressions d'un type particulier.

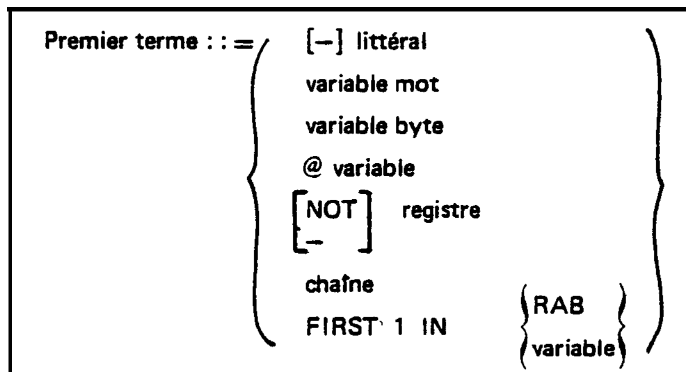
1) Assignation de l'accumulateur

Forme générale :

RA : = expression RA ;

Le premier terme est chargé dans RA puis de la gauche vers la droite chaque opérateur rencontré opère sur RA ou sur RA et le terme suivant.

On distingue le terme chargé dans l'accumulateur, c'est-à-dire le premier terme de l'expression des termes suivant.



variable byte : est cadrée à droite dans RA et précédée de zéro

@ variable : RA est chargé avec l'adressa effective à la variable désignée ; en particulier, si cette variable est un tableau il s'agit de l'adresse effective du mot désigné par l'indice qui est obligatoire ici, et qui est un indice de mot. On notera la différence avec la notion de "constante adresse" utilisée dans les expressions entières (page 73).

FIRST 1 IN... RA est chargé par le rang du premier bit à 1 rencontré sur l'accumulateur étendu RAB, après chargement éventuel de la variable désignée. Cette opération normalement possible sur l'index RX seul, est générée par le compilateur en passant par le registre RX.

sauvegarde de RX	SAVE (RX) ;
chargement RX par le rang	RX : = FIRST 1 IN ;
chargement de RA par RX	RA : = RX ;
restauration de RX	RESTORE (RX) ;

Opérateurs binaires et termes :

opérateur binaire	::=	opérateur arithmétique / opérateur logique
opérateur arithmétique	::=	+ - * /
opérateur logique	::=	AND [NOT] / OR / XOR
Terme	::=	{ [—] littéral / variable mot / registre / carry / chaîne }

- CARRY n'est possible qu'avec les opérations d'addition et de soustraction (+ et -).
- registre est interdit après l'opérateur de multiplication *
- AND NOT n'est permis que si le terme avec lequel il opère est un registre.

Enfin les opérateurs unaires sont :

opérateur unaire	::=	NEG / NOT / SWAP /
		op shift litteral [(index)] (1)
opérateur shift	: : =	SARS / SLLS / SLRS / SCRS / SCLS

Exemples :

RA : = A + B NEG ;	
RA : = NOT RY ;	
RA : = A + A * - 3 NOT;	
RA : = RA +CARRY XOR RX ;	
RA : = TAB (RX) * 3 OR '8000 ;	
RA : = RA * 5;	
RA : = RA AND NOT RB SLRS 4 (RX) ;	
RA : = RY SWAP ;	$RA_{0-7} := RY_{8-15}, RA_{8-15} := RY_{0-7}$
RA : = @TAB (2) + RY ;	l'index (2) est obligatoire pour un tableau
RA : = '7FFF ;	
RA : = TAB (3) + 5 ;	

Ecritures interdites

RA : = @TAB + 2 ;	TAB étant un identificateur de tableau ; il manque l'index (cf. page 66)
RA : = 2 + @TAB (2)	TAB (2) ne peut être qu'un premier terme car une adresse ne peut être que seulement chargée dans RA.

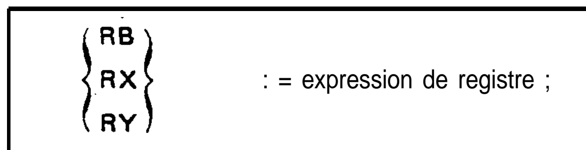
Remarque commune à tous les termes :

Littéral : s'il est supérieur à 127, ou multiplicateur, ou diviseur, il est généré dans la section locale accessible

Chaîne : est limité à deux caractères.

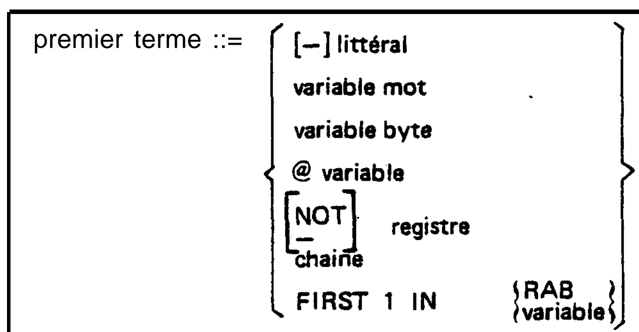
(1) index (cf. page 55)

2) Assignation des registres RB, RX, RY



Le premier terme est chargé dans le registre, puis de la gauche vers la droite chaque terme et opérateur rencontré opérant sur le registre.

- Le premier terme d'une expression de registre, est le même que celui d'une expression de RA,



FIRST 1 IN... même remarque que pour l'accumulateur RA, quant à l'affectation de ce terme aux registres RB, RY

@ variable
variable byte : ces termes ne sont chargeables directement que dans l'accumulateur RA pour les autres registres le compilateur génère le chargement en utilisant l'accumulateur.

sauvegarde RA

chargement de RA

chargement du registre par RA
restauration de RA

@ variable : même remarque que pour l'accumulateur : il s'agit d'une adresse effective, c'est-à-dire que l'indice est obligatoire dans le cas d'une variable tableau.

opérateur binaire : : =	opérateur arithmétique / opérateur logique
opérateur arithmétique : : =	{ + / − }
opérateur logique : : =	AND [NOT] / OR / XOR
terme : : =	{ chaîne / registre / [−] littéral / CARRY }

- chaîne ne comporte ici qu'un seul caractère et n'est possible qu'associé à un opérateur arithmétique
- littéral n'est possible que pour les opérations arithmétiques et si sa valeur absolue est inférieure à 127.
- CARRY n'est possible qu'avec les opérations arithmétiques.

Les opérateurs unaires sont :

opérateur unaire ::= NOT / NEG / SWAP

Exemples :

RY : = A + (RB := B) AND RX ;
RB : = RB + CARRY ;
RX : = A - (RY : = B) + 5
RX : = FIRST 1 IN A ;
RA : = FIRST 1 IN RAB ;

Ecritures interdites

RY : = RB − RX OR '7F ;	opérations logiques entre registres seulement
RK : = RA ;	seuls RA, RB. RX, RY peuvent intervenir dans une assignation
RX : = @TAB (3) + A ;	Addition de la variable A impossible sur RX.

Remarques :

1) Les extensions de "expression RA" par rapport à une "expression de registre" portent sur :
- les opérateurs binaires * et /
- les opérateurs de décalage
- un terme peut être une variable ou un littéral supérieur à 127.

Exemple :

L'expression

RX + RY + A * 3 peut être assignée à RA mais pas à RY.

2) Dans une programmation avec registres, l'utilisateur doit toujours avoir à l'esprit l'ordre d'évaluation d'une expression. Nous pouvons ainsi voir sur un exemple simple que :

- RY : = RA + RY ;

se traduit par RY : = RA

et RY : = RY + RY

le résultat final sera RY = 2 * RA

Alors que :

RY : = RY + RA ;

est l'affectation à RY de la somme RY + RA

3) Assignation de l'accumulateur étendu et de variable long-

RAB : =
LONG

{variable byte / variable mot / variable longue / RAB} [Ob Opérande.....]

avec

Opérande ::= terme OU

terme ::= variable mot / littéral

Ob ::= * / /

ou ::= opérateur shift littéral [(indice)]

opérateur shift ::= SARD / SLLD / SLRD / SCRD / SCLD.

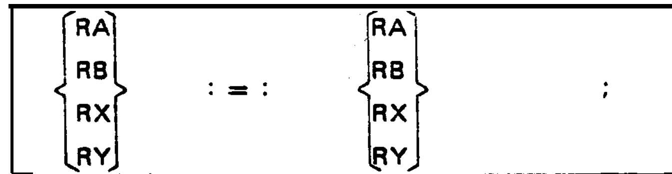
Exemples :

RAB : = RAB/DIX ;

RAB : = VL SCRD 16 (A) ; A étant une variable mot chargeable dans RX

RAB : = TABFLAGS ;

4) Echange de registres



Les contenus des registres cités sont échangés

Exemples :

RA :=: RB ;
RX :=: RY ;

5.3.3 - ASSIGNATION DE VARIABLES

Forme générale :

variable : = expression Parenthésée ;

où variable peut être :

- une variable mot
- une variable octet
- une variable flottante

Le cas des variables longues ayant été vu précédemment.

- l'écriture d'une expression parenthésée est indépendante des possibilités directes du calculateur, elle ne comporte donc pas les restrictions d'une expression de registre.
- l'évaluation d'une expression parenthésée se fait de gauche à droite avec priorité seulement sur l'évaluation des termes parenthésés et des fonctions.

L'évaluation d'une expression parenthésée utilise :

- les registres RA, RB, RY
- la pile générale (KSTORE section)
- le premier mot, ou les deux premiers mots de la section LOCAL accessible, suivant que l'expression est de type entier ou flottant.
- le premier mot du "common" pour le compte rendu d'arithmétique flottante

Ces mots devront donc être réservés par l'utilisateur (par RES)

- les notions caractéristiques de cette forme d'expression sont :

  . son type, entier ou flottant
  . les parenthèses (d'où le nom de l'expression)
  . les fonctions

Un terme d'une expression parenthésée peut être :
- une expression entre parenthèses : (A + B)

Un opérateur d'une expression parenthésée peut être un opérateur de type "Fonction", agissant sur l'expression entre parenthèses qui suit : IFIX (A + B)

Outre les fonctions propres à l'arithmétique flottante (IFIX, FLT, NORM, FABS) les opérations de complément logique et de complément arithmétique peuvent être réalisées aussi bien par des fonctions que par des opérateurs unaires :

On écrira : "expression entière" pour : "expression éventuellement parenthésée portant sur des entiers" et "expression flottante" si l'expression porte sur des variables flottantes.

Exemple :

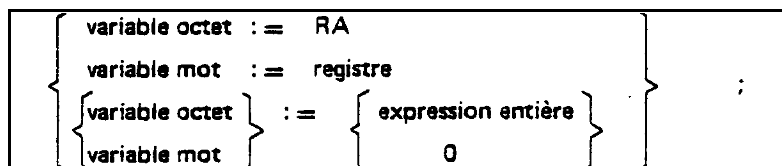
Les écritures suivantes conduisent au même résultat :

(A + B) NOT	opérateur unaire sur résultat précédent
NOT (A + B)	fonction NOT sur terme parenthésé

de même :

(A + B) NEG	opérateur unaire sur résultat précédent
- (A + E)	fonction - sur terme parenthésé

1) Assignation de variable courte



On peut ainsi soit charger une variable avec le contenu d'un registre, soit la charger avec le résultat d'une expression soit la remettre à zéro, zéro ne peut donc être le premier terme d'une assignation.

Les termes d'une expression parenthésée entière peuvent être :

<table>
<tr><td rowspan="7">terme ::=</td><td>[fonction] (expression entière)</td></tr>
<tr><td>variable octet</td></tr>
<tr><td>variable mot</td></tr>
<tr><td>[−] littéral</td></tr>
<tr><td>@ variable</td></tr>
<tr><td>CARRY</td></tr>
<tr><td>chaîne</td></tr>
</table>

- @ variable. Une constante translatable initialisée par l'adresse de la variable est alors générée dans la section LOCAL accessible qui doit exister. Dans le cas d'un tableau cette constante comporte le bit index, de plus on ne spécifiera pas d'indice.

- variable octet n'est permis que comme premier terme de l'expression.

Les opérateurs et fonctions sont :

opérateurs binaires	: :=	+ - / x AND OR XOR
opérateurs unaires.	: :=	NEG / NOT / SWAP / opérateur shift
opérateur shift	: :=	SARS / SLLS / SLRS / SCRS / SCLS
fonction	: :=	{ NOT / − }

Exemples :

```
.......
  . LOCAL SECTION L1
      RES 1 ;          << RESERVATION 1 MOT POUR EXPRESSION ENTIERE
      WORD A, B, C, D, E, F ;
      POINTER WORD PTW ;
      ARRAY 5 WORD  TAB ;
      POINTER ARRAY WORD PTAB ;
  . USING LOCAL = L1 ;

      A := A + B * C ;        }  identiques
      A := (A + B) * C ;      }  différentes
      A := A + (B * C) ;      }

      A := - (A + TAB (2)) ;

      RX := A + 1 ;           }  équivalent à B := TAB((RX := A + 1));
      B  := TAB (RX) ;        }

      A  := - (A + B) * - (C + D) / E      SLLS  4  ;

  TAB (2)  := NOT (A AND B) OR NOT ( C AND D) NOT ;
  TAB (1)  := (A + B * (C + D)) - (E - F) ;

PTAB  := 2 + @ TAB ;            la variable A contiendra l'adresse du tableau TAB (avec
                                le bit index) augmentée de 2. Ceci pourrait être par exemple
                                l'initialisation du pointeur tableau PTAB

PTW   :=@ TAB + 3 AND '7FFF ;  le bit index est ici éliminé ; PTW est ainsi initialisé avec
                                l'adresse du 4e élément de TAB.

& PTAB (1) := - (A + B) * NOT (D) ;
  & PTW    := - 4;
    A      := A NEG * - (C OR D) ;     }  identiques
    A      := -(A) * - (C OR D) ;      }
& PTAB (3) := A XOR B ;
```

Ecritures interdites

```
    A    := 2 +    TAB (2)          l'indice est interdit dans un t me "adresse" d'une
                                                expression entière
    A    := A + B +@ TAB + 5 + RX
                                    l'apparition d'un registre est interdite
    A    := A NEG + OCT             si OCT est un octet.
```

Remarque :

Il faut distinguer :

- la fonction négation qui s'écrit "-" et qui précède un terme parenthésé
- l'opérateur binaire "-" qui opère sur 2 termes
- le signe "-" qui peut précéder un littéral.

Exemples :

RA := - 5	nombre "- 5"
RA := C - 5	opérateur binaire soustraction
A := - (C + D)	fonction négation

C'est pour une raison de facilité que l'écriture "- 5" est permise, il en résulte les conséquences suivantes :

L'écriture: A := - B est autorisée ;

les écritures :

C := A + (-B)
C := A + (- (B))
C := A + - B sont équivalentes

mais, pour raison de clarté, la deuxième écriture est conseillée.

2) Assignation de l'accumulateur flottant et de variable flottante

L'accumulateur flottant, RFL, est constitué par les registres RA et RB, il est donc équivalent au registre RAB, mais syntaxiquement l'usage de "RFL" est obligatoire.

Assignation de RFL : ou de variable flottante	: : =	{ RFL variable "FLOAT" }	: = expression flottante ;
expression flottante	: : =	terme [opérateur terme ...]	
terme	: : =	{ [fonction] (expression flottante) variable flottante }	
Opérateur	: : =	+ - * /	
fonction	: : =	FNEG / FABS / IFIX / FLT / NORM	

- le premier terme d'une expression flottante peut être RFL
- les fonctions n'opèrent que lorsque l'expression qui suit a déjà dévaluée dans RFL, ce sont :

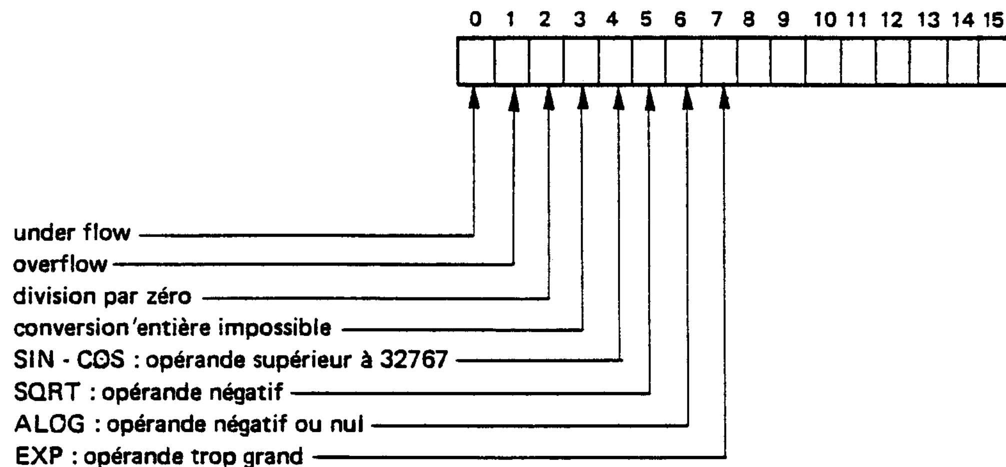
FNEG	négation de RFL
FABS	valeur absolue de RFL
IFIX	conversion flottante, entière - le résultat est dans RA
FLT	conversion entière, flottante - l'opérande est supposé être dans RA.
NORM	normalisation de RFL - (voir manuel présentation SOLAR 16)

- dans le cas de la fonction FLT, l'expression entre parenthèse qui suit ne peut être que RFL.

Remarques :

1) Les fonctions standard de la bibliothèque flottante sont accessibles comme toute procédure externe.
2) L'opérateur "-" est ici uniquement l'opérateur binaire, de la soustraction
3) Rappelons que la programmation de calcul portant sur des variables flottantes nécessite :
   - la réservation en tête du local de 2 mots (RES 2 ;) nécessaire à l'évaluation des expressions
   - la réservation en tête du "common" d'un mot qui sert d'indicateur rémanent à l'arithmétique flottante

indicateur rémanent :



Exemples :

```
          . . . . . . . . .
     .    LOCAL SECTION L1
               RES 2 ;                          «RESERVATION 2 MOTS POUR
               WORD A ;                         «CALCUL EXPRESSION FLOTTANTE
               FLOAT F1, F2, F3, F4, F5, F6, F7 ;

     .    USING LOCAL = L1 ;
               RA     :=  A ;
               F2     :=  FLT (RFL) ;
               F1     :=  FNEG (F2 + F3) * F3 ;
               F3     :=  F1 + F2 - F4 ;
               F1     :=  FABS (F1 * F2) / F3 ;
               F1     :=  FABS (F3 * (F4 + F5) x (F6 + F7) ) - F1 ;
               RA     :=  A ;
               RFL    :=  FLT (RFL) + F1 + F2 ;
               RFL    :=  FABS (F1) ;
               RFL    :=  FNEG (F1) ;
               RFL    :=  IFIX (F1) ;           } Assignation à la variable A
               A      :=  RA ;                  } de la partie entière de F1
               RFL    :=  NORM (F1) ;
               RFL    :=  RFL + FLT (A) ;
               RFL    :=  FNEG (ABS (F1 + F2) - F3) ;
               RFL    :=  F1 * F2 / F3 ;
```

Remarque générale :

Dans une assignation multiple :

V1, V2,... Vn : = expression

Le type de l'expression située à droite doit être conforme au type de la variable Vn précédant immédiatement le symbole (:=)

Si Vn est un registre on doit trouver une expression de registre, si Vn est une variable courte on doit trouver une expression entière...

Exemples :

A, B, RA	: =	expression RA	
A. B, RY	: =	expression de registre	
RA, A, B	: =	expression entière	
RY, A	: =	expression entière	
RFL, F1	: =	expression flottante	
A, RA	: =	RA + CARRY ;	
RA, A, B	: =	RA + CARRY ;	INTERDIT car ce n'est pas une expression entière
A, RA	: =	@TAB (2) + B + RX ;	
RA, A	: =	@TAB + B ;	
RA, A	: =	@TAB (2) + 6 ;	INTERDIT car l'index n'est pas autorisé dans un terme adresse d'une expression entière
RA, A	: =	@TAB + B + RX ;	INTERDIT car un registre ne peut apparaître dans une expression entière
RY, RA	: =	A + B + 2 ;	
RA, RY	: =	A + B + 2 ;	INTERDIT car cette expression n'est pas une expression de registre

par contre,

RA, RY	:=	A + (RX : = B) + 2 ;	est autorisé car on est alors en présence d'une expression de registre correcte.

5.3.4 - POSITIONNEMENT D'INDICATEURS, TRAITEMENT DU BIT

L'instruction :

```
SET CARRY ;
```

permet de positionner à 1 l'indicateur IC (carry) du registre d'état du calculateur.

Les instructions :

{ SET / RESET / NEGATE }	BIT littéral	[(index)] OF	{ variable WORD / variable LONG / RAB }	;

permettent respectivement de positionner à 1, à 0 ou de complémenter un bit d'une variable mot, longue ou du registre RAB.

Exemples :

```
SET      BIT   10      OF   A;      A étant un WORD
SET      BIT   31      OF   C;      C étant un LONG
RESET    BIT   16 (A)  OF   RAB ;
NEGATE   BIT   18      OF   C;
```

Si l'identificateur B est un byte l'écriture :

```
SET      BIT   2       OF   6;      est INTERDITE.
```

Remarque :

L'un seulement des termes de cette instruction peut être indexée.

```
SET   BIT   O (I)   OF   A       «correct
SET   BIT   O       OF   T (j)   «correct
SET   BIT   O (I)   OF   T (j)   «incorrect
```

5.4 - INSTRUCTION CONDITIONNELLE : IF

L'instruction IF est une instruction composée, comportant la définition de 1 ou 2 traitements partiels dont l'exécution est conditionnée par le résultat d'un test apparaissant en tête de l'instruction.

Forme générale :

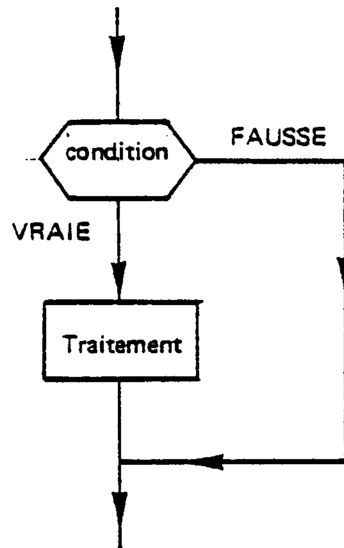
IF condition THEN bloc [ELSE bloc] END ;

L'instruction conditionnelle se présente donc sous l'une des deux formes suivantes :

* IF condition THEN bloc END ;

qui spécifie l'exécution conditionnelle du traitement défini dans le bloc

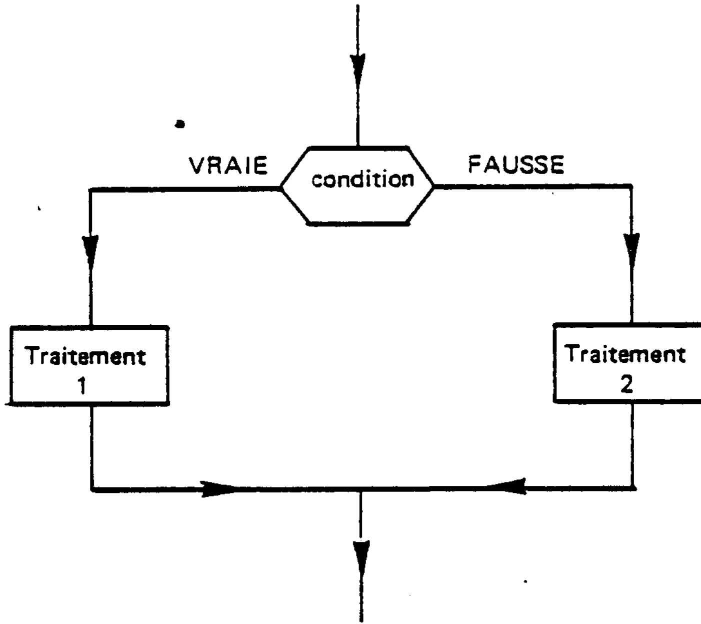
Si condition alors faire traitement sinon l'ignorer.



* IF condition THEN bloc1 ELSE bloc2 END ;

qui spécifie que si la condition est satisfaite le premier traitement sera exécuté tandis que le second ignoré. Si la condition n'est pas satisfaite le premier traitement sera ignoré et le second exécuté.

Si condition alors faire traitement1 sinon faire traitement2



La condition pouvant apparaître dans l'instruction est définie comme suit :

condition ::= { condition AND | condition OR }

condition AND ::= (condition simple) [AND (condition simple) AND . . .]

condition OR ::= (condition simple) [OR (condition simple) OR . . . .]

5.4.1 - CONDITIONS SIMPLES

a) sur registres

L'écriture de ces conditions est liée aux possibilités directes du calculateur, ce sont :

RA opérateur de relation { [−] { 0 | littéral | Word | byte | registre | chaîne } } /

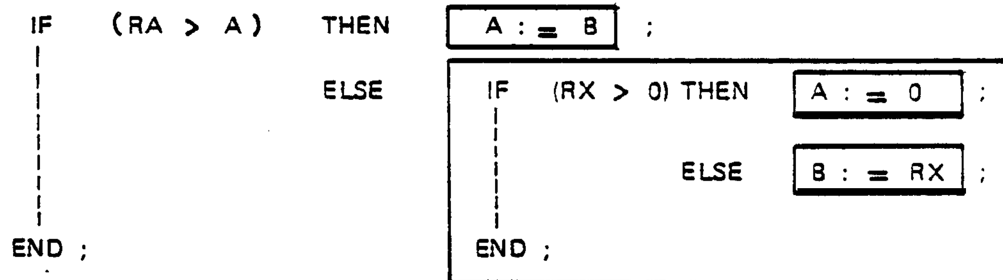
{ RB | RX | RY } opérateur de relation { registre | 0 }

opérateur de relation ::= { >= | > | = | <= | < | /= }

Elles permettent donc de comparer :

- l'accumulateur à un registre, à 0, à un littéral ou à une variable de type mot ou octet.
  Un littéral supérieur à 255 est généré dans la section locale accessible ; si celle-ci est complète ou n'existe pas il y a erreur.
- un registre simple quelconque à un autre registre ou à 0.

Exemples :




```
IF (RY > RX)    THEN    RY := 0 ;  END ;

IF (RA > 8)     THEN    RA := RA – 8 ;  END ;
```

L'écriture :

IF (RX > 8) THEN .... est interdite car une telle comparaison ne peut être faite sur le SOLAR.

b) sur variables entières

Les possibilités d'écriture sont plus générales.

Elles permettant de comparer :

. une variable de type mot ou octet à 0 ou à une expression de variable entière (voir assignation de variable courte).

Exemples :

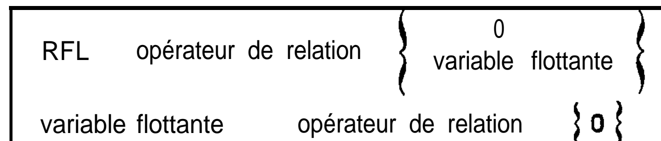
```
IF      (V1 < V2)   THEN    V1 := MEM ;
|
|                   ELSE    IF   (V1 > 0)   THEN    V1 := 1 ;
|                           |
|                           |               ELSE    V1 := 0 ;
|                           |
|                           END ;
|
END ;

IF      (A = B + C * (D - E))     THEN ...........      END ;
```

L'écriture IF (A / = RB) THEN .... END ;

et peut être remplacée par :

```
IF      ((RA := A) / = RB)     THEN .............      END ;
```

C) sur variables flottantes et registre flottant



RFL opérateur de relation { 0 / variable flottante }

variable flottante opérateur de relation { 0 }

Elles permettent respectivement de comparer :

- l'accumulateur flottant à 0, à une variable flottante
- une variable flottante à 0

Exemples :

```
IF      (RFL > 0)     THEN     F1 := F1 + F2 ;
                      ELSE     F1 := F1 - F2 ;
END ;

IF      (RFL = F2)    THEN     F3 := F2 - F1 END ;

IF      (F1 / = 0)    THEN     RFL := F1 ;
                      ELSE     RFL := F2 ;

END ;
```

d) test d'indicateurs :

```
Indicateur ::= [NOT]   CARRY
               [NOT]   OVERFLOW                     ⎧ variable LONG ⎫
               [NOT]   BIT littéral   [index]   OF  ⎨ variable WORD ⎬
                                                    ⎩ RAB           ⎭
```

Exemples :

```
IF  (CARRY)    THEN  RA := 2 ;
               ELSE  IF (NOT OVERFLOW) THEN RA : = 1 ;
                                       ELSE RA := 3 ;
                     END ;
END ;

IF  (BIT 2 (A) OF V)   THEN   .......   END ;
IF  (BIT 5 OF RAB)     THEN   .......   END ;
IF  (BIT 5 (RX) OF V) THEN    .......   END ;
IF  (NOT BIT 6 OF TAB (0))  THEN  ......  END ;
```

Remarques :

1) voir aussi, le texte de l'indicateur après certaines instructions au chapitre 5 paragraphe 2.

2) comme pour les instructions SET et RESET, l'emploi de l'index pour le test sur BIT est permis seulement pour l'un des opérandes, à la fois. Ainsi, l'écriture :

IF (NOT BIT 6 (RX) OF TAB (0)) . . . . . . est interdite.

5.4.2 - CONDITIONS COMPOSEES AVEC OR ET AND

La condition testée peut être l'union ou l'intersection de plusieurs conditions simples mais pas une combinaison des deux.

Exemples :

```
IF    (A = B)  OR  (RX = RY)    THEN      .......     END ;

IF    (CARRY)  OR  (OVERFLOW)THEN         .......     END ;

IF    (RA > 0)  OR  (RB > 0)  OR  (RX < 0)   THEN    A :=0 ;
|                                            ELSE
|              IF  (RA < 0)  AND  (RB > RY)   THEN   B := 0 ;
|              |                              ELSE   A := B ;
|              END ;
END ;

IF    ((RA := 100) > RX)  AND  (RX / = 0)    THEN    .......    END ;
IF    (V1 > V2)  OR  (V1 < V3)  OR  (V1 < V4)  THEN  V1 := 0 ;
|                                            ELSE
|          IF  (V1 = 'F)  AND  (V3 = V4)  AND  (V4 / = 0)  THEN  V2 := 1 ;
|          |                                               ELSE
|          |                    V1, V2 := V3 ;
|          END ;
END ;

IF    (& PTA <0)  OR  (& PTAB (2) >0)   THEN    ......     END ;
IF    (A = (B + C)  /  (D + 2))  AND  (F = 0)    THEN    .......    END ;
```

L'écriture suivante :

IF (A = B) OR (C = D) AND (A > C) THEN ....END ;

est ainsi interdite.

5.5 - INSTRUCTION DE CHOIX : CASE OF

```
CASE { RX           } / littéral OF instructions END ;
     { variable mot }
```

où littéral est le nombre d'instructions comprises entre OF et END.

L'instruction exécutée est celle dont le numéro d'ordre est égal à la valeur de RX ou de la variable.
- si la variable ou le registre à la valeur k, c'est la kième instruction qui est exécutée ;
- si la valeur est nulle toutes les instructions sont ignorées.

Une instruction composée compte pour une instruction

Exemples :

```
CASE (I) /4 OF
        A := B + C ;                        ] 1e instruction
        BEGIN                               ]
            CALL PROC (A) ;                 ] 2e instruction
            B := A + C ;                    ]
        END ;                               ]
        IF (A <B) THEN B := B + 1           ]
                  ELSE I := J ;             ] 3e instruction
        END ;                               ]
        A := 0 ;                            ] 4e instruction
END ;
```

Cette séquence est équivalente, par exemple, à :

```
IF    (I = 1)   THEN    A :=B + C ;
|               ELSE    IF   (I = 2)   THEN CALL PROC (A) ;
|                       |                   B := A + C ;
|                       |              ELSE IF    (I = 3)
|                       |                   |                  etc . . .
|                       |                   END ;
|                       END ;
END ;
```

6.6 - INSTRUCTIONS DE BRANCHEMENT

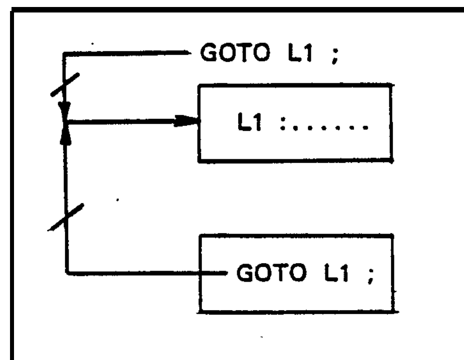
En permettant de se renvoyer à telle instruction ou telle séquence d'instructions, étiquetées ou non, elles rompent l'exécution séquentielle d'un programme.

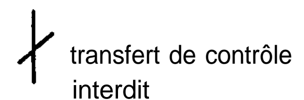
```
{ GOTO      nom de LABEL direct [ON condition OR]          }
{ GOTO      nom de LABEL indirect                          }
{ {EXIT }                                                  } ;
{ {CYCLE}   [ nom de LABEL ]   [ON condition OR]           }
{ EXIT      nom de procédure                               }
```

. l'instruction GOTO permet un branchement direct ou indirect à une instruction repérée par une étiquette.

Lorsque l'instruction GOTO porte sur une étiquette déclarée indirecte il y a branchement indirecte, branche ment est direct (relatif à l'instruction courante) dans les autres cas.

Les étiquettes, comme les autres identificateurs, sont inconnues à l'extérieur du bloc où elles sont définies ; il est donc impossible d'entrer dans un bloc par une instruction GOTO à une étiquette définie dans ce bloc :





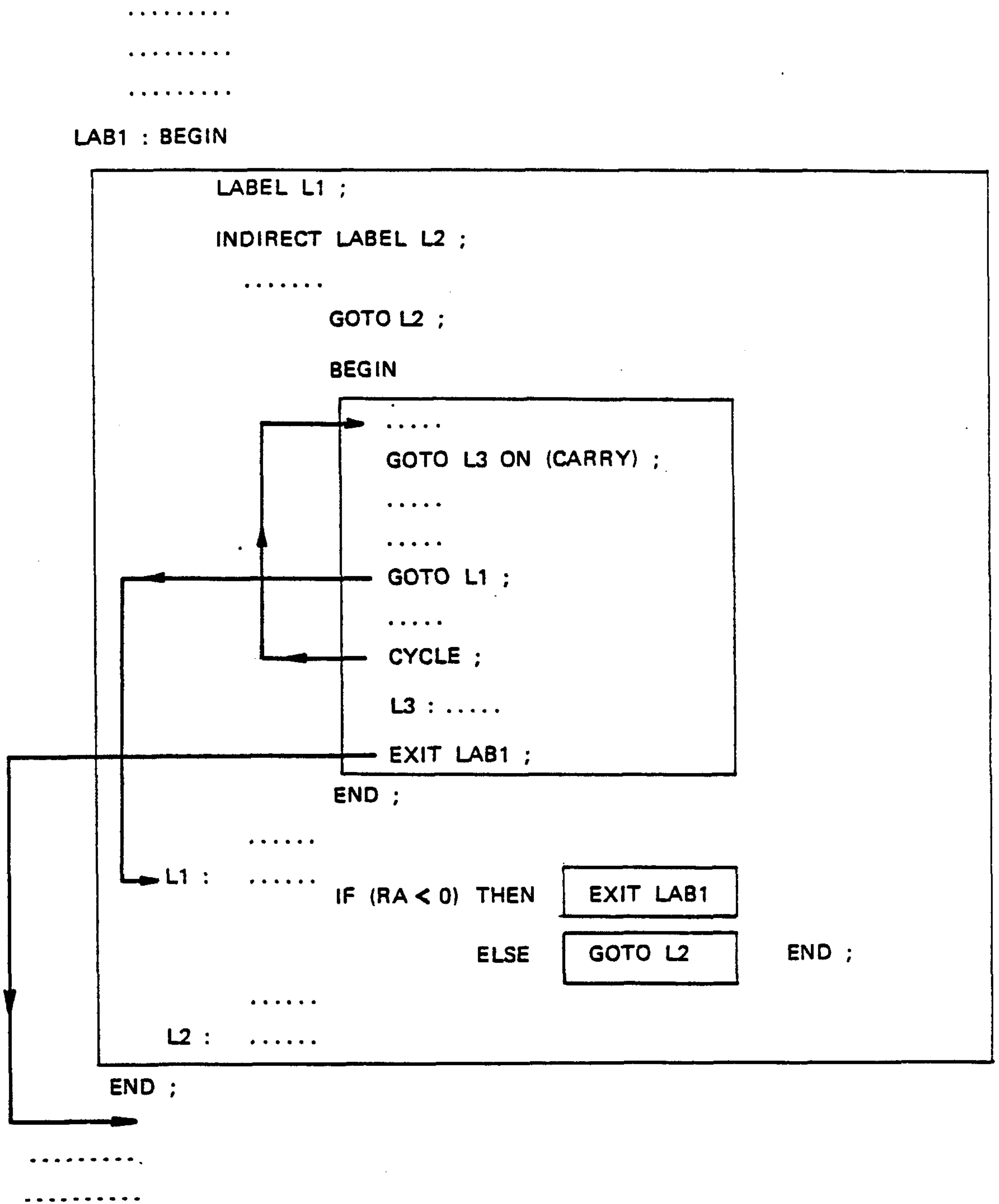
- l'instruction EXIT est une instruction permettant de sortir de l'instruction composée en cours (si le label est omis) ou étiquetée par le nom suivant EXIT. Le branchement se fait sur la première instruction suivant le END de l'instruction dont l'on veut sortir. On peut ainsi désactiver plusieurs blocs à la fois.
- l'instruction CYCLE est du même type que EXIT mais le branchement est effectué sur la première instruction de l'instruction composée en cours ou nommée.
- l'option ON condition signifie que l'instruction de rupture de séquence ne sera satisfaite que si la condition est vraie. La condition (qui doit être une condition OR) (1) se décrit d'une manière identique à celle d'une instruction conditionnelle. Cette forme d'écriture permet d'obtenir un code plus performant qu'en utilisant l'instruction IF, seulement dans le cas d'un label direct (branchement court).
- l'instruction EXIT permet de sortir de la procédure courante nommée et de revenir derrière l'instruction d'appel. C'est l'instruction normale de sortie de procédure, elle est cependant inutile lorsque l'exécution de celle-ci se termine par la dernière instruction écrite dans la déclaration de procédure.

Cas où la rupture de séquence s'accompagne d'une sortie de bloc

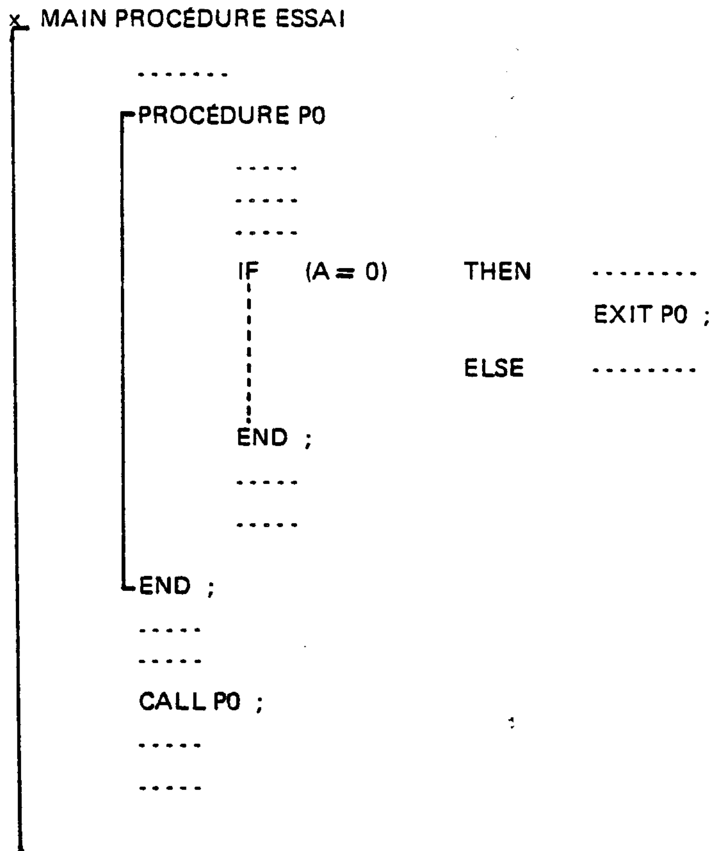
La restitution du contexte de travail du bloc appelant (section de données accessibles) est générée automatiquement par le compilateur à la sortie d'un bloc, si cela est nécessaire (d'autres précisions à ce sujet seront données dans la remarque du paragraphe 5.8).

(1) voir page 81

Exemples :



Noter la déclaration de l'étiquette L1, dans le bloc ou elle est définie, et parce qu'elle est utilisée avant sa définition, dans un bloc plus interne.



Les instructions CYCLE et EXIT permettent de limiter le nombre des étiquettes dans un programme.

Les écritures suivantes provoquent des erreurs :

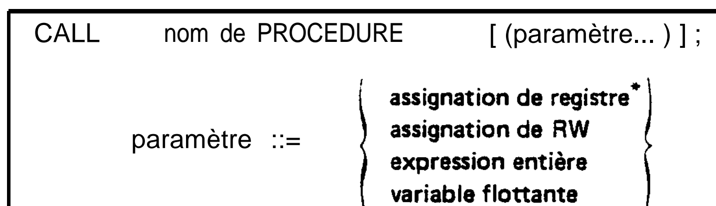
. GOTO L1 ON (RA = 0) AND (RB < 0) ; condition AND interdite ici

. INDIRECT LABEL L2 ;
GOTO L2 ON (CARRY) ON condition interdite avec label indirect

. EXIT P0 ON (RA < 0) ---- ou avec une procédure.

5.7 - APPEL DE PROCEDURE

Cette instruction permet d'exécuter un traitement déclaré ailleurs. Elle réalise le branchement vers la première instruction de la procédure nommée, et provoque ainsi l'entrée dans le bloc défini par cette procédure.



- le point d'entrée dans la procédure est la première instruction du bloc qu'elle définit
- l'adresse de retour à l'instruction suivant l'appel est sauvegardée dans la pile
- le retour du sous-programme défini par une procédure peut intervenir de plusieurs façons :
  . exécution de END délimitant la procédure
  . exécution de l'instruction EXIT procédure
  . une instruction GOTO qui réalise un transfert de contrôle hors de la procédure (dans ce cas la gestion des bases et du registre RK qui pointe la pile est laissée à l'utilisateur cf. remarque du paragraphe 5.8)
- les paramètres effectifs passés par expressions entières ou variables flottantes utilisent les registres ; ils initialisent dans le même ordre d'apparition les paramètres formels déclarés lors de la déclaration de la procédure. Ils sont passes par l'intermédiaire de la pile pointée par RK.
- les assignations de registres permettent d'initialiser les registres servant de paramètres à la procédure appelée (RA, RB,...).

Remarque :

Une procédure est accessible même si la section dans laquelle elle a été déclarée ne l'est pas, à condition qu'il existe une section locale accessible dans laquelle le compilateur générera un relais (cf. annexe II).

Exemples :

Si P0, P1, P2, P3, P4 sont des noms de procédures

```
CALL  P0 ;
CALL  P1 (A) ;
CALL  P0 (RA : = 2, RX : = CPT + 2) ;
CALL  P3 (A, B + C, C * (D + E)) ;
CALL  P4 (F1) ;
CALL  P4 (F1, RW : = WORK) ;        «sauvegarde et chargement de RW
...... ;                             «restauration de RW
```

* Les assignations des registres doubles (RAB et RFL) sont interdites.

5.8 - BRANCHEMENT ET APPELS DE PROCEDURES, PAR POINTEUR

De tels branchements et appels de procédure sont réalisés par des instructions GOTO et CALL portant sur des pointeurs ou des éléments de tableaux respectivement d'étiquettes et de procédures.

Forme générale :

```
GOTO    { nom de POINTER LABEL      [(indice)]   }
        { nom de ARRAY LABEL         (indice)    }

CALL    { nom de POINTER PROCEDURE  [(indice)]   }
        { nom de ARRAY PROCEDURE     (indice)    }   [ (paramètres) ] ;
```

La possibilité d'assigner des pointeurs et des éléments de tableaux permet de réaliser ces transferts de contrôle dynamiques.

Remarque :

Nous avons vu (paragraphe 4.4) qu'une déclaration de pointeur ou de tableau de procédures ne spécifie pas de liste de paramètres formels. Cependant l'appel de procédure par leur intermédiaire peut comporter une liste de paramètres effectifs si la procédure ainsi appelée en utilise.

Exemples :

```
. . . . . . . . .
POINTER LABEL ADLANC ;
ARRAY 3 LABEL TABETIQ = (@ L1, @ L2, @ L3) ;
POINTER ARRAY LABEL PTAB = (@ TABETIQ + 1) ;
POINTER WORD SPTAB SYN PTAB ;
. . . . . . . . .
POINTER PROCEDURE PTPROC = (@ PROC 5) ;
ARRAY 4 PROCEDURE TABPRO = (@ PROC1, @ PROC2, @ PROC3, @ PROC4) ;
. . . . . . . . .
ARRAY 4 WORD TABP SYN TABPRO ;
. . . . . . . . .
```

. . . .

CALL PTPROC ;

CALL TABPRO (2) ; l'élément du tableau ainsi pointé est @ PROC3

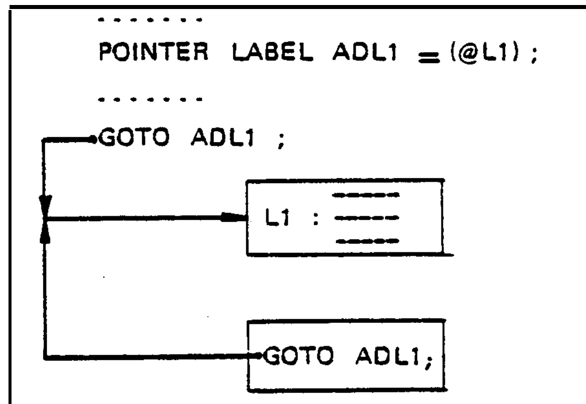
TABP (2) : = @ PROC6 ;

. . . .

CALL TABPRO (2) (A, B) ; la procédure PROC6 ainsi appelée a 2 paramètres

CALL TABPRO (1) (RA : = 2, RB : = A) ;
CALL TABPRO (1) (RA := TAB (4)) ;

On remarque que ces branchements, permettent d'entrer dans un bloc autrement que par sa première instruction, ce qui doit être utilisé avec prudence.



```
MAIN PROCÉDURE ESSAI
 . LOCAL SECTION L1
 POINTER PROCÉDURE PTP0 = (@LAB1) ;
 PROCÉDURE P0
   . LOCAL SECTION CONTINUE L1
     WORD A ;
   . USING LOCAL IS L1 ;
     ........
     ........
     LAB1 :.....
     ........
     ........
     EXIT P0 ;
     ........
     ........
     ........
 END ;
 . USING LOCAL = L1 ;
 ......
 ......
 ......
 CALL P0 ;
 ......
 CALL PTP0 ;
 ......
END.
```

Dans ce dernier exemple, la procédure P0 comporte deux points d'entrée :

- par l'instruction CALL P0 l'exécution se poursuit sur l'instruction USING LOCAL IS L1
- par l'instruction CALL PTP0 l'exécution se poursuit à l'instruction étiquetée par LAB1. La gestion des bases est ici inexistante car P0 utilise le même contexte de travail que la procédure principale.

Exemples d'utilisation

Les tableaux d'étiquettes ou de procédures permettent la réalisation d'aiguillages sur des étiquettes ou des procédures.

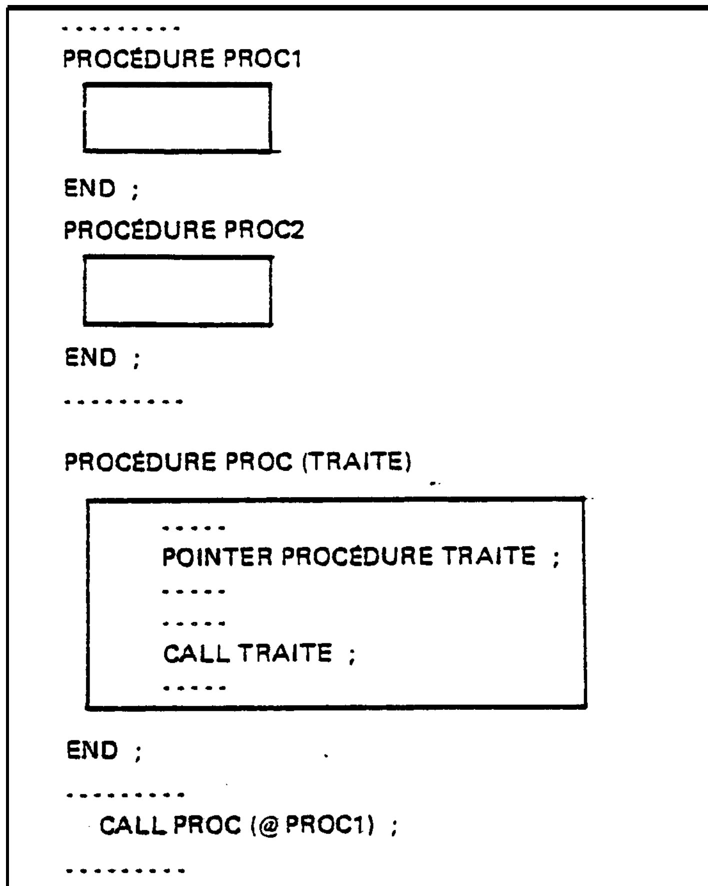
```
MAIN PROCÉDURE ESSAI
    ......
    PROCÉDURE PROC1
    [          ]
    END ;
    PROCÉDURE PROC2
    [          ]
    END ;
    PROCÉDURE PROC3
    [          ]
    END ;

    ARRAY 3 PROCÉDURE
       TABPRO= @PROC1,   PROC2,
       @PROC3) ;
    WORD I ;
       .......

    CALL TABPRO (I) ;
    ........

    ........
    ........
END.
```

```
MAIN PROCÉDURE ESSAI
    ......
    PROCÉDURE PROC1
    [          ]
    END ;
    PROCÉDURE PROC2
    [          ]
    END ;
    PROCÉDURE PROC3
    [          ]
    END ;
    WORD I ;
    ........
    ........

    RX := I + 1 ;
    CASE (RX) / 3 OF
          CALL PROC1 ;
          CALL PROC2 ;
          CALL PROC3 ;
    END ;
    .........
    .........
END.
```

Ces deux exemples sont deux écritures différentes pour la réalisation du même traitement. Cependant, on constate que la première écriture est plus simple que la seconde.

Les pointeurs permettent aussi de passer en paramètres des adresses d'étiquettes ou de procédures de tableaux.

MAIN PROCEDURE ESSAI



END.

Remarque :

Lors de ces branchements indéfinis le compilateur ne peut pas détecter une sortie de bloc ; ii n'y a donc pas génération automatique de la restauration du contexte de l'appelant ; la gestion des bases est par suite laissée aux soins du programmeur.

Il en est de même pour l'entrée dans un bloc autrement que par sa première instruction. Si le bloc appelé utilise le même contexte que l'appelant il n'y a pas de problème sinon le programmeur devra charger les bases à l'entrée dans le bloc.

Des précisions sur la méthode utilisée par le compilateur pour charger les bases seront données en annexe.

5.9 - INSTRUCTIONS DE BOUCLES

```
DO   clause   ;   bloc   END ;
```

Cette instruction composée provoque l'exécution, un certain nombre de fois, de la séquence d'instructions du bloc qu'elle définit.

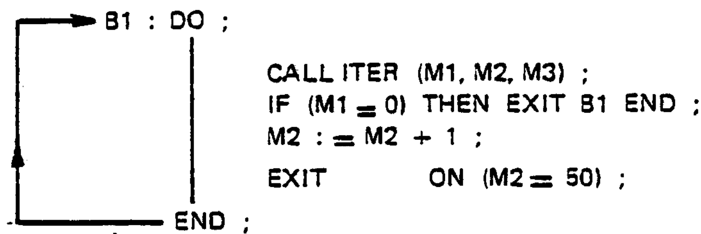
La clause permet d'indiquer ce nombre de fois ou du moins la condition de réexécution de la boucle. Elle se présente sous quatre formes :

```
           ⎧ vide
clause ::= ⎨ WHILE  condition
           ⎪ {INCR} {variable de type mot}  TIMES
           ⎪ {DECR} {        RX          }
           ⎩ FOR  assignation STEP ± pas UNTIL limite
```

5.9.1 - CLAUSE VIDE

Elle correspond à une boucle illimitée dont on sort par une instruction de branchement GOTO ou EXIT.

Exemple :



Le bloc est ainsi exécuté tant que M1 est différent de 0 ou que M2 est différent de 50.

Dans une instruction composée BEGIN la méthode à suivre est inverse. La bloc est exécuté une seule fois et le soin est laissé au programmeur de faire boucler son exécution autant de fois qu'il le désire. L'exemple précédent pourrait ainsi s'écrire :

```
B1 : BEGIN
     CALL ITER (M1, M2, M3) ;
     IF (M1 = 0) THEN EXIT B1 END
     M2 : = M2 + 1 ;
     CYCLE ON (M2 < 50) ;
   END ;
```

5.9.2 - CLAUSE WHILE

Elle indique que la séquence est répétée tant que la condition suivant le mot "WHILE" est satisfaite. Cette condition se décrit de la même manière que celles des instructions conditionnelles et le test est fait avant l'exécution des instructions.

Exemple :

```
D0 WHILE (M1 / = 0) AND (M2 < 50) ;
   |  CALL ITER (M1, M2. M3) ;
   |  M2 : = M2 + 1 ;
END ;
```

5.9.3 - CLAUSE TIMES

ou clause "nombre de fois". La séquence est alors exécutée autant de fois que le précise le contenu du compteur. lequel peut être une variable mot ou le registre RX. Suivant que le facteur est précédé du mot DECR OU INCR, après chaque exécution de la boucle, le compteur qu'il indique est décrémenté ou incrémenté de un et si la valeur est encore positive ou négative la boucle est de nouveau exécutée et ainsi de suite. Si la valeur initiale du facteur est négative positive ou nulle avant l'exécution de la boucle, celle-ci sera toutefois exécutée une fois.

Exemple :

```
      I := 50 ;

B1 : DO DECR  I TIMES ;
        | CALL ITER (M1, M2, M3) ;
        |  IF (M1 = 0) THEN EXIT B1 END ;
        | M2 := M2 + 1 ;
     END ;
```

Dans cet exemple, la variable I peut être remplacée par le registre RX :

```
B1 : DO DECR (RX : = 50) TIMES ;
```

Remarque :

La séquence suivant une clause "TIME" est toujours exécutée au moins une fois, le test de fin de boucle étant réalisé en fin de séquence.

5.9.4 - CLAUSE FOR

ou clause "assignation". Elle consiste à donner une valeur de départ à un élément (mot, octet ou registre). Cette valeur sera augmentée ou diminuée de la valeur du "pas" après chaque exécution de la séquence. On sort de la boucle lorsque l'élément dépasse la valeur limite précisée derrière le mot "UNTIL". On peut ainsi schématiser les étapes d'une instruction DO comportant cette clause :

1) L'instruction d'assignation est exécutée
2) Si le signe du pas est positif (négatif) et si la valeur de l'élément assigné n'est pas supérieure (inférieure) à la limite, l'exécution continue en séquence, sinon l'exécution est terminée
3) L'ensemble des instructions suivant la clause est exécuté.
4) Le pas est ajouté (soustrait) à l'élément assigné et l'exécution reprend en 2.

Les "pas" et "limite" sont fonction du type de l'assignation : assignation de registre ou assignation de variable.

```
                   { assignation RA             }
assignation  ::=   { assignation registre       }
                   { assignation variable(1)    }
```

1) assignation de registre. Dans ce cas on a :

```
pas ::=      littéral     (inférieur à 127)

             { 0        }
limite ::=   {          }
             { registre }
```

En effet, pour les registres, seules l'addition immédiate et la comparaison à 0 ou à un autre registre sont possibles.

Exemple :

```
DO FOR RB : = 50 STEP - 1 UNTIL (RY : = 1) ;
    CALL ITER (M1, M2, M3) ;
    EXIT ON (M1 = 0) ;
    M2 : = M2 + 1 ;
END ;
```

L'écriture suivante :

DO FOR RB : = 1 STEP + 1 UNTIL 5 ; est INTERDITE.

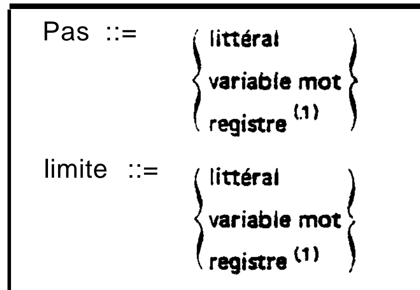
Elle peut être remplacée par :

DO FOR RB : = 1 STEP + 1 UNTIL (RY : = 5) ;

(1) Assignation par expression exclusivement

2) assignation RA et assignation de variable

Si l'assignation est une assignation de l'accumulateur ou de variable (mot ou octet) les possibilités d'écriture sont :



Exemples :

```
LIMITE : = 1 ;
DO FOR I : = 50 STEP - 1 UNTIL LIMITE ;
   | CALL ITER (M1, M2, M3) ;
   | EXIT ON (M1 = 0) ;
   | M2 : = M2 + 1 ;
END ;
```

On peut écrire, la même boucle d'une manière plus optimale :

```
RY : = 1 ;        «LIMITE
RB : = 1 ;        «PAS

DO FOR RA : = 50 STEP - RB UNTIL RY ;
   | SAVE (RA) ;
   | ---------
   | ---------
   | RESTORE (RA) ;
END ;
```

Les instructions SAVE et RESTORE qui permettent ici d'utiliser RA, à la fois comme indice de boucle et comme accumulateur dans le bloc, sont vues au paragraphe 5.11 - 12 page 105

Remarque sur le fonctionnement général des boucles

Pour que la séquence soit exécutée au moins une fois il faut, à l'entrée dans l'instruction composée DO :

- que la condition suivant 'WHILE" soit satisfaite,
- elle le sera toujours au moins une fois pour la clause "TIMES",
- que la valeur assignée à l'élément soit inférieure (ou supérieure) à la valeur donnée comme limite (clause "FOR").

(1) RB, RX, RY

5.9.5 - EXEMPLES

1)

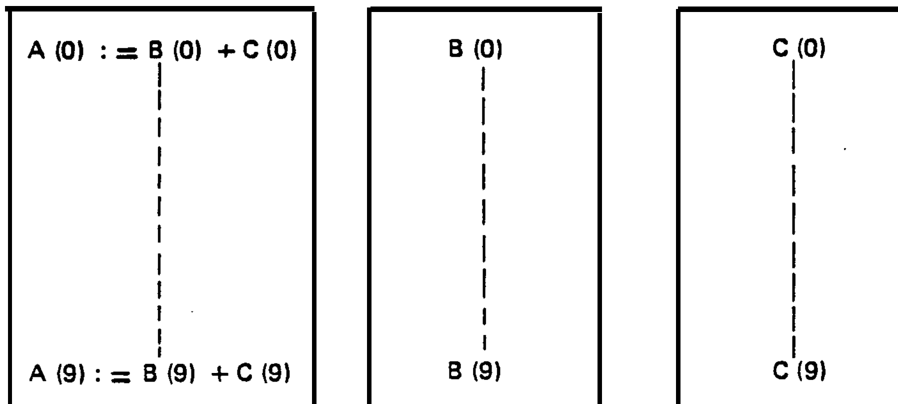
```
L1 : DO WHILE (A < 5) ;
     CASE ((RX := A)) / 5 OF
              EXIT L1 ;
              BEGIN
                   A, RA := A + B ;
                   CYCLE ON (RA < 10) ;
              END ;
              GOTO L2 ;
     L2 :     A, RA := A + C ;
         RESUL := B + C ;
     END ;
     A := A + C / 3 ;
     C := B + 1 ;
     B := C + A / 3 ;
END ;
GOTO L1 ON (RES = 0) ;
--------
```

2) boucles et traitements des tableaux

Soit à additionner les éléments de 2 tableaux :

- ARRAY 10 WORD B
- ARRAY 10 WORD C

et ranger les résultats dans un troisième : ARRAY 10 WORD A.



Ce traitement peut être décrit indifféremment par de nombreuses écritures, parmi lesquelles on peut citer :

```
DO FOR I := 0 STEP + 1 UNTIL 9 ;
|   A (I) := B (I) + C (I) ;
END ;
```

ou d'une manière plus optimale :

```
DO FOR RX := 0 STEP + 1 UNTIL (RY := 9) ;
|   A (RX) := B (RX) + C (RX) ;
END ;
```

ou encore

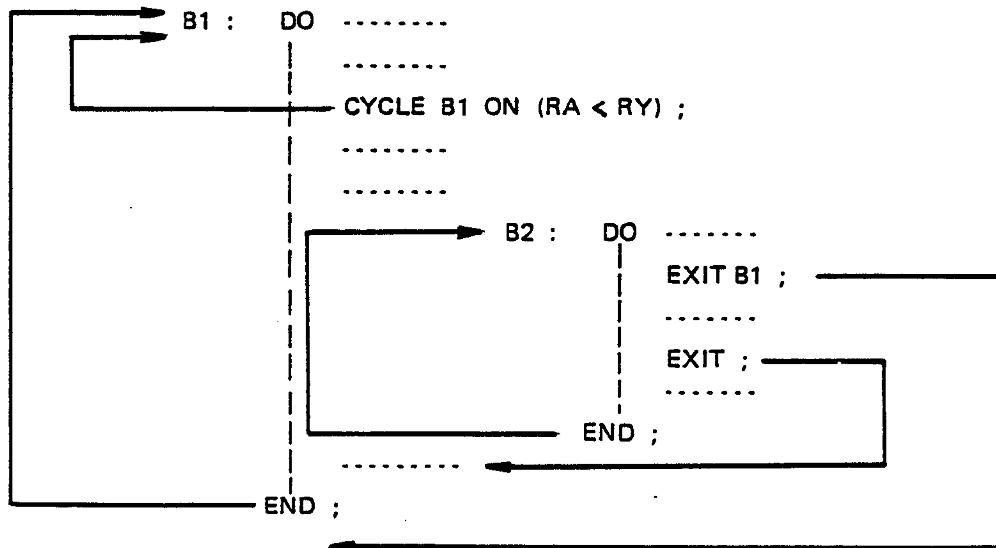
```
RX := -1 ;
RB := 9 ;
DO WHILE ((RX := RX+1)  <= RB) ;
|   A ( RX ) := B ( RX ) + C ( RX ) ;
END ;
```

Si le tableau était un tableau de flottants on aurait :

```
DO FOR RX := 0 STEP + 2 UNTIL (RY := 18) ;
|   A (RX) := B (RX) + C (RX) ;
END ;
```

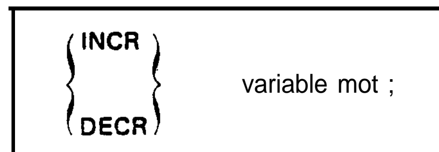
car chaque élément occupe alors 2 mots.

3) boucles imbriquées



5.10 - INSTRUCTIONS INCR ET DECR

Ces deux instructions permettent respectivement d'incrémenter ou de décrémenter une variable, de la valeur 1.



Ces deux instructions positionnent un indicateur que l'on peut tester directement par :

= si la variable est nulle après l'opération

< si la variable est négative après l'opération

> si la variable est positive après l'opération.

On peut ainsi écrire des séquences de la forme :

```
INCR VAR ;
IF (=) THEN CALL ERREUR
        ELSE IF (>) THEN GOTO POSITIF END ;
END ;
```

qui permet une traduction en langage machine beaucoup plus efficace que la séquence :

```
VAR : = VAR + 1 ;
IF (VAR = 0) THEN  - - - - -
                   ......
```

5.11 - INSTRUCTIONS SUR PILE GENERALE

Elles permettent l'utilisation de la pile générale pointée par le registre K du calculateur.

5.11.1 - CHARGEMENT DU REGISTRE RK

Celui-ci s'effectue en tête du programme principal car il doit avoir lieu avant toute instruction nécessitant l'utilisation de la pile générale soit :

- avant tout chargement de base par l'instruction USING
- avant tout appel de procédure
- avant tout calcul d'expression entière
- avant toute instruction sur la pile : sauvegarde ou restitution de registre.

Ce chargement est réalisé par l'instruction suivante :

```
. USING  { RK     }  =  nom de KSTORE SECTION ;
         { KSTORE }
```

Elle a la même forme qu'une instruction de chargement de base. Le nom de SECTION donné ici est l'identification qui apparaît au moment de la déclaration KSTORE SECTION.

Exemple :

```
MAIN PROCEDURE EXEMPL
  • COMMON SECTION C0
    -------
  • LOCAL SECTION L0
    -------
    -------
    -------
  • KSTORE SECTION PILE
    RES 10 ;
  • USING RK = PILE, COMMON = C0, LOCAL = L0 ;
    -------
    -------
    -------
END.
```

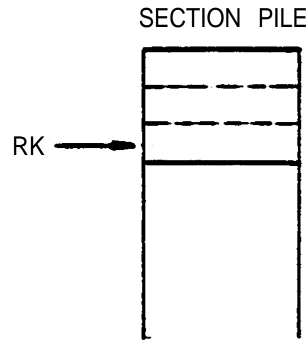
5.11.2 - SAUVEGARDE ET RESTAURATION DE REGISTRE

Les instructions SAVE et RESTORE agissent sur la pile générale déclarée. Elles correspondent respectivement à la sauvegarde dans la pile ou à la restauration à partir de la pile des registres indiqués.

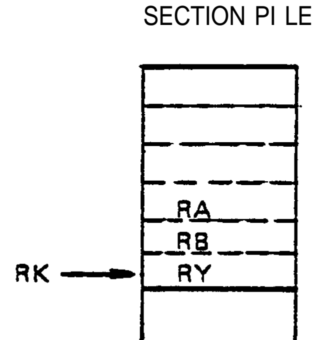
Remarque : “registre” désigne ici la suite RA, RB, RX, RY, RC, RL, RW, RK.

Exemples :

SAVE (RA, RB, RY) ;

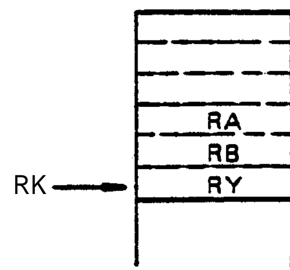


état de la pile avant
l'instruction SAVE

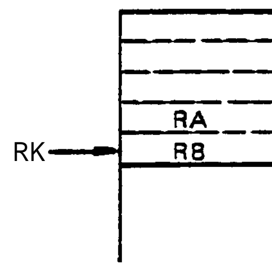


état de la pile après
l'instruction SAVE

RESTORE (RX) ;



état de la pile avant



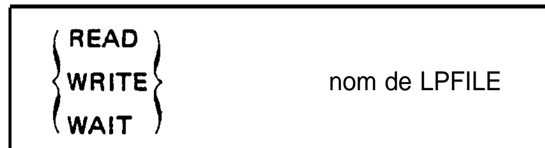
état de la pile après

RX reçoit ainsi la valeur qui était stockée en haut de la pile

5.12 - ENTREES-SORTIES ET FONCTIONS MONITEUR (cf. notices de BOS et IOCS)

5.12.1 - OPERATIONS D'ENTREES-SORTIES

Nous citons dans ce paragraphe les instructions qui permettent d'utiliser les échanges PL16 définis dans les déclarations de LPFILE, par "nom de LPFILE" on entend aussi nom de pointeur de LPFILE



L'instruction READ permet de lire un fichier dont le MODE a été défini INPUT dans la déclaration de LPFILE.

Exemple :

```
- - - - - - -
LPFILE  COMAND  =      (MODE : INPUT, EM ; EU : PC ;
                        DATA : BUFFER ;
                        EOE : 4 ;
                        CONTROL)  ;
- - - - - - - - -
READ COMAND ;           lecture de 4 caractères, à partir de l'unité d'échange PC ; ils sont rangés
                        dans  BUFFER.
```

L'instruction WRITE permet d'écrire sur un fichier dont le MODE a été défini OUTPUT dans sa déclaration.

```
- - - - - - - - -
LPFILE  OBJET  =       (MODE : OUTPUT, IB ; EU : BO ;
                        DATA : BUFOBJ ;
                        EOE : 80) ;
- - - - - -
WRITE  OBJET            écriture de 80 caractères sur BO.
```

Remarque :

Les instructions :

READ OBJET ;
ou WRITE COMAND ; donnent lieu à un message d'erreur

Les instructions READ et WRITE se traduisent par :

- chargement dans RA de l'adresse de la table d'échange
- SVC IOCS

L'instruction WAIT permet de se mettre en attente d'une fin d'opération d'entrée-sortie effective avec retour immédiat (elle a été définie par l'option IM dans la déclaration de LPFILE).

Cette instruction est traduite par :

. chargement dans RA de l'adresse de la table d'échange correspondant à l'opération d'entrée-sortie dont on attend la fin
- SVC WEIO

5.12.2 - COMPTE RENDU D'ÉCHANGE

Lorsqu'une opération d'entrée-sortie est effectuée :
- soit avec retour en fin d'échange (EM)
- soit avec retour immédiat (IM)

Le mot (3) de la table d'échange contient le compte rendu d'échange qui permet à l'utilisateur de connaître comment s'est déroulée l'entrée-sortie.

L'instruction PL16

```
Registre : =      STATUS OF      nom de LPFILE ;
```

permet de charger dans un registre ce compte rendu.

5.12.3 - MODIFICATION DES CARACTERISTIQUES D'ÉCHANGE

La table d'échange définie dans la déclaration de LPFILE peut être modifiée dynamiquement au cours de l'exécution du programme grâce à l'instruction PL16 suivante :

```
         ⎧ nom de variable ⎫
ASSIGN   ⎨                 ⎬   TO   nom de LPFILE ;
         ⎩ unité d'échange ⎭
```

Celle-ci permet de modifier l'unité attachée à un fichier ou de modifier la zone sur laquelle se fera l'E/S.

Exemples :

```
---------
LPFILE  COMAND  =     (MODE : INPUT, EM ; EU : PC ;
                       RES : 1 ;
                       EOE : 4 ;
                       CONTROL) ;
ARRAY 4 BYTE BUF1 ;
LONG BUF2 ;
---------
ASSIGN BUF1 (10) TO COMAND ;
READ COMAND ;                          lecture sur PC
----------
ASSIGN BUF2 TO COMAND ;
READ COMAND ;                          lecture sur PC
---------
ASSIGN TK TO COMAND ;
READ COMAND ;                          lecture sur TK
```

5.12.4 - FONCTIONS MONITEUR

Ce sont les instructions PL16 suivantes :

```
STOP ;
TEST ;
START ;
```

- STOP correspond à "retour au superviseur" provoquant la reprise de contrôle par ce dernier. C'est cet appel qui mettra toujours fin au déroulement d'un programme. Cette instruction génère la requête programmée SVC ABOS.
- TEST. Par cet appel le programme peut tester, en des points où il est interruptible, les appels superviseur ou la présence d'un défaut périphérique et rendre éventuellement le contrôle au superviseur. Un programme ainsi interrompu peut être réactivé. Cette instruction génère la requête programmée SVC TAPS.
- START transmet au superviseur l'adresse de la table des points d'entrée du programme et des clés associées. Après cet appel le superviseur se met en attente d'une commande de l'utilisateur. Cette instruction génère la requête programmée SVC CAMO.

A ce dernier appel est liée l'existence de la table des clés et des points d'entrée associés ; son adresse doit être chargée dans l'accumulateur avant l'instruction START.

Remarque :

Le contexte d'un programme étant détruit après le passage des clés, il devra donc être rechargé au moment du lancement d'un processus. Nous donnons en annexe des exemples montrant comment on peut en PL16 utiliser la notion de processeur définie par BOS.

6 - SEGMENTATION D'UN PROGRAMME PL16

Un programme PL16 peut être organisé de façon modulaire ; il est alors découpé en "segments" de programme.

Cette organisation repose sur les justifications suivantes :

- structure de traitements
- la mise au point d'un programme peut être réalisée par étapes, segment après segment
- l'édition, la correction, la documentation des programmes s'en trouvent facilitées. On peut ainsi, en cours de mise au point, ne recompiler qu'une faible partie d'un programme, ou bien encore le configurer plus facilement
- l'utilisation des notions "d'overlay" de tâche

La syntaxe du langage apporte alors toutes les facilités pour effectuer cette segmentation des programmes et permet en plus de conserver pour le programme ainsi "découpé" sa logique.

Les deux notions ainsi introduites sont celles de :

- unité de compilation ou segment procédure
- référence à des noms définis dans une autre unité afin de réaliser la communication entre segments de programme compilés séparemment.

Jusqu'ici, nous avions défini un programme PL16 comme étant une "MAIN procédure" ; d'une manière plus générale ii sera constitué d'un ou plusieurs segments de programme encore appelés unités de compilation, chaque segment décrivant un traitement.

6.1 - UNITE DE COMPILATION

C'est l'unité de base contenant du texte source sur laquelle travaille le compilateur PL16. Elle est constituée d'une "segment procédure" qui est :

- soit le programme principal, elle est alors appelée

  MAIN PROCEDURE

- soit une procédure, elle est alors appelée

  SEGMENT PROCEDURE

Un segment procédure est défini de la même façon qu'une procédure classique ; sa déclaration comporte :

- un en-tête
- un bloc
- une fin.

6.2 - PROGRAMME PL16

Il est ainsi défini :

Programme ::= Segment procédure [; segment procédure. . . .] .

Toutes les segments procédures qui constituent un programme sont séparées par des caractères point-virgules. (;) et un point (.) termine le programme. Il est ainsi compilable en une seule fois (cf 9.2.4). Mais tous ces élèments sont compilables séparément et un code objet est généré pour chacun-d'eux. Dans le cas d'une compilation séparée pour chaque unité, chaque segment procédure doit se terminer par un point (.)

```
Segment procédure ::= { SEGMENT } PROCEDURE nom [(paramètre.... )]
                      { MAIN    }
            bloc
          END
```

6.2.1 - MAIN PROCEDURE

Ce segment définit le programme principal, c'est-à-dire celui qui sera lancé en fin de chargement en mémoire. Le point d'entrée de cette procédure est une instruction "USING" comportant nécessairement l'initialisation du registre pilé RK.

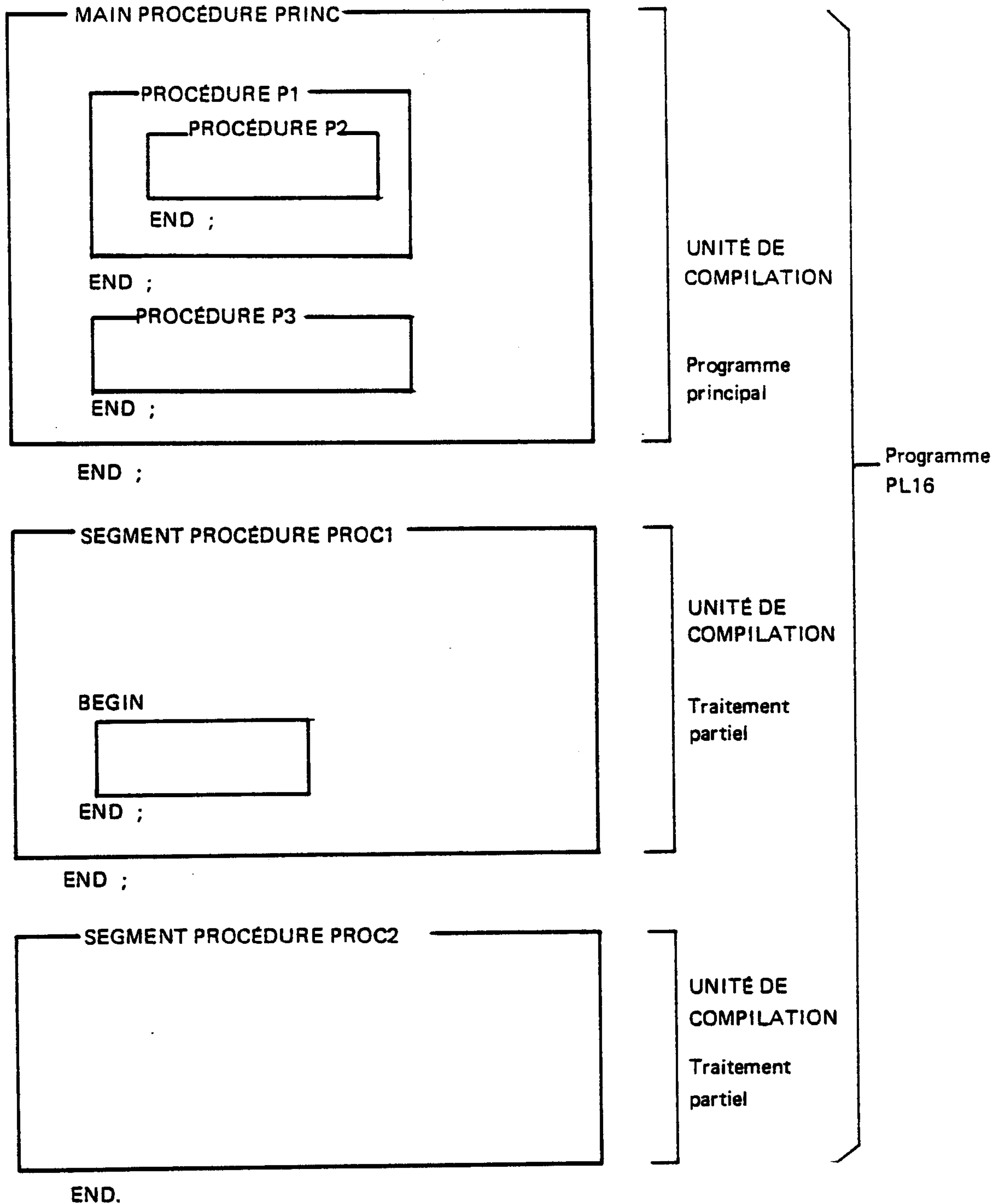
- USING RK = ---------

à l'entrée dans une MAIN PROCEDURE aucune sauvegarde n'est faite du contexte antérieur non plus que de restauration en sortie.

La MAIN PROCEDURE ne peut évidemment pas être définie avec paramètres.

6.2.2 - SEGMENT PROCEDURE

Ce segment définit une procédure connue et appelable de tout autre segment du programme PL16. Il s'agit donc simplement d'une procédure, compilable indépendamment et définie implicitement comme externe ; à ce titre, l'accès au nom du segment procédure nécessite une déclaration d'externe, même à l'intérieur de ce segment (appel récursif, ou déclaration de tâche (cf. 8.1 page 146)).



Le programme ainsi écrit est compilable en une seule fois (cf. 9.2.4). Chacune des 3 unités peut être compilée séparemment à condition de la terminer par un point (.) au lieu d'un point-virgule (;)

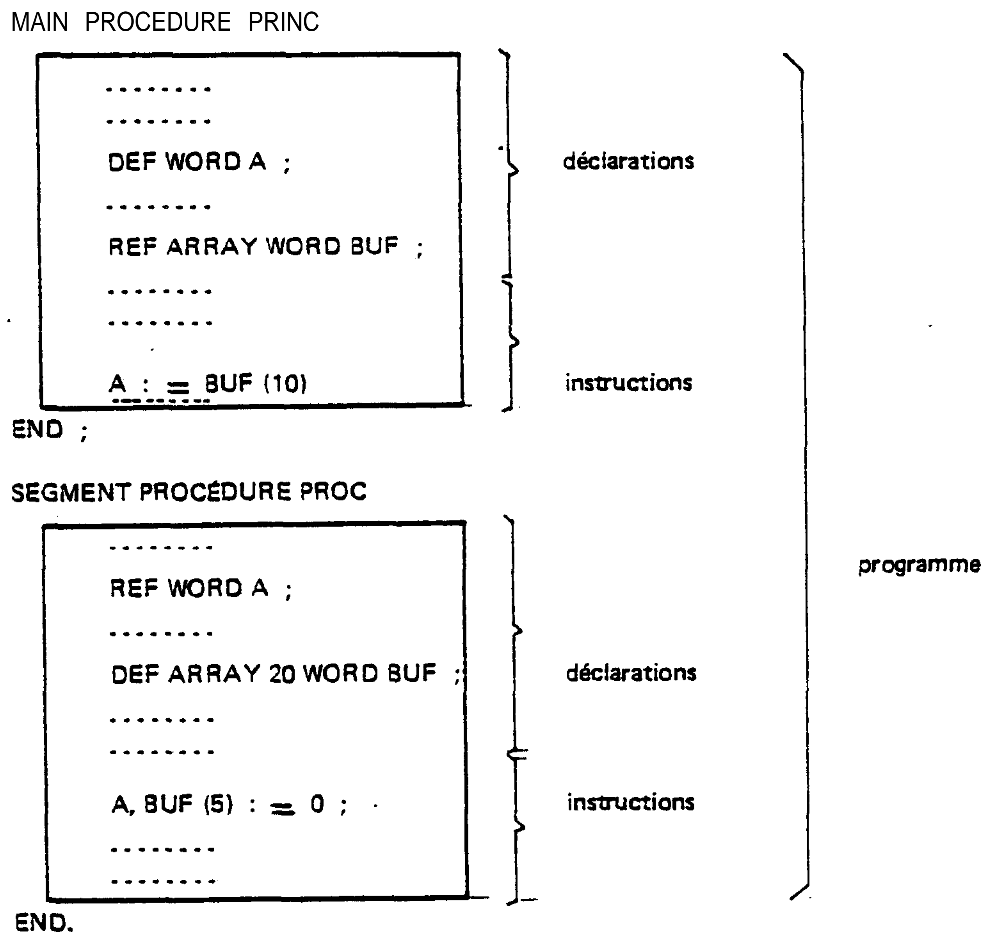
6.3 - IDENTIFICATEURS COMMUNS A PLUSIEURS UNITES PL16

Un programme PL16 découpé en segments procédures doit pouvoir garder son unité. Aussi le langage permet-il, à partir d'un segment procédure d'accéder à tout identificateur (de variable, procédure, section...) déclaré dans un autre segment procédure ; la communication entre segments est ainsi possible.

Pour cela :

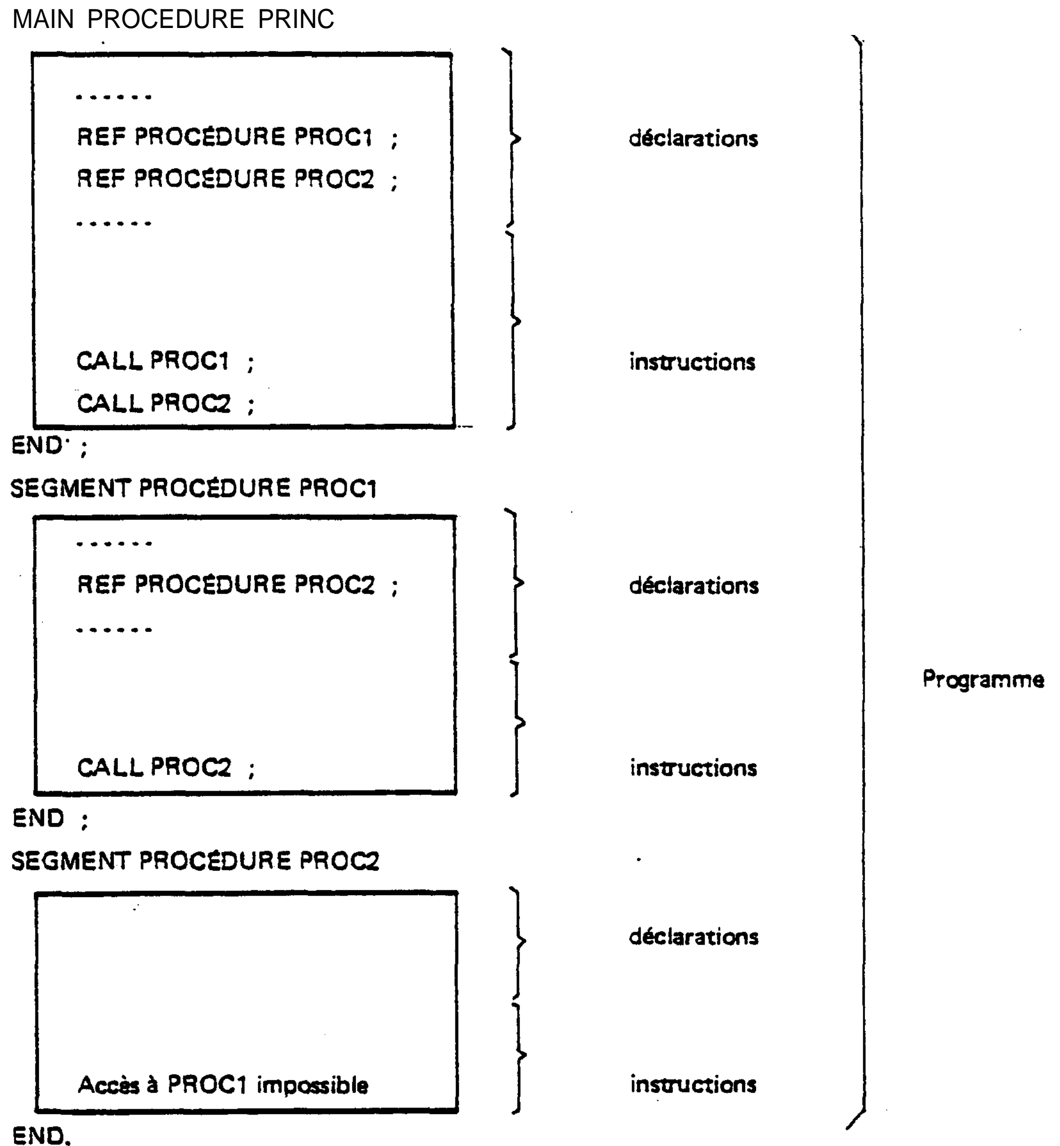
- la déclaration, dans un segment, d'un identificateur qui doit être accessible de l'extérieur est précédée du symbole DEF (définition d'externe)
- la déclaration de cet identificateur, dans un autre segment, doit être précédée du symbole REF ou du symbole EXT (cf. chapitre 7 paragraphe 2).

Exemple :



Il en est de même pour l'utilisation du nom d'un segment procédure par une autre ; mais la déclaration d'un segment procédure ne doit pas être précédée du symbole DEF car la définition d'externe est implicite. En effet un segment procédure doit être accessible de l'extérieur.

Exemple :



Remarque:

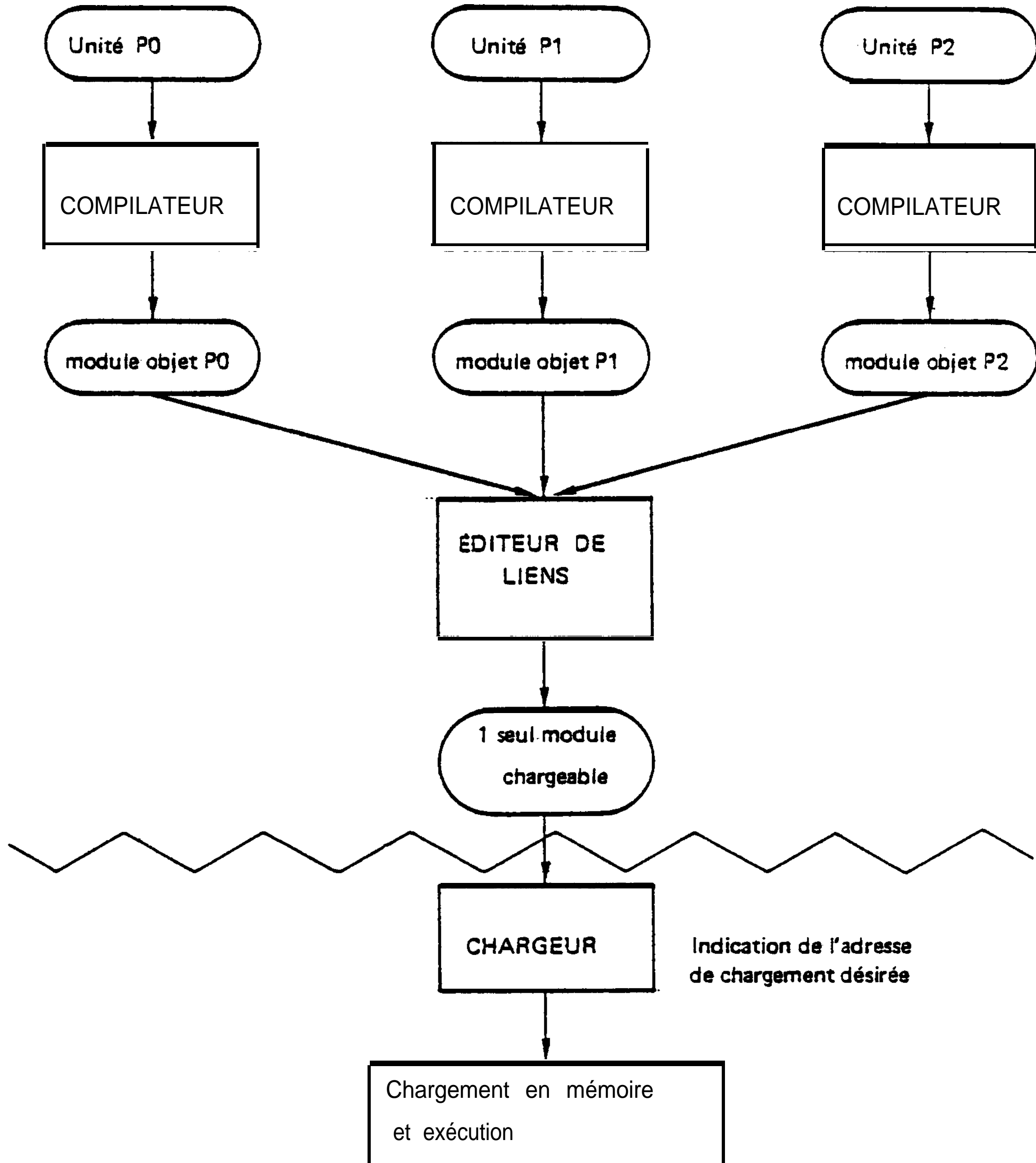
L'utilisation du nom de procédure (Main procédure, Segment procédure ou procédure) à l'intérieur de cette procédure nécessite la déclaration explicite de ce nom en externe (EXT nom proc) avant son utilisation.

6.4 - ÉDITION DE LIENS

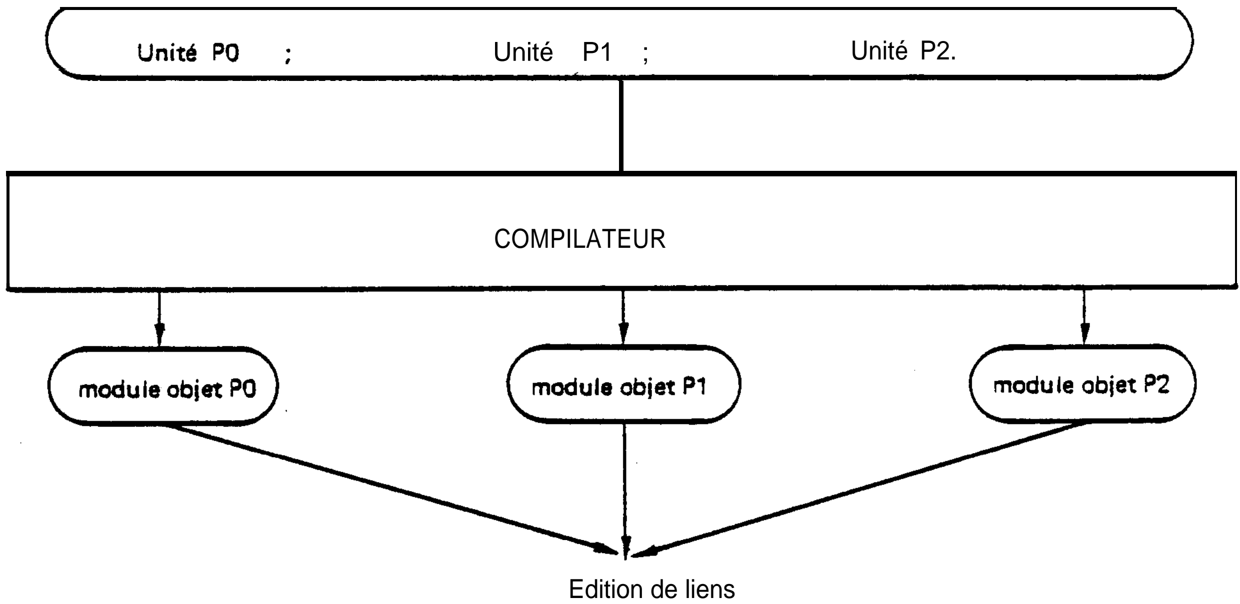
La compilation de chaque unité de code source aura pour résultat une unité de code objet ou module objet Il faudra ensuite, rétablir l'unité du programme en combinant tous les produits de compilation des différents unités de manière à obtenir un seul module, chargeable en machine. La programme qui, par la résolution des références externes, établit les liens entre différentes unités est l'éditeur de liens.

Exemple :

Soit à obtenir un code chargeable pour un programme composé de 3 unités.

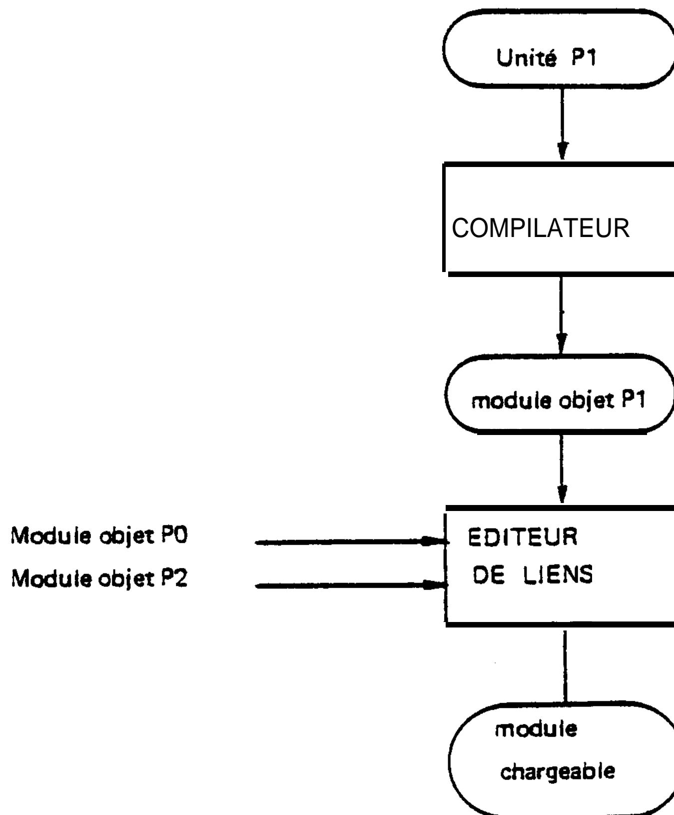


ou bien avec une seule compilation (cf. 9.2.4) :



Edition de liens

En cas d'erreur détectée dans l'unité P1, par exemple, il suffira, après correction, de recompiler cette unité et de refaire une édition de liens avec les autres codes objet inchangés.



6.5 - EXEMPLES

a) le programme de calcul que nous avons vu au chapitre 2 pourrait ainsi se découper en plusieurs segments :

MAIN PROCEDURE CALCUL

```
    (.....
     ....
    . KSTORE SECTION PILE
          RES 50 ;
    . LOCAL SECTION LOC
          WORD A, B ;
          WORD S ;
          WORD I ;
      REF PROCÉDURE LIRE ;
      REF PROCÉDURE ÉCRIRE ;
      REF PROCÉDURE SOMME ;
    . USING KSTORE = PILE, LOCAL = LOC ;
          DO FOR ------
              ------
              ------
          END ;
END ;
```

Programme principal

```
SEGMENT PROCÉDURE LIRE (X, Y)
    . LOCAL SECTION LOC
          ------
    . USING RL = LOC ;
          ------
END ;
```

Traitement LIRE

```
SEGMENT PROCÉDURE ÉCRIRE (RESUL)
    . LOCAL SECTION LOC
          ------
          ------
    . USING LOCAL = LOC ;
END ;
```

Traitement ECRIRE

```
SEGMENT PROCÉDURE SOMME (X, Y, Z)
    . LOCAL SECTION LOC
          -------
          -------
    . USING LOCAL = LOC ;
          -------
END.
```

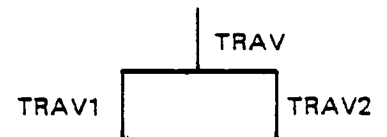
Traitement SOMME

b) Nous présentons ici trois réalisations possibles d'un programme théorique PL16.

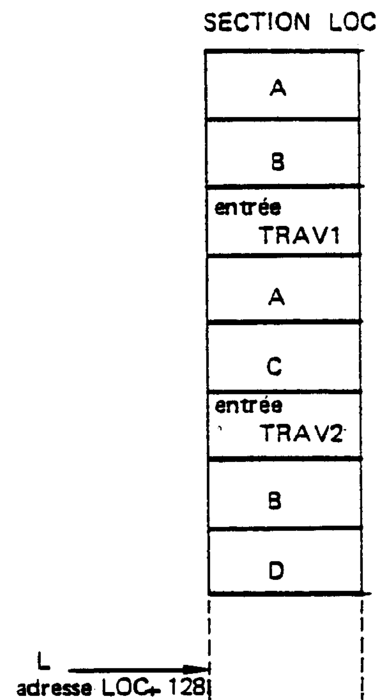
1) Programme en une seule unité de compilation

```
MAIN PROCÉDURE TRAV
 . LOCAL SECTION LOC
   WORD A, B ;
      PROCÉDURE TRAV1
         . LOCAL SECTION CONTINUE LOC
              WORD A, C ;
         . USING LOCAL IS LOC ;
              --------
              Accès à A ; (de TRAV1)
              Accès à C ;
              Accès à B ; (de TRAV)
      END ;

      PROCÉDURE TRAV2
           . LOCAL SECTION CONTINUE LOC
              WORD B, D ;
           . USING LOCAL IS LOC ;
              Accès à A ;   (de TRAV)
              Accès à B ;   (de TRAV2)
              Accès à D ;
              -------
      END ;
      ------
      ------
  . USING RL = LOC ;
      Accès à A ;
      Accès à B ;
      Accès à TRAV1 ;
      Accès à TRAV2 ;
    END .
```

Structure du programme



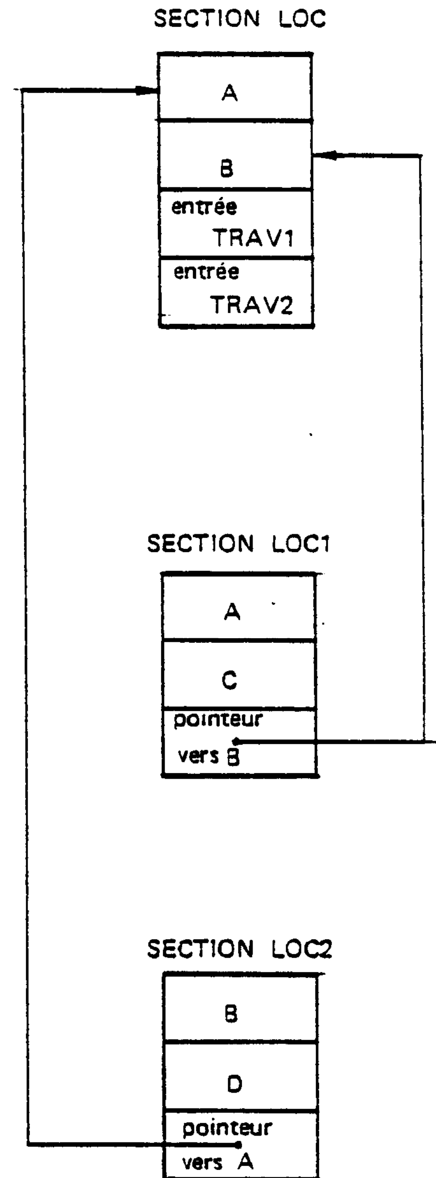
Structure des données



La base L permettrait d'accéder à toutes les variables mais, la structure de bloc intervenant les accès aux identificateurs s'en trouvent limités.

2) même programme en 3 unités de compilation

```
MAIN PROCÉDURE TRAV
   . LOCAL SECTION LOC
     DEF WORD A
     DEF WORD B ;
     REF PROCÉDURE TRAV1 ;
     REF PROCÉDURE TRAV2 ;
   . USING LOCAL = LOC ;
       Accès à A ;
       Accès à B ;
       Accès à TRAV1 ;
       Accès à TRAV2 ;
END ;

SEGMENT PROCÉDURE TRAV1
   . LOCAL SECTION LOC1
       WORD A, C ;
     REF WORD B ;
   . USING RL = LOC1 ;
       Accès à A ;
       Accès à C ;
       Accès à B ;
END ;

SEGMENT PROCÉDURE TRAV2
   . LOCAL SECTION LOC2
       WORD B, D ;
     REF WORD A ;
   . USING RL = LOC2 ;
       Accès à A ;
       Accès à B ;
       Accès à D ;
END.
```



On a défini comme étant à portée externe les noms utiles dans les différents segments.

3) Même programme en 3 unités de compilation

```
MAIN PROCÉDURE TRAV
  . DEF COMMON SECTION COM
        WORD A, B ;
        REF PROCÉDURE TRAV1, TRAV2 ;
  . USING COMMON = COM ;
        Accès à A ;
        Accès à B ;
        Accès à TRAV1 ;
        Accès à TRAV2 ;
END ;
```

SECTION COM

A
B
entrée TRAV1
entrée TRAV2

```
SEGMENT PROCÉDURE TRAV1
  . REF COMMON SECTION COM
        RES 1 ;
        WORD B ;
  . LOCAL SECTION LOC1
        WORD A, C ;
  . USING RC IS COM, LOCAL = LOC1 ;
        Accès à A ;
        Accès à B ;     (de COMMON)
        Accès à C ;
END ;
```

SECTION LOC1

A
C

```
SEGMENT PROCÉDURE TRAV2
  . REF COMMON SECTION COM
        WORD A ;
  . LOCAL SECTION LOC2
        WORD B, D ;
  . USING RC IS COM, LOCAL = LOC2 ;
        Accès à A ;     (de COMMON)
        Accès à B ;
        Accès à D ;
END.
```

SECTION LOC2

B
D

On a défini comme appartenant à une section “common” les éléments utiles dans différents segments.

Remarques sur l'exemple 3)

- DEF COMMON SECTION permet de rendre la section COM accessible par une autre segment procédure. Cette segment procédure doit alors la déclarer par REF et la redécrire.

- A l'intérieur de la procédure TRAV1 on ne peut pas redéclarer A dans la REF COMMON SECTION car on aurait une double déclaration de l'identificateur A dans le bloc TRAV1 ; il suffit d'indiquer par RES ␣ qu'on réserve simplement un mot et qu'on ne se servira pas dans ce bloc de la mémoire correspondante : ceci permet au compilateur de calculer le déplacement de B dans la section. Le même raisonnement vaut pour l'identificateur B dans TRAV2, à la différence que RES est alors inutile pour la place de B car aucune déclaration ne le suit.

- Cette méthode est, en général, moins coûteuse en place mémoire que celle vue dans 2). En effet pour une section déclarée par REF il suffit d'un pointeur vers la section déclarée ailleurs par DEF, tandis que pour des variables il y aura autant de pointeurs que de variables déclarées par REF.

6.6 - CONCLUSION

Un programme PL16 peut être divisé en plusieurs unités de compilation. Le programme conserve alors la même logique car on peut sauvegarder les liens existants entre les différentes parties de programme par la notion de REFERENCE. L'unité du programme sera rétablie par l'éditeur de liens.

7 - COMPLEMENTS POUR UNE PROGRAMMATION PLUS GENERALE

7.1 - OPERATIONS D'ENTREES-SORTIES : GENERALISATION

7.1.1 - DECLARATION D'ECHANGE PAR IOCB, OU POINTEURS

IOCB

La table nécessaire à la réalisation des instructions d'entrée-sortie peut être déclarée de deux manières différentes suivant que l'utilisateur désire formaliser un échange avec contrôle syntaxique, en vue d'une utilisation précise (déclaration de LPFILE) ou qu'il entend seulement bénéficier des facilités symboliques d'initialisation par une suite de mots ; dans ce dernier cas la déclaration d'IOCB permet de réserver et d'initialiser cette suite de mots en donnant un nom au premier d'entre eux.

La syntaxe d'une telle déclaration diffère de celle d'un LPFILE (cf. 4.5) seulement par la définition du "MODE" qui peut se réduire à un nombre.

```
                                  ( nombre       )
IOCB   nom  =    ( MODE :        {               }     ;
                                  ( comme LPFILE )
                       ----------
                       ----------) ;
```

POINTEURS

Considérés comme "variable mot" ils en ont les mêmes propriétés. Utilisés dans des instructions d'entrée-sortie, ils peuvent être initialisés ou chargés avec l'adresse d'une zone initialisée par ailleurs et doivent comporter le symbole d'indirection : &

Ils sont utilisés soit comme LPFILE, soit comme IOCB

```
              ( ( INPUT  )                )
POINTER      {  {        }     LPFILE      }    ;
              ( ( OUTPUT )                )
                               IOCB
```

Exemples

1) Opérations d'entrée-sortie

```
IOCB ENTER = (MODE : '88 ; EU : BI ;
              DATA : BUFFER ; EOE : 80 ; CONTROL) ;
```

La table ainsi réservée représente un échange effectif qui est une entrée sur BI avec arrêt sur compte d'octets, sans suppression des NULL

```
........
........
CONSTANT IN = 0 ;                  « ENTRÉE NORMALE
CONSTANT NSNULL = '08 ;            « NON SUPPRESSION DES NULL
CONSTANT EMOD = '80 ;              « ÉCHANGE EFFECTIF
                                   « ET RETOUR FIN D'ÉCHANGE
CONSTANT STOPCD = '40 ;            « ARRET SUR CODE
CONSTANT MODRESUL = EMOD + STOPCD + IN + NSNULL ;
                                   « L'OCTET DE FONCTION QUI
                                   « EN RESULTE EST 'C8
ARRAY 4 BYTE TABCOD = ("A", "B", "!", '0) ;
IOCB ENTRÉE = (MODE : MODRESUL ; EU : SI ;
               DATA : BUFFER ; EOE : TABCOD ; RES : 2) ;
```

La table ainsi réservée représente un échange effectif qui est une entrée sur SI avec arrêt sur code, sans suppression des NULL

```
........
IOCB SORMES1 = (MODE : OUTPUT, CA, EM ; EU : SO ;
                RES : 1 ; EOE : TABCOD ; RES : 2) ;
```

L'option absente DATA devra être chargée avant l'exécution de l'échange.

```
IOCB SORMES2 = (MODE : OUTPUT, EM ; EU : SO ; RES : 4) ;
```

Les options absentes DATA et EOE devront être chargées avant l'exécution de l'échange ; les deux derniers mots réservés servent au compte rendu d'échange.

```
IOCB SORTIE = (RES : 5) ;
```

Toutes les options devront être chargées avant l'échange.

```
CONSTANT CM = '20                    « FU chaîne de mesure
IOCB MESURE = (MODE : INPUT, EM ; EU : CM ;
               DATA : BUFFER ;
               EOE : 50 ;
               RES : 3) ;
POINTER        INPUT LPFILE PFILE ;
POINTER        IOCB PENTREE = (@ ENTREE) ;
```

2) Fonctions spéciales

L'IOCB décrivant une demande de réalisation de fonction spéciale est composé d'un seul mot.

IOCB HORTENS = (MODE : '47 ; EU : HR) ;

Le mot ainsi réservé permettra de mettre hors tension l'unité physique pointée par l'unité HR.

IOCB SAUTPAGE = (MODE : '42 ; EU : LO) ;

Saut de page sur l'unité "list output".

IOCB AVANCEBANDE = (MODE : '41 ; EU : BO) ;

Avance bande sur l'unité "binary output".

7.1.2 - UTILISATION

Les opérations que l'on peut réaliser à l'aide d'un IOCB sont exprimées par les instructions :

{ EXECUTE / WAIT }	nom d'IOCB ;

La première permet d'exécuter la fonction décrite dans la table d'échange, la seconde d'attendre la fin d'une opération d'entrée-sortie avec retour immédiat (mode IM).

Lorsque certaines options manquent dans la table d'échange, il faut les charger avant l'exécution de l'instruction EXECUTE. Le moyen le plus simple pour le faire est l'utilisation d'un pointeur initialisé avec l'adresse du mot représentant l'option manquante.

Les opérations que l'on peut réaliser sur pointeur de LPFILE (IOCB) sont :

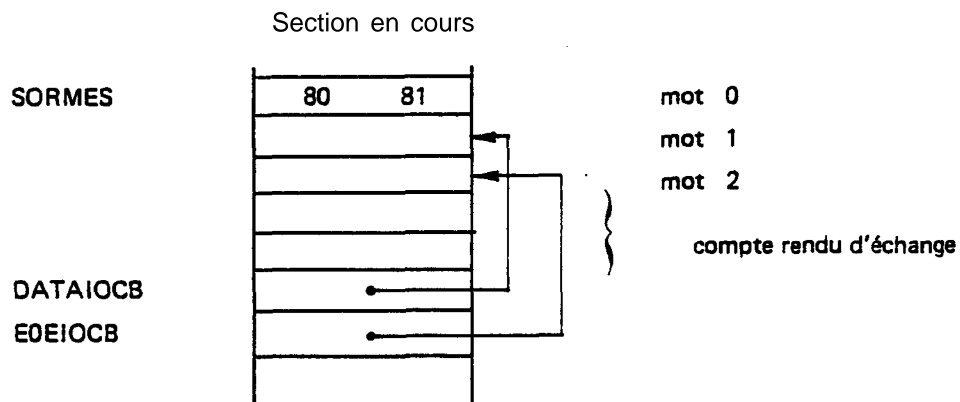
- les instructions : READ, WRITE, EXECUTE, WAIT, le pointeur étant alors précédé du symbole d'indirection.

Remarque :

Ne pas confondre les instructions d'entrées-sorties, avec les instructions portant sur la déclaration d'échange elle-même comme ASSIGN, ou assignation de registre par compte rendu d'échange.

Exemple :

```
--------
--------
IOCB SORMES = (MODE : OUTPUT, EM ; EU : SO ; RES : 4) ;
POINTER WORD DATAIOCB = (@ SORMES + 1) ;
POINTER WORD EOEIOCB = (@ SORMES + 2) ;
ARRAY 12 BYTE MES1 = ("IL FAIT BEAU") ;
ARRAY 8 BYTE MES2 = ("IL PLEUT") ;
POINTER OUTPUT LPFILE PSORM = (@ SORMES) ;
--------
--------
--------
& DATAIOCB : = @ MES1 AND '7FFF ;        «RANGEMENT DANS L'IOCB
                                         «DE L'ADRESSE DU BUFFER
& EOEIOCB : = 12 ;                       «A SORTIR, ET DU COMPTE
                                         «D'OCTETS.
WRITE & PSORM ;                          «SORTIE PREMIER MESSAGE
---------
---------
---------
& DATAIOCB) : = @ MES2 AND '7FFF ;
& EOEIOCB : = 8 ;
EXECUTE SORMES ;                         «SORTIE SECOND MESSAGE
---------
```



Le même IOCB permet ainsi de travailler sur des zones de données différentes.

7.1.3 - COMPTE RENDU D'ÉCHANGE

Le mot qui, dans l'IOCB, contient le compte rendu d'échange peut être testé après son chargement dans un registre par l'instruction.

```
Registre : = STATUS OF nom d'IOCB ;
```

Exemple :

```
RA : = STATUS OF SORMES ;
RY : = STATUS OF SORTIE ;
```

7.1.4 - INSTRUCTIONS D'ENTRÉES-SORTIES SPÉCIALES

Par l'existence de ces instructions le langage permet de demander au moniteur d'entrées-sorties certaines fonctions spéciales sans avoir à décrire l'IOCB correspondant.

Elles concernent :

- la lecture du mot d'état d'un périphérique :

```
                   ⎧ unité symbolique   ⎫ (1)
RA : = STATUS OF  ⎨                     ⎬          ;
                   ⎩ unité fonctionnelle ⎭
```

Cette instruction charge dans RA le mot d'état de l'unité physique pointée par l'unité nommée.

Exemples :

```
RA : = STATUS OF SI ;
RA : = STATUS OF HR ;
```

- les fonctions spéciales de positionnement d'une unité physique qui peut être la bande magnétique ou le disque à accès séquentiel :

```
⎧ REWIND                                          ⎫
⎪ ⎧ FORWARD  ⎫        ⎧ BLOCK ⎫                   ⎪
⎪ ⎨          ⎬ SKIP   ⎨       ⎬   ON              ⎪   ⎧ unité symbolique   ⎫ (1)
⎨ ⎩ BACKWARD ⎭        ⎩ FILE  ⎭                   ⎬   ⎨                     ⎬ ;
⎪                                                 ⎪   ⎩ unité fonctionnelle ⎭
⎪ WRITE ⎧ EOF ⎫       ON                          ⎪
⎩       ⎩ GAP ⎭                                   ⎭
```

(1) Nom ou numéro

- les fonctions ainsi réalisées sont (cf. manuel de référence de IOCS) :

  . pour REWIND : le rebobinage

  . pour WRITE EOF : écriture “Fin de Fichier”

  . pour BACKWARD SKIP BLOCK : saut arrière d'un enregistrement ou d'un secteur disque
  FORWARD SKIP BLOCK : saut avant d'un enregistrement ou d'un secteur disque

Dans le cas où l'unité physique pointée par l'unité symbolique est une bande magnétique la génération d'un espace par WRITE GAP ou les sauts avant et arrière de fichiers (. . . . . . SKIP FILE. . . .) sont possibles.

Remarque :

Pour ces instructions le compilateur génère dans la section locale accessible à cet instant un IOCB d'un mot. Si cette section n'existe pas ou est complète il y a erreur.

7.2 - SYMBOLES EXTERNES

Nous avons vu dans le chapitre précédent comment, par la notion d'externe, on peut avoir accès à un identificateur à partir de segments procédures différentes.

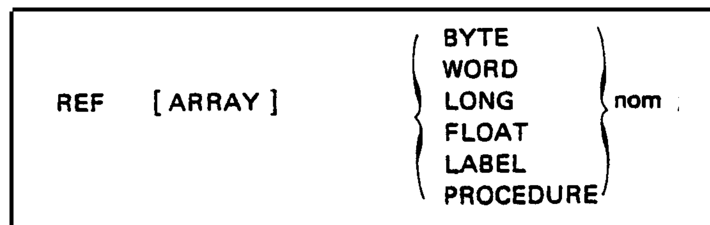
7.2.1 - DEFINITION D'EXTERNE

Elle consiste à faire précéder du symbole DEF la déclaration de l'identificateur (1) qui doit être accessible de l'extérieur. Cependant pour un LABEL ou un INDIRECT LABEL on fera précéder de DEF sa définition et non sa déclaration éventuelle.

Référence à un externe. A l'extérieur du segment où il a été défini comme externe l'utilisation de ce symbole (1) doit être précédée de sa déclaration par REF ou EXT.

7.2.2 - DECLARATION PAR REF

1) Déclaration de variables



Une déclaration de variable par référence réserve un mot dans la section où elle apparaît et initialise ce mot avec l'adresse de la variable dont le nom suit.

Implicitement l'accès à la variable se fera d'une manière indirecte à travers ce mot. Cette variable peut être une variable simple ou un tableau ; une déclaration de POINTER par REF est interdite car elle conduirait à une double indirection, ce qui est interdit par le code d'ordre SOLAR 16. Les attributs SYN et = sont interdits dans une déclaration par référence.

2) Déclaration d'étiquettes

Cette déclaration réserve un mot initialisé avec l'adresse de l'étiquette dont le nom suit. Ce mot servira de relais lors d'une instruction de branchement à cette étiquette. De plus, lors d'un tel branchement, la gestion des bases est laissée à l'utilisateur car le compilateur ne peut détecter quel sera le bloc d'arrivée.

(1) Rappel : un tel identificateur, ne peut être suivi d'une indication de synonyme.

Exemples :

MAIN PROCEDURE PRINC

```
. LOC SECTION LOC
  DEF ARRAY 5 WORD TAB ;
  DEF WORD A ;                          } déclarations
  .....
  REF LABEL L1 ;
  .....
  DEF ARRAY 4 LABEL TABETI ;
  .....
. USING LOCAL = LOC ;
  .....
  .....
  GOTO L1 ;                             } instructions
  .....
  .....
  GOTO TABETI (2) ;
```

END ;

SEGMENT PROCÉDURE PROC

```
  LOCAL SECTION LOC
  .....
  REF ARRAY WORD TAB ;
  REF WORD A ;                          } déclarations
  .....
  REF ARRAY LABEL TABETI ;
  .....
  USING LOCAL = LOC ;
  LABEL L1 ;
    GOTO TABETI (3) ;
    .....                               } instructions
                    GOTO L1 ;
    .....
  DEF L1 : ......
```

END.

3) Déclarations de procédures

REF	PROCÉDURE	nom	;

Cette déclaration réserve un mot dans la section où elle apparaît et l'initialise avec l'adresse de la procédure dont le nom suit. Ce nom doit être le nom d'une procédure déclarée par une déclaration directe précédée du mot DEF dans une autre segment procédure, ou bien encore le nom d'une segment procédure.

Exemple :

dans segment procédure P1

```
. . . . .
. . . . .
DEF PROCÉDURE PROC (X)
      . . . . . .
      . . . . . .
      . . . . . .
END ;
. . . . .
DEF ARRAY 5 PROCÉDURE TAPRO ;

CALL PROC (A) ;
. . . . .
CALL TAPRO (2) ;
. . . . .
```

dans segment procédure P2

```
. . . . . .
. . . . . .
REF PROCEDURE PROC ;
. . . . . .
. . . . . .
. . . . . .


REF ARRAY PROCEDURE TAPRO ;
. . . . . .
CALL TAPRO (4) ;
. . . . . .
CALL PROC (5) ;
. . . . . .
```

Remarqua :

La déclaration par REF d'une procédure ne doit pas faire apparaître la liste de paramètres qui peut éventuellement se trouver dans la déclaration de la procédure par DEF.

4) Déclarations d'échanges

```
      ⎧ ⎧ INPUT  ⎫   LPFILE ⎫
REF   ⎨ ⎩ OUTPUT ⎭          ⎬   nom   ;
      ⎩   IOCB              ⎭
```

Cette déclaration réserve un mot dans la section où elle apparaît et l'initialise avec l'adresse de la table nécessaire à IOCS pour réaliser l'échange. La description complète de l'échange est faite dans la déclaration DEF dans une autre segment procédure. Une déclaration par référence de LPFILE spécifie simplement si le mode est INPUT ou OUTPUT afin que le compilateur puisse effectuer une vérification syntaxique d'utilisation.

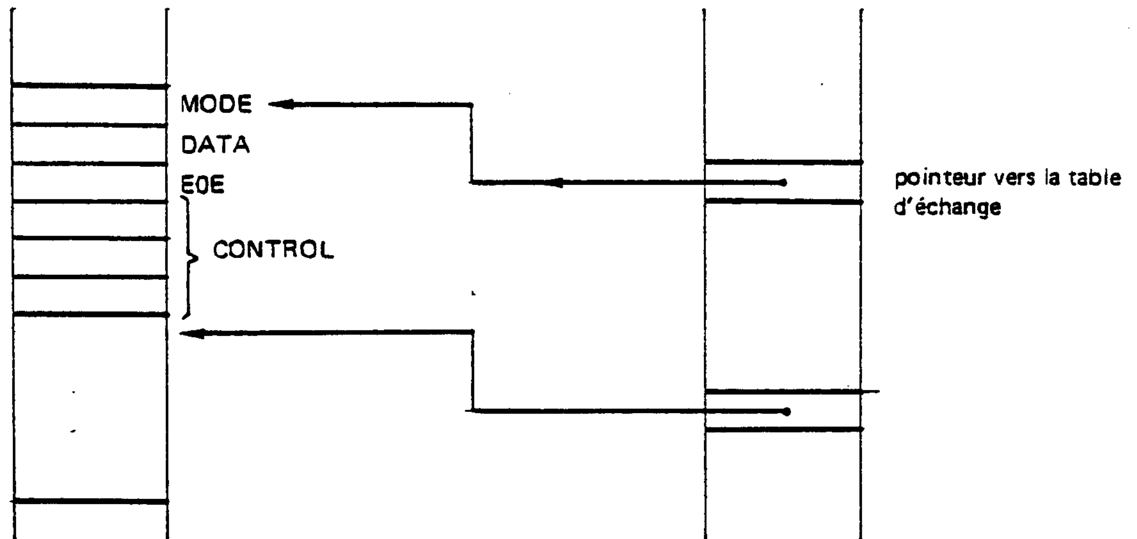
Exemple :

dans segment procédure P1

```
......

DEF LPFILE COMAND = (
      MODE : INPUT, CA, EM, NSN ;
      EU : PC ;
      DATA : BUFFER ;
      EOE : TABCODE ;
      CONTROL) ;
DEF IOCB ENTREE = (RES : 5) ;
......
READ COMAND ;
......
```

dans segment procédure P2

```
.......
.......
REF INPUT LPFILE COMAND ;

.......
.......

REF IOCB ENTREE :
......
READ COMAND ;
EXECUTE ENTREE ;
......
```



4) Déclaration de sections

Lorsque l'ouverture d'une section de données est précédée du mot REF, cela signifie que cette section est déclarée par DEF dans une autre segment procédure. Les déclarations qui suivent permettent seulement d'en définir les éléments pour le compilateur, mais aucune place n'est réservée.

Les initialisations des variables déclarées dans cette section sont donc interdites.

Ainsi par cette déclaration on redécrit une section existant ailleurs afin d'avoir accès aux variables qu'elle contient ; son adresse ne sera connue qu'a l'édition de liens des différentes segments procédures.

Cette méthode évite de déclarer par REF et DEF toutes les variables d'une section.

Exemples :

```
MAIN PROCEDURE P1
  . DEF COMMON SECTION CI
    WORD A ;
    BYTE B ;
    ARRAY 4 WORD TAB ;
    .......
  . USING COMMON
    .......
    .......

    TAB (0) : = TAB (1) + A ;
    .......
```

```
SEGMENT PROCEDURE P2
  . REF COMMON SECTION CI
    WORD D ;
    BYTE B ;
    ARRAY 4 WORD TAB ;
    .......
  . LOCAL SECTION LOC2
    .......
    .......

    .......
    TAB (3) : = D ;
    .......
```

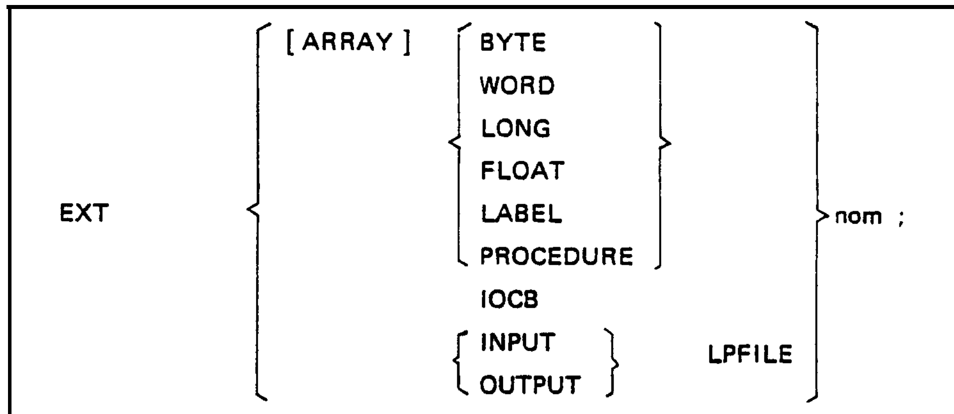
Les noms donnés aux variables dans une REF section peuvent être différents de ceux donnés aux variables correspondantes dans la DEF section. Ainsi dans l'exemple précédent les noms A et D permettent d'accéder au même mot mémoire qui a été réservé lors de la déclaration de A.

Remarque :

La déclaration par REF d'une section de données réserve dans la dernière section ouverte ou si elle n'existe pas dans la section de code un mot initialisé avec l'adresse de la section - 128. Ainsi dans l'exemple précédent ce mot est réservé en tête de la section de code de P2.

Nous verrons une utilisation de la déclaration REF KSTORE.... au paragraphe 9.2.3 c).

7.2.3 - DECLARATION PAR EXT



Contrairement à la déclaration par REF une déclaration EXT ne réserve aucun mot dans aucune section du programme compilé, mais indique au compilateur qu'un élément de même nom et même type existe. Toute apparition de ce nom sera précédée du symbole “adresse” (@), elle correspondra à une constante adresse initialisée avec l'adresse de définition de cet externe.

Utilisée en partie initialisation d'une déclaration, l'adresse d'un externe ne peut être combinée avec une constante ou une autre adresse.

L'écriture :

POINTER ARRAY PIOCB = (@ ENTREE + 1) ; est interdite.

Exemples :

dans segment procédure P1

```
. . . . . . . .
. . . . . . . .
DEF WORD A ;
. . . . . . . .
. . . . . . . .
```

dans segment procédure P2

```
. . . . . . . .
. . . . . . . .
EXT WORD A ;
. . . . . . . .
POINTER WORD PA = (@A) ;
. . . . . . . .
RA : = & PA + 2 ;
```

dans segment procédure P1

```
. . . . . . . . .
. . . . . . . . .
DEF PROCEDURE PROC1
      . . . . . .
      . . . . . .
      . . . . . .
END ;
. . . . . . . . .
. . . . . . . . .
```

dans segment procédure P2

```
. . . . . . . .
EXT PROCEDURE PROC1 ;
POINTER PROCEDURE PROC = o PROC1) ;

. . . . . . . .

. . . . . . . .

CALL PROC
. . . . . . . .
```

dans segment procédure P1

```
. . . . . . .
DEF IOCB ENTREE =
(MODE : INPUT, EM ; EU :
SI ; RES : 4) ;

. . . . . . .
. . . . . . .
EXECUTE ENTREE ;

-------
```

dans segment procédure P2

```
. . . . . . .
EXT IOCB ENTREE ;
POINTER ARRAY WORD PIOCB = (@ ENTREE OR '8000)
POINTER OUTPUT FILE PENTREE = (@ ENTREE) ;
ARRAY 4 WORD BUF ;
. . . . . . . .
. . . . . . . .
& PIOCB (1) : = @ BUF AND '7FFF ;
READ PENTREE ;
```

Une déclaration par EXT permet d'éviter des conflits de noms dans un bloc :

```
. . . . . . .
. LOCAL SECTION LOC1
      EXT WORD A ;
      POINTER WORD PA1 = (@A) ;

. . . . . . . .
. LOCAL SECTION LOC2
      POINTER WORD PA2 = (@ A) ;
. . . . . . .
. . . . . . .
. USING LOCAL = LOC1 ;
      &PA1 := 4 ;
      . . . . . .
      . USING RL = LOC2 ;
        & PA2 : = TOTO ;
        -------
. . . . . . .
```

La déclaration : REF WORD A dans chacune des sections, serait une double définition d'identificateur dans un même bloc.

7.3 - GENERATIONS IMPLICITES

7.3.1 - DE DONNEES

En cours de compilation, le compilateur peut être amené à générer, sans réservation explicite par des déclarations, des mots contenant des informations nécessaires à la compilation.

Dans ce cas, il le fait toujours dans la section locale accessible à cet instant, si celle-ci n'existe pas ou est complète il y a diagnostic d'une erreur.

Ceci se produit dans les cas suivants :

- génération d'un mot pour une constante ne tenant pas sur un octet (paragraphe 5.1 page 56)
- génération d'un mot dans le cas d'une division ou d'une multiplication par un littéral (paragraphe 5.3.2 page 67'"
- génération d'un mot initialise par une adresse dans le cas des expressions entières (paragraphe 5.3.3 page 73).
- génération des IOCB d'un mot dans le cas d'instruction d'entrées-sorties n'utilisant pas de tables d'échange prédéclarées (paragraphe 7.1.4 paragraphe 8.2.4)
- génération d'un relais dans le cas d'un branchement arrière long
- génération de deux mots dans le cas de l'instruction CASE OF.
- nous avons vu qu'une déclaration de procédure réserve dans la section en cours un mot initialisé avec l'adresse du point d'entrée de cette procédure si cette section n'est pas accessible au moment de l'appel de la procédure le compilateur génère automatiquement un relais dans la section locale accessible à cet instant. Si cette dernière n'existe pas ou est complète l'utilisateur a encore la possibilité de déclarer un "pointer" (POINTER PROCEDURE) dans une des sections accessibles pour rendre cet appel possible.
- génération des adresses des sections à charger dans les registres de base lors d'une instruction USING = dans le bloc d'une instruction composée (cf. chargement des bases, en annexe).

7.3.2 - D'INSTRUCTIONS

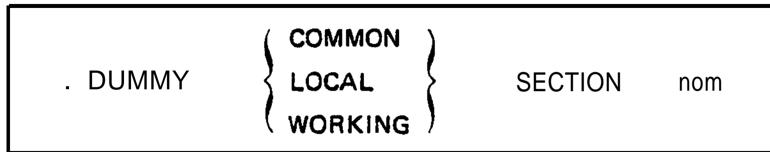
Nous ne citons ici que les cas spéciaux ou la puissance du langage nécessite l'utilisation de registre ou de mémoire non cités explicitement par l'utilisateur, ce sont :

- l'évaluation d'expressions parenthésées entière ou flottante utilise les registres RA, RB, RY les mémoires d'adresse 0 et 1 dans le local courant, la KSTORE section.
- le chargement d'un registre par le premier terme d'une expression peut nécessiter le passage par un autre registre, après sauvegarde de celui-ci (expression de registre et de RA page 66 et 68)
- le calcul de tout indice est fait dans le registre index RX supposé libre pour cet usage.

7.4 - COMPLEMENTS SUR LES DECLARATIONS DE SECTIONS

7.4.1 - SECTIONS DUMMY

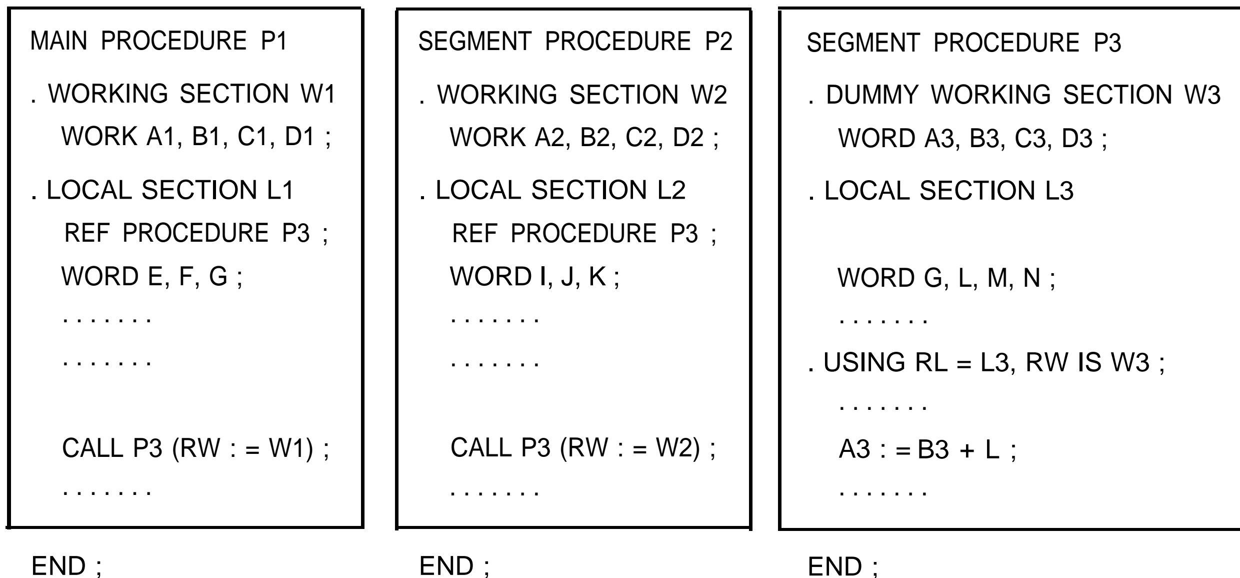
La déclaration de telles sections se fait de la manière suivante :



Lorsque l'ouverture d'une section est précédée du mot DUMMY, cela signifie que la section en question est en réalité l'image d'une ou de plusieurs autres sections dont l'adresse sera passée comme paramètres.

Les déclarations qui suivront ne réserveront aucune place mais définiront, pour le compilateur, la structure de ces sections c'est-à-dire l'emplacement de chaque élément déclaré dans la section. Ainsi aucune initialisation de variable ne sera possible dans une dummy section.

Exemple :



La notion de section DUMMY permet ainsi à une procédure de travailler sur une zone qui ne lui appartient pas ; la base correspondante est chargée par l'appelant dans un appel de la procédure avec assignation de base. De cette manière la zone réelle dont la section dummy est l'image est fournie à la procédure.

L'instruction USING portant sur une DUMMY SECTION ne peut qu'être USING IS et non USING = puisque la section est virtuelle. Le chargement de la base n'a lieu qu'au moment de l'appel de la procédure.

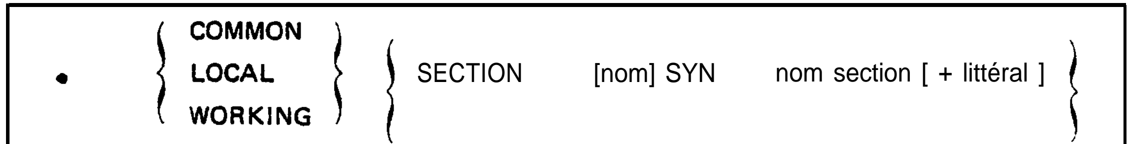
Ainsi, dans notre exemple, W3 est l'image des sections W1 et W2, celles-ci étant les sections paramètres de la procédure P3. Le chargement de la base RW a lieu au moment de l'appel de P3.

Remarque :

Dans le cas d'une DUMMY section on définit une section virtuelle dont l'adresse sera connue à l'exécution du programme alors que dans le cas d'une REF section, on définit une section dont l'adresse sera connue à l'édition de liens.

7.4.2 - SECTIONS SYNONYMES

L'attribut SYN dans une déclaration de section a la même signification que pour une variable simple. Il signifie que les déclarations suivantes sont une restructuration ou une réutilisation de la section dont le nom suit :

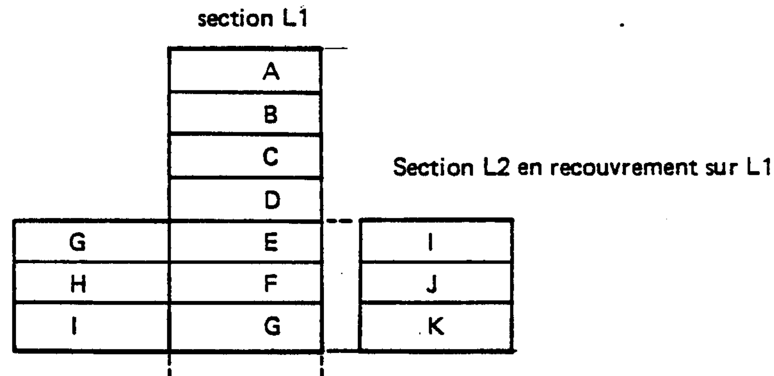


On peut ainsi avoir des zones de données qui se recouvrent, ce qui permet à plusieurs traitements indépendants entre eux d'utiliser une même zone de travail sans risque d'erreur.

Une section ouverte par SYN peut recevoir un nom ou pas ; dans le premier cas une instruction USING sur la section ainsi nommée sera nécessaire pour la rendre accessible car il s'agit d'une nouvelle section ; dans le deuxième cas il s'agit simplement d'un repositionnement dans la section dont le nom suit.

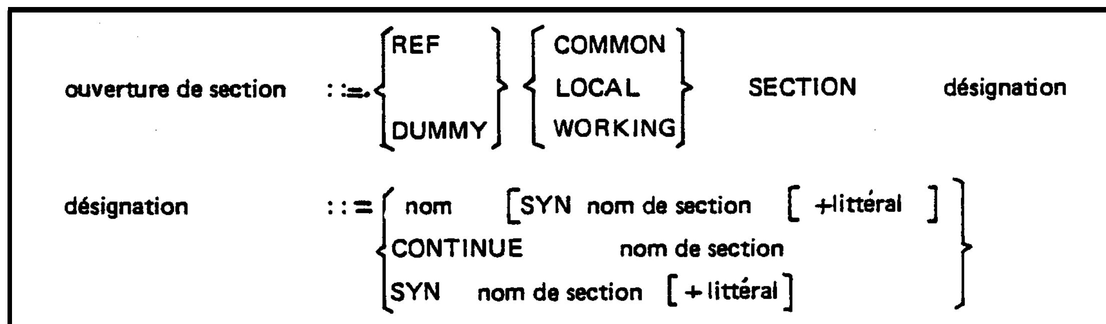
Exemple :

```
        ........
      81 : BEGIN
        . LOCAL SECTION L1
              WORD A, B, C, D, E, F, G ;
        . USING LOCAL = L1 ;                          Accès à la section L1
              .......
              .......
        B2 : BEGIN
            . LOCAL SECTION L2 SYN L1 + 4
              WORD I, J, K ;
            . USING LOCAL = L2 ;                      Accès à la section L2

            END ; « Fermeture définitive de L2, L2 n'est plus
              - - - « accessible ; L1 redevient accessible
        B3 : BEGIN
            . LOCAL SECTION SYN L1 + 4
                        « réutilisation de la section L1
              WORD G, H, I ;                          Accès à la section L1
            . USING LOCAL IS L1 ;
              A  : = B ;
              H  : = I + G ;
            END ;
              .....
        END ;                  « Fermeture définitive de L1
        ......
        ------
```



Dans cet exemple B1 sous-traite une partie de ses données à B2 et B3 (E, F, G) B2 ouvre une nouvelle section (L2) et devra donc charger la base L ; B3 utilise la même section que B1. Ainsi B2 et B3 n'ont pas à connaître le nom des variables de B1.

7.4.3 - FORME GENÉRALE POUR UNE OUVERTURE DE SECTION



Ce tableau résume tout ce qui a été vu au sujet des déclarations de sections.

Les options SYN ou CONTINUE sont interdites dans une déclaration précédée d'une spécification REF ou DUMMY.

Une section réouverte par l'option CONTINUE a les mêmes caractéristiques que la section précédemment ouverte DUMMY ou REF. Toute section ouverte dans un bloc est définitivement fermée par la fermeture du bloc dans laquelle elle a été ouverte (END du bloc).

Les déclarations sont toujours chargées dans la dernière section ouverte ; ainsi une section est momentanément abandonnée à l'ouverture ou la réouverture d'une autre section.

Une déclaration de section comportant un nom dans la partie désignation correspond à l'ouverture d'une nouvelle section de données ; en l'absence de ce nom ii s'agit d'une réouverture ou d'une redéfinition par les options CONTINUE et SYN d'une section qui a été abandonnée précédemment et qui n'est pas encore fermée.

Exemple :

```
MAIN PROCÉDURE ESSAI
 .  DEF COMMON SECTION COM          « ouverture de COM pour déclarations
      WORD CC, CD, CE ;
 .  LOCAL SECTION LOC               « ouverture de LOC pour déclarations
      WORD LC, LD ;                 « abandon de COM
 .  USING RC = COM, RL = LOC;       « accès à COM et LOC
      .......                       « pour les instructions

      BEGIN
       .  LOCAL SECTION L1          « ouverture de L1
            WORD A, B, C, D ;
       .  USING RL = L1 ;           « accès à L1 pour les instructions
          ......
            BEGIN
             .  LOCAL SECTION L2
                  WORD E, F, G, H ;
             .  USING RL = L2 ;        « accès à L2 pour les instructions
                ......
            END ;                      « fermeture définitive de L2
            ......                     « la section locale courante est L1
            BEGIN
             .  LOCAL SECTION CONTINUE L1
                  WORD I, J, S ;
                .......
            END ;
            BEGIN
             .  LOCAL SECTION SYN L1 + 4
                  WORD K, L, M ;
                  ......
            END ;

            ......
            BEGIN
             .  LOCAL SECTION CONTINUE L1
                  WORD P, Q ;
             .  ......
                  BEGIN
                   .  WORKING SECTION W1
                        WORD WORK1, WORK2 ;
                   .  USING RW = W1 ;
                      .......
                  END ;
```

Section COM

CC
CD
CE

Section LOC

LC
LD

Section L1

A B C D	recouvrement
I J S	K L M
P Q	

Section W1

work1
work2

```
|       |
|     END ;                     « fermeture de L1
| .......                       « la section locale courante est LOC
|     BEGIN
|       |  . LOCAL SECTION L1     « ouverture nouvelle section L1
|       |       WORD A, B, C, D, E ;
|       |  . USING RL = L1 ;
|       |    ......
|       |    ......
|     END ;                     « fermeture de L1
| .......
| .......
END.        (Fermeture COM et LOC)
```

7.5 - INSTRUCTIONS MACHINE

PL16 permet l'utilisation directe, dans un programme, des instructions machine. Il est, en effet, possible d'attacher un nom à un code ou à une valeur numérique et d'insérer celui-ci avec ses paramètres dans le code généré. Ceci permet d'inclure dans PL16 un assembleur et d'atteindre les instructions machine que le langage n'utilise pas (par ex. l'instruction PTY qui est le calcul de la parité d'un octet...)

Ceci est rendu possible par :

- la déclaration d'instruction
- la génération du code de cette instruction lors de son utilisation qui peut comporter ou non de paramètres

La déclaration de la plupart de ces "instructions" est évidemment inutile pour l'efficacité, comme pour la puissance du langage, elle est seulement rendu possible par la nécessité d'avoir à en déclarer certaines.

Les instructions effectivement utilisées sont soulignées, en face des formats correspondants (voir ci-dessous), ce sont :

- ADR, LR qui permettent l'accès aux registres RC, RL, RW, RK protégés par le langage
- SVC appel des sous-programmes superviseur (accès fichier)
- PTY partie d'octet

7.5.1 - DECLARATION

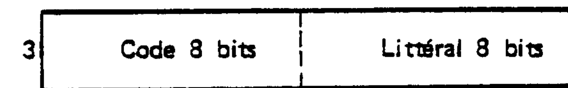
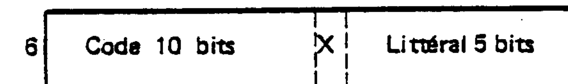
déclaration d'instruction ::= INSTRUCTION nom (format, code) ;

format indique le format de l'instruction, c'est-à-dire le type des paramètres qui pourront apparaître lors de l'utilisation de l'instruction.

format ::= 0, 1, 2, 3, 4, 5, 6, 7,

code est un littéral tenant sur un mot, représentant le code instruction avec certains champs initialisés ou non. En général on mettra dans le code le champ invariant de l'instruction. Au moment de l'utilisation de l'instruction les champs variables seront chargés par le compilateur suivant la valeur des paramètres indiqués, en effectuant un "OU" logique entre le code donné et la valeur des paramètres de l'instruction.

nom est un symbole quelconque qui peut être le même que le code mnémonique machine, pour plus de clarté.

Formats autorisés	Instructions correspondantes
	Littéral 16 bits RSR, RSV, ACQ, ACK, HALT, PTY, STEP, ROMB. SCY, ACTD, QUIT, DIT, EIT, RDHV, IPI, LAR, STAR, RDOE, WOE, MVTS, MVTM, DBP, SBP, RBP, SST, RST, RDSI, RCDA, WCDA, RBTM, SBTM, DRBM, INSQ, SFQ, SUPQ, SLQ.
	Registre registre XR, LR, ORR, ANDR, EORR, CLSR, CMR, ADR, SBR, NGR, ADCR, SBCR, CPR, CPZR, SWBR, XIMR
	Littéral 8 bits ou label SVC
	Variable LA, LB, LX, LY, LAD, STA, STB, STX, STY STZ, OR, AND, EOR, AD, SB, MP, DV, CP, CPZ, IC, DC, LBY, STBY, CPBY, BR, BSR, SIO, XM
	Registre, littéral 8 bits ADRI
	Indexation, littéral 5 bits RBT, SBT, IBT, TBT, DBT, SLRS, SLLS, SCRS, SCLS, SARS, SLRD, SLLD, SCRD, SCLD, SARD
	Littéral 8 bits signé LAI, LBI, LXI, LYI, CPI, ORI, ANDI, EORI.

7.5.2 • UTILISATION

Une instruction machine qui a été déclarée peut alors apparaître dans les instructions PL16 d'un programme
Une instruction machine sera donc :

```
nom d'instruction [ (attribut, attribut...) ] ;
```

Les attributs correspondent dans le cas où ils existent aux champs de l'instruction non initialisés lors de la déclaration. Ils peuvent être suivant les formats et les champs qu'ils sont amenés à initialiser :

```
attribut ::=  [-]  littéral
              nom de variable
              nom de procédure
              nom de label
              registre quelconque
```

Ce sont en fonction des formats :

Format	0 :	aucun attribut
Format	1 :	un littéral 16 bits
Format	2 :	un nom de registre [, un nom de registre]
Format	3 :	un littéral 8 bits ou 7 bits signés ou label (relatif)
Format	4 :	un nom de variable, de label ou de procédure avec indication possible du symbole d'indirection &.
Format	5 :	un littéral 8 bits ou 7 bits signés, un nom de registre
Format	6 :	[RX,] littéral 5 bits
Format	7 :	littéral 8 bits signés

Quel que soit le format, les attributs sont facultatifs ; s'ils sont absents le compilateur génère le code donné à la déclaration. Par contre s'ils sont présents ils doivent être conformes au format donné dans la déclaration.

Remarque :

Dans une instruction de format 2, si un seul registre apparaît lors de son utilisation le compilateur le prend pour registre "source". Pour les instructions ou le seul registre paramètre est un registre "destination", on aura intérêt à utiliser des déclarations avec format 3 ou 1, le paramètre étant le numéro de registre.

Exemples :

```
(*)---------
    INSTRUCTION PTY (0, '1E07) ;
    INSTRUCTION ACTD (0, 1E05) ;
    INSTRUCTION CP32 (0, '1520) ;        le code ainsi initialisé correspond à l'instruction machine
                                         CPI 32
    INSTRUCTION ADD (2, '2C00) ;
    INSTRUCTION SVC (3, '1C00) ;
    INSTRUCTION SIO (4, '0700) ;
    INSTRUCTION ADRI (5, 0800) ;
    INSTRUCTION CPI (7, '1500) ;
    INSTRUCTION DECAL (6, '2980) ;
    INSTRUCTION ECHR (2, '2B80)
    CONSTANT FMS = '38
    WORD DISPACK = ('5C02) ;
        ADD (RX, RA) ;
        ADD (RC, RA) ;
        DECAL (4) ;
        DECAL (RX, 4) ;
        SIO (DISPACK) ;
        PTY ;
        ACTD ;
        SVC (FMS) ;               « le code génère est 1C38
        CP 32 ;                   « le code génère est '1520
  L1 :  CPI (-2) ;
        ADRI (-2, RB) ;
```

(*) Les registres RK, RC, RL, RW qui, dans le langage, ne peuvent apparaître que dans les instructions USING ou d'assignation de base pourront être atteints dans un programme PL16 par les instructions machine.

7.5.3 - CODES INSTRUCTIONS SOLAR 16

Nous ne donnons ci-dessous, les codes des instructions non accessibles explicitement par le langage et qu'il peut être utile de connaître.

Il importe de se souvenir, que les instructions concernant les registres RK, RL, RC, RW sont à utiliser avec la plus grande prudence et dans des cas très particuliers.

Une telle utilisation est en effet contraire au principe qui consiste à laisser au compilateur le soin de générer les instructions qui traduisent la structure du programme.

Pour les instructions du type “avec opérande mémoire” (format 4) la partie mode d'adressage (base, indirection) est complété par le compilateur en fonction de l'opérande paramètre.

format	instruction	code
0,1	ACQ	1E00
	ACK	1E0D
	ACTD	1E05
	PTY	1E07
	QUIT	1E06
	RSR	1E02
2	LR	2B00
	ADR	2D00
3	SVC	1C00
4	SIO	0700
	BIO	0300
	BSR	0600
5	ADRI	0800
6	–	–
7	–	–

8 - OPTION "SCHEDULER" ET INSTRUCTIONS DIVERSES

Remarque préliminaire :

Bien que la notion de "tâche hardware" existe, en dehors de cette option, nous présentons seulement dans ce chapitre l'ensemble des possibilités du langage PL16 quant à la définition et l'utilisation de tâches.

8.1 - DECLARATION DE TACHE

Cette déclaration permet de réserver et d'initialiser, dans la section en cours, la PST (16 mots) d'une tâche :

```
déclaration de tâche : :  { HARD } TASK [priorité de la tâche] (attributs) ;
                          { SOFT }

attributs ::              RA = expression de constante,
                          RB = expression de constante,
                          RX = expression de constante,
                          RY = expression de constante,
                          RC = nom de section,
                          RL = nom de section,
                          RW = nom de section,
                          RK = nom de section,
                          START = { nom de label     }  (1)
                                  { nom de procédure }
                          [MASTER],
                          RSLO = expression de constante,
                          RSLE = expression de constante,
                          XTNUSR = expression de constante,
                          XAPPMAX = expression de constante,
                          XPARTITION = expression de constante,
                          XSYSTEME = expression de constante.
```

Si la priorité de le tâche n'est pas indiquée dans la déclaration, ii sera demandé au moment du chargement en mémoire.

Il doit y avoir au moins un attribut ; lorsqu'un registre est absent il est initialisé à 0. Le registre d'état est initialisé à '8000 dans le cas où l'attribut MASTER est présent, à 0 (mode esclave) lorsqu'il est absent.

Dans le cas où l'option DRIP 16 de SOLAR 16 est présente, les registres RSLO et RSLE (registres SLO et SLE) du calculateur) contiennent respectivement les adresses de début et de fin d'implantation d'une tâche esclave ; ils définissent la zone accessible par cette tâche. Au moment du chargement en mémoire d'une tâche esclave, le chargeur met dans le registre SLO de la PST l'adresse d'implantation de la tâche et translate SLE de l'adresse d'implantation.

(1) le nom utilisé ici a dû apparaître avant la déclaration TASK, dans une déclaration d'externe.

Les quatre derniers mots concernent les systèmes RTES-D et RTES-C et sont respectivement :

- numéros de tâche et d'usager
- nombre d'appel maximal
- numéro de partition
- réservé au système.

Exemples :

SOFT TASK 2 (RL = LOC, MASTER) ;

Réservation de la PST de la tâche "software" numéro 2 avec initialisation du registre L et du registre d'état.

SOFT TASK (RC = C1, RK = PILE MASTER) ;

Réservation de la PST d'une tâche "software" dont la priorité ne sera connue qu'au chargement en mémoire.

```
SOFT TASK 125   (RB    =   3,
                 RC    =   C1,
                 RL    =   L1,
                 RW    =   W1,
                 RK    =   PILE,
                 START =   ETIQ1,
                 RSLO  =   0
                 RSLE  =   @ ADFIN) ;
```

Réservation de la PST de la tâche "software" de priorité 125. Cette tâche démarre en ETIQ1 et se déroulera en mode esclave. Ses limites inférieures et supérieures au moment de son exécution seront respectivement :

. adresse d'implantation

. adresse d'implantation + @ ADFIN

Exemples

```
SEGMENT PROCEDURE TRAITE2
  . LOCAL SECTION LOC
      PROCEDURE TACHE2              « TRAITEMENT NIVEAU2
        | .......
        | . USING LOCAL IS LOC ;
        | .......
        | .......
      END ;
  HARD TASK 2 (RA = 1, RB = 2,
               RL = LOC,
               START = TACHE2
               MASTER) ;
. USING LOCAL = LOC ;
  .......                           « ATTENTE INTERRUPTION NIVEAU 2
END ;
```

La PST ainsi décrite est celle de la tâche "hardware" de niveau 2 ; elle démarre en TACHE2 et se déroulera en mode maître. La base RL est initialisé par l'adresse de la section utilisée par la tâche.

La procédure TACHE2 décrit une tâche qui sera lancée à un instant donné ; la PST sera alors chargée dans les registres. De cette manière il n'est pas nécessaire de charger les bases dans la tâche ; une instruction USING 15... sera donc suffisante.

. . . . . .

HARD TASK 4 (RB = 0) ;

Tous les registres sont initialisés à 0. Un paramètre au moins étant nécessaire on a choisi simplement la place de la PST, celle-ci sera chargée au moment d'une sauvegarde de contexte.

8.2 - MULTITRAITEMENT

Le langage PL16 permet d'utiliser les possibilités SOLAR 16 concernant la gestion des tâches et des ressources

8.2.1 - DECLARATION DE RESSOURCE

```
déclaration de ressource ::= { RESSOURCE   nom = ENTRY nombre, PRMAX    nombre  } ;
                             { PRIVATE     nom = ENTRY nombre                   }
                             { PRIVATE     nom = ENTRY nombre, MESSMAX  nombre  }
```

Ces 3 possibilités permettent de définir respectivement :

- un sémaphore d'exclusion et la file d'attente de ses demandeurs
- un sémaphore privé
- un sémaphore privé avec liste de paramétres.

Elles réservent respectivement dans la section en cours

- **1 + ((PRMAX + 16)/16) mots** : [| compteur] [File de 8 mots (128 bits) maximum]
- **1 seul mot** [| compteur]
- **1 + ((MESSMAX + 16)/16) mots :** [| compteur] [File des paramètres]

(bits 0 ... 15)

Le nombre qui suit ENTRY initialise le compteur noté ci-dessus, ce sera :

- dans le cas d'un sémaphore d'exclusion, le nombre d'accès simultanés c'est-à-dire le nombre maximum d'utilisateurs qu'elle peut satisfaire. En général ce nombre d'accès est égal à 1.
- dans le cas d'un sémaphore privé, le nombre d'activation non exécutées c'est-à-dire en attente. En général ce nombre est initialisé à 0 car la tâche gérée par ce sémaphore n'a pas encore été mise en attente, ni activée.
- PRMAX permet de définir le numéro maximum de la tâche demandeur.
- MESSAMX définit le numéro maximum du paramètre qui sera passé au moment de l'activation de la tâche gérée par ce semaphore privé.

```
déclaration de                 { POINTER [MESS] PRIVATE nom ; }
pointeur de ressource ::=      { POINTER RESSOURCE nom ;      }
```

Comme pour toute autre déclaration du même genre, la déclaration par pointeur réserve seulement un mot, accessible en tant que tel, ou utilisable, précédé du symbole d'indirection, comme Ressource ou Private dans les instructions appropriées. On peut ainsi, comme pour les entrées-sorties, réaliser soi-même les réservations et initialisations nécessaires à la définition de ressources (initialisation de tableau), tout en utilisant les instructions adéquates grâce à des pointeurs.

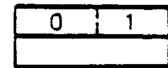
Exemples :

. . . .

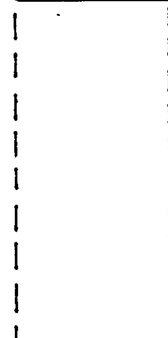
. LOCAL SECTION LOC

SECTION LOC

RESSOURCE IMPRIMANTE = ENTRY 1, PRMAX 12 ;



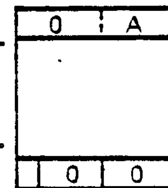
Cette déclaration définit le moyen de travail IMPRIMANTE qui sera utilisé en concurrence par 12 demandeurs au maximum.
Elle réserve un mot dont l'octet droit est initialisé à 1 et un mot pour la file des demandeurs (12 bits, les 4 derniers bits ne seront pas utilisés).



. . . .

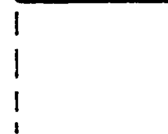
. . . .

RESSOURCE VISUS = ENTRY 10, PRMAX 96 ;



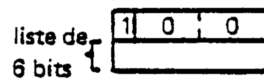
La file des demandeurs est ici une file de 96 bits (soit 6 mots)

PRIVATE SOFT2 = ENTRY 0 ;

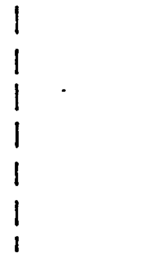


Le sémaphore privé SOFT2 ainsi défini permettra de gérer la tâche qui "s'appropriera" le sémaphore lors d'une instruction WAIT

PRIVATE SOFT10 = ENTRY 0 ; MESSMAX 6 ;

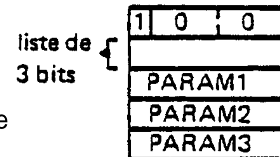


La sémaphore privé avec liste de paramètres ainsi défini, permettra la gestion d'une tâche. Un numéro de paramètre allant de 0 à 5 sera enregistré dans la liste en même temps que la demande d'activation de la tâche qu'il gère.



. . . .

PRIVATE SOFT24 = ENTRY 0, MESSMAX 3 ;
WORD PARAM1, PARAM2, PARAM3 ;



La passage d'un paramètre lors de l'appel d'une tâche gérée par sémaphore privé doit se faire dans une convention fixée par le programme
Une solution peut être de faire suivre le sémaphore de n mots (si n est le nombre de bits de la liste) où le demandeur place ses paramètres

L'adresse de sa table de paramètres
(ou son paramètre s'il est unique)
Dans cet exemple il y a 3 paramètres possibles pour la tâche gérée par le sémaphore SOFT24. Celle-ci doit tester l'état de la liste lors de son activation pour connaître le numéro du paramètre passé. En PL16 le programmeur pourra tester la liste par l'intermédiaire d'un pointeur

```
POINTER WORD PTFILE = (@ SOFT24 + 1) ;

POINTER ARRAY WORD TABPARAM = (@SOFT24 + 2 OR '8000) ;
                                   « le pointeur pointe un tableau
                                   « contenant les 3 paramètres
.........
.........
RA : = &PTFILE ;                  « RA contient la file de paramètres
RX : = FIRST 1 IN RAB ;           « RX contient ainsi le numéro du paramètre passé
RA : = &TABPARAM (RX) ;           « RA contient le paramètre,
                                  « il pourra être exploité par
.........                         « la tâche activée
.........
```

8.2.2 - INSTRUCTIONS PRIVILEGIEES

Les instructions PL16 qui permettent la synchronisation des tâches sont :

```
ARM (no tâche)
QUIT
ACTIVATE (nom de private [, n° de paramètre ] ) ;
WAIT (nom de private ) ;
REQUEST (ressource) ;
RELEASE (ressource) ;
```

Excepté QUIT ces instruction; ne peuvent être utilisées qu'en mode "MAITRE"

Elles permettent respectivement (cf. manuel de référence SOLAR 16) :

- d'armer une tâche "software" de priorité donnée
- d'acquitter une tâche "software"
- de demander l'activation de la tâche "software" qui s'est "approprié" le sémaphore privé. Cette demande spécifie un numéro de paramètre qui est passé par le registre RY (RY sera donc détruit) lorsque le sémaphore privé a été défini avec une liste de paramètres.
- à une tâche "software" de se mettre en attente et de "s'approprier" le sémaphore privé indiqué (l'instruction WAIT est interdite sous niveau "hardware").

Le jeu des instructions ACTIVATE et WAIT permettent de synchroniser des tâches

- de demander un accès à la ressource qu'on indique (l'instruction REQUEST est interdite sous niveau "hardware")
- de rendre une ressource et ainsi de rendre un accès à cette ressource

Le jeu des instructions REQUEST et RELEASE permettent de gérer une ressource et de synchroniser les tâches qui demandent des accès à cette ressource.

Exemples :

En reprenant les déclarations du paragraphe précédent on peut écrire :

TACHE SOFT 2

```
E2     : ------
--------
REQUEST (IMPRIMANTE) ;
WRITE MESSAGE ;

--------          « UTILISATION
--------          « IMPRIMANTE

RELEASE (IMPRIMANTE) ;

--------

--------

--------

WAIT (SOFT2) ;    «MISE EN
                  « ATTENTE DE
                  « LA TACHE
                  « SOFT 2
--------

--------

--------

--------

QUIT ;            « ACQUITTEMENT
                  « TACHE SOFT2

GOTO E2           «POUR RELANCE
                  «ULTERIEURE
```

TACHE SOFT 6

```
---------
E6      :-------
------
REQUEST (IMPRIMANTE) ;
------
------
ARM (2) ;
------
------
RELEASE (IMPRIMANTE) ;

------

------

ACTIVATE (SOFT2) ;   « DEMANDE
                     « D'ACTIVATION
                     « DE LA TACHE
                     « GERÉE PAR
                     « SOFT 2
------

------

------

QUIT ;
GOTO E6 ;
```

Dans cet exemple la tâche 6 peut être lancée par ailleurs ; elle se synchronisera ensuite avec la tache 2 plus prioritaire suivant que la ressource IMPRIMANTE sera libre et selon le jeu des instructions WAIT et ACTIVATE.

Un programme PL16 comportant ces 2 tâches pourrait s'écrire :

```
MAIN  PROCEDURE  TACBOS
   . KSTORE  SECTION  PILE1
         RES 20 ;
   . LOCAL SECTION LOC
         RESSOURCE IMPRIMANTE = ENTRY . . . . .
         PRIVATE SOFT2 = . . . . . . .
         KSTORE  SECTION  PILE2 ;
         ------  RES  50  ;
         PROCEDURE   TACHE2
           . LOCAL SECTION
                 CONTINUE LOC
              . . . . . . .
           . USING RL IS LOC ;
              E2 :
              . . . . . . .
         END ;
         KSTORE  SECTION  PILE6 ;
         . . . . . . . RES 5O ;
         PROCEDURE   TACHE6
           . LOCAL  SECTION
                 CONTINUE  LOC
              . . . . . . .
           . USING  RL  IS  LOC ;
              E6 :
         END ;
         SOFT TASK 2 (RL = LOC, RK = PILE2,
                        START = @TACHE2 MASTER) :
         SOFT TASK 6 (RL = LOC, RK = PILE6,
                        START = @TACHE6, MASTER) ;
         . . . . . . .
   . USING RK = PILE1, RL = LOC ;
         DEBUT : ARM (06) ;
         QUIT ;
         GOTO DÉBUT ;
   END.
```

On peut écrire... IS LOC car lors de l'activation de la tâche sa PST est chargée dans les registres, donc L se trouve correctement chargée. (Commentaire sur TACHE2)

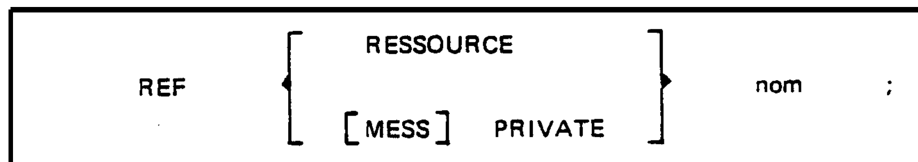
Tâche "sofware" 6

« la main procédure TACBOS
« est lancée sous le niveau
« du superviseur BOS

On peut aussi vouloir écrire une segment procédure par tâche. Dans ce cas, il est nécessaire de pouvoir accéder à des sémaphores à partir de divers segments procédure. Aussi le langage permet-il de définir en externe des RESSOURCES et des PRIVATE.

8.2.3 - DÉCLARATION DE RESSOURCES EXTERNES

1) par REF

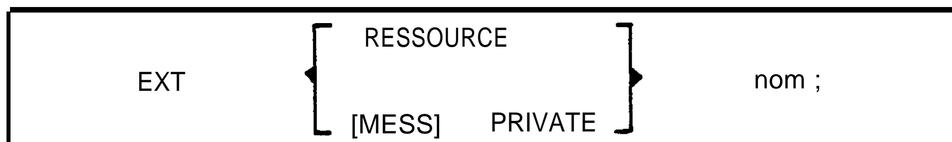


La ressource est alors définie ailleurs dans une déclaration par DEF. L'option MESS permet au compilateur de vérifier la bonne utilisation du sémaphore privé ; en effet dans ce cas il faudra spécifier un numéro de paramètre.

Exemple :

```
SEGMENT PROCEDURE P0
   . LOCAL SECTION LOC
         REF RESSOURCE IMPRIM ;
         REF PRIVATE SOFT2 ;
         REF MESS PRIVATE SOFT3 ;
   . . . . . . . .
   . USING LOCAL IS LOC ;
   . . . . . . . .
      REQUEST (IMPRIMA) ;
      ACTIVATE (SOFT2) ;
   . . . . . . . .
      ACTIVATE (SOFT3, 3) ;
END.
```

2) par EXT



Le nom d'une ressource déclarée par EXT apparaîtra précédé du symbole adresse @.

Exemple :

```
.....
EXT MESS PRIVATE SOFT3 ;
POINTER WORD PTSEMAPHORE = (@ SOFT3) ;
INSTRUCTION ACT (4, '1C00) ;
.....
ACT (&PTSEMAPHORE) ;
.....
```

3) Exemple :

Reprenons l'exemple du paragraphe 9.2.2 ; il peut :

MAIN PROCÉDURE TASKA

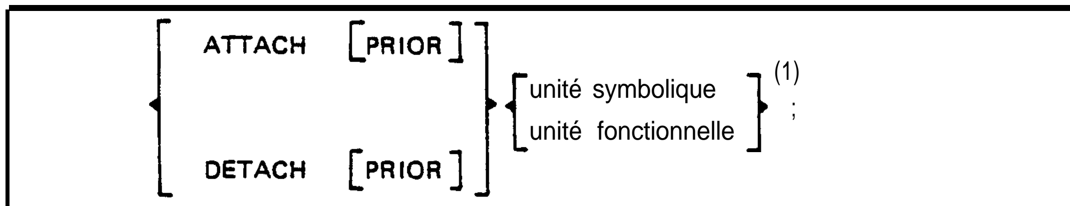
```
. DEF KSTORE SECTION PILE
     RES 20 ;
. LOCAL SECTION LOC
  DEF RESSOURCE IMPRIM
            = ENTRY 1,....      ;
  DEF PRIVATE SOFT2 =....       ;

  PROCÉDURE TACHE2

    « Tâche soft 2

  END ;
  SOFT TASK 2 (RL = LOC, RK   PILE
            START= TACHE2,
            MASTER) ;
. USING RK= PILE, RL = LOC ;
  ......
  ARM (10) ;
END.
```

SEGMENT PROCÉDURE TASKB

```
. REF KSTORE SECTION PILE
     RES 20 ;
. LOCAL SECTION LOC
  REF RESSOURCE IMPRIM
  REF PRIVATE SOFT2 ;

  PROCÉDURE TACHE10

    « Tâche soft 10

  END ;
  SOFT TASK 10 (RL = LOC, RK = PILE
            START = TACHE 10,
            MASTER) ;
  .....
. USING LOCAL IS LOC ;
  .....
  .....
END ;
```

8.2.4 - OPERATIONS D'ENTRÉES-SORTIES

Dans le cas où le système d'exploitation possède l' IOCS multitâches les instructions suivantes sont permises :



Elles font partie des fonctions spéciales réalisées par IOCS. Ce sont respectivement les fonctions d'attachement et de détachement (avec priorité dans le cas de l'option PRIOR) de l'unité physique pointée par l'unité symbolique ou fonctionnelle indiquée à la tâche qui exécute cette instruction (cf. notice IOCS).

Le compilateur génère alors l'IOCB d'un mot, nécessaire à cette fonction, dans la section locale accessible à cet instant.

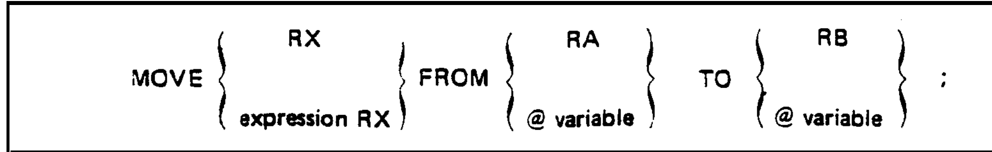
Exemples :

Tâche T1	Tâche T2
------	------
ATTACH L0 ;	ATTACH LL ;
------	------
Echanges sur L0	Echanges sur LL
------	------
DETACH L0 ;	DETACH LL ;
------	------

Si les unités symboliques L0 et LL sont associées à la même unité physique (imprimante par exemple), la tâche T2 pourra être suspendue tant que T1 n'aura pas détaché L0 ou inversement.

(1) nom ou numéro

8.3 - INSTRUCTION DE TRANSFERT MEMOIRE



Cette instruction permet le déplacement d'une zone mémoire (cf. instruction machine MOVE) :

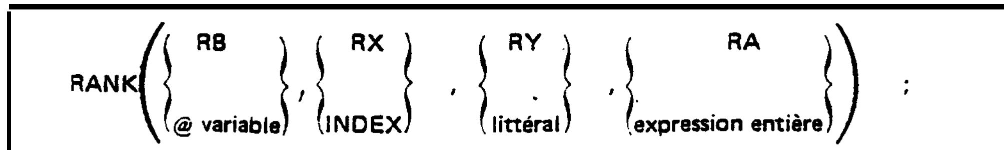
- RX où expression de RX définit le nombre de mots à déplacer
- RA où variable définit l'adresse de début de la zone origine
- RB où variable définit l'adresse de début de la zone destination.

Exemple :

```
--------
WORD ORIGINE, DESTINATION ;
WORD NB ;
ARRAY 10 WORD TAB1 ;
ARRAY 20 WORD TAB2 ;

NB : = 50 ;
RA : = ORIGINE ; RB : = DESTINATION ;
MOVE NB FROM RA TO RB ;
--------
MOVE (RX : = 10) FROM @ TAB1 (0) TO @ TAB2 (10) ;
--------
```

8.4 - RECHERCHE D'UN OCTET DANS UNE ZONE

L'instruction PL16 " RANK " permet de localiser dans une zone un octet donné en paramètre. Les paramètres de recherche sont donnés dans les registres.



RB : adresse de la zone concernée
RX : index portant sur l'adresse donnée dans RB
RY : nombre d'octets de la zone
RA : octet recherché (cadré à droite de RA)

Apres exécution de cette instruction :

- le rang de l'octet trouvé est dans le registre RX et l'indicateur carry vaut 0,
- si l'octet n'a pas été trouvé, la valeur de RX est égale au nombre d'octets de la zone et l'indicateur carry vaut 1.

Exemple :

```
ARRAY 20 BYTES ORIGINE ;
WORD I ;
..............
..............
RB := @ ORIGINE (RX : = @) ;
RANK (RB, 0, (RY : = 10) , (RA : = '7F)) ;
IF (CARRY) THEN ----
RANK (@ ORIGINE (1), 10, 4, ORIGINE (3))
```

8.5 - INSTRUCTIONS SUR PILE UTILISATEUR

L'utilisateur qui possède l'option "scheduler" peut, en plus de la pile générale pointée par le registre K, travailler sur des piles qui lui sont propres. Le langage PL16 lui donne la possibilité de déclarer de telles piles et le moyen de les utiliser par des instructions spéciales.

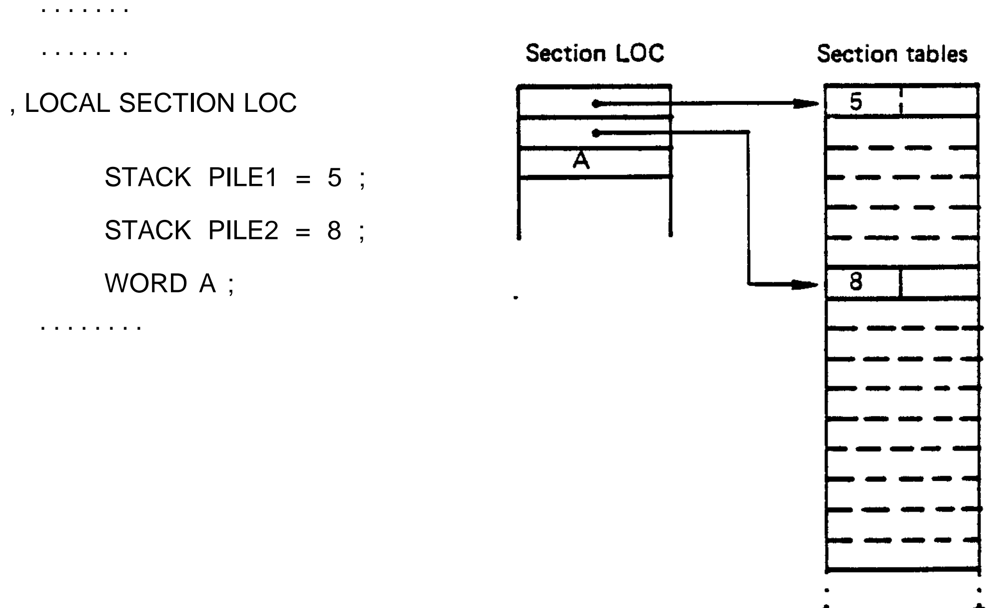
8.5.1 - DECLARATION DE PILE

Forme générale :

```
STACK    nom    =    longueur    ;
```

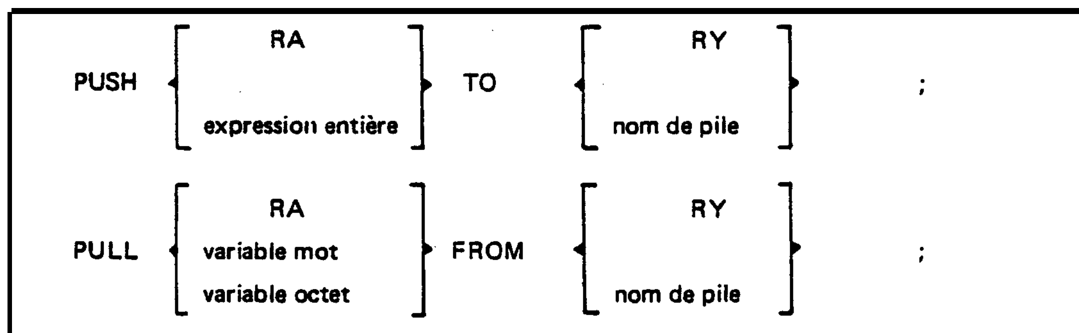
La longueur donnée ici est le nombre de mots N maximum que pourra contenir la pile. Cette déclaration réserve donc dans la section des tables les N mots de la pile et initialise l'octet gauche du premier mot de la pile avec cette longueur. L'octet droit de ce mot contiendra au cours de l'exécution du programme l'état de la pile c'est-à-dire le nombre de mots occupés. Elle réserve de plus dans la section ouverte à cet instant un pointeur vers la pile.

Exemple :



8.5.2 - INSTRUCTIONS SUR PILE

Les instructions PL16 qui agissent sur les piles déclarées sont :



Elles permettent respectivement :

- l'empilement du contenu de RA ou du résultat de l'expression entière dans la pile nommée ou dont l'adresse est donnée par RY.

Le CARRY est positionné :

à 0 si la pile n'est pas saturée

à 1 si la pile est saturée après exécution du PUSH

- le dépilement de la pile nommée ou dont l'adresse est donnée par RY ; la valeur ainsi recueillie est chargée soit dans RA soit dans une variable.

Exemples :

Si l'on reprend les noms des piles précédemment déclarées on peut écrire :

```
  ......
PUSH RA TO PILE1 ;            « EMPILE RA
PUSH TOTO + 2 TO PILE2 ;      « EMPILE TOTO

TOTO : = ...... ;
  ......

PULL RA FROM PILE1 ;          « RESTAURE RA
PULL TOTO FROM PILE2 ;        « RESTAURE TOTO
PUSH A + B + C TO PILE1 ;
  ......
```

9 - LE COMPILATEUR PL16

9.1 - DIRECTIVES AU COMPILATEUR

A l'intérieur d'un programme PL16 on peut trouver des ordres qui ne sont ni des déclarations ni des instructions PL16, ce sont des directives données au compilateur.

Une directive apparaît dans un enregistrement précédée du symbole (!) elle est cadrée en colonne 2.

Les directives reconnues par le compilateur PL16 sont :

```
! ON  _ _  (CAR)
! OFF   _  (CAR)
```

Le premier caractère d'un enregistrement n'est jamais analysé comme faisant partie d'une phrase PL16
La présence d'un caractère en début d'enregistrement entraîne la compilation de cet enregistrement suivant qu'il a été défini "ON" ou "OFF" dans une directive. Par défaut seuls les caractères espace et (!) sont "ON", tous les autres caractères reconnus dans le langage étant "OFF".

Il s'en suit que : tout enregistrement d'un programme ne comportant aucune directive !ON devra commencer par un espace ou !.

Ces deux directives permettent ainsi la compilation conditionnelle de certaines parties d'un programme PL16.

Exemple :

Nous présentons ici quelques cartes supportant du code source PL16. :

```
1 2 3
 !ON   T
 SEGMENT PROCEDURE P0

        . LOCAL SECTION LOC
T          WORD A ;                    « Ces deux enregistrements
T          BYTE B ;                    « seront compilés

C          WORD C ;                    « Ces deux enregistrements
C          BYTE D ;                    « ne le seront pas

! ! OFF  T                             « DIRECTIVE
T          WORD E ;                    « Ces 3 enregistrements
C          WORD D ;                    « ne seront pas compilés
WORD G ;
```

Ne pas confondre le caractère ! qui fait partie de toute directive avec le caractère ! pouvant apparaître en colonne 1.

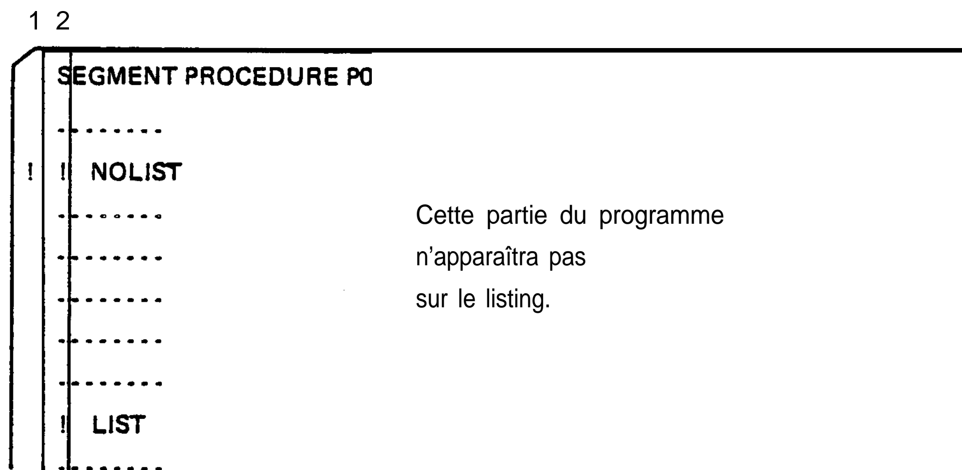
! LIST
! NOLIST

Ces deux directives permettent respectivement d'obtenir ou non un listing du programme au moment de sa compilation.

On peut ainsi, à tout moment, demander ou arrêter la sortie du listing. Nous présentons la forme du listing dans le paragraphe 10.2.4

L'option prise par défaut est ! LIST.

Exemple :



! LTTY
! LIMP

Dans le cas où l'on travaille en LIST, ces deux directives permettent respectivement d'obtenir

. une liste courte (le listing pourra alors sortir sur un téléimprimeur)

. une liste longue (qui permet une meilleure utilisation d'une imprimante rapide).

Par défaut l'option prise par le compilateur est LIMP

! TSYM
! NOTSYM

Ces deux directives permettent de contrôler l'impression par l'éditeur de liens d'une table des symboles du programme

Par défaut l'option prise est !NOTSYM.

! TRACE n ! NOTRACE	Ces directives permettent respectivement la génération ou non d'un certain nombre de visualisation en cours d'exécution d'un programme PL16. (entrée procédure, sortie procédure, passage sur label). Ces directives sont détaillées dans le manuel de référence de DRIP16. Par défaut l'option prise par le compilateur est NOTRACE.
! DEGRE n	Cette directive définit le degré n des visualisations réalisées en cours d'exécution d'un programme PL16. Cette directive est détaillée dans le manuel de référence de DRIP16.
! PAGE	Cette directive permet la réalisation d'un saut de page sur le listing. Par défaut le compilateur réalise un saut de page toutes les 54 lignes.
! EØT	Cette directive est une demande d'arrêt de la compilation avec retour au superviseur - la compilation peut être relancée. après changement éventuel de support source.
! OBJ ! NOBJ	Ces directives permettent l'arrêt et la reprise de la génération du programme objet L'option par défaut est ! GEN.
! GLRM ! NOGLRM	Ces directives qui n'existent que pour la version PL 1 164 001 01/ permettent soit la génération de l'instruction LRM sur .USING =, soit la génération telle qu'elle est présentée en ANNEXE II (chargement des bases).

Remarque :

Toutes ces directives ne sont valides que pour l'unité de compilation en cours.

9.2 - LE COMPILATEUR PL16

9.2.1 - PRESENTATION

Le compilateur du langage PL16 existe en deux versions :

PL16-R	compilateur résident mémoire fonctionnant sous tous les superviseurs SOLAR 16. La taille mémoire laissée au compilateur doit être supérieure à 10K.
PL16-S	compilateur segmenté gestion de la table des symboles sur disque (superviseur disque). La taille mémoire laissée au compilateur doit être au moins de 6K.

Caractéristiques d'une compilation :

Pour chaque unité de compilation supportée par un fichier source, le compilateur PL16 fournit :

- éventuellement un fichier listing (si option LIST)
- un fichier objet composé d'un binaire link éditable qui constitue un module pour l'éditeur de liens EDILE. Il contient :
  . le code objet de l'unité de compilation pouvant éventuellement comporter sa table des symboles (option TSYM)
  . la table d'implantation des sections de l'unité de compilation.

Une unité de compilation PL16 peut comporter au maximum 14 sections de données hormis les sections DUMMY qui sont des sections virtuelles. Le compilateur numérote les sections dans l'ordre où elles sont ouvertes :

- de 2 à 'F pour les sections de données réelles
- à partir de '10 pour les sections DUMMY.

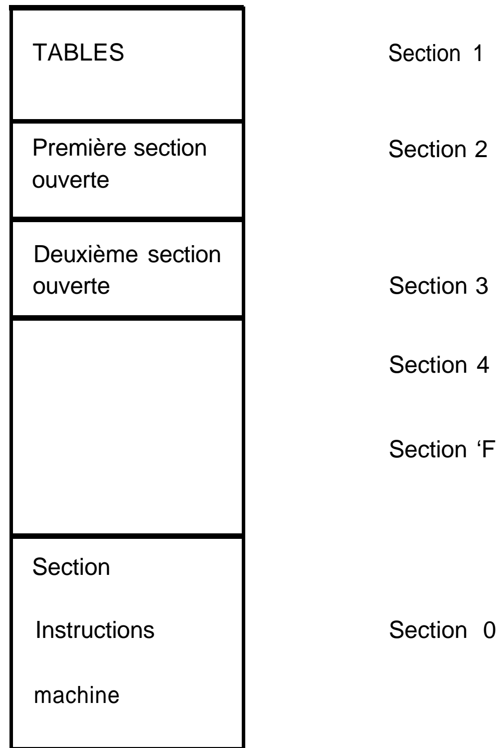
Les sections 0 et 1 sont réservées respectivement aux sections instruction et tables.

En fin de compilation d'une unité, le compilateur connaît la longueur exacte de :

- la section de code (instructions machine qui traduisent les instructions PL16)
- la section des tables (où sont rangés les tableaux)
- des différentes sections de données du programme.

La table que le compilateur est alors capable de fournir, communique l'implantation relative suivante pour les sections 0 à 'F :

Adresse 0



Section 1

Section 2

Section 3

Section 4

Section 'F

Section 0

Remarque :

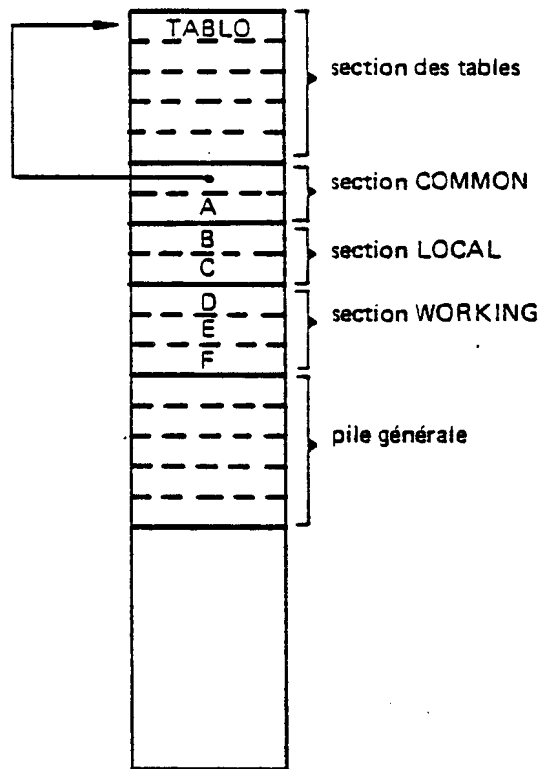
Une KSTORE section est numérotée comme une section de données ordinaire (entre 2 et 'F)

Exemple :

Les différentes sections de l'unité de compilation suivante :

```
MAIN PROCEDURE P0
    . COMMON SECTION C0
        ARRAY 5 WORD TABLO ;
        WORD A ;
    . LOCAL SECTION L0
        WORD B ;
        WORD C ;
    . WORKING SECTION W0
        WORD D, E, F ;
    . KSTORE SECTION PILE
        RES 5 ;
    . USING RC = C0, RL   L0, RW = W0, RK = PILE ;
END.
```

seront implantées de la manière suivante :



Pour l'ensemble des modules objet issus de la compilation de toutes les unités d'un programme, l'éditeur de liens fournit :

- un module de binaire translatable
- un listing où sont imprimés les symboles des unités compilées avec l'option TSYM.

Le module issu de l'édition de lien peut alors être chargé en mémoire à l'adresse désirée par l'utilisateur.

9.2.2 - LES SUPPORTS D'ENTREES-SORTIES

Les unités symboliques utilisées par le compilateur sont :

- l'unité (SI) pour le code source
- l'unité (BO) pour le code objet issu de la compilation
- l'unité (LO) pour le listing et les messages d'erreur

Ces unités symboliques peuvent être affectées à différents supports physiques, par commandes au superviseur.

9.2.3 - PRODUCTION DE PROGRAMMES EXECUTABLES

a) **Chargement du compilateur PL16**

Sous les superviseurs sans disque, le compilateur est chargé, comme tout programme utilisateur, à partir d'un ruban perforé ou de la bande magnétique.

Sous BOS-D le chargement est automatique, depuis le disque, par la commande CALL PL après qu'une image mémoire (PL-:S) ait été crée par le "builder" par la commande SLOD,,15 (version R) ou SLOD, 6, 10 (version S). Si nécessaire (nombre flottant dans le texte source, et absence opérateur câblé) l'arithmétique flottante programmée doit également être chargée en mémoire.

b) Activation du compilateur

Les commandes reconnues par le compilateur sont :

IPLC (Initialise PL Compiler) qui permet de lancer le compilateur en le réinitialisant

CPLC (Continue PL Compiler) permet de continuer une compilation à la suite d'un arrêt momentanée (appel superviseur, défaut périphérique, directive !EOT)

Exemple :

```
*  SICR          (entrée lecteur de cartes)
*  LOLP          (liste imprimante rapide)
*  BOHP          (objet perforateur de ruban)
*  IPLC          (lance compilateur)
                 ←— appel pupitre
*  BOZE          (retour superviseur, annule l'affectation sortie)
*  CPLC          (suite compilation)
*                (fin compilation, retour superviseur).
```

c) Compilation

Une unité de compilation est une procédure compilable séparemment le résultant de sa compilation est un module objet, "link-éditable" séparemment.

Une compilation est lancée par la commande IPLC, elle transforme un texte source en un texte objet.

Le texte source d'une compilation est constitué d'une ou de plusieurs unités de compilation, la dernière (ou la seule) unité étant terminée par END. (au lieu de END ;).

Le texte objet d'une compilation est constitué de un ou plusieurs modules objet.

Exemple :

Texte source	Commande compilation
MAIN PROCEDURE U1	* SICR
|	* LOLP
END ;	* BOHP
	* IPLC
	*
SEGMENT PROCEDURE U2	
|	
END ;	
SEGMENT PROCEDURE U3	
|	
END.	

Le compte rendu de fin de compilation, qui peut être testé par commande IF - est le nombre total d'erreurs détectées.

9.2.4 • EDITION DE LIEN

Elle consiste, à partir d'un texte objet lu sur BI à produire un binaire translatable sur BO.

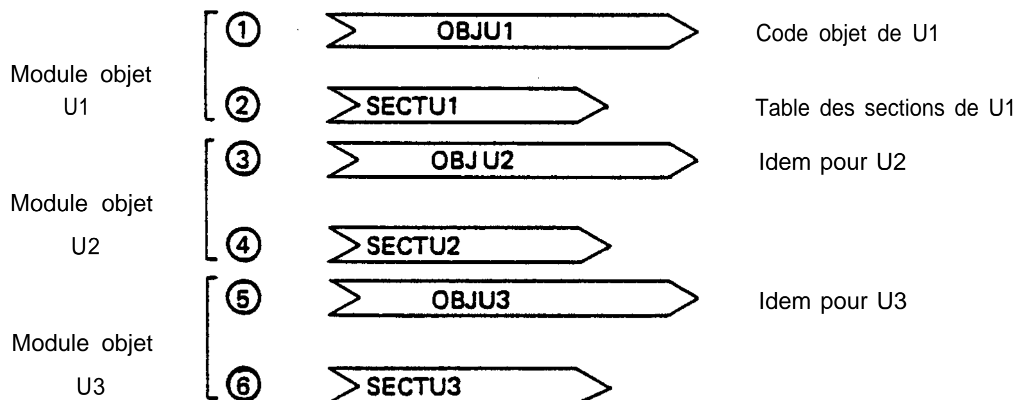
La structure du texte objet est légèrement différente, suivant que l'unité de sortie (BO) de la compilation était HP (perforateur de ruban), Ti (bande magnétique), ou un fichier temporaire disque (BO *).

a) sortie sur ruban perforé (BO HP)

Chaque module objet produit est composé de deux parties séparées par une "avance bande".

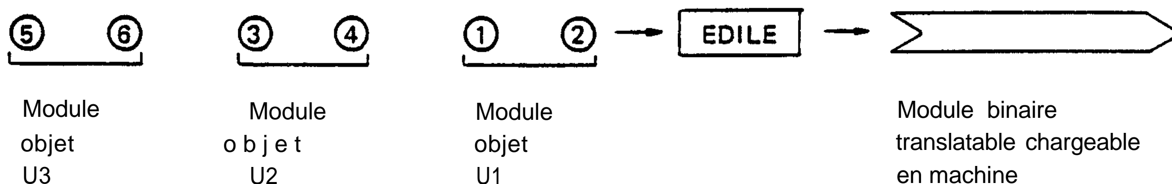
Partie 1 programmes
Partie 2 table d'implantation de section de données

Ces deux parties étant à lire dans l'ordre inverse par l'éditeur de lien.



Les commandes permettant l'édition de lien, sont alors :

* I L N K	Lecture SECTU1 par EDILE
* C L N K	Lecture OBJU1
* C L N K	Lecture SECTU2
* C L N K	Lecture OBJU2
* C L N K	Lecture SECTU3
* C L N K	Lecture OBJU3
* E L N K	Fin d'édition de liens
*	



b) unité de sortie affectée au dérouleur de bande (BO Ti)

Chaque module objet, ainsi que chaque partie de module est suivi d'une marque de bande, l'organisation générale étant la même que sur ruban perforé. Il y a donc lieu, avant toute commande à l'éditeur de lieu, de se positionner convenablement sur la bande.

OBJU1	•	SECTU1	•	OBJU2	•	SECTU2	•	— — — — — —

Après compilation en BO T1 par exemple, on écrira :

```
* BO HP
* TPIO REWI T1          rebobinage bande
* TPIO SKIP 1           positionnement sur tables 1er module
* ILNK
* TPIO REWI T1
* CLNK
* TPIO SKIP 3           positionnement sur tables 2ème module
* CLNK
   |
   |
* ELNK
```

c) unité symbolique de sortie affectée au disque (BO * ou BO NOM)

Cette possibilité n'existe évidemment que sous le système BOS-D.
Il y a lieu, dans ce cas, de ne compiler qu'une seule unité à la fois. La compilation produit directement un programme objet prêt à être traité par l'éditeur de liens EDILE en une seule commande. En reprenant l'exemple précédent, on devrait donc terminer chaque unité de compilation par END, et créer par compilations successives, trois fichiers objet indépendants à "Link-éditer", soit :

```
* CALL PL
* BO MODUL 1 sortie fichier objet MODUL1
* SI SOUR1
* IPLC
* SI SOUR2
* BO MODUL2 sortie fichier objet MODUL2
* IPLC

* CALL EDILE
* BO HP
* BI MODUL1
* ILNK                  édition de liens de chaque programme objet par
* BI MODUL2             une seule commande
* CLNK
   |
   |
* ELNK
```

Le chargement du module translatable obtenu par éditeur de liens, permet d'implanter le programme en mémoire à l'adresse désirée. L'adresse de lancement automatique du programme est le point d'entrée de la dernière MAIN PROCÉDURE donnée à l'éditeur de liens.

9.2.5 • LE LISTING

Le listing du code source obtenu en cas d'option LIST a la forme suivante :

LIGNE	RC	RL	RW	DONNEES	TABL	PROG	PF	BN	:x SOURCE
NNNN	CC	LL	WW	DD AAAA	TTTT	PPPP	FF	BB	SOURCE

où :

NNNN représente le numéro (en décimal) de l'enregistrement liste

C C représente le numéro (en hexadécimal) de la section COMMON courante

LL représente le numéro (en hexadécimal) de la section LOCAL courante

W W représente le numéro (en hexadécimal) de la section WORKING courante

DD représente le numéro (en hexadécimal) de la section dans laquelle les données sont rangées à un instant donné : la dernière ouverte ou, lors des instructions, la section locale accessible.

AAAA représente le déplacement courant (en hexadécimal) atteint dans cette dernière section. On connaîtra ainsi pour chaque élément déclaré son emplacement dans la section.

TTTT représente le déplacement courant (en hexadécimal) atteint dans la section des tables.

PPPP représente le déplacement courant (en hexadécimal) atteint dans la section instruction. On peut ainsi connaître le nombre d'instructions machine générées pour une ligne source PL1600.

Les sections COMMON, LOCAL et WORKING ainsi représentées sont :

- lors des déclarations, les sections ouvertes. On peut ainsi à un instant donné, avoir 3 sections de type différents ouvertes : une section COMMON, une local, une WORKING courante
- lors des instructions, les sections accessibles par les registres de base. Trois sections sont respectivement accessibles par les bases C, L et W, à un instant donné.

Lorsqu'une section d'un type donné est fermée ou inaccessible son numéro est égal à '1F.

FF représente le niveau d'imbrication du bloc

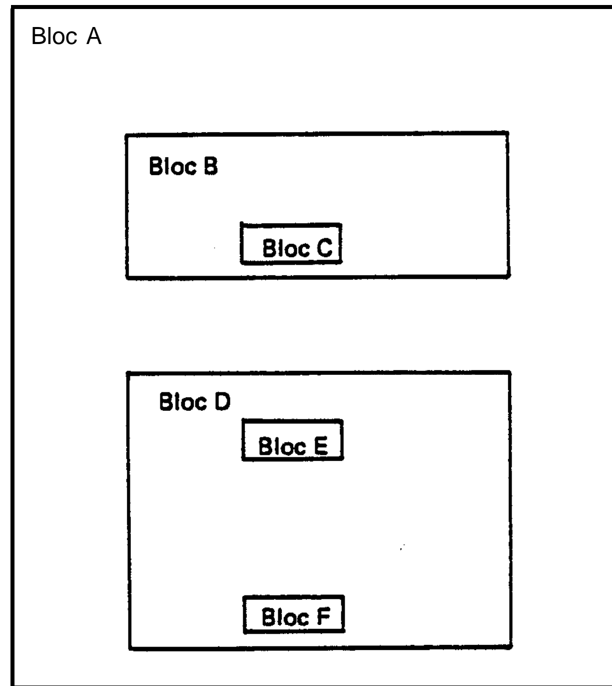
BB représente le numéro du bloc courant ; les blocs sont numérotés dans l'ordre où ils apparaissent ; seuls ceux qui contiennent des déclarations ou des étiquettes sont numérotés.

Exemple :

```
LIGNE RC  RL  RW  DONNEES    TABL  PROG  PF  BN :*SOURCE
0001  1F  1F  [illegible]  1F  0000  0000  0000  00  00 : SEGMENT PROCEDURE PO
0002  1F  1F  1F  1F  0000  0000  0000  00  00 : .LOCAL SECTION LOC
0003  1F  02  1F  02  0000  0000  0000  01  01 : WORD A;
0004  1F  02  1F  02  0001  0000  0000  01  01 : BYTE B;
```

L'information "profondeur de bloc" est très utile pour aider à la lecture du programme source, en même temps elle permet une correction rapide des erreurs signalées par le compilateur et qui peuvent êtres dues à des fautes d'imbrication (END ignoré, ou en trop...) l'information "numéro de bloc" est à mettre en concordance avec le listing des symboles, donné par l'éditeur de lien, lorsqu'on a compilé avec l'option !TSYM (voir chapitre 10).

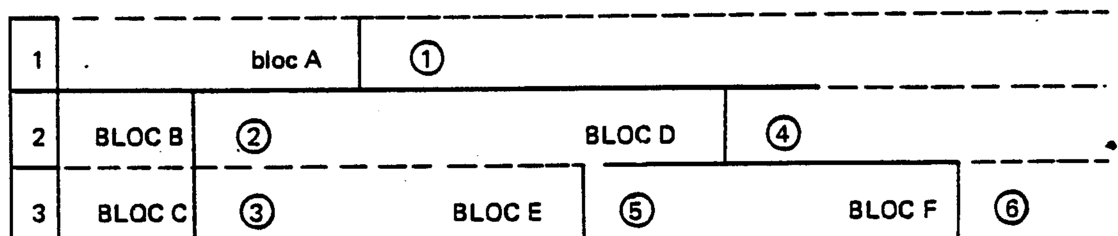
Nous donnons ci-dessous un exemple différenciant ces deux notions :



Les blocs B et D sont au niveau d'imbrication 2, les blocs C, E et F sont au niveau 3 :

Si tous les blocs contiennent des déclarations, les blocs A, B, C, D, E, F seront numérotés respectivement 1, 2, 3, 4, 5, 6.

On peut représenter cet exemple par le schéma suivant :



1, 2, 3 sont les niveaux d'imbrication

ⓘ sont les numéros de blocs.

```
LIGNE RC  RL  RW  DONNEES    TABL  PROG  PF  BN :*SOURCE
0001  1F  1F  1F  1F  0000  0300  0000  00  00 : MAIN PROCEDURE LISTE          <<EXEMPLE DE LISTE
0002  1F  1F  1F  1F  0000  0000  0000  00  00 :  .COMMON SECTION COM                    <<OUVERTURE COMMON
0003  02  1F  1F  02  0000  0000  0000  01  01 :          WORD A;
0004  02  1F  1F  02  0001  0000  0000  01  01 :          WORD B;
0005  02  1F  1F  02  0002  0000  0000  01  01 :  .LOCAL SECTION LOC                     <<OUVERTURE LOCAL
0006  02  03  1F  03  0000  0000  0000  01  01 :          WORD C;
0007  02  03  1F  03  0001  0000  0000  01  01 :          WORD D;
0008  02  03  1F  03  0002  0000  0000  01  01 :          PROCEDURE PO
0009  02  03  1F  03  0003  0000  0000  01  01 :              .LOCAL SECTION LO          <<OUVERTURE AUTRE LOCAL
0010  02  04  1F  04  0000  0000  0000  02  02 :               WORD E;                   <<ABANDON LOCAL LOC
0011  02  04  1F  04  0001  0000  0000  02  02 :               WORD F;
0012  02  04  1F  04  0002  0000  0000  02  02 :              .USING LOCAL=LO;            <<ACCES AU LOCAL LO
0013  1F  04  1F  04  0002  0000  0007  02  02 :               E:=F;
0014  1F  04  1F  04  0002  0000  0009  02  02 :               E:=E*F;
0015  1F  04  1F  04  0002  0000  0000  02  02 :          END;                           <<FERMETURE LO
0016  02  03  1F  03  0003  0000  000F  01  01 :   •      WORD E;                        <<REPRISE LOC
0017  02  03  1F  03  0004  0000  000F  01  01 :          WORD F;
0018  02  03  1F  03  0005  0000  000F  01  01 :  .USING RC=COM,RL=LOC;                  <<ACCES AU COMMON ET LOCAL LOC
0019  02  03  1F  03  0005  0000  0014  01  01 :          A:=A+B;
0020  02  03  1F  03  0005  0000  0017  01  01 :          A:=A*2;
0021  02  03  1F  03  0006  0000  001B  01  01 :          C:=C+D+A;
0022  02  03  1F  03  0006  0000  001F  01  01 :          E:=E+F+B*A;
0023  02  03  1F  03  0006  0000  0025  01  01 :          CALL PO;
0024  02  03  1F  03  0006  0000  0026  01  01 :          BEGIN
0025  02  03  1F  03  0006  0000  0026  01  01 :            .WORKING SECTION WORK        <<OUVERTURE WORKING
0026  02  03  05  05  0000  0000  0026  02  03 :              WORD W1;
0027  02  03  05  05  0001  0000  0026  02  03 :              WORD W2;
0028  02  03  05  05  0002  0000  0026  02  03 :            .USING WORKING=WORK;         <<ACCES WORKING
0029  02  03  05  03  0007  0000  0028  02  03 :              W1:=W2+C;                  « LA SECTION LOCALE ACCESSIBLE EST LOC
0030  02  C3  0   03  0007  0000  002E  02  03 :              W2:=A+B+D;
0031  02  03  05  03  0007  0000  0032  02  03 :          END;                           <<WORKING PLUS ACCESSIBLE
0032  02  03  1F  03  0007  0000  0033  01  01 :          A:=A-B;
0033  02  03  1F  03  0007  0000  0036  01  01 :          B:=B/2;
0034  02  03  1F  03  0007  0300  0039  01  01 :          BEGIN
0035  02  03  1F  03  0007  0000  0039  01  01 :           .WORKING SECTION WORK         <<OUVERTURE AUTRE WORKING
0036  02  03  06  06  0000  0000  0039  02  04 :            WORD A;
0037  02  03  06  06  0001  0000  0039  02  04 :            WORD B;
0039  02  03  06  06  0002  0300  0039  02  04 :           .USING RW=WORK;               <<ACCES WORKING
0039  02  03  06  03  0008  0000  003E  02  04 :             A:=B;
0040  02  03  06  03  0008  0000  0040  02  04 :          END;                           <<WORKING PLUS ACCESSIBLE
0041  02  03  1F  03  0009  0000  0041  01  01 :          A:=A+B*2;
0042  02  03  1F  03  0008  0000  0046  01  01 : END.                                    <<FERMETURE COMMON ET LOCAL LOC
FIN DE COMPILATION  0000 ERREUR(S) *0058 MOTS
```

```
LIGNE RC  RL  RW  DONNEES   TABL  PROG  PF  BN :*SOURCE
0001  1F  1F  1F  1F  0000  0000  0000  00  00 : MAIN PROCEDURE CALCUL
0002  1F  1F  1F  1F  0000  0000  0000  00  00 :      << LE TRAITEMENT GLOBAL EFFECTUE PAR CETTE PROCEDURE
0003  1F  1F  1F  1F  0000  0000  0000  00  00 :      << EST LA REPETITION DES TRAITEMENTS PARTIELS SUIVANTS:
0004  1F  1F  1F  1F  0000  0000  0000  00  00 :      <<      LECTURE DE 2 NOMBRES COMPRIS ENTRE 0 ET 9
0005  1F  1F  1F  1F  0000  0000  0000  00  00 :      <<      CALCUL DE LEUR SOMME
0006  1F  1F  1F  1F  0000  0000  0000  00  00 :      <<      ECRITURE DU RESULTAT S'IL DIFFERE DE 0
0007  1F  1F  1F  1F  0000  0000  0000  00  00 :      << LE TRAITEMENT S'ARRETE SI LA SOMME EST NULLE
0008  1F  1F  1F  1F  0000  0000  0000  00  00 :      <<  OU SI LE CALCUL A ETE FAIT 10 FOIS
0009  1F  1F  1F  1F  0000  0000  0000  00  00 :   .KSTORE SECTION  PILE
0010  1F  1F  1F  02  0000  0000  0000  01  01 :    RES 10;
0011  1F  1F  1F  02  000A  0000  0000  01  01 :   .LOCAL SECTION LOC
0012  1F  03  1F  03  0000  0000  0000  01  01 :      WORD A,B ;                        << VALEURS A LIRE
0013  1F  03  1F  03  0002  0000  0000  01  01 :      WORD S ;                          << RESULTAT
0014  1F  03  1F  03  0003  0000  0000  01  01 :      WORD I ;                          << MEMOIRE DE TRAVAIL
0015  1F  03  1F  03  0004  0000  0000  01  01 :      PROCEDURE LIRE(X,Y)
0016  1F  03  1F  03  0005  0000  0000  01  01 :         .LOCAL SECTION CONTINUE LOC
0017  1F  03  1F  03  0005  0000  0000  02  02 :           POINTER WORD X,Y;            << PARAMETRES
0018  1F  03  1F  03  0007  0000  0000  02  02 :           ARRAY 2 BYTE CHIFFRES ;      << CARACTERES ASCI
0019  1F  03  1F  03  0008  0001  0000  02  02 :           LPFILE NOMBRES=(MODE :INPUT,EM;EU:SI;
0020  1F  03  1F  03  0009  0001  0000  02  02 :           DATA:CHIFFRES; EOE: 2 ; CONTROL) ;
0021  1F  03  1F  03  000E  0001  0000  02  02 :         .USING LOCAL IS LOC ;
0022  1F  03  1F  03  000E  0001  0000  02  02 :           READ V3MBRES ;
0023  1F  03  1F  03  000E  0001  0008  02  02 :           @X:=CHIFFRES(0) AND 'F;     <<MOTS POINTES PAR X,Y=VALEURS
0024  1F  03  1F  03  000E  0001  000F  02  02 :           @Y:=CHIFFRES(1) AND 'F ;    <<DONNEES PAR LES CARACT. ASCII
0025  1F  03  1F  03  000E  0001  0013  02  02 :      END ;
0026  1F  03  1F  03  000E  0001  0014  01  01 :      PROCEDURE ECRIRE(RESUL)
0027  1F  03  1F  03  000F  0001  0014  01  01 :         .LOCAL SECTION CONTINUE LOC
0028  1F  03  1F  03  000F  0001  0014  02  03 :           WORD RESUL ;                 << PARAMETRE
0029  1F  03  1F  03  0010  0001  0014  02  03 :           WORD DIX=(10) ;              << POUR DIVISION PAR 10
0030  1F  03  1F  03  0011  0001  0014  02  03 :           ARRAY 8 BYTE SORTIE=('8D,'DA,"S=  ",'8D,'0A) ;
0031  1F  03  1F  03  0012  0005  0014  02  03 :           POINTER WORD PTSUM=(@SORTIE+2 AND '7FFF);
0032  1F  03  1F  03  0013  0005  0014  02  03 :           LPFILE RESULTAT=(MODE:OUTPUT,EM;EU:SO;
0033  1F  03  1F  03  0014  0005  0014  02  03 :           DATA:SORTIE; EOE: 8 ; CONTROL) ;
0034  1F  03  1F  03  0019  0005  0014  02  03 :         .USING LOCAL IS LOC ;
0035  1F  03  1F  03  0019  0005  0014  02  03 :           RA:=0 ; RB:=RESUL ; RAB:=RAB/DIX ; << TRANSFORMATION
0036  1F  03  1F  03  0019  0005  001E  02  03 :           RA:=RA OR '30 SLLS 8 OR RB OR '30 ; << DU RESULTAT EN
0037  1F  03  1F  03  0019  0005  0022  02  03 :           @PTSUM:=RA ;                 << CARACTERES ASCII
0038  1F  03  1F  03  0019  0005  0023  02  03 :           WRITE RESULTAT ;
0039  1F  03  1F  03  0019  0005  0025  02  03 :      END ;
0040  1F  03  1F  03  0019  0005  0026  01  01 :      PROCEDURE S3MME(X,Y,Z)
0041  1F  03  1F  03  001A  0005  0026  01  01 :         .LOCAL SECTION CONTINUE LOC
0042  1F  03  1F  03  001A  0005  0026  02  04 :           WORD X,Y;POINTER WORD Z;     << PARAMETRES
0043  1F  03  1F  03  001D  0005  0026  02  04 :         .USING LOCAL IS LOC ;
0044  1F  03  1F  03  001D  0005  0026  02  04 :           @Z:=X+Y;                    << CALCUL DE LA
0045  1F  03  1F  03  001D  0005  0034  02  04 :      END ;                             <<SOMME
0046  1F  03  1F  03  001D  0005  0035  01  01 :   .USING LOCAL=LOC,KSTORE=PILE;        << PROGRAMME PRINCIPAL
0047  1F  03  1F  03  001D  0005  0038  01  01 :      DO FOR I:=1 STEP +1 UNTIL 10 ;
0048  1F  03  1F  03  001D  0005  003F  01  01 :         CALL LIRE(@A,@B);
0049  1F  03  1F  03  001F  0005  0044  02  04 :         CALL SO44E(A,B,@S);
0050  1F  03  1F  03  0020  0005  0048  02  04 :         IF(S/=0) THEN CALL ECRIRE(S)
0051  1F  03  1F  03  0020  0005  0050  03  04 :         ELSE STOP END ;
0052  1F  03  1F  03  0020  0005  0052  02  04 :      END ;
0053  1F  03  1F  03  0020  0005  0055  01  01 : END.
FIN DE COMPILATION  0000 ERREUR(S) '0086 MOTS
```

9.2.6 - LES ERREURS

Lorsqu'une erreur est détectée au moment de la compilation d'un enregistrement de code source, un message d'erreur est imprimé sous l'enregistrement listé en cause.

Ce message est composé d'un numéro d'erreur et du caractère x dont la position correspond au symbole syntaxique qui a provoqué l'erreur.

En option LIST, les messages d'erreur apparaissent sur le listing ; en NOLIST, la détection d'une erreur provoque l'impression de l'enregistrement erroné suivie de celle du message d'erreur.

Exemple :

```
LIGNE RC  RL  RW  DONNEES   TABL  PROG  PF  BN :*SOURCE
0001  1F  1F  1F  1F  0000  0000  0000  00  00 : SEGMENT PROCEDURE PO
0002  1F  1F  1F  1F  0000  0000  0000  00  00 : .LOCAL SECTION LOC
0003  1F  02  1F  02  0000  0000  0000  01  01 : WORD A;
0004  1F  02  1F  02  0001  0000  0000  01  01 : BYTE B;
0005  1F  02  1F  02  0002  0000  0000  01  01 : LONG A;
************************ERREUR NO 80                  *
0006  1F  02  1F  02  0002  0000  0000  01  01 : .USING LOCAL=LOC;
0007  1F  02  1F  02  0002  0000  0007  01  01 : A:=B;
0008  1F  02  1F  02  0002  0000  0009  01  01 : END.
FIN DE COMPILATION  0001 ERREUR(S) '000E MOTS
```

La liste complète des messages d'erreur est donnée en annexe page 198.

9.2.7 - CARACTERISTIQUES PROPRES A PL16-S

1) Caractéristiques générales

Ce compilateur fonctionne sous tous les systèmes d'exploitation SOLAR16 supportant un disque. Le langage compilé est strictement le PL16 les messages d'erreur sont identiques à ceux de la version résidente.
Il occupe une partition mémoire de 6 K au minimum. Les performances sont optimales s'il s'exécute dans une partition de 7,5 Kmots.

Son support externe est constitué de deux rubans perforés :
- ruban binaire translatable à charger par le "builder" comme pour le PL16-R (commande SLOD, 6, 10,
- un fichier direct TRANSS-PL à charger par l'utilitaire FUP3 sur la FU disque D2.

2) Caractéristiques de fonctionnement

Le compilateur PL16-S utilise 3 fichiers :
- 2 fichiers temporaires, ouverts sur la FU disque associée au "job" en cours, avec les numéros '70 et '71,
- 1 fichier permanent ouvert sur la FU D2, avec le numéro '72 (TRANSS-PL).
Ces fichiers sont fermés en fin de compilation.

3) Remarques d'utilisation

- Contrairement à la version PL16-R, toute valeur donnant lieu à une génération implicite, est générée à chaque utilisation dans le même bloc.
  Il peut donc être intéressant de déclarer comme "WORD" les valeurs utilisées fréquemment.
  Exemple :

```
au lieu de   CONSTANT   H8000 = '8000 ;
écrire       WORD       H8000 = ('8000) ;
```

- Certaines libertés d'écritures acceptées par la version PL16-R, ne le sont pas pour la version PL16-S. Ce sont celles où un identificateur est utilisé sur une ligne sans avoir été complètement défini par la ligne précédente.
  Exemple :

```
                  CONSTANT A = 10
                           ,B = A + 2 ;
                                 erreur
il faut écrire    CONSTANT A = 10,
                           B  = A + 2 ;
```

10 - AIDE A LA MISE AU POINT DES PROGRAMMES PL16

10.1 - MISE AU POINT SOUS LE CONTROLE DE AID

10.1.1 - EXPLOITATION DU LISTING

Au moment du chargement d'un programme en mémoire les adresses d'implantation des sections sont imprimées sur l'unité symbolique EL. Dans le cas d'un programme PL16, on aura pour chaque unité de compilation :

- l'adresse de la section instruction de cette unité : elle suit le nom de la segment procédure constituant cette unité
- les adresses des différentes sections de données de cette unité : elles suivent les noms de ces sections.

Grâce aux déplacements indiqués sur le listing le programmeur peut connaître :

- **l'adresse d'une donnée,** il lui suffit pour cela d'ajouter à l'adresse de la section où est rangée la donnée, le déplacement de cette donnée dans la section.
- **l'adresse en mémoire de la séquence d'instructions** machine qui correspond à une instruction PL16 : il suffit d'ajouter à l'adresse de la section instruction le déplacement courant dans cette section.

Ces renseignements permettent en mise au point sous AID :

- de visualiser ou charger des variables
- de mettre des points d'arrêt dans le programme dans les mêmes conditions que pour un programme obtenu par assemblage.

Exemple :

Soit à mettre en oeuvre le programme de CALCUL décrit au premier chapitre du manuel mais découpé en deux unités de compilation (la procédure SOMME étant devenue une SEGMENT PROCEDURE).

L'enchaînement des commandes sous le contrôle d'un système sans disque :

* IPLC	Compilation
* ILNK	Edition de liens
* CLNK	- lecture tables des sections de CALCUL
0054 CALCUL	- code objet module CALCUL
ERL 09	
* CLNK	- lecture table des sections de SOMME
* CLNK	- code objet module SOMME
0077 SOMME	
* ELNK	- fin d'édition de liens

'54 est l'adresse de définition de l'externe CALCUL (nom de la MAIN PROCEDURE).
'77 est l'adresse de définition de l'externe SOMME (nom de la SEGMENT PROCEDURE)
ERL 09 : il existe des externes non résolus (REF PROCEDURE SOMME).

```
* IRLD                              Chargement
OK?  Y

ADR? '7E00
MOD? M
CALCUL '7E2C                        - implantation section de code CALCUL
PILE    '7E05                       - implantation pile générale
LOC     '7E0F                       - implantation section LOC (de CALCUL)
SOMME  '7E76                        - implantation section de code SOMME
LOC     '7E73                       - implantation section LOC (de SOMME)
RUN     '7E54                       - adresse de lancement
             12
S =  03                             Exécution
23
S = 05
34
S = 07
45
S = 09
56
S = 11
67
S = 13
78
S = 15
89
S = 17
90
S = 09
00
*                                   retour à BOS
```

```
LIGNE RC  RL  RW  DONNEES  TABL  PROG  PF  BN :*SOURCE
0001  1F  1F  1F  1F  0000  0000  0000  00  00 ::ITSYM
0002  1F  1F  1F  1F  0000  0000  0000  00  00 : MAIN PROCEDURE CALCUL
0003  1F  1F  1F  1F  0000  0000  0000  00  00 :       << LE TRAITEMENT GLOBAL EFFECTUE PAR CETTE PROCEDURE
0004  1F  1F  1F  1F  0000  0000  0000  00  00 :       << EST LA REPETITION DES TRAITEMENTS PARTIELS SUIVANTS:
0005  1F  1F  1F  1F  0000  0000  0000  00  00 :       <<      LECTURE DE 2 NOMBRES COMPRIS ENTRE 0 ET 9
0006  1F  1F  1F  1F  0000  0000  0000  00  00 :       <<      CALCUL DE LEUR SOMME
0007  1F  1F  1F  1F  0000  0000  0000  00  00 :       <<      ECRITURE DU RESULTAT S'IL DIFFERE DE 0
0008  1F  1F  1F  1F  0000  0000  0000  00  00 :       << LE TRAITEMENT S'ARRETE SI LA SOMME EST NULLE
0009  1F  1F  1F  1F  0000  0000  0000  00  00 :       <<  OU SI LE CALCUL A ETE FAIT 10 FOIS
0010  1F  1F  1F  1F  0000  0000  0000  00  00 :    .KSTORE SECTION  PILE
0011  1F  1F  1F  02  0000  0000  0000  01  01 :      RES 10;
0012  1F  1F  1F  02  000A  0000  0000  01  01 :    .LOCAL SECTION LOC
0013  1F  03  1F  03  0000  0000  0000  01  01 :       WORD A,B ;                          << VALEURS A LIRE
0014  1F  03  1F  03  0002  0000  0000  01  01 :       WORD S ;                            << RESULTAT
0015  1F  03  1F  03  0003  0000  0000  01  01 :       WORD I ;                            << MEMOIRE DE TRAVAIL
0016  1F  03  1F  03  0004  0000  0000  01  01 :       PROCEDURE LIRE(X,Y)
0017  1F  03  1F  03  0005  0000  0000  01  01 :          .LOCAL SECTION CONTINUE LOC
0018  1F  03  1F  03  0005  0000  0000  02  02 :            POINTER WORD X,Y;               << PARAMETRES
0019  1F  03  1F  03  0007  0000  0000  02  02 :            ARRAY 2 BYTE CHIFFRES ;         << CARACTERES ASCI
0020  1F  03  1F  03  0008  0001  0000  02  02 :            LPFILE NOMBRES=(MODE :INPUT,EN;EU:SI;
0021  1F  03  1F  03  0009  0001  0000  02  02 :            DATA:CHIFFRES; EOE: 2 ; CONTROL) ;
0022  1F  03  1F  03  000E  0001  0000  02  02 :          .USING LOCAL IS LOC ;
0023  1F  03  1F  03  000E  0001  0000  02  02 :            READ NOMBRES ;
0024  1F  03  1F  03  000E  0001  0008  02  02 :            @X:=CHIFFRES(0) AND 'F;         <<MOTS POINTES PAR X,Y=VALEURS
0025  1F  03  1F  03  000E  0001  000F  02  02 :            @Y:=CHIFFRES(1) AND 'F ;        <<DONNEES PAR LES CARACT. ASCII
0026  1F  03  1F  03  000E  0001  0013  02  02 :       END ;
0027  1F  03  1F  03  000E  0001  0014  01  01 :       PROCEDURE ECRIRE(RESUL)
0028  1F  03  1F  03  000F  0001  0014  01  01 :          .LOCAL SECTION CONTINUE LOC
0029  1F  03  1F  03  000F  0001  0014  02  03 :            WORD RESUL ;                    << PARAMETRE
0030  1F  03  1F  03  0010  0001  0014  02  03 :            WORD DIX=(10) ;                 << POUR DIVISION PAR 10
0031  1F  03  1F  03  0011  0001  0014  02  03 :            ARRAY 8 BYTE SORTIE=('8D,'0A,"S=  ",'8D,'0A) ;
0032  1F  03  1F  03  0012  0005  0014  02  03 :            POINTER WORD PTSUM=(@SORTIE+2 AND '7FFF);
0033  1F  03  1F  03  0013  0005  0014  02  03 :            LPFILE RESULTAT=(MODE:OUTPUT,EN;EU:SO;
0034  1F  03  1F  03  0014  0005  0014  02  03 :            DATA:SORTIE; EOE: 8 ; CONTROL) ;
0035  1F  03  1F  03  0019  0005  0014  02  03 :          .USING LOCAL IS LOC ;
0036  1F  03  1F  03  0019  0005  0014  02  03 :            RA:=0 ; RB:=RESUL ; RAB:=RAB/DIX ; << TRANSFORMATION
0037  1F  03  1F  03  0019  0005  001E  02  03 :            RA:=RA OR '30 SLLS 8 OR RB OR '30 ; << DU RESULTAT EN
0038  1F  03  1F  03  0019  0005  0022  02  03 :            @PTSUM:=RA ;                    << CARACTERES ASCII
0039  1F  03  1F  03  0019  0005  0023  02  03 :            WRITE RESULTAT ;
0040  1F  03  1F  03  0019  0005  0025  02  03 :       END ;
0041  1F  03  1F  03  0019  0005  0026  01  01 :       REF PROCEDURE SOMME;
0042  1F  03  1F  03  001A  0005  0026  01  01 :    .USING LOCAL=LOC,KSTORE=PILE;          << PROGRAMME PRINCIPAL
0043  1F  03  1F  03  001A  0005  002C  01  01 :       DO FOR I:=1 STEP +1 UNTIL 10 ;
0044  1F  03  1F  03  001A  0005  0030  01  01 :          CALL LIRE(@A,@B);
0045  1F  03  1F  03  001C  0005  0035  02  03 :          CALL SOMME(A,B,@S);
0046  1F  03  1F  03  001D  0005  003C  02  03 :          IF(S/=0) THEN CALL ECRIRE(S)
0047  1F  03  1F  03  001D  0005  0041  03  03 :          ELSE STOP END ;
0048  1F  03  1F  03  001D  0005  0043  02  03 :       END ;
0049  1F  03  1F  03  001D  0005  0046  01  01 : END;
FIN DE COMPILATION  0000 ERREUR(S) '0074 MOTS
```

```
LIGNE RC  RL  RW  DONNEES  TABL  PROG  PF  BN :*SOURCE
0001  1F  1F  1F  1F  0000  0000  0000  00  00 ::ITSYM
0002  1F  1F  1F  1F  0000  0000  0000  00  00 : SEGMENT PROCEDURE SOMME(X,Y,Z)
0003  1F  1F  1F  1F  0000  0000  0000  00  00 :         .LOCAL SECTION LOC
0004  1F  02  1F  02  0000  0000  0000  01  01 :           WORD X,Y;POINTER WORD Z;     << PARAMETRES
0005  1F  02  1F  02  0003  0000  0000  01  01 :         .USING LOCAL = LOC ;
0006  1F  02  1F  02  0003  0000  0007  01  01 :           @Z:=X+Y;
0007  1F  02  1F  02  0003  0000  0015  01  01 : END.                                   <<SOMME
FIN DE COMPILATION  0000 ERREUR(S) '0018 MOTS
```

A partir du listing précédent et des adresses données par le chargeur on peut trouver l'adresse de la variable RESUL par exemple :

- le listing indique : déplacement 'F dans la section nº 3 (section LOC)
- le chargeur indique : adresse de LOC : '7E0F

d'où adresse de RESUL : '7E0F + 'F = '7E1E

On pourra, sous AID, visualiser cette variable par la commande : '7E1E/

Soit à retrouver l'adresse de lancement du programme :

celui-ci se lance sur l'instruction . USING LOCAL = LOC....

Soit en '26 de la section de code CALCUL :

'7E2C + '26 = '7E52

Il faut ajouter à cela le nombre de registres initialisés par l'instruction USING (les valeurs des bases sont en effet générés dans la section de code : cf annexe) ; dans notre cas RK et RL sont initialisés, d'où l'adresse de lancement :

'7E52 + '2 = '7E54

Pour faire du pas à pas, le pas étant une instruction PL16, il faut connaître le nombre d'instructions machine générées pour une instruction PL16 ; celui-ci se déduit facilement de la profondeur indiquée dans la section de code :

ainsi pour l'instruction WRITE RESULTAT, ii y a :

'25 – '23 = 2 instructions machine générées.

La pose de points d'arrêt est aussi facile :

l'instruction CALL SOMME (A, B, @ S) est implantée en :

'7E2C + '35 = '7E61

On pourra, sous AID, poser un point d'arrêt à cette adresse :

DBI '7E61

10.1.2 - UTILISATION DE LA TABLE DES SYMBOLES

Si la directive !TSYM est présente en tête d'un module, le code objet "linkeditable" comporte la table des symboles de ce module.

A l'édition de liens EDILE imprime pour chaque module leur table des symboles après avoir recalculé les adresses de ceux-ci en fonction de l'arrangement relatif de ces modules. Les symboles sont regroupés suivant leur appartenance à un bloc ; le numéro du bloc qui les contient est imprimé avant la liste des symboles de ce bloc. Les adresses des symboles sont des adresses relatives à 0. Ainsi dans l'exemple précédent la directive !TSYM conduit au moment de l'édition de liens à l'impression du listing suivant :

* *	
** MODULE NO 01	Module CALCUL
* *	
** BLOC NO 02 (02)	Bloc procédure LIRE
0015 Y	
0014 x	
0017 NOMBRES	
0016 CHIFFRES	
** BLOC NO 03 (03)	Bloc procédure ECRIRE
0021 PTSUM	L'adresse de chargement étant
0020 SORTIE	'7E00, l'implantation de RESUL
0022 RESULTAT	est '7E00 + '1E = '7E1E
001E RESUL	
001F DIX	
** BLOC NO 01 (01)	Bloc procédure CALCUL
000F LOC	
001D ECRIRE	
0011 s	
0013 LIRE	Le relais vers la procédure LIRE est à
0005 P I L E	l'adresse :
0012 I	'7E00 + '13 = '7E13
0010 B	
000F A	
**	
** MODULE NO 02	Module SOMME
* *	
** BLOC NO 01 (04)	Le numéro qui figure entre parenthèses est
	le numéro de bloc remis à jour en fonction du
0073 LOC	nombre de blocs du module précédent
0075 Z	
0074 Y	
0073 X	

10.2 - MODULE DRIP16 (Debugging Real time Interactive Program)

Nous présentons ici l'interface langage PL16 - DRIP16. Se reporter au manuel de référence de DRIP16 pour ce qui concerne le fonctionnement, la mise en œuvre et l'utilisation de cet outil.

C'est un processeur indépendant qui doit être chargé sous le système d'exploitation avant le chargement du programme comportant les appels à DRIP16. Ce module fournit une aide à la mise au point complète des programmes écrits en PL16, en visualisent certaines informations, au cours du déroulement du programme ; il permet aussi une maintenance aisée des programmes PL16.

Il existe en 2 versions :

DRIP16 - A version minimum (associée au produit compilateur PL16) fonctionnant sous tout superviseur
DRIP16 - B version complète (associée aux produits RTES-D et RTES-C) fonctionnant sous tout superviseur supportant un disque.

Le module DRIP16 est activé :

- soit par le programme lui-même s'il comporte les directives ou instructions appropriées (1)
- soit par des commandes données à la console.

10.2.1 - DIRECTIVES AU COMPILATEUR

! TRACE [n]

Cette directive porte sur toutes les définitions de label ou de procédure qui suivent. Pour chacune de ces définitions, le compilateur PL16 génère systématiquement un appel approprié au module DRIP16. Cet appel est de "degré" n si n est spécifié. Si n est absent il est de degré 1 ou du degré indiqué dans la dernière directive DEGRE. Si le module DRIP16 est à l'état "actif" (voir instructions) il donne, pour chaque appel un message de *l'un* des types suivants :

- <NOM DE PROCEDURE >
  qui signifie que l'on est entré dans la procédure ou sorti de celle-ci
- <NOM DE LABEL>
  qui signifie que l'on est passé par l'étiquette nommée.

! NOTRACE

Cette directive termine la portée de la directive précédente, c'est-à-dire que les appels au module DRIP16 ne sont plus générés.

! DEGRE [d]

Cette directive définit le "degré" d des appels au module DRIP16 générés sur les instructions de mise au point qui sont décrites dans le paragraphe suivant.

$1 \leqslant n, d \leqslant 255$

(1) Cette activation réalisée par une instruction "SVC + 7" générée par le compilateur, détruit l'indicateur "carry".

10.2.2 - INSTRUCTIONS

SILENCE [d]	Cette instruction désactive les appels suivants : entrée/sortie de procédure, passage par label.
NOSILENCE [d]	Cette instruction active les appels suivants : entrée/sortie de procédure, passage par label.
RSNAP [d]	permet la visualisation du contenu des registres.
XPRINT [d] chaine	permet l'impression d'un message en cours d'exécution du programme.

SNAP [d] ({ [@] nom de variable [+ depl] / RA } , n [,F]) avec F = 1, 2, 3

permet la visualisation de n mots à partir de l'adresse donnée par nom de variable ou contenue dans le registre RA, dans le format

1 - Hexadécimal
2 - ASCII
3 - Décimal

XSYST [d] n [,liste de nombres]

permet la visualisation d'informations systèmes sous RTES-D ou RTES-C

n = 1	Files du "sheduler"
= 2	État des événements
= 3	Phases d'avancement
= 4	Occupation des partitions
= 5	Taux d'attente des partitions
= 6	" d'occupation des zones dynamiques
= 7	Portion de ZDR (cet appel comporte 2 paramètres : rang du 1er mot à "dumper" ; nombre de mots à "dumper")
= 8	État périphérique (cet appel comporte 1 paramètre : numéro de FU ou SU)
= 9	BCT d'une tâche (cet appel comporte 1 paramètre : numéro de tâche)
= 10	Liste des opérations différées et périodiques
= 11	" des tâches en attente d'événement
= 12	Liste des ressources de l'application
= 13	"Dump" des zones de FMS
= 14	"Dump" de zone système (cet appel comporte 2 paramètres : adresse absolue du 1er mot à "dumper" nombre de mots à "dumper").

Remarque :

Le "degré" d'un appel est spécifié par d ou, si ce dernier est absent, par celui apparaissant dans la dernière directive DEGRE ou encore est défini égal à 1 par défaut.

Nomenclature des appels à DRIP16

E/S procédure passage par label	appels implicites	appels locaux	
SILENCE NOSILENCE SNAP RSNAP XPRINT	appels explicites par instruction		
XSYST		appel système	

Conclusion :

Grâce aux visualisations données par le module DRIP16, l'utilisateur peut suivre de très près le déroulement d'un programme lancé en mode continu ; il dispose ainsi d'un outil de mise au point rapide, utilisable dans tout contexte de programmation et doté, par les commandes de sélection qu'il offre, d'une très grande souplesse.

10.2.3 - COMMANDES AU MODULE DRIP16

MULTI — Définition du contexte d'utilisation de DRIP16 : mono ou multiprogrammation.

ONTR / OFTR [, Pi, Pj] — Validation et inhibition des appels locaux (sauf SILENCE et NOSILENCE qui sont toujours effectifs) pour toute priorité ou les priorités indiquées.

DEGR, L1, L2, D1, D2 [,P] — $1 \leq Li, Di \leq 255$

Définition des plages de "degré" valides pour toute priorité ou la priorité P.
plage L1, L2 informations éditées sur U5
plage D1, D2 informations stockées sur fichier circulaire créé sur U6

ONSYS / OFSYS [, Ni, Ni] — Validation et inhibition des appels système Ni, Nj ou de tous les appels système

DFIL [, taille] — Création du fichier circulaire de taille Kmots (30 K par défaut).

DISC , R , FICNAM - PW — Catalogage fichier plein "au repos"

DISC , A , FICNAM - PW — Catalogage fichier actif en cours.

FLTR	"Dump" des "flag" de DRIP16
REGS	Dump des registres sur EL après un appel superviseur
CONT	Reprise du programme après un appel superviseur

Remarque :

Pour la version DRIP16 - A les seules commandes admises sont :

MULTI

ONTR
OFTR sans paramètres

DEGR, L1, L2

REGS et CONT

10.3 • TABLE DE CORRESPONDANCE DES RÉFERENCES

Pour la version PL 1 164 001 01/, il est possible d'éditer sur LO la table des références croisées.
Cette édition est demandée par la commande XREF placée devant les commandes de compilation IPLC ou CPLC. Elle reste effective jusqu'à la commande NOXF.

Exemple :

```
CALL PL
LO LP
SI SOURCE1
BO BINAIRE
XREF
IPLC
              ——————→ Production de la table des références
S1 SOURCE2
NOXF
CPLC
              ——————→ pas de table des références
```

Remarque :

La table des références est fabriquée sur le fichier temporaire affecté par SO *.
Lors du CALL PL, cette affectation est automatique.
Lors d'un RUN PL, cette affectation doit être demandée (commande SO *).

Présentation de la table des références

Les références sont rangées par ordre alphabétique et présentées ainsi :

Etiquette	N° ligne du listing de la définition de la référence	**Type**	N° lignes du listing de la définition de la référence et de l'utilisation de la référence

Exemples

VAR	265	REF. PROCEDURE	265	268	412	
ZBAR	144	WORD	144	254	455	465
TABLO	484	ARRAY WORD	484	1517	1577	

ANNEXE

	Pages
MOT RESERVES DU LANGAGE	185
CHARGEMENT DES BASES	186
PASSAGE DE “CLES’ AU SUPERVISEUR	188
TRANSFERT DE PARAMETRES	191
ACCES AUX IDENTIFICATEURS	196
OPERANDES, OPERATEURS ET REGISTRES	197
ERREURS DETECTEES A LA COMPILATION	198
INDEX ALPHABÉTIQUE	200

I - LISTE DES MOTS RESERVES DU LANGAGE PL16

A
ACTIVATE
AND
ARM
ARRAY
ASSIGN
ATTACH

B
BACKWARD
BEGIN
BI
BIT
BO
BYTE

C
CA
CALL
CARRY
CASE
CC
COMMON
CONSTANT
CONTINUE
CONTROL
CR
CYCLE

D
DU1
DU2
DU3
DU4
DU5
DU6
DU7
DU8
DATA
DECR
DEF
DETACH
DO
DUMMY

E
EC
EL
ELSE
EM
END
ENTRY
EOE
EOF
EU
EXECUTE
EXIT
EXT

F
FABS
FIRST
FLOAT
FLT
FNEG
FOR
FORWARD
FROM

G
GAP
GOTO

H
HARD
HP
HR

I
IB
IF
IFIX
IM
IN
INCR
INDIRECT
INPUT
IOCB
INSTRUCTION
IS

K
KSTORE

L
LABEL
LL
LO
LOCAL
LONG
LP
LPFILE

M
MAIN
MASTER
MESS
MESSMAX
MODE
MOVE

N
NEG
NEGATE
NORM
NOSILENCE
NOT
NSN

O
OF
ON
OR
OUTPUT
OVERFLOW

P
PC
POINTER
PRIOR
PRIVATE
PRMAX
PROCEDURE
PULL
PUSH

Q
QUIT

R
RA
RAB
RANK
RB
RC
READ
REF
REGISTER
RELEASE
REQUEST
RES
RESET
RESSOURCE
RESTORE
REWIND
RFL
RK
RL
RLSE
RSLO
RSNAP
RX
RY
RW

S
SARD
SARS
SAVE
SCLD
SCRD
SCLS
SCRS
SECTION
SEGMENT
SET
SI
SILENCE
SKIP
SLLD
SLRD
SLLS
SLRS
SNAP
SO
SOFT
STACK
START
STATUS
STEP
STOP
SWAP
SYN

T
TU1
TU2
TASK
THEN
TK
TEST
TIMES
TO
TP
TR
TRACE
TS
TSYM

U
U1
U2
U3
U4
U5
U6
UNTIL
USING

W
WAIT
WHILE
WORD
WORKING
WRITE

X
XOR
XTNUSR
XPARTITION
XPRINT
XAPPMAX
XSYST
XSYSTEME

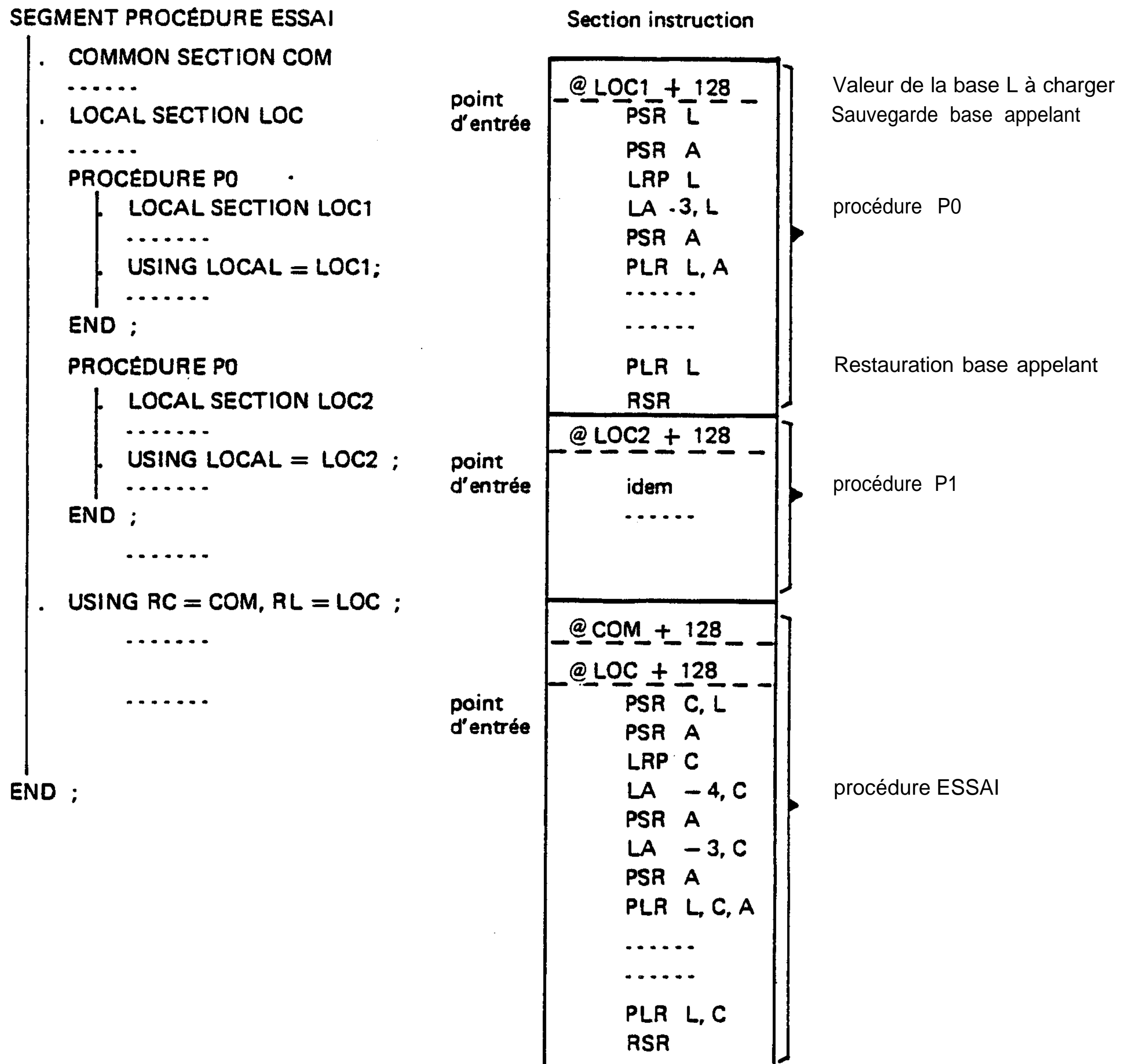
Z
ZE

II - CHARGEMENT DES BASES

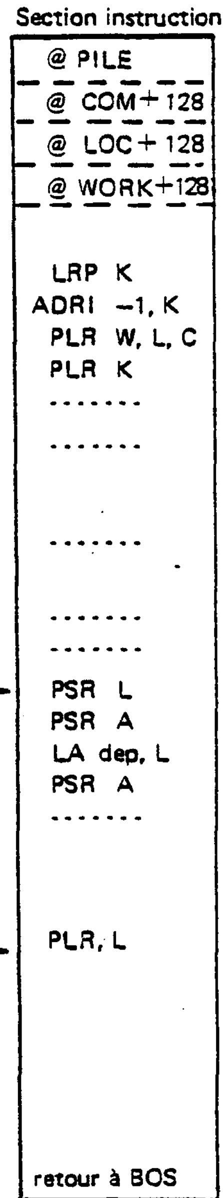
Nous présentons dans ce paragraphe le code objet généré par le compilateur lors d'une instruction USING :

- si le bloc qui contient cette instruction est défini par une instruction composée il doit exister une section de données locale accessible et non encore complète où le compilateur générera automatiquement l'adresse de la section à charger.
- si le bloc est une procédure il n'y a pas de restriction car l'adresse de la section ou les adresses des sections à charger sont générées dans la section instruction.

Exemples :



```
MAIN PROCEDURE ESSAI
  . KSTORE SECTION PILE
      RES 10 ;
  . COMMON SECTION COM
      ......
  . LOCAL SECTION LOC
      ......
  . WORKING SECTION WORK
      ......
  . USING RC = COM, RL = LOC, RW = WORK
    RK = PILE ;
      ......
      ......
      ......
      ......
            BEGIN
              . LOCAL SECTION L1
                  ......
                  ......
              . USING LOCAL = L1 ;  ——→
                  ......
                  ......
            END ;                   ——→
      ......
      ......
  END.
```



pas de sauvegarde
de bases car programme
principal

Sauvegarde de L
génération dans la section
locale accessible (LOC)
de l'adresse de L1
(déplacement dep)

Restauration L

Remarque :

Pour la version PL 1 164 001 01/, la directive !!GLRM permet de générer dans une unité de compilation LRM suivi du ou des mots à charger sur l'instruction .USING =.
La directive !!NOGLRM ou la compilation d'une autre unité sans directive !!GLRM génère ce qui est présenté ci-dessus.

III UTILISATION DE L'INSTRUCTION START

Exemples :

1) Cas d'une seule clé

```
MAIN PROCÉDURE ESSAI
  . KSTORE SECTION PILE
        RES 20 ;
  . LOCAL SECTION L0
        WORD INDIC =(0) ;
        EXT PROCEDURE ESSAI ;          « NECESSAIRE POUR AVOIR ACCES AU NOM
        ARRAY 4 WORD TABCLE = (1,      « NOMBRE DE CLES
                              "CLEF",
                              @ ESSAI) ;
        . . . . . .
  . USING RK = PILE, LOCAL = L0 ;
        GOTO CLEDEJAPASSEE ON (INDIC /= 0) ;
        RA := @ ABCLE (0) ; INCR INDIC ;
        START ;
     CLEDEJAPASSEE :                   « PROCESSEUR
END.
```

Le point de lancement de ce programme au chargement et le point de lancement par la clé "CLEF" donnée sous le contrôle "du superviseur" à la place de "BOS" sont identiques ; ils correspondent au point d'entrée de la MAIN PROCEDURE. Le test d'un indicateur permet de savoir si la clé a déjà été passée au superviseur ou non.

2) Cas de plusieurs clés

```
MAIN PROCEDURE ESSAI
 . COMMON SECTION CO
   ......
 . LOCAL SECTION LOC
   ......
   ......
   WORD INDIC = (0)  , P ;
   EXT  PROCÉDURE ESSAI  ;     « NECESSAIRE POUR ACCES AU NOM
   POINTER LABEL ENTRÉE   = (@ ESSAI) ;
   ARRAY 7 WORD TABCLE    = (   2     ,
                             "CLE1"   ,
                             @ENT1    ,
                             "CLE2"   ,
                             @ ENT2)  ,
   ......
   PROCÉDURE ENT1                             « ENTREE PROCESSEUR1
"CLE1" --->  USING LOCAL = LOC  ;             « RY = 1
             P,RY : = 1 ;
             GOTO ENTREE  ;
   END  ;
"CLE2" --->PROCÉDURE ENT2                     « ENTREE PROCESSEUR2
             USING LOCAL = LOC  ;             « RY = 2
             P, RY : = 2 ;
             GOTO ENTREE
   END  ;
 . KSTORE SECTION PILE
     RES  20  ;
   USING KSTORE = PILE, COMMON  = CO, LOCAL = LOC ;
   GOTO PROCESS ON  (INDIC  / = 0)  ;
   RA  : = @TABCLE (0)  ;     INCR  INDIC  ;
   START  ;
 PROCESS  : GOTO PROC2  ON  ( P   = 2)  ;
 PROC1  :  .....
           .....
           .....                              « PROCESSEUR 1
           .....
 PROC2  :  .....
           .....
           .....
           .....                              « PROCESSEUR 2
           .....
END.
```

Le lancement du programme, après le chargement, a lieu au point d'entrée de la MAIN procédure. Les clés sont passées au superviseur après le chargement des bases et du registre RK.

Le lancement d'un processeur par une clé a lieu au point d'entrée d'une procédure, la base RL est chargée pour permettre le branchement au point d'entrée de la MAIN procédure. Un indicateur a été positionné pour permettre l'aiguillage vers l'un ou l'autre des processeurs.

Cette méthode permet d'utiliser la même séquence de chargement des bases et du registre RK, quel que soit le point d'entrée dans le programme : on sait en effet que le chargement de RK doit se trouver en tête de la MAIN procédure.

IV - PASSAGE DE PARAMETRES ENTRE PROCEDURES PL16

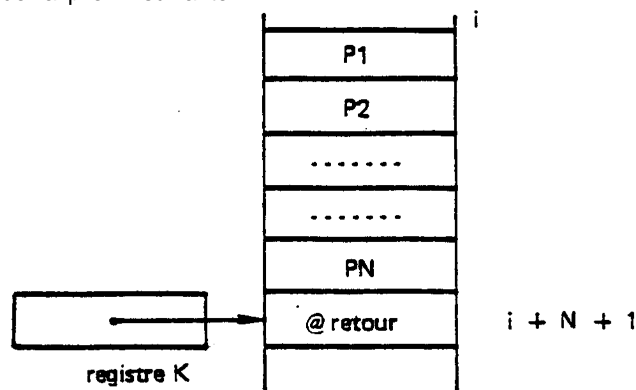
Nous précisons ici comment sont compilés l'appel et la définition d'une procédure PL16. L'utilisateur disposera alors des éléments lui permettant soit :

- d'écrire en langage assembleur un sous-programme externe pouvant être appelé depuis un programme PL16
- d'appeler, depuis un programme en langage assembleur, une procédure externe définie en PL16.

Nous donnons enfin les règles à respecter quand aux types de paramètres pour une compatibilité avec les appels et définition FORTRAN.

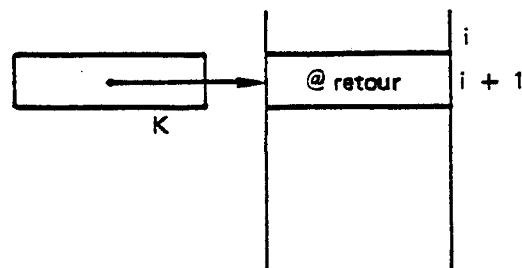
- APPEL D'UNE PROCEDURE PL16

L'appel d'une procédure PROC, par l'instruction CALL PROC (P1, P2, ... PN), consiste à lui fournir la configuration de la pile K suivante :



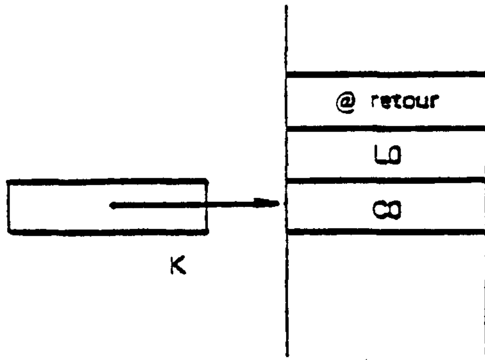
- RÉCUPÉRATION DES PARAMETRES PAR UNE PROCEDURE PL16

Elle consiste, au début de la procédure PROC, à recopier les paramètres effectifs dans les variables correspondantes de la procédure (ce sont les N paramètres formels) et à tasser la pile



Remarque :

Il a pu y avoir, dans la procédure, initialisation des registres de base, avant la récupération des paramètres. Les bases correspondantes de l'appelant ont alors été sauvegardés et l'état de la pile après récupération est le suivant :



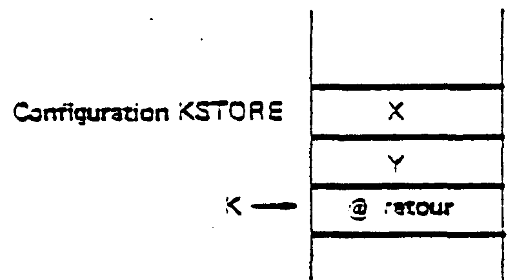
En conclusion. une procédure externe écrite en assembleur ou en Fortran devra récupérer les paramètres selon la méthode décrite dans le paragraphe 8.8.2 ; une procédure externe écrite en PL16 devra recevoir les paramètres dans la figure du paragraphe 8.8.1

Exemple :

Séquence de récupération de paramètres générée par le compilateur PL1600.

1) Appel d'une procédure

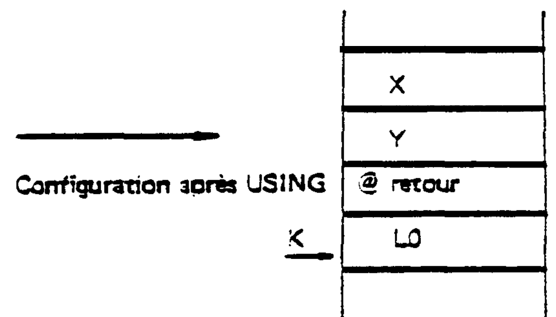
```
    CALL PROC (X, Y) ;
  LA    X
  PSR   A
  LA    Y
  PSR   A
  BSR   relais   procédure
retour.....
```



On remarque que l'accumulateur est détruit

2) Définition de la procédure

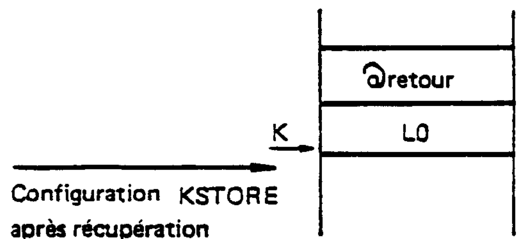
```
  PROCEDURE PROC (P1, P2),
     . LOCAL SECTION LOC
       WORD P1, P2 ;
     , USING RL = LOC ;
       ......
  END ;
```



L0 est la valeur de la base L de l'appelant

3) Séquence générée après l'instruction "USING"

```
ADRI    -2, K
PLR     A
STA     P2
PLR     A
STA     P1
ADRI    4, K
PLR     B, A
ADRI    -2, K
PSR     A, B
```



- COMPATIBILITE FORTRAN - PL16 (cf manuel de référence du langage FORTRAN)

Un appel PL16, une procédure PL16 sont compatibles respectivement à un sous-programme FORTRAN, un appel FORTRAN si :

- les paramètres transmis ou reçus sont des adresses (pointeurs en langage PL16). En effet en FORTRAN un paramètre effectif est toujours une adresse.
- l'appel d'un sous-programme FORTRAN donne dans RA le nombre de paramètres transmis :

exemple :

```
WORD X, Y;
POINTEUR WORD Z ;
CALL SUBFOR (@X, @Y, Z, RA : = 3) ;
```

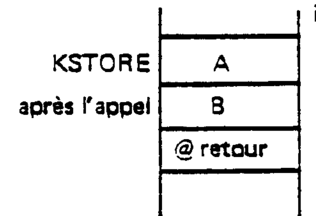
- EXEMPLE D'APPEL DE SOUS-PROGRAMME ASSEMBLEUR PAR UN PROGRAMME PL16

Sous réserve de respecter les conditions qui viennent d'être vues, un programme PL16 peut appeler des sous-programmes écrits en assembleur ou en Fortran.

Nous donnons ici un exemple de sous-programma assembleur appelé par un programme PL16.

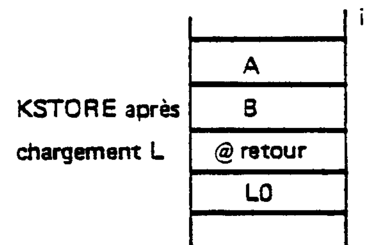
PROGRAMME PRINCIPAL EN PL16

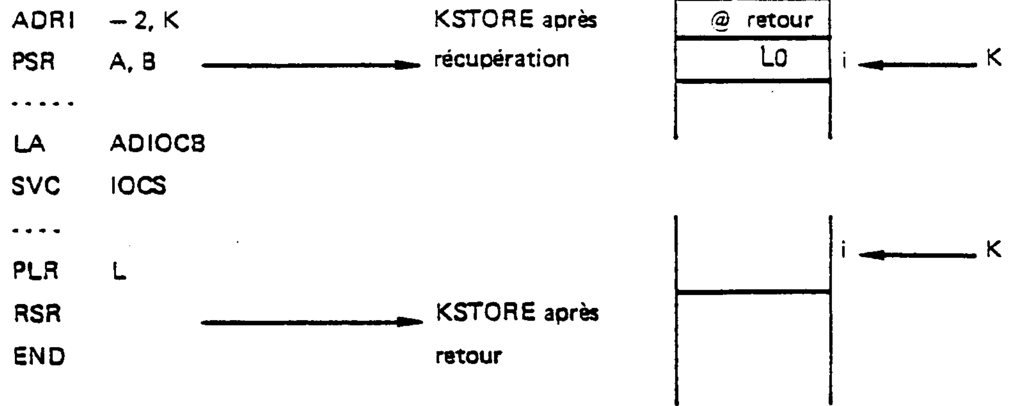
```
MAIN PROCEDURE ESSAI
  . LOCAL  SECTION  LOC
     REF PROCÉDURE REPONS ;
     DEF  LPFILE  COMAND  =  (MODE  :  INPUT,  EM  ;  EU  :  PC  ;  DATA  :  BUFFER  ;
                               EOE  :  10  ;  CONTROL)  ;
     DEF  ARRAY  10  BYTE  BUFFER  ;
     WORD  A,  B  ;
     ......
  . USING  LOCAL  =  LOC  ;
     ......
     CALL  REPONS  (A,  B)  ;
     ......
END.
```



SOUS-PROGRAMME EN ASSEMBLEUR

```
         LOCAL  DONNEE
         ENT REPONS               < L'EXTERNE REPONS est défini dans ce programme
         EXT  COMAND
         EXT  BUFFER              < COMAND et BUFFER sont définis ailleurs
IOCS   :  VAL 8
PARAM1 : WORD 0
PARAM2 : WORD 0
ADIOCB : WORD COMAND < ADRESSE DU LPFILE
ADBUF : WORD BUFFER, X < ADRESSE DU BUFFER AVEC BIT INDEX

            PROG  REPOND
            WORD  PARAM1  +  128     < VALEUR DE LA BASE L DE L'APPELE
REPONS  :    PSR    L                < SAUVE BASE L DE L'APPELANT : L0
            PSR    A                 < ET CHARGE LA BASE L DE L'APPELE
            LRP    L
            LA     -3, L
            PSR    A
            PLR    A, L
            ADRI   -2, K
            PLR    A
            STA    PARAM2            < RECUPERATION DES PARAMETRES
            PLR    A
            STA    PARAM1
            ADRI   4, K
            PLR    A, B
```




ADRI -2, K
PSR A, B
.....
LA ADIOCB
SVC IOCS
....
PLR L
RSR
END
KSTORE après
récupération
KSTORE après
retour
@ retour
LO
i
K
i
K

Accès aux Identificateurs (Ecriture)

Type déclaré	Adresse	Valeur	Valeur pointée
Variable simple A	@ A	A	X
tableau : T	@ T (expression entière) @ T (I) (adresse effective) T (buffer E/S)	(pointeur implicite non accessible)	T (I) (accès indirect, implicite)
Pointeur de variable : PTA	Non	PA	& PTA
Pointeur de tableau de variable : PTA	**Non**	PTA	& PTA (I)
Label L Procédure P	L (inst. branchement) P (CALL)	X	X
Pointeur : Label PL Procédure PP	Non	PL (assignation ou PP initialisation)	PL (inst. branchement) PP (CALL)
Tableau de Label TL Procédure TP	Non	Non	TL (I) (branchement) TP (I) (CALL)
Pointeur tableau Label PTL Procédure PTP	Non	Non	PTL (I) (branchement) PTP (I) (CALL)

Tableau des compatibilités - opérandes, opérateurs, élément assigné

Elément d'une expression	Élément assigné				
	RA	RB, RX, RY	variable entière	variable longue, RAB	variable flottante RFL
var. FLOAT					terme
var. LONG				terme	
{− / NOT} registre	1er terme	1er terme			
@ variable	1er terme (ad. effective)	1er terme (ad. effective)	terme (Cte adresse)		
Registre	terme	terme	terme unique (RA, si var. BYTE)		
CARRY	après +, −				
Chaîne	2 caractères	1 caractère	2 caractères		
[−] Littéral	terme	terme (littéral court)	terme	terme	
Variable mot	terme	1er terme	terme	terme	
Variable byte	1er terme	1er terme	1er terme	1er terme	
+, −	oui	oui	oui		oui
*, /	oui		oui	oui	oui
AND, OR, XOR	oui	sur registre	oui		
AND, NOT	devant un registre	devant un registre			
NEG, NOT, SWAW	oui	oui	oui		
Décalages	simples			doubles	
fonction {− / NOT}			oui		

ERREURS DETECTEES A LA COMPILATION

I - Erreurs n'interrompant pas l'analyse

40	Erreur de syntaxe niveau 1
41	Néant
42	Nombre de sections de données supérieures à 14
43	Une section synonyme ne peut être déclarée DUMMY ou REF
44	Une section ne peut être synonyme d'une section de type différent
45	Une section ne peut être synonyme d'une section DUMMY ou REF
46	Initialisation interdite dans section DUMMY, REF
	Déclaration de procédure interdite ailleurs que dans le corps d'une procédure
47	Un élément ne peut être synonyme d'un élément déclaré par REF
48	Il ne peut y avoir synonyme entre éléments simples et tableaux
49	L'adresse de l'élément synonyme dépasse les limites de la section
4A	Seule une variable de type mot peut être initialisée avec une adresse
4B	Trop de valeur d'initialisation pour la taille de l'élément
4C	Etiquette définie dans une profondeur autre que celle où elle est déclarée
4D	Etiquette déjà définie
4E	Branchement relatif trop long
4F	La variable est dans une section inaccessible
50	Pas de local accessible pour génération
51	Section de donnée saturée
52	Un élément d'un "dummy section" ne peut être définie comme externe
53	Un élément défini comme externe ne peut être déclaré synonyme
54	Section déjà assignée par USING
55	Plus de 6 lettres pour un nom d'externe
56	Néant
57	Opérande interdit
58	Indirection interdite
59	Paramètre incorrect (opérations optionnelles et Entrées-Sorties)
5A	2 caractères au maximum dans une chaîne utilisée comme opérande
5B	Index interdit
5C	Index obligatoire
5D	On ne peut sortir par EXIT, procédure que de la procédure en cours
5E	Trop d'instruction dans un CASE . . . . OF
5F	Pas assez d'instruction dans un CASE . . . . OF
60	L'accès indirect à un LPFILE est interdit pour ASSIGN
61	Paramètres mal ordonnés (IOCB, LPFILE)
62	Un label ne peut être défini comme externe que lors de sa définition
63	Taille de la KSTORE supérieur à 256
64	Un calcul sur adresse dans une expression de constante ne peut concerner un externe ou une adresse indéfinie
65	CYCLE sur bloc procédure interdit ou manque nom de procédure dans l'instruction EXIT qui fait sortir du bloc procédure
66	Valeur d'initialisation trop grande
67	Paramètre non déclaré
68	Initialisation du registre RK interdit, ailleurs que dans MAIN PROCEDURE
69	Indexation double interdite
6A	Changement de contexte interdit sur un branchement conditionnel
6B	Degré d'une instruction spécifique à DRIP16 hors de la plage [1,255]
6C	Paramètre d'une instruction d'appel à DRIP16 incorrect
6D	Longueur section table > 16K (version PL 1 164 001 01/)
6E	Longueur section programme 16K (version PL 1 164 001 01/)
6F	Utilisation RA comme Registre index dangereuse (version PL 1 164 001 01/)
80	Erreur de syntaxe niveau 2

II - Erreurs provoquant l'abandon de la compilation

F1	Débordement pile de travail (niveau d'imbrication trop grand)
F2	Débordement table des symboles
F3	Fin pile de travail (sortie de bloc incorrecte)
F4	Saturation mémoire 1
F5	Imbrication de bloc incorrecte
F6	Imbrication de bloc incorrecte
F7	Saturation mémoire 2
F8	Erreur de lecture permanente
F9	Débordement pile KSTORE du compilateur
FA	Fichier TRANSS - PL absent sur la FU D2 (P16 -S)
FB	Partition mémoire donnée au compilateur P16-S inférieur à 6 K.

Remarque sur l'erreur «syntaxe»

Le sens général de cette erreur est :

- L'élément syntaxique trouvé est interdit à cet endroit

Erreur 40 : l'analyse est poursuivie

Erreur 80 : l'analyse est poursuivie mais dans certains cas seulement à partir du premier «;» rencontré.

INDEX ALPHABETIQUE

A
Accès aux données 22, 43, 196
ACTIVATE 149
Adresse de variable 34, 56, 66, 73
Aide à la mise au point 173
Algorithme 23
ARM 149
ASSIGN 107
Assignation (généralités) 62
ATTACH 154

B
Base (cf. section de données
Bit (instructions sur...) 79, 84
Bloc définition 9
instruction composée et... 57
contexte d'un... 58, 88
profondeur de... 169
numéro de... 169
Boucle 97
Branchements 87, 92
Byte 35, 36, 66

C
Cadrage (d'opérande) 65
CASE....OF 86
Chaines 7
Choix (instruction de...) 86
Commentaire 6
Common (cf. section)
Communication (système) 108, 165
Compilateur PL16 162
Compilation 165
Composée (instructions) 13, 57
Compte rendu (échange) 125
Conditions 81
Constante : type 7
déclaration 33
... adresse 34, 73
CONTINUE 29, 137
Contexte : de bloc 58
sauvegarde de 88, 94, 96
restauration de 88, 96
CYCLE 87

D
Décalages 67, 70
Déclarations 11, 27
associées à l'option scheduler 146, 156
DECR 103
DEF 112, 127
Définition - d'étiquette 47
d'éléments externes 112, 127
DETACH 154
Directives 159
DO 97
DRIP16 178
Dummy (section) 134

E
Echange decl. 45, 107, 121
Edition de lien 114
exemple 167
Entrées-sorties - inst. 106, 123, 125
ENTRY 147
Erreur exemple 172
liste 198
Etiquette déclaration 47
référence a . . . 47
définition d' . . . 47
EXECUTE 123
EXIT 87
Externe - symbole 112, 127
Expression déconstante 33
d'assignation 62
de RA 66
de registre 68
flottante 75
évaluation d' . . . 64, 65, 70, 197

F
Fonctions flottantes 76
moniteur 108
speciales, E/S 123
FOR 97
Format texte source 5
d'instruction machine 141

G
Générations implicites 134
de données 50, 53, 129, 132, 134
d'instructions 53, 66, 68, 134
GOTO 87, 92

H
Hard tache 145

I
Identificateur écriture 7
portée 27, 112
accès 196
IF
INCR 103
Indicateur flottant 76
positionnement 78
test 84, 103
Indice - synonyme 40
Index - accès aux variables 55

Initialisation 39, 131, 134
INPUT cf. écriture
Instruction a 13, 54
  composées 57
  section d'... 61
  déclaration d'... 140

K
Kstore déclaration de 53
  initialisation 104

L
Label déclaration 47
  définition 47
Listing 168
  exemple 170
  exploitation 176, 179
Littéral 8, 134
Local (cf. section)
LPFILE 45

M
MAIN Procédure 110
MASTER (cf. tache
MESS, MESSMAX
  (cf. ressource)
MODE (cf. échange)
MOT réservés 184
MOVE 155
O
Objet (programme) 166
Opérandes 64, 197
Opérateurs 64, 65, 197
Options (cf. directives) 159
OUTPUT (cf. échange)
Overflow (cf. condition)
N
Nom 7
Nombre 7
NOSILENCE 178
P
Paramètre formel 50
  effectif 91
Pile Kstore section 53
  utilisateur 156
  instruction sur... 104, 157
Pointeur déclaration 36
  utilisation 43
  d'échange 107
  de ressource 147
PORTEE d'identificateurs 22, 27
PRIOR 154
PRIVATE 147
Privilégiées - instructions 149

PRMAX 147
Procédure
  notion de 15
  imbrication 23
  déclaration 49
  appel de.. 91, 92
  sortie de... 88
Programme exemple 17
  production de 165

Q
Quit 149

R
RANK 155
Récursif (appel) 26
READ 106
REF 112, 127
Référence
  à identificateur externe 127
Registre remarque 4
  nom de 54
  assignation de 63, 65
  utilisation implicite 134
RELEASE 149
REQUEST 149
RES 42
RESET 78
Réservés (mots) 184
Ressource (déclaration de) 147
RESTORE 105
REWIND 125
RSNAP 178
S
SAVE 105
SILENCE 178
Section
  de données 22, 28
  déclaration 29, 137
  caractéristique 32
  dummy 134
  synonyme 135
Segmentation - de programme 109
Sémaphore 147
SET 78
SKIP 125
SNAP 179
SOFT 145
STACK 156
START 108, 145
STATUS 125
STEP 97
STOP 108

Structure de programme		23
Symbole	table des . . .	160
	de base	5
	externe	112
Synonyme	de variable	40
	de section	134
SWAP opérateur		64
T		
Table	des symboles	
	option	160
	utilisation	176
Tableau	déclaration	36
	accès	42, 55, 196
Tache	déclaration	145
TASK		145
TEST		108
Test	condition	80
	bit	84
	indicateur	84, 103
TIMES		97
Trace		
	option	161, 178
TSYM		159
U		
Using	instruction	58, 104
Unité	d'échange	46
	de compilation	109
V		
Variable		
	déclaration	35
	accès	42, 55, 196
	assignation de . . .	71
W		
WAIT		106, 123, 149
WHILE		97
WORKING (cf. section)		
WRITE		106, 125
X		
XPRINT		178
XSYST		179