

# **SOLAR**

**ASM16**

**Assembleur pour langage machine**

**LOGICIEL**

**LOGICIEL**

**LOGICIEL**

**LOGICIEL**

**LOGICIEL**

# LANGAGE ASM16

## MANUEL DE RÉFÉRENCE

SOMMAIRE	Pages
1 - INTRODUCTION	1 - 1
1.1 - PRESENTATION	1 - 1
1.1.1 - Caractéristiques de l'assembleur	1 - 1
1.1.2 - Caractéristiques du langage symbolique	1 - 2
1.1.3 - Procédure d'utilisation de l'assembleur	1 - 2
1.2 - DEFINITIONS	1 - 3
1.2.1 - Instruction	1 - 3
1.2.2 - Constante numérique	1 - 3
1.2.3 - Constante alphanumérique	1 - 3
1.2.4 - Table	1 - 3
1.2.5 - Constante adresse pour l'adressage indirect	1 - 4
1.2.6 - Directive	1 - 4
1.2.7 - Phrase	1 - 4
2 - SECTIONNEMENT DES PROGRAMMES	2 - 1
2.1 - UTILISATION IMPLICITE DES BASES	2 - 2
2.2 - POSSIBILITE D'EXTENSION DES ZONES	2 - 2
2.3 - ZONE ACCESSIBLE PAR UNE BASE	2 - 2
3 - REGLES D'ECRITURE DES PROGRAMMES	3 - 1
3.1 - ECRITURE DES OPERANDES	3 - 4
3.1.1 - Ecriture des nombres	3 - 4
3.1.2 - Symboles	3 - 5
3.1.3 - Expressions	3 - 5
3.1.4 - Exemples d'expressions	3 - 6
3.1.5 - Ecriture du mode d'adressage	3 - 7
3.1.6 - Référence au compteur d'assemblage	3 - 7
3.2 - ECRITURE DES INSTRUCTIONS	3 - 8
3.2.1 - Codes opérations	3 - 8
3.2.2 - Instructions avec référence mémoire	3 - 8
3.2.3 - Instructions avec opérande immédiat	3 - 9
3.2.4 - Instructions de saut conditionnel	3 - 10
3.2.5 - Instructions de décalage et de traitement de bits	3 - 11
3.2.6 - Instructions entre registres	3 - 12
3.2.7 - Instructions d'arithmétique flottante	3 - 12
3.2.8 - Instructions diverses	3 - 13

4 - DIRECTIVES DE SECTIONNEMENT DES PROGRAMMES	4 - 1
4.1 - DESCRIPTION GENERALE	4 - 1
4.1.1 - Généralités	4 - 1
4.1.2 - Sections de données	4 - 1
4.1.3 - Sections de tables	4 - 1
4.1.4 - Sections de programme	4 - 2
4.2 - DIRECTIVE COMMON	4 - 2
4.3 - DIRECTIVE LOCAL	4 - 3
4.4 - DIRECTIVE TABLE	4 - 3
4.5 - DIRECTIVE PROG	4 - 3
4.6 - DIRECTIVE DSEC	4 - 4
4.7 - DIRECTIVE USE	4 - 5
5 - DIRECTIVES DE GENERATION DE DONNEES	5 - 1
5.1 - DIRECTIVE WORD	5 - 1
5.2 - DIRECTIVE BYTE	5 - 2
5.3 - DIRECTIVE FLOAT (E)	5 - 3
5.4 - DIRECTIVE ASCI (E)	5 - 3
5.5 - DIRECTIVE DZS	5 - 4
5.6 - DIRECTIVE DO (E)	5 - 5
6 - DIRECTIVES DE DEFINITION DE SYMBOLES	6 - 1
6.1 - DIRECTIVE EQU	6 - 1
6.2 - DIRECTIVE VAL	6 - 2
6.3 - CHARGEMENT DU COMPTEUR D'ASSEMBLAGE	6 - 2
7 - ASSEMBLAGE CONDITIONNEL : IF (E)	7 - 1
8 - DIRECTIVES RELATIVES AUX PST : PSTS et PSTH	8 - 1
9 - DIRECTIVES DE LIAISON INTERPROGRAMMES : EXT et ENT	9 - 1
10- DIRECTIVES DE SECTIONNEMENT ET D'IDENTIFICATION DES PROGRAMMES	10-1
10.1 - DIRECTIVE EOT	10-1
10.2 - DIRECTIVE END	10-2
10.3 - DIRECTIVE IDP (E)	10-3

11	-	DIRECTIVES RELATIVES A LA TABLE DES SYMBOLES	11 - 1
11.1	-	DIRECTIVE EST (E)	11 - 1
11.2	-	DIRECTIVE NDS	11 - 1
11.3	-	DIRECTIVE DST	11 - 2
12	-	LISTE D'ASSEMBLAGE	12 - 1
13	-	ORGANISATION DES PROGRAMMES	13 - 1
13.1	-	INTRODUCTION	13 - 1
13.2	-	ORGANISATION MODULAIRE DES PROGRAMMES	13 - 1
13.2.1	-	Organisation modulaire des instructions	13 - 1
13.2.2	-	Organisation modulaire des données	13 - 2
1.3.3	-	SEGMENTS ET ZONE DE DONNEES COMMUNES	13 - 2
1.3.4	-	ADRESSAGE DES OPERANDES EN MEMOIRE	13 - 3
13.4.1	-	Utilisation des registres de base	13 - 3
13.4.2	-	Initialisation des bases C et L	13 - 4
1.3.5	-	ADRESSAGE DES TABLES	13 - 4
13.5.1	-	Adressage indirect d'une table	13 - 5
13.5.2	-	Adressage indirect post-indexé d'une table	13 - 5
13.5.3	-	Adressage basé d'une table	13 - 6
13.5.4	-	Réservation des tables	13 - 7
13.5.5	-	Tables communes et tables locales	13 - 8
13.6	-	ADRESSAGE LORS DES RUPTURES DE SEQUENCE	13 - 9
13.6.1	-	Ruptures de séquence entre segments	13 - 9
13.6.2	-	Ruptures de séquence à l'intérieur d'un même segment	13 - 9
13.7	-	SOUS-PROGRAMMES	13 - 10
13.7.1	-	Zone de rangement commune	13 - 10
13.7.2	-	Sous-programmes communs	13 - 11
13.7.3	-	Transmission des paramètres : méthode générale	13 - 11
13.7.4	-	Transmission des paramètres : méthode utilisant les registres	13 - 13
13.7.5	-	Transmission des paramètres : paramètres placés derrière BSR	13 - 13
13.7.6	-	Utilisation de la zone de rangement pour des opérandes (sous-programmes réentrants)	13 - 13

14	-	UTILISATION DE L'ASSEMBLEUR	14 - 1
14.1	-	INTRODUCTION	14 - 1
14.2	-	ACTIVATION DE L'ASSEMBLEUR	14 - 2
14.2.1	-	Commande IASM	14 - 2
14.2.2	-	Commande LASM (E)	14 - 3
14.2.3	-	Commande CASM	14 - 3
14.3	-	CORRECTION DES ERREURS	14 - 4
14.4	-	PASSAGE EN ASSEMBLAGE CLAVIER	14 - 6
14.5	-	CORRECTION D'UN PROGRAMME PAR SURCHARGE	14 - 7
14.6	-	INTERRUPTION D'UN ASSEMBLAGE EN COURS	14 - 8
14.7	-	UTILISATION DES ASSEMBLEURS ASM et ASM-E	14 - 8
14.7.1	-	Chargement par l'intermédiaire du chargeur autonome	14 - 8
14.7.2	-	Chargement par l'intermédiaire du chargeur sous système	14 - 9
14.7.3	-	Particularité concernant l'assembleur ASM-E	14 - 10
14.8	-	UTILISATION DE L'ASSEMBLEUR ASM-D	14 - 11
15	-	ANNEXES	
15.1	-	Liste des directives	15 - 1
15.2	-	Liste des numéros d'erreur	15 - 3
15.3	-	Codes instructions	15 - 6

## 1 - INTRODUCTION

### 1.1 - PRESENTATION

#### 1.1.1 - Caractéristiques de l'assembleur

L'assembleur est un programme conversationnel qui transforme les instructions écrites dans le langage symbolique source en un programme objet translatable.

L'assembleur présente plusieurs particularités qui facilitent la programmation.

— **Assemblage en un seul passage :**

L'assembleur délivre un programme objet en ne faisant qu'une seule lecture du programme symbolique.

— **Assemblage conversationnel :**

Les erreurs relatives à une phrase sont détectées immédiatement après la lecture de celle-ci et signalées sur un organe d'impression. L'utilisateur peut alors apporter toute correction utile, avant de demander la poursuite de l'assemblage.

Ceci élimine en grande partie les pertes de temps dues aux fautes de frappe ou aux erreurs d'écriture.

— **Assemblage incrémental :**

Il est possible, par des assemblages successifs, de compléter un programme déjà assemblé, car la table des symboles peut être conservée d'un assemblage à l'autre.

— **Assemblage paramétré et conditionnel :**

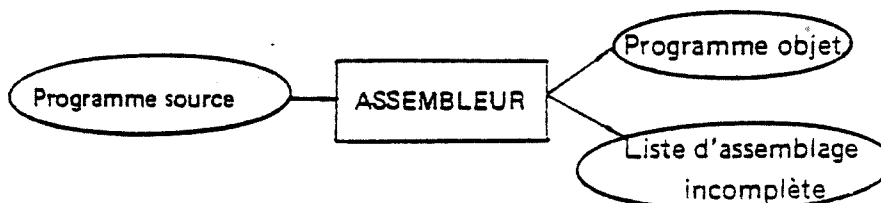
Le fait de paramétrer un programme permet d'adapter celui-ci à toute configuration particulière.

— **Edition d'une liste d'assemblage :**

La présence de références en avant donne lieu à une liste incomplète.

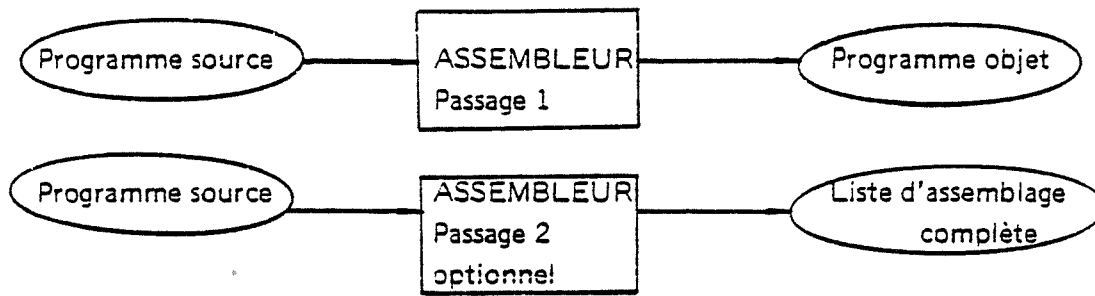
Une liste d'assemblage complète peut être obtenue lors d'un second passage.

a)





b)



### 1.1.2 - Caractéristiques du langage symbolique

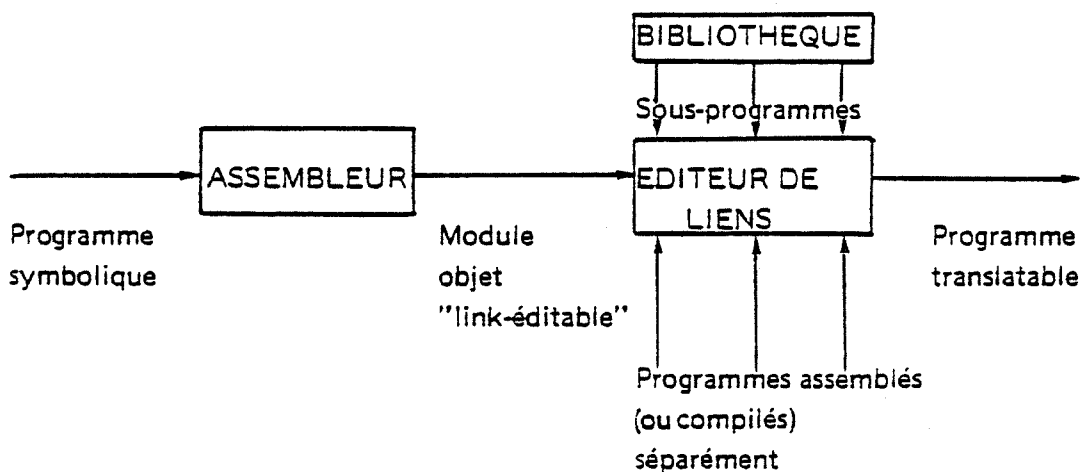
L'assembleur permet de disposer, pour l'écriture des programmes, des simplifications apportées par l'emploi d'un langage symbolique :

- utilisation de codes opérations mnémoniques,
- utilisation de symboles réservés pour indiquer le mode d'adressage,
- utilisation de noms symboliques définis par le programmeur pour écrire les adresses, les opérandes immédiats,
- possibilité de réserver des tables, d'inclure des commentaires,
- écriture des constantes sous forme décimale, hexadécimale ou alphanumérique.

### 1.1.3 - Procédure d'utilisation de l'assembleur

A partir d'un programme symbolique, l'assembleur produit un programme objet en binaire translatable. L'implantation d'un tel programme n'est déterminée qu'au moment de son chargement en mémoire. Il est assemblé à partir de l'adresse zéro. Toutes les adresses sont calculées à partir de cette origine. Elles sont traduites au moment du chargement pour correspondre aux adresses mémoire.

Le langage d'assemblage donne la possibilité de découper un programme en une ou plusieurs unités assemblées séparément. Les modules objets ainsi obtenus sont ensuite reliés entre eux par l'éditeur de liens.





Trois versions de l'assembleur sont disponibles :

- ASM, assembleur de base,
- ASM-E, assembleur étendu, disposant d'un plus grand nombre de directives que l'assembleur ASM,
- ASM-D, version disque de l'assembleur ASM-E.

L'assembleur ASM s'exécute sous le contrôle du système d'exploitation BOS-A sur une configuration minimale de 4 K mots.

Les assembleurs ASM et ASM-E s'exécutent sous le contrôle de BOS-B et BOS-C sur une configuration minimale de 8 K mots.

L'assembleur ASM-D s'exécute sous le contrôle de BOS-D et BACKM, moniteur permettant une activité Background sous RTES-C/D.

**Remarque :**

Dans la suite du manuel le symbole **E** signifie que la fonction décrite n'est disponible que sous ASM-E et, par conséquent, sous ASM-D.

## 1.2 - DEFINITIONS

### 1.2.1 - Instruction

Mémoire du programme comportant un code instruction, interprétée en tant que pas de programme lors de l'exécution.

En plus du code opération une instruction peut comporter une adresse avec indication du mode d'adressage ou un opérande immédiat.

### 1.2.2 - Constante numérique


Mémoire du programme destinée à servir d'opérande, dont le contenu correspond à une valeur numérique exprimée en décimal ou en hexadécimal, cette valeur étant fixée avant le début de l'exécution du programme.

### 1.2.3 - Constante alphanumérique

Mémoire du programme comportant les codes d'un ou plusieurs caractères utilisés dans les entrées-sorties, son contenu étant fixé avant le début de l'exécution du programme.

### 1.2.4 - Table

Plusieurs mémoires consécutives comportant des constantes, des résultats de calculs, etc. . . , que le programme adresse en utilisant un indice.

**Bull**  1.2.5 - Constante adresse pour l'adressage indirect

Mémoire du programme contenant l'adressage d'une autre mémoire sous la forme d'une adresse sur 15 bits et éventuellement une indication d'indexation.

1.2.6 - Directive

Ordre donné à l'assembleur, que celui-ci exécute pendant l'assemblage. Par exemple, la directive END indique la fin du programme source et arrête l'assemblage.

1.2.7 - Phrase

Une instruction, une constante, une réservation de table, une directive constituent une phrase du programme source. Chaque phrase est terminée par un retour chariot (une seule phrase par ligne) lorsque le symbolique est sur ruban papier ou la fin de la carte si le symbolique est sur cartes (une seule phrase par carte).

**Remarque :**

Dans la suite les crochets sont utilisés pour délimiter les éléments optionnels lors de l'écriture des phrases sources.

**Exemple :**

USE Base [, Expression translatable]

## 2 - SECTIONNEMENT DES PROGRAMMES

Ce point sera repris en détail au chapitre 13.

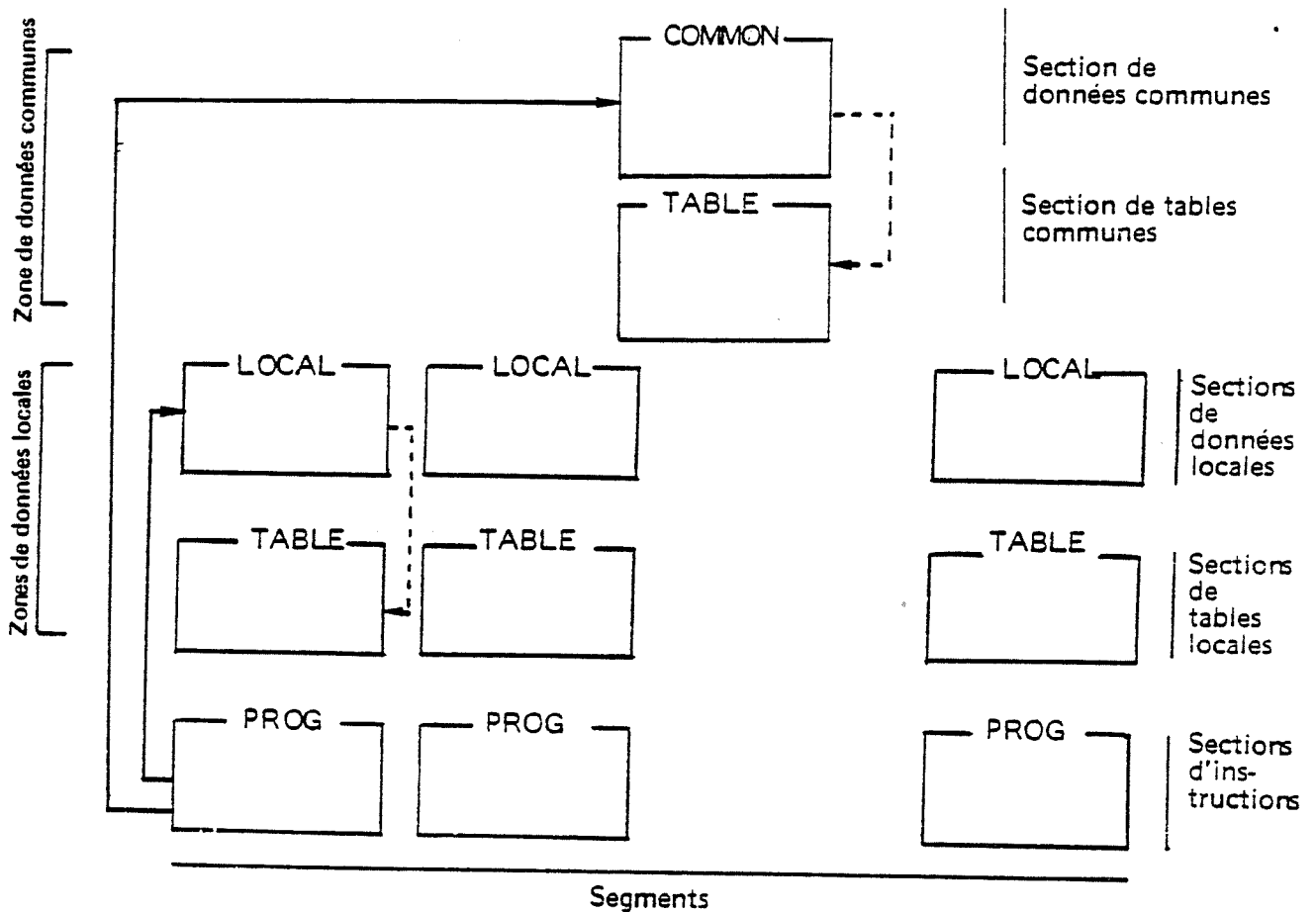
Un programme peut se décomposer en un ensemble de segments et une zone de données communes aux divers segments.

Un segment constitue un module du programme. Il comporte une zone de données locales et une section d'instructions.

Chaque zone de données (communes ou locales) comprend :

- une section de données auxquelles on accède directement par l'intermédiaire d'un registre base,
- une section contenant toutes les tables auxquelles on accède par adressage indirect post-indexé.

Un certain nombre de directives permettent le sectionnement d'un programme. Ce sont COMMON, LOCAL, PROG et TABLE.



## **Bull** 1 - UTILISATION IMPLICITE DES BASES

Chaque registre base de l'unité de traitement permet l'accès direct à 256 mots consécutifs.

Dans la plupart des cas les sections de données ont une longueur inférieure ou égale à cette limite.

Pour l'assembleur implicitement la base C pointe sur la section commune et la base L sur la section locale du segment en cours. Il suppose :

- la base C chargée par l'adresse de début de la section commune incrémentée de 128,
- la base L chargée par l'adresse de début de la section locale incrémentée de 128.

Pour utiliser ce mode d'adressage, trois règles sont à respecter :

- limitation des sections de données à 256 mots,
- chargement en début de programme de la base C,
- chargement à l'entrée dans chaque segment de la base L.

Afin de limiter la taille des sections de données, il est important d'en exclure les tables et de les introduire dans des sections spéciales notées TABLE. Un moyen d'accéder alors à une table consiste à utiliser l'adressage indirect post-indexé à partir de constantes adresses appartenant aux sections locales ou commune. On peut également utiliser un adressage basé par W.

### 2.2 - POSSIBILITE D'EXTENSION DES ZONES

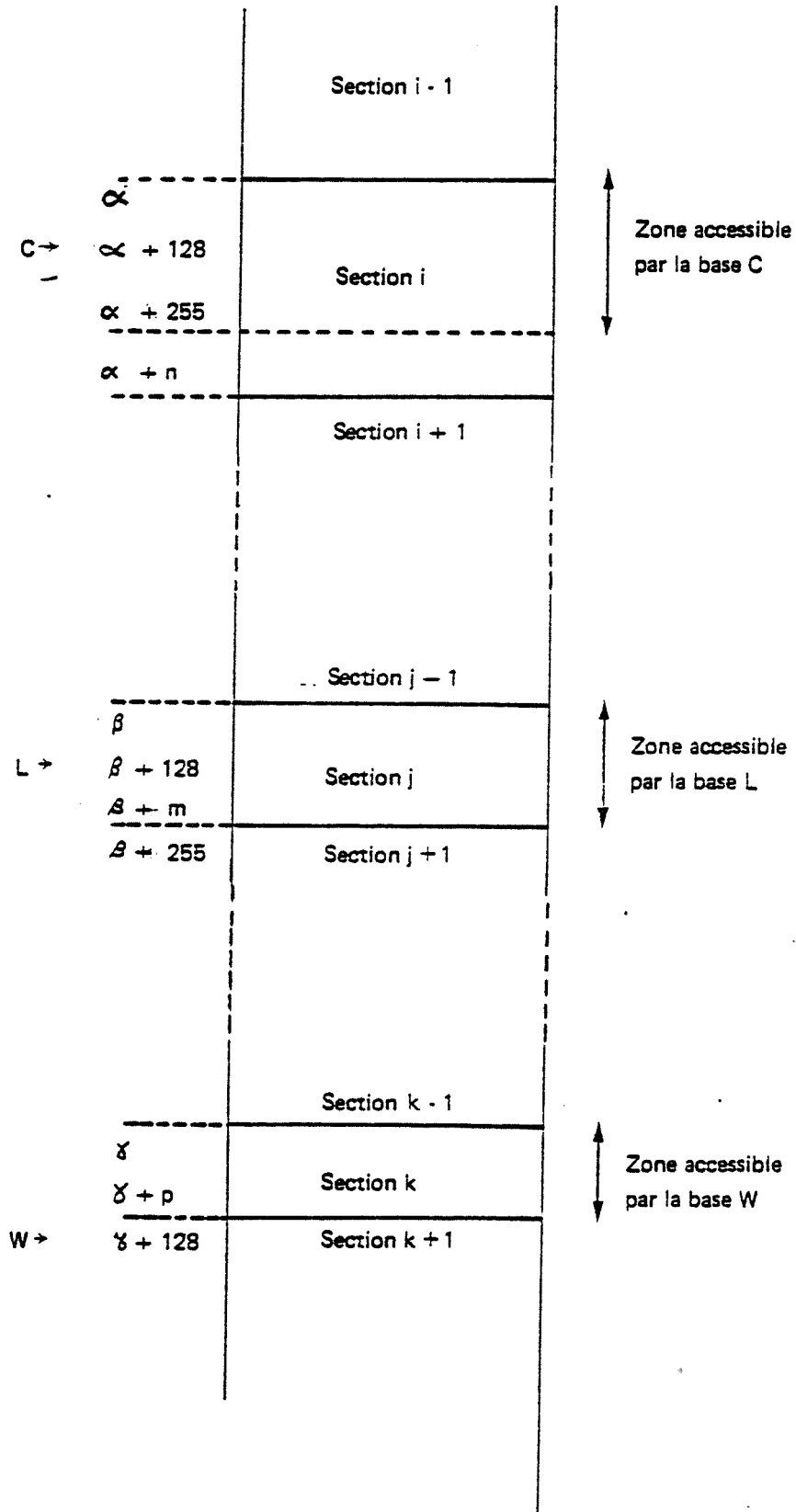
Certains programmes peuvent nécessiter des sections de données comportant plus de 256 mots.

Un certain nombre d'instructions permettent de réinitialiser les bases. Il suffit alors d'indiquer à l'assembleur le changement intervenu. Ceci est réalisé par le jeu des directives USE (voir chapitre 4).

Ce procédé n'est à employer que dans des cas limités. La mise au point des programmes est, en effet, facilitée par une décomposition plus fine en plusieurs segments.

### 2.3 - ZONE ACCESSIBLE PAR UNE BASE

Quelle que soit la longueur de la section à laquelle est affectée une base, la zone accessible par l'intermédiaire de cette base est limitée à la section. On ne peut donc atteindre directement des mots extérieurs à la section.



### 3 - REGLES D'ECRITURE DES PROGRAMMES

Les programmes symboliques sont écrits en utilisant certains des caractères du code ASCII :

- les lettres A, B, C, . . . . ., N, O, P, . . . . ., Z
- les chiffres 0, 1, . . . . . 9
- les caractères spéciaux suivants :





	retour chariot	
	espace	
	point	·
	deux points	:
	virgule	,
(E)	point virgule	;
	et	&
	dollar	\$
	plus	+
	moins	-
	multiplier	*
	diviser	/
	inférieur	<
	apostrophe	'
	double apostrophe	"
	flèche à gauche	←
	flèche en haut	↑
	dièse	#


Les autres codes sont soit ignorés (codes non imprimables), soit signalés comme erreur. Toutefois les commentaires et les chaînes de caractères peuvent comporter n'importe quel caractère imprimable autre que ← et ↑.

Un programme symbolique est une succession de phrases.

**Bull** assembleur tronque toute phrase source comportant plus de 72 caractères de telle sorte que, dans le cas de cartes, les colonnes 73 à 80 sont ignorées.

Deux caractères spéciaux facilitent l'écriture des programmes :

- la flèche  annule le début de l'enregistrement,
- la flèche  annule le caractère qui la précède, quel que soit ce caractère (sauf pour les caractères ,  et retour chariot).

Une phrase comporte généralement une instruction ou une directive destinée à l'assembleur. Elle peut également ne comporter qu'un commentaire. Le premier caractère d'un commentaire est toujours .

Chaque phrase peut comporter quatre zones auxquelles aucun cadrage n'est imposé. Il suffit de respecter l'ordre suivant :

- zone étiquette
- zone opération
- zone opérande
- zone commentaire.

Les zones étiquette et commentaire sont facultatives.

Les deux caractères d'annulation  et  sont reconnus par l'assembleur, quelle que soit la zone dans laquelle ils apparaissent.

Les deux écritures suivantes sont équivalentes :

STA AUX < COM  ETIQ : LAI "A   30  ETIQ : LAI 30

#### Zone étiquette :

Une étiquette est un symbole défini par l'utilisateur auquel est associée la valeur du compteur d'assemblage. Elle est utilisée pour repérer la ligne du programme source où elle est définie. Il est interdit de redéfinir un symbole étiquette déjà défini dans le programme source.

Une étiquette est composée de caractères alphanumériques et doit respecter les règles suivantes :

- le premier caractère est alphabétique,
- elle comporte six caractères au maximum,
- elle est suivie immédiatement de deux points,
- tout nom symbolique correct non encore employé peut être utilisé comme étiquette.

#### Zone opération :

Elle renferme soit un code opération, soit une directive d'assemblage. L'espacement entre les zones étiquette et opération est facultatif. Par contre, au moins un espace doit séparer les zones opération et opérande.

**Exemple :**

```
ALPHA:      LA      MEM
BETA:  STA      AUX
LB      MEM1
```

**Zone opérande :**

L'opérande est séparé du code opération ou de la directive par au moins un espace.

Plusieurs informations peuvent être nécessaires pour préciser un opérande :

- le déplacement et la base dans une instruction avec référence mémoire,
- l'adresse et l'indication d'indexation dans une constante adresse,
- les registres dans une instruction sur registres.

Les divers éléments sont séparés par des virgules (aucun espace n'est admis entre eux). La virgule joue ainsi le rôle de séparateur d'éléments relatifs au même opérande.

D'autre part les deux directives de génération de données WORD et BYTE (voir chapitre 5) peuvent comporter plusieurs opérandes. Dans ce cas, la séparation des divers opérandes est réalisée par des points virgules.


Cette possibilité conduit à un allègement de l'écriture des programmes en évitant la répétition de lignes relatives à la même directive.

E

Les deux écritures suivantes sont ainsi équivalentes :

```
WORD      1
WORD      TOTO, X      ⇔      WORD      1 ; TOTO, X ; - 224
WORD      - 224
```



Le point virgule joue le rôle de séparateur d'opérandes relatifs à la même directive.


La zone opérande peut être vide ; cela est réalisé lorsque le premier caractère rencontré après la zone opération est le caractère  ou le retour chariot.

**Exemple :**

```
HALT  ATTENTE D'INTERRUPTION
```

**Zone commentaire :**

La zone commentaire est facultative. Elle peut contenir tout caractère ASCII, exceptés le retour chariot reconnu comme fin d'enregistrement,  et  reconnus comme caractères d'annulation.

Elle commence par le caractère .

Une ligne de programme peut ne comporter que la zone commentaire.



**Bull** Exemple :

GAMMA : LAI +3 < CHARGEMENT DE  
< L'ACCUMULATEUR AVEC LA VALEUR +3

**3.1 - ECRITURE DES OPERANDES**

**3.1.1 - Ecriture des nombres**

**a) Ecriture décimale**

Les nombres décimaux positifs sont précédés ou non du signe + , les nombres négatifs étant toujours précédés du signe -.

**Exemple :**

+ 512  
- 43  
1247



**b) Ecriture hexadécimale**

Les nombres hexadécimaux sont toujours précédés d'une apostrophe. Il n'est pas nécessaire de préciser les poids forts qui sont pris à zéro par défaut.

Les deux écritures suivantes sont ainsi équivalentes :

'1A  $\Leftrightarrow$  '001A

**c) Chaine de caractères**

Utilisée en tant que nombre une chaine de caractère est composée de 1 ou 2 caractères ASCII autres que  ,  et retour chariot, encadrés par des doubles apostrophes.

**Exemple :**

"AB"  
"&"

Le nombre de caractères d'une chaine peut cependant être supérieur à la limite ci-dessus lorsqu'elle apparait dans la zone opérande des directives ASCII et IDP (voir chapitres 5 et 10).

### 3.1.2 - Symboles

Un opérande peut contenir un nom symbolique, à condition que ce dernier soit défini dans le programme ou déclaré externe par l'intermédiaire de la directive EXT.

#### a) Symbole translatable

Une étiquette est un symbole défini par l'utilisateur. Cependant la valeur qui lui est associée est déterminée par l'assembleur. Elle représente une adresse.

Il s'agit d'un symbole translatable en ce sens que sa valeur est relative à l'adresse origine du programme assemblé.

La valeur définitive ne sera connue qu'au moment du chargement du programme objet en mémoire. Elle est fonction de l'implantation en mémoire.

#### b) Symbole absolu

La directive VAL permet de définir des symboles absolus. Leur valeur est indépendante de l'implantation en mémoire.

Les symboles translatables et absolus ne peuvent donc pas être employés l'un à la place de l'autre.

### 3.1.3 - Expressions

Symboles et nombres peuvent être combinés au moyen des opérateurs +, -, \*, et / pour former des expressions. Une expression est évaluée de la gauche vers la droite, sans priorité d'opérateurs. L'expression est la forme généralisée de l'opérande. Par la suite elle sera la seule utilisée. Une expression peut être absolue ou translatable.

Elle respecte les règles suivantes :

#### a) Expression absolue

Une expression absolue est une combinaison de termes absolus au moyen des quatre opérateurs. Un nombre, un symbole absolu ou la différence de deux symboles translatables sont des termes absolus.

La valeur d'une expression absolue est indépendante de l'implantation mémoire.



## b) Expression translatable

Une expression translatable est la somme d'un symbole translatable et d'une expression absolue ne comportant aucun opérateur  $\times$  et  $/$ . Le symbole translatable est le premier de la somme. Il ne peut être précédé d'un signe  $-$ .

La valeur d'une expression translatable est interprétée comme l'adresse d'un mot de la section où est défini le symbole translatable.

## c) Règles de la multiplication et de la division

Le caractère  $\otimes$  apparaissant dans une expression indique la multiplication des éléments situés à sa gauche (multiplicande) par l'élément situé à sa droite (multiplicateur).

$15 - \text{ABSOL} \otimes 3 + \text{TRANS1} - \text{TRANS2}$

multiplicateur  
multiplicande

**Multiplicateur et multiplicande doivent être absolus**

Le caractère  $\oslash$  apparaissant dans une expression indique la division entière des éléments situés à sa gauche (dividende) par l'élément situé à sa droite (diviseur).

**Diviseur et dividende doivent être absolus.**

## 3.1.4 - Exemples d'expressions

Soient ABSOL un symbole absolu de valeur 13, TRANS1 et TRANS2 deux symboles translatables de valeurs respectives 22 et 7.

Les deux expressions :

$- \text{ABSOL} / 5 + \text{TRANS1} - \text{TRANS2}$   
 $\text{TRANS1} - \text{TRANS2} \otimes 2 / \text{ABSOL}$

sont absolues. Leurs valeurs respectives sont 13 et 2.

L'expression suivante est translatable. Sa valeur est 30.

$\text{TRANS1} + \text{ABSOL} - 5$

Par contre l'assembleur interdit les expressions :

$\text{TRANS1} \otimes \text{ABSOL}$  (multiplicande non absolu)  
 $\text{TRANS1} - \text{ABSOL} / 3$  (expression translatable incorrecte)

**Remarque :**

Une expression vide est considérée par l'assembleur comme étant absolue et de valeur nulle.

LAI < CHARGEMENT DE A PAR LA VALEUR 0

## 3.1.5 - Ecriture du mode d'adressage

- Pour indiquer qu'une instruction utilise l'adressage indirect, on utilise le caractère  $\textcircled{\&}$  accolé à l'opérande.

Exemple :

LA  $\textcircled{\&}$ INDIR + 3 < RELAI D'INDIRECTION = INDIR + 3

- Pour indiquer qu'une constante adresse ou une instruction de décalage ou sur bit utilisent l'indexation, on se sert de la lettre X séparée de l'opérande par une virgule.

Exemple :

SLRS 13,X

L'adressage indirect n'est autorisé que pour les instructions avec référence mémoire.

## 3.1.6 - Référence au compteur d'assemblage

Lorsqu'on veut exprimer une adresse à l'aide d'un déplacement par rapport à l'instruction que l'on est en train d'écrire, on utilise pour cela le symbole \$ dont la valeur est celle de l'adresse relative où sera chargé le mot du programme correspondant.

Par exemple les deux écritures ci-dessous sont équivalentes :

$\textcircled{1}$	<pre>JMP SUITE LAI "*" SUITE : STA AUX</pre>	$\longleftrightarrow$	$\textcircled{2}$	<pre>JMP \$ + 2 LAI "*" STA AUX</pre>
-------------------	--	-----------------------	-------------------	---------------------------------------

Le seul intérêt de la seconde écriture est d'éviter la définition de l'étiquette SUITE.

En fait, le symbole \$ est un symbole toujours défini dont la valeur est celle du compteur d'assemblage, compteur initialisé à zéro en début d'assemblage et ensuite incrémenté après chaque instruction ou directive générant des données.

Le symbole \$ est assemblé suivant les mêmes règles qu'un symbole translatable. Il peut être utilisé dans les expressions.

Dans une expression translatable il est donc obligatoirement le premier élément. Une exception : le symbole \$ peut être associé dans une différence à un autre élément translatable pour former un terme absolu.

Exemple :

WORD TOTO + \$ - TRANS.



## 3.2 - ECRITURE DES INSTRUCTIONS

### 3.2.1 - Codes opérations

Chaque code opération du calculateur a une représentation symbolique.

Les codes opérations sont classés en sept catégories qui correspondent à des instructions de types différents :

- instructions avec référence mémoire
- instructions avec opérande immédiat
- instructions de saut conditionnel
- instructions de décalage et de traitement de bits
- instructions entre registres
- instructions d'arithmétique flottante
- instructions diverses.

### 3.2.2 - Instructions avec référence mémoire

Les instructions avec référence mémoire comportent une adresse d'opérande. Cette adresse se compose d'une base et d'un déplacement par rapport à cette base.

L'assembleur admet la spécification des adresses sous ce format explicite.

Une expression absolue correspond au déplacement. Déplacement et base doivent apparaître dans cet ordre. Ils sont séparés par une virgule.

Définissant les huit bits poids faibles de l'instruction, l'expression absolue doit avoir une valeur comprise entre - 128 et 127.

**Exemple :**

```
STA 123,L  
LB &12,C
```

On peut cependant utiliser une méthode plus commode pour l'écriture des adresses. Il s'agit de la forme symbolique.

Dans ce cas, le déplacement ainsi que la base sont générés par l'assembleur.

On suppose qu'une section de données comporte la séquence :

```
PUIS10 : WORD 1  
        WORD 10  
        WORD 100  
LETRA  : WORD "A"
```

et que la base L pointe sur PUIS10.

Pour adresser la constante étiquetée par LETRA on peut écrire par exemple :

LA LETRA

L'assembleur détermine la base par laquelle la mémoire est accessible et le déplacement par rapport à cette base (dans l'exemple ci-dessus 3,L).

Plus généralement, l'adresse symbolique d'un opérande est représentée par une expression translatable dont tous les éléments doivent être définis. De plus, les conditions suivantes doivent être respectées :

- le premier élément de l'expression (élément translatable) appartient à une section de données sur laquelle pointe l'une des trois bases,
- la valeur de l'expression translatable représente l'adresse d'un opérande de cette même section de données,
- l'adresse est telle que l'opérande est accessible, c'est-à-dire le déplacement par rapport à la base est compris entre - 128 et 127.

Remarques :

- une adresse d'opérande écrite sous forme symbolique ne doit pas comporter de spécification de base. Ainsi l'écriture :

LA LETRA,C

est interdite lorsque LETRA est un symbole translatable. La base est générée par l'assembleur.

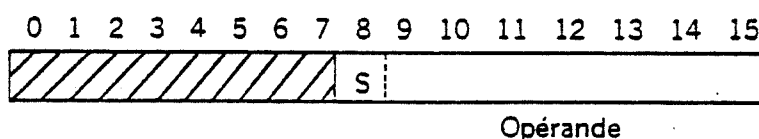
- les instructions avec référence mémoire peuvent être indirectes,
- lorsqu'une même zone est accessible par plusieurs bases l'assembleur choisit dans l'ordre : W, L et C.
- avec une adresse d'opérande écrite sous forme explicite aucun contrôle n'est effectué par l'assembleur.

### 3.2.3 - Instructions avec opérande immédiat

Les codes opérations des instructions avec opérande immédiat comportent en dernière lettre I. Ils sont combinés à une expression absolue.

Deux sous-ensembles sont à considérer, selon que l'opérande est codé sur 8 ou 9 bits.

#### a) Opérande immédiat 8 bits



C'est le cas de l'instruction d'addition immédiate à un registre ADRI, dont le champ opérande comporte dans cet ordre : une expression absolue, une virgule, un registre.

**Bull** La valeur de l'expression doit être comprise entre - 128 et 127.

L'évaluation étant effectuée sur 16 bits, cela revient à dire que les 9 bits de poids forts doivent être identiques.

Les écritures :

ADRI -128,A

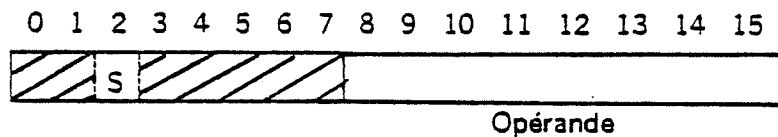
ADR! 'FFF,Y

sont autorisées.

Par contre l'assembleur interdit :

ADRI 'FF,B

b) Opérande immédiat 9 bits



La valeur de l'expression absolue doit être comprise entre - 256 et 255.

Exemples :

LBI "A"

LAI 'FFF4

ANDI 7

ORI 'FE

Remarque :

Lorsque sont introduites des limitations sur la valeur d'expressions absolues, seul intervient le résultat : les valeurs individuelles des divers éléments qui constituent l'expression absolue peuvent ne pas respecter les limites indiquées.

Ainsi l'écriture :

LAI -400 + '100

est autorisée. Elle est équivalente à :

LAI -144

3.2.4 - Instructions de saut conditionnel

Cette catégorie regroupe en fait toutes les instructions de saut qui sont relatives au pointeur P.

Elles comportent l'adresse de la rupture à effectuer lorsqu'une condition est réalisée. Cette adresse s'écrit sous forme d'une expression translatable dont le premier élément appartient obligatoirement à la section d'instructions en cours.

Le premier élément peut être le caractère \$ qui appartient à la section en cours d'assemblage.

La rupture ne peut s'effectuer qu'à une distance comprise entre -128 et 127 mots de l'instruction de saut.

**Exemple :**

On suppose la section de programme implantée comme suit :

```
'0100      RUPT : LAI      -20
              -
              -
              -
'0110      JAL      RUPT
```

Lors de l'assemblage de l'instruction d'adresse '0110, l'assembleur va générer le déplacement 'FO (-16).

### 3.2.5 - Instructions de décalage et de traitement de bits

Les instructions de décalage comportent toujours quatre lettres dont la signification est la suivante :

Shift	}	Logical	}	}	Right	}	Single
		Circular			Left		Double
		Arithmetical					

soit SLRS, SLRD, etc. . .

Elles comportent une valeur indiquant le nombre de décalages à effectuer.

Les instructions de traitement de bits comportent une valeur indiquant le rang du bit à traiter. Dans les deux cas, la valeur s'écrit sous la forme générale d'une expression absolue de valeur positive, inférieure ou égale à 31.

Ces instructions peuvent être indexées.

**Exemple :**

```
SCLD  11
SBT   30,X
```

**Remarque :**

L'instruction DBT (recherche du premier bit à 1 de AB) ne comporte aucun opérande.





### 3.2.6 - Instructions entre registres

Les instructions qui entrent dans cette catégorie réalisent une opération portant sur deux registres. Elles comportent un registre source et un registre destination.

Les règles suivantes ont été adoptées :

- les registres qui peuvent être employés sont notés A, B, X, Y, C, L, W, K.
- le champ opérande comporte le registre source, une virgule, le registre destination.

**Exemple :**

ADR A,B

- lorsque les registres source et destination sont le même registre, il est inutile de l'écrire deux fois.

Ainsi les deux écritures sont équivalentes :

EORR B,B  $\iff$  EORR B

- les instructions ADRP et LRP effectuent des opérations implicites sur P. Elles ne comportent donc qu'un seul registre.

**Exemple :**

ADRP X

Il en est de même des instructions ADCR, SBCR, CPZR et XIMR.

### 3.2.7 - Instructions d'arithmétique flottante E

Elles sont codées sur deux mots, le premier, toujours codé '3800, caractérisant une instruction de ce type.

#### a) Instructions avec référence mémoire

C'est le cas des instructions FLD, FST, FAD, FSB, FMP, FDV, FCAM et FCMZ.

Elles comportent une adresse d'opérande pouvant être écrite sous une forme explicite ou symbolique.

**Exemple :**

FLD 13,L  
FST MEM

**b) Instructions sur A et B**

C'est le cas des instructions FNEG, FABS, FIX, FLT, NORM et FCAZ.  
Elles ne comportent pas d'opérande.

**3.2.8 - Instructions diverses**

**a) Instructions sans opérande**

C'est le cas d'un certain nombre d'instructions telles que ACK, ACQ, HALT, RSR.

**b) Instructions LRM, PSR et PLR**

Ces instructions peuvent comporter dans un ordre quelconque jusqu'à huit registres séparés par des virgules.

**Exemple :**

PSR W,A,Y  
LRM C,L,W,K

**c) Instruction SVC**

L'instruction SVC comporte le numéro d'un sous-programme du superviseur qui est défini par une expression absolue donc la valeur doit être positive et inférieure ou égale à 255.

**Exemple :**

SVC 25  
ou SVC SP11

avec SP11 symbole valeur, inférieur ou égal à 255.

**d) Instruction ACTD**

L'instruction ACTD peut comporter un paramètre pouvant s'écrire sous la forme d'une expression absolue de valeur positive, inférieure ou égale à 7.

**Exemple :**

ACTD  
ACTD 4

**Instructions optionnelles** **(E)**

Les instructions introduites par les options CDA (Zone intertâches/Traitement de listes) et VSS 16 (Système câblé d'adressage virtuel) sont codées sur deux mots.

Elles appartiennent à l'une des deux catégories suivantes :

— instructions sans opérande

C'est le cas, par exemple, des instructions RCDA et WCDA.

— instructions avec opérande immédiat

Le second mot comporte alors une valeur, codée sur les bits 1 à 15, le bit 0 pouvant être utilisé en tant que bit d'indexation. Les instructions de ce type comportent donc un paramètre pouvant s'écrire sous la forme d'une expression absolue de valeur positive, inférieure ou égale à 32 767.

**Exemples :**

RBTM 153  
SBTM 0,X

## 4 - DIRECTIVES DE SECTIONNEMENT DES PROGRAMMES

### 4.1 - DESCRIPTION GENERALE

#### 4.1.1 - Généralités

Les mémoires utilisées dans un programme appartiennent à l'une des catégories suivantes :

- données

Elles sont générées par des directives spécialisées et peuvent être absolues ou translatables.

- zones de travail

Définie par l'utilisateur à l'aide de la directive DZS, une zone de travail est constituée de mémoires consécutives initialisées à zéro par l'assembleur.

- instructions.

#### 4.1.2 - Sections de données

Données et zones de travail sont seules autorisées à l'intérieur d'une section de données.

Les constantes adresses peuvent comporter des noms symboliques avant que ceux-ci aient été définis, avec ou sans déplacement. Elles peuvent utiliser une adresse appartenant à n'importe quel type de section autre que DSEC.

Toutes les phrases d'une section de données peuvent être étiquetées.

#### 4.1.3 - Sections de tables

Les sections de tables permettent de placer les tables à l'extérieur des sections de données. Elles obéissent aux mêmes règles que les sections de données.



#### 4.1.4 - Sections de programme

Une section de programme peut contenir des instructions et des constantes.

Les opérandes accessibles par les instructions avec référence mémoire doivent appartenir à des sections définies précédemment.

Ceci implique que toute expression translatable apparaissant dans le champ opérande de telles instructions soit définie.

Deux situations peuvent se produire :

##### 1) Utilisation implicite des bases C et L

Lorsque les sections de données sont limitées à 256 mots, la base C permet d'accéder aux données communes et la base L aux données locales du segment en cours.

##### 2) Extension des bases

La directive USE permet d'indiquer à l'assembleur un positionnement des bases. En ce qui concerne W son utilisation est indispensable puisqu'aucun positionnement implicite n'est réalisé pour cette base.

A un instant donné, la base spécifiée par une directive USE permet l'accès à un maximum de 256 mots d'une section de données (voir paragraphe 2.3).

#### 4.2 - DIRECTIVE COMMON

Syntaxe	<table border="1"><tr><td></td><td>COMMON</td><td>[Nom de section]</td></tr></table>		COMMON	[Nom de section]
	COMMON	[Nom de section]		
Fonction	La section de données communes est précédée par la directive COMMON pouvant comporter le nom de la section.			
Exemple	COMMON TACHE1			
Remarques	Après la rencontre de COMMON, l'assembleur suppose affectée à la base C la valeur du compteur d'assemblage incrémentée de 128. L'organisation des programmes introduite au chapitre 2 conduit aux contraintes suivantes : <ul style="list-style-type: none"><li>— il ne peut exister qu'une directive COMMON au plus par programme,</li><li>— exceptions faites pour DSEC et TABLE, aucune autre directive de sectionnement ne peut la précéder.</li></ul>			

## 4.3 - DIRECTIVE LOCAL

Syntaxe	<table border="1"><tr><td></td><td>LOCAL</td><td>[Nom de section]</td></tr></table>		LOCAL	[Nom de section]
	LOCAL	[Nom de section]		
Fonction	Toute section de données locales est précédée de la directive LOCAL pouvant comporter le nom de la section.			
Exemple	LOCAL    SPCONV			
Remarques	Après chaque rencontre de LOCAL, l'assembleur suppose affectée à la base L la valeur du compteur d'assemblage incrémentée de 128. L'organisation des programmes conduit aux contraintes suivantes : <ul style="list-style-type: none"><li>— il n'existe qu'une directive LOCAL au plus entre le commun et la première section de programme,</li><li>— il n'en existe qu'une au plus entre deux sections de programme (c'est-à-dire par segment).</li></ul>			

## 4.4 - DIRECTIVE TABLE

Syntaxe	<table border="1"><tr><td></td><td>TABLE</td><td>[Nom de section]</td></tr></table>		TABLE	[Nom de section]
	TABLE	[Nom de section]		
Fonction	Toute section de tables est précédée de la directive TABLE pouvant comporter le nom de la section.			
Exemple	TABLE    TABSOM			
Remarque	Le nombre de sections de tables dans un segment n'est pas limité par l'assembleur. Il semble cependant raisonnable de les regrouper dans une section unique.			

## 4.5 - DIRECTIVE PROG

Syntaxe	<table border="1"><tr><td></td><td>PROG</td><td>[Nom de section]</td></tr></table>		PROG	[Nom de section]
	PROG	[Nom de section]		
Fonction	Toute section de programme est précédée de la directive PROG pouvant comporter le nom de la section.			
Exemple	PROG    PRCONV			
Remarque	Deux sections de programme peuvent être consécutives. Ce cas se produit lorsqu'un segment ne comporte ni données locales, ni tables locales et ne travaille qu'avec le commun.			

**Bull** 16 - DIRECTIVE DSEC**Syntaxe**

	DSEC	[Nom de section]
--	------	------------------

**Fonction**

La directive DSEC permet de décrire l'arrangement d'une zone de mémoires sans réserver effectivement des mémoires. Elle définit une section de données comme "fictive".

Les données apparaissant dans une DSEC ne sont pas générées dans le programme objet. Seules sont prises en compte par l'assembleur les définitions de symboles. A cet effet un compteur est utilisé pour déterminer les adresses associées aux symboles. Il est initialisé à zéro en début de section et progresse par la suite de la même manière que le compteur d'assemblage.

Pour utiliser les symboles définis dans la DSEC deux conditions doivent être satisfaites :

- un registre base donne accès à la DSEC par l'intermédiaire de la directive USE,
- ce même registre est chargé avec l'adresse d'une zone mémoire.

Les symboles de la DSEC permettent alors de décrire cette zone.

**Exemples**

- utilisation de la même section de données par plusieurs programmes assemblés séparément.
- description de l'organisation d'une zone de mémoires dont l'emplacement varie au cours de l'exécution.
- description de la structure d'une table.

**Remarques  
générales**

- les directives de sectionnement ne peuvent pas être étiquetées.
- il n'existe pas de directive "fin de section". Toute directive de sectionnement ouvrant une nouvelle section clot implicitement la section précédente.
- lorsqu'une directive de sectionnement comporte un nom de section, celui-ci peut être utilisé ultérieurement comme opérande des directives EST et DST (voir chapitre 11). Il est unique dans le programme.  
D'autre part, au moment du traitement du programme objet par le chargeur traducteur ou le chargeur disque, les noms des diverses sections sont imprimés ainsi que leurs adresses d'implantation en mémoire.
- les symboles translatables définis dans une DSEC ne peuvent être utilisés que dans les expressions absolues (différence de deux symboles translatables) et les expressions translatables opérandes d'instructions avec référence mémoire.

#### 4.7 - DIRECTIVE USE

Syntaxe

USE	Base [, Expression translatable définie]
-----	--

Fonction

La directive USE permet d'indiquer à l'assembleur les registres de base qu'il peut utiliser pour l'adressage des opérandes, dans le cas des instructions avec référence mémoire.

Elle spécifie la valeur qui sera chargée dans la base par le programme à l'exécution mais n'en effectue pas le chargement.

Le champ opérande est constitué du nom de la base et d'une expression translatable définie. Une virgule les sépare.

Exemple 1

```
USE    C, MEM + 3
```

Cette directive indique que la base C aura, à l'exécution, la valeur de l'expression translatable MEM + 3.

L'étiquette MEM doit appartenir à une section de données, de tables ou à une DSEC mais en aucun cas à une section d'instructions.

A partir de cet instant les références mémoire remplissant les deux conditions suivantes :

- se reporter à un mot de la section où est définie l'étiquette MEM,
  - correspondre à une adresse telle que le déplacement par rapport à MEM + 3 est compris entre - 128 et 127,
- sont résolues par l'assembleur à l'aide de la base C.

Exemple 2

COMMON
DEBUT :    WORD
_____

PROG
_____
LA DEBUT    < ASSEMBLE COMME : LA   - 128,C
_____
USE C,DEBUT
_____
LA DEBUT    < ASSEMBLE COMME : LA   0,C
.....



**Bull** Remarques

- une directive USE portant sur une base annule le positionnement précédent de la base

Il peut s'agir, en particulier, du positionnement implicite dans le cas des bases C et L. L'utilisateur peut indiquer qu'il vient de revenir à ce positionnement par une directive USE ne comportant que le nom de la base.

Ainsi la directive :

```
USE C
```

indique que le positionnement précédent de la base C (dans l'exemple précédent en DEBUT) est annulé et que la base C pointe à nouveau sur le 128ème mot du COMMON.

- l'écriture :

```
USE Base,OFF
```

indique que la base spécifiée n'est plus utilisable pour l'adressage d'opérandes.

```
-----  
TABLO : DZS 50  
-----  
Utilisation de W          USE W,TABLO  
en tant que base          -----  
                           LA  TABLO  < ASSEMBLE  
                           -----  < COMME : LA 0,W  
  
Utilisation de W          USE W,OFF  
en tant que registre      -----  
de travail                LA  TABLO  < PHRASE  
                           -----  < ERRONÉE
```

- la directive USE ne peut pas être étiquetée.
- jusqu'à la rencontre d'une directive USE relative à la base W l'assembleur s'interdit l'emploi de cette base : les adresses symboliques doivent appartenir aux sections basées par C et L.
- la directive USE ne peut apparaître que dans une section PROG.

## 5 - DIRECTIVES DE GENERATION DE DONNEES

## 5.1 - DIRECTIVE WORD

Syntaxe	[Étiquette]	WORD	Expression [; Expression . . .]
---------	-------------	------	---------------------------------

avec

Expression ::= Expression absolue | Expression translatable [,X]

**Fonction** La directive WORD permet de générer des données occupant un mot mémoire (16 bits)

Elle peut avoir plusieurs opérandes, séparés par des points virgules.

**E** Les opérandes se présentent sous la forme d'expressions dont les valeurs, calculées par l'assembleur, sont affectées à des mots d'adresses consécutives.

Les données générées sont cadrées à droite dans les mots.

**Exemple** La directive :

```
WORD '1A; 5
```

engendre les mots consécutifs dont les contenus hexadécimaux sont :

```
'001A
'0005
```

**Remarques**

- associée à une expression absolue, la directive WORD a pour but la génération d'une donnée absolue. La valeur de l'expression doit être comprise entre - 32768 et 32767.
- avec une expression translatable, il s'agit alors d'une constante adresse utilisée dans une section de programme. Elle peut être indexée.

```
WORD ADRES1,X
WORD ADRES2
```

- lorsque la directive est étiquetée, le symbole est associé au premier mot généré.

Dans l'exemple suivant l'étiquette ETIQ sert à référencer le mot mémoire dont le contenu est 1 :

```
ETIQ : WORD 1 ; ADRES1
```

**Bull**  2 - DIRECTIVE BYTE

**Syntaxe**

[Etiquette]	BYTE	Expression absolue [; Expression absolue. ..]
-------------	------	---

**Fonction**

La directive BYTE permet de générer des données occupant un demi-mot mémoire (octet).

Elle peut avoir plusieurs opérandes, séparés par des points-virgules.

(E)

Les opérandes se présentent sous la forme d'expressions absolues dont les valeurs, calculées par l'assembleur, sont affectées à des octets consécutifs.

Les données générées sont cadrées à droite dans les octets.

L'assembleur calcule la valeur des expressions absolues et ne conserve que les 8 bits de poids faibles.

Lorsque le nombre d'opérandes d'une directive BYTE est impair, le dernier mot est complété par des zéros à droite.

**Exemples**

La directive :

```
BYTE '1A ; 'B ; 9
```

engendre les deux mots dont les contenus hexadécimaux sont :

```
'1A0B
```

```
'0900
```

La séquence :

```
BYTE '1A
```

```
BYTE 9
```

engendre les deux mots :

```
'1A00
```

```
'0900
```

**Remarque**

Lorsque la directive BYTE est étiquetée le symbole est associé au premier mot généré.

## 5.3 - DIRECTIVE FLOAT (E)

Syntaxe

[Etiquette]	FLOAT	Constante flottante
-------------	-------	---------------------

Fonction

La directive FLOAT permet de générer des données occupant deux mots mémoire, utilisés en tant que nombres en virgule flottante par les instructions d'arithmétique flottante.

L'opérande se présente sous la forme d'une constante comportant 3 parties :

- un signe (+ ou -),
- un nombre décimal (la partie entière est séparée de la partie décimale par un point),
- un exposant (la lettre E suivie d'un entier positif ou négatif représentant une puissance de 10).

Exemples

+ 31415926E - 7       $\longleftrightarrow$       31415926 x 10<sup>-7</sup>  
 -12.4E + 2       $\longleftrightarrow$       - 12.4 x 10<sup>2</sup>

Remarques

- un certain nombre de simplifications sont permises :
  - . le signe + peut être omis devant un nombre ou un exposant positif  
6.28E2
  - . l'exposant peut être absent  
16.281
  - . la partie entière ou la partie décimale peuvent être vides, mais non les deux  
.381E4  
132E - 3
- l'ensemble partie entière - partie décimale constitue une chaîne de 1 à 8 chiffres décimaux.
- les valeurs représentées sont, en valeur absolue, comprises entre 0.15 x 10<sup>-38</sup> et 1.7 x 10<sup>38</sup>.

## 5.4 - DIRECTIVE ASCII (E)

Syntaxe

[Etiquette]	ASCII	Chaîne de caractères
-------------	-------	----------------------

Fonction

Cette directive permet d'implanter en mémoire un libellé quelconque.

La chaîne de caractères ASCII figurant en zone opérande, encadrée par des doubles apostrophes, est rangée à raison de deux caractères par mot.

**Bull** exemple

```

ASCI "CECI EST UN TEXTE"

```

- Remarques**
- lorsque le nombre de caractères est impair, le dernier mot est complété à droite par un octet nul.
  - lorsque la directive est étiquetée, le symbole est associé au premier mot généré.

**5.5 - DIRECTIVE DZS****Syntaxe**

[Etiquette]	DZS	Expression absolue
-------------	-----	--------------------

**Fonction**

La directive DZS permet de définir une zone de travail à l'intérieur d'une séquence.

Elle a les deux effets suivants :

- réservation du nombre de mémoires égal à la valeur de l'expression absolue constituant l'opérande (en faisant progresser de cette valeur le compteur d'assemblage),
- remise à zéro de ces mémoires.

**Exemple**

```

ALPHA : WORD 5
BETA  : DZS 100
GAMMA : WORD TOT,X

```

Dans cet exemple, les mémoires étiquetées ALPHA et GAMMA sont séparées par 100 mémoires remises à zéro.

Il est possible d'étiqueter la directive DZS, le nom symbolique utilisé prenant alors comme valeur l'adresse de la première mémoire de la table (dans l'exemple l'étiquette BETA correspond à la première des 100 mémoires).

**Remarques**

- la directive DZS ne sert qu'à définir une zone de travail. Si l'on désire un tableau de constantes, on écrira par exemple :
 

```

NOMBRE : WORD 10 ; 100 ; 1000 ; 10000

```
- une table peut être de longueur nulle :
 

Ainsi :

```

AFIN : WORD 4
ADEB : EQU AFIN
ALPHA: DZS AFIN - ADEB

```

où AFIN et ADEB sont deux étiquettes de même valeur.

## 5.6 - DIRECTIVE DO (E)

## Syntaxe

[Etiquette]	DO	Expression absolue
-------------	----	--------------------

## Fonction

La directive DO est utilisée pour générer la ligne suivante autant de fois que la valeur de l'expression absolue figurant dans le champ opérande.

## Exemple

```
DO 5  
WORD - 1
```

## Remarques

- lorsque l'expression absolue est nulle, la phrase suivante est ignorée.
- la ligne à répéter ne doit pas être étiquetée. Par contre, si la directive DO est étiquetée, le symbole correspondant est associé à la première ligne générée.

Ainsi les deux écritures suivantes sont équivalentes :

```
SYMB : DO 3            ↔    SYMB : BYTE 1 ; 2 ; 3  
          BYTE 1 ; 2 ; 3                    BYTE 1 ; 2 ; 3  
                                              BYTE 1 ; 2 ; 3
```

## 6 - DIRECTIVES DE DEFINITION DE SYMBOLES

## 6.1 - DIRECTIVE EQU

Syntaxe	Symbole :	EQU	Expression translatable définie
---------	-----------	-----	------------------------------------

**Fonction** La directive EQU permet de définir le symbole figurant dans la zone étiquette en lui associant la valeur de l'expression translatable définie figurant dans la zone opérande.

Le symbole ainsi défini est translatable. Il appartient à la même section que l'expression translatable.

**Exemple** Dans l'exemple suivant :

```

LOCAL      L1
ALPHA:     WORD
-----
PROG       P1
-----
BETA:      EQU      $ + 1
GAMMA :    EQU      ALPHA + 1
-----

```

Les symboles translatables BETA et GAMMA appartiennent respectivement aux sections P1 et L1.

**Remarque** Il est impossible de redéfinir par équivalence un symbole étiquette déjà défini. Il est par contre possible d'avoir deux symboles étiquettes différents désignant la même mémoire ainsi que le montre l'exemple suivant :

```

MEM1:      WORD      4
EQUIV:     EQU       MEM1

```

**Bull** 2 - DIRECTIVE VAL

Syntaxe

Symbole :	VAL	Expression absolue
-----------	-----	--------------------

Fonction

La directive VAL permet de définir un symbole absolu. Le nom symbolique à gauche de VAL est rangé dans la table des symboles ainsi que la valeur (sur 16 bits) de l'expression absolue située dans la zone opérande.

Un symbole absolu ne peut être utilisé dans le programme qu'après avoir été défini.

Il peut prendre successivement des valeurs différentes au cours d'un même assemblage, chaque nouvelle valeur lui étant affectée par une directive VAL.

Exemple

```
V1:    VAL  10          < VALEUR  = 10
V2:    VAL  V1 + 2     < VALEUR  = 12
V2:    VAL  V2 + V1    < VALEUR  = 22
```

### 6.3 - CHARGEMENT DU COMPTEUR D'ASSEMBLAGE

La valeur initiale du compteur d'assemblage est zéro. L'assembleur le fait évoluer par la suite lorsqu'il rencontre soit une instruction, soit l'une des directives WORD, BYTE, FLOAT, ASCII ou DZS.

On peut d'autre part interrompre le déroulement normal en spécifiant l'adresse où doit se poursuivre l'assemblage.

On utilise pour cela la directive EQU avec le symbole \$ dans la zone étiquette et une expression translatable définie dans la zone opérande.

Exemple 1 :

La directive :

```
$ EQU $ + 10
```

a pour effet d'incrémenter le compteur d'assemblage de 10, sans aucune remise à zéro de mémoire, à la différence de la directive :

```
DZS 10
```

Elle est utilisée, en particulier, lorsque le contenu initial d'une zone mémoire est sans influence lors de l'exécution du programme.



**Exemple 2 :**

```
          LOCAL   L1
CSTE:    WORD    4
          -----
          PROG
          -----
$        EQU     CSTE
```

La directive EQU a pour effet un retour dans la section L1 en CSTE.

**Remarques :**

— la directive EQU ne peut en aucun cas être suivi d'une expression absolue.

L'écriture suivante est interdite :

```
$ EQU 12
```

— l'évolution du compteur d'assemblage doit être telle qu'il n'y ait pas recouvrement d'une partie du programme par une autre. On risque en effet de détruire une chaîne que devra remonter le chargeur.

7 - ASSEMBLAGE CONDITIONNEL : IF E

## Syntaxe

	IF	Expression absolue, [Symbole] , [Symbole] , [Symbole]
--	----	--

## Fonction

IF est une directive d'assemblage conditionnel. Elle permet d'assembler ou pas certaines parties de programme en fonction de la valeur affectée en début d'assemblage à certains symboles.

Elle comporte dans la zone opérande une expression absolue et trois symboles, séparés par des virgules.

## Exemple

IF VAL1 - VAL2, SYMB1, SYMB2, SYMB3

L'assemblage est interrompu jusqu'à la phrase comportant dans sa zone étiquette le premier, le second ou le troisième symbole suivant que l'expression absolue est respectivement négative, nulle ou positive.

Cette phrase est assemblée alors que toutes celles qui la séparent de la directive IF sont ignorées par l'assembleur.

## Remarques

— deux des symboles peuvent être égaux, ce qui permet de réaliser les six conditions :

< 0    = 0    > 0    ≤ 0    ≥ 0    ≠ 0

— il n'est pas indispensable d'étiqueter la phrase qui suit la directive IF.

En effet, à la condition que le nombre de virgules soit respecté, l'assembleur interprète une absence de symbole comme une demande de poursuite de l'assemblage.

```

IF      VALEUR, SYMB1,,SYMB2
DZS    50    < POURSUITE DE L'ASSEMBLAGE SI
          < VALEUR NULLE
SYMB1 : EQU  $-1 < REPRISE DE L'ASSEMBLAGE SI
          < VALEUR NEGATIVE
SYMB2 : VAL  '1FF < REPRISE DE L'ASSEMBLAGE SI
          < VALEUR POSITIVE

```

— la directive IF ne peut pas être étiquetée.

— la recherche d'un symbole est arrêtée par la rencontre de l'une des directives END ou EOT. Dans le dernier cas la recherche se poursuivra après la réactivation de l'assembleur.

— le symbole recherché après une directive IF peut être associé à l'une des directives EQU ou VAL.

## 8 - DIRECTIVES RELATIVES AUX PST : PSTS ET PSTH

### Syntaxe

	PSTS	Expression absolue
	PSTH	Expression absolue

### Fonction

Les directives PSTS et PSTH permettent l'identification des zones mémoire constituant respectivement les PST des tâches software et hardware.

Elles donnent lieu, lors du chargement du programme objet, à la mise à jour des tables de PST.

La valeur de l'expression absolue figurant dans la zone opérande représente le niveau de priorité de la tâche. Il doit être inférieur ou égal à 127 pour PSTS, à 15 pour PSTH.

### Exemple

```
PSTS    10      < PST DE LA TACHE SOFTWARE
          < DE NIVEAU 10
WORD    1       < VALEUR INITIALE DU REGISTRE A
WORD    < VALEUR INITIALE DU REGISTRE B
WORD    < VALEUR INITIALE DU REGISTRE X
WORD    11      < VALEUR INITIALE DU REGISTRE Y
WORD    BASEC  < VALEUR INITIALE DE LA BASE C
WORD    BASEL  < VALEUR INITIALE DE LA BASE L
WORD    < VALEUR INITIALE DE LA BASE W
WORD    KSTOR  < VALEUR INITIALE DU REGISTRE K
WORD    STAR   < VALEUR INITIALE DU REGISTRE P
WORD    '8000  < VALEUR INITIALE DU REGISTRE S
DZS 2      < VALEUR INITIALE DES REGISTRES SLO - SLE
```

### Remarques

- les directives PSTS et PSTH ne peuvent pas apparaître dans une DSEC
- elles ne peuvent pas être étiquetées.

9 . DIRECTIVES DE LIAISON INTERPROGRAMMES :  
EXT ET ENT

Syntaxe

	EXT	Symbole [, Symbole . . .]
	ENT	Symbole [, Symbole. . .]

Fonction

Un symbole peut être défini dans un programme et référencé dans un autre, assemblé (ou compilé) séparément. Les chainages sont effectués par l'éditeur de liens.

Dans le programme où le symbole est défini il faut le déclarer au moyen de la directive ENT.

De la même manière le programme se référant à un symbole externe doit le déclarer au moyen de la directive EXT. Il est alors identifié par l'assembleur comme référence externe.

La zone opérande de ces deux directives comporte un ou plusieurs symboles séparés par des virgules.

Exemple

```

ENT      RESUL
EXT      RAC2, N
COMMON
ARAC2:   WORD      RAC2
AN:      WORD      N
-----
TABLE
RESUL:   DZS       5
-----
PROG
-----
STA      &AN
BSR      ARAC2
-----

```

Programme 1



	LOCAL	
	ENT	RAC2, N
	EXT	RESUL
N:	WORD	
ARESUL:	WORD	RESUL, X
	-----	-----
	PROG	RACINE
	-----	-----
RAC2:	LA	N
	-----	-----
	STY	&ARESUL
	-----	-----
	STB	&ARESUL
	-----	-----
	RSR	

Programme 2

Les symboles doivent être déclarés par EXT ou ENT avant leur emploi dans le programme.

**Remarques**

- les directives EXT et ENT ne peuvent pas être étiquetées,
- les symboles déclarés par ENT sont des symboles translatables,
- une référence externe ne peut être utilisée que sous forme de constante adresse (indexée ou non mais toujours sans déplacement) associée à la directive WORD.

Ainsi la séquence :

	EXT	SYMB
	-	
	-	
	-	
CSTE :	WORD	SYMB + 3

provoquera l'émission d'un message d'erreur.

## 10 - DIRECTIVES DE SECTIONNEMENT ET D'IDENTIFICATION DES PROGRAMMES

### 10.1 - DIRECTIVE EOT

#### Syntaxe



#### Fonction

Lorsque le support utilisé pour le programme source est, par exemple, le ruban de papier, il est intéressant d'avoir le programme en plusieurs bandes. Les corrections sont ainsi facilitées.

Chaque bande intermédiaire doit se terminer par la directive EOT qui indique une fin de bande physique mais non la fin logique du programme. La dernière se termine par la directive END.

De manière plus générale la directive EOT permet de fractionner tout programme symbolique écrit en langage assembleur, avec utilisation éventuelle de supports différents pour les diverses fractions : cartes, rubans perforés, fichiers disque, . . .

La table des symboles est unique pour l'ensemble du programme. Elle n'est pas réinitialisée après une directive EOT.

#### Exemple

Bande 1		COMMON
		-----
		EOT
Bande 2		LOCAL
		-----
		PROG
		-----
		EOT
Bande 3		LOCAL
		-----
		PROG
		-----
		END

**Bull** Remarque

- lorsqu'il rencontre une directive EOT, l'assembleur commande l'impression du mot EOT (en fait la dernière phrase du programme source) et le superviseur reprend le contrôle.

L'utilisateur peut alors monter la suite du programme sur l'organe de lecture avant de demander la continuation de l'assemblage.

- la directive EOT ne peut pas être étiquetée.

## 10.2 - DIRECTIVE END

**Syntaxe**

	END	[Expression translatable définie]
--	-----	-----------------------------------

**Fonction**

La directive END indique la fin logique et physique du programme source.

Elle peut comporter un opérande interprété comme l'adresse de lancement du programme assemblé, après le chargement du programme objet en mémoire. Il s'agit d'une expression translatable, adresse d'un mot appartenant à une section de programme.

**Exemple**

	LOCAL	
	-----	
	PROG	
DEPART :	-----	
	-----	
	END	DEPART

**Remarques**

- la directive END termine l'assemblage. Elle donne lieu à deux actions, différentes suivant que le programme comporte ou non des références non définies :
  - . toutes les références sont définies  
L'assembleur commande l'impression du mot END (en fait la dernière phrase du programme source) et le superviseur reprend le contrôle.
  - . il existe des références non définies  
L'assembleur commande l'impression d'un message d'erreur.
- la directive END ne peut pas être étiquetée.

10.3 - DIRECTIVE IDP E

**Syntaxe**

	IDP	Chaine de caractères
--	-----	----------------------

**Fonction**

La directive IDP permet l'identification des programmes objets générés par l'assembleur.

La chaîne de caractères figurant dans la zone opérande est en effet imprimée lors du traitement du programme objet par le chargeur ou l'éditeur de liens.

**Exemple**

La séquence :

```
IDP    "PROGRAMME D'ESSAI"  
IDP    "30-5-72 DURAND"
```

provoquera, au moment du chargement du programme objet, l'impression du message :

```
PROGRAMME D'ESSAI  
30-5-72 DURAND
```



## 11 - DIRECTIVES RELATIVES A LA TABLE DES SYMBOLES

### 11.1 - DIRECTIVE EST E

Syntaxe

	EST	[Nom de section]
--	-----	------------------

Fonction

L'utilisateur peut demander l'édition :

- de tous les symboles, par la directive EST sans opérande,
- des symboles définis dans une section, par la directive EST suivie du nom de la section.

Les symboles sont édités par sections et dans les sections par ordre alphabétique sur le premier caractère.

Exemple

COMMON	C1	
-----		
PROG	P1	
-----		
EST	C1	< EDITION DES SYMBOLES DEFINIS DANS C1
EST		< EDITION DE TOUS LES SYMBOLES

Remarque

La directive EST ne peut pas être étiquetée.

### 11.2 - DIRECTIVE NDS

Syntaxe

	NDS	
--	-----	--

Fonction

La directive NDS provoque l'édition de tous les symboles encore non définis.

**Bull** 1.3 - DIRECTIVE DST

**Syntaxe**

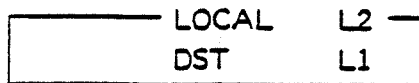
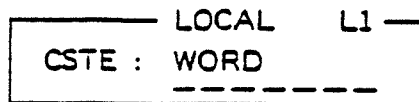
	DST	Nom de section
--	-----	----------------

**Fonction**

L'utilisateur peut remettre à zéro tous les symboles définis dans une section, par l'intermédiaire de la directive DST suivie du nom de la section.

**Exemple**

Lorsque la place dont on dispose dans la table des symboles est insuffisante pour qu'on y conserve jusqu'à la fin de l'assemblage tous les symboles d'un programme, on peut ainsi en éliminer ceux qui sont inutiles pour la suite de l'assemblage.



**Remarques**

- un symbole éliminé par une directive DST peut être à nouveau défini dans une autre section.
- la directive DST ne peut pas être étiquetée.

## 12 - LISTE D'ASSEMBLAGE

L'assembleur qui réalise l'assemblage en un seul passage permet l'édition d'une liste d'assemblage lors de cette lecture du programme symbolique. La présence de références en avant donne lieu à une liste incomplète.

L'assembleur peut d'autre part fournir une liste complète au cours d'une seconde lecture du programme source, les valeurs affectées aux symboles ayant été enregistrées au premier passage (ASM—E et ASM—D exclusivement).

L'édition est réalisée sur le périphérique associé à l'unité symbolique LO (List Output).

Chaque ligne de la liste d'assemblage comporte quatre zones :

- la valeur hexadécimale du compteur d'assemblage,
- la valeur hexadécimale du mot généré,
- dans le cas des sections de données communes et locales (directives COMMON et LOCAL) la valeur hexadécimale du déplacement qui correspond au positionnement implicite des bases C et L,
- la phrase du programme source.

Les caractères % et \* peuvent précéder la valeur générée. Ils ont les rôles suivants :

- % précise qu'il s'agit d'une constante adresse ; une translation peut être effectuée par l'éditeur de liens, le chargeur translateur et le chargeur disque.
- \* indique un chainage ; il s'agit soit d'une référence en avant, soit d'un symbole externe au programme.

**Remarque :**

La directive PAGE permet un saut à la page suivante de la liste d'assemblage. Elle n'est effective que si l'organe de sortie est une imprimante.



Exemple :

< SOUS-PROGRAMME SOUPRO  
< SAUVEGARDE DE LA BASE L  
< RESULTAT DANS X  
<  
< DONNEES LOCALES  
<

			LOCAL		
			EXT	SPROG	< REFERENCE EXTERNE
			ENT	CHKSUM	
	0010	LG :	VAL	16	
0000	% 8013 (80)	ABUF1 :	WORD	BUF1,X;BUF2,X	
1	% 8023 (81)				
	0001	ABUF2 :	EQU	\$ - 1	
2	00BB (82)	PVIRG :	WORD	";"	
3	* 7FFF (83)	ASPROG :	WORD	SPROG	
4	0000 (84)	CHKSUM :	WORD	O	

<  
< PROGRAMME  
<

			PROG	SOUPRO	
5	1A04	DEPART	PSR	L	< SAUVEGARDE
6	1D24		LRM	X, L	< INITIALISATIØNS
7	0010		WORD	LG	
8	0080		WORD	ABUF1+128	
9	B080	BOUCLE :	LA	&ABUF1	
A	9582		CP	PVIRG	
B	0203		JNE	\$ + 3	
C	1B20		PLR	L	< RESTITUTION
D	1E02		RSR		< SORTIE SOUPRO
E	A181		STBY	&ABUF2	
F	8984		AD	CHKSUM	< CALCUL CHECKSUM
0010	2D80		ADCR	A	< ADDITION REPORT
1	8D84		STA	CHKSUM	
2	19F7		JDX	BOUCLE	
3	8583		BR	ASPROG	< RUPTURE EN SPROG

<  
<  
< TABLES  
<

			TABLE	
	0013	BUF1 :	EQU	\$ - 1
4			DZS	LG
	0023	BUF2 :	EQU	\$ - 1
0024			DZS	10
			END	

## 13 - ORGANISATION DES PROGRAMMES

### 13.1 - INTRODUCTION

Ce chapitre, qui reprend un certain nombre de points détaillés dans les chapitres précédents, propose une organisation des programmes SOLAR 16 écrits en langage assembleur. Cette organisation constitue une synthèse et une justification des dispositifs hardware et des directives de l'assembleur suivantes :

- adressage basé, indirect et indirect post-indexé,
- adressage par rapport à P,
- instructions de rupture inconditionnelle et conditionnelle,
- registre K et instructions BSR-RSR, PSR-PLR, SVC-RSV,
- directives COMMON, LOCAL, PROG, TABLE,
- directives DZS, WORD, BYTE, FLOAT,
- directive DSEC.

Il ne traite pas des autres organisations possibles de programmes (utilisation différente de bases, permise par la directive USE) qui peuvent compléter ou remplacer la structure proposée.

### 13.2 - ORGANISATION MODULAIRE DES PROGRAMMES

L'organisation modulaire des programmes, grâce aux avantages qu'elle apporte dans la conception, le codage, la mise au point, la documentation, l'évolution des programmes, est maintenant très répandue.

Sur SOLAR 16 :

- l'utilisation de l'overlay, du "swap", de la relocation dynamique, de la notion de tâche,
- l'impossibilité pour des instructions codées sur 16 bits d'adresser directement l'ensemble de la mémoire,

constituent des raisons supplémentaires de l'utiliser.

#### 13.2.1 - Organisation modulaire des instructions

Pour organiser un programme d'une manière modulaire on répartit les instructions en sous-ensembles ayant un rôle précis dans le fonctionnement de ce programme.

**Bull** 3.2.2 - Organisation modulaire des données

Simultanément à la structuration des instructions il devient nécessaire de classer les données du programme en données locales, propres à chaque sous-ensemble d'instructions, et données communes, utilisées par l'ensemble du programme.

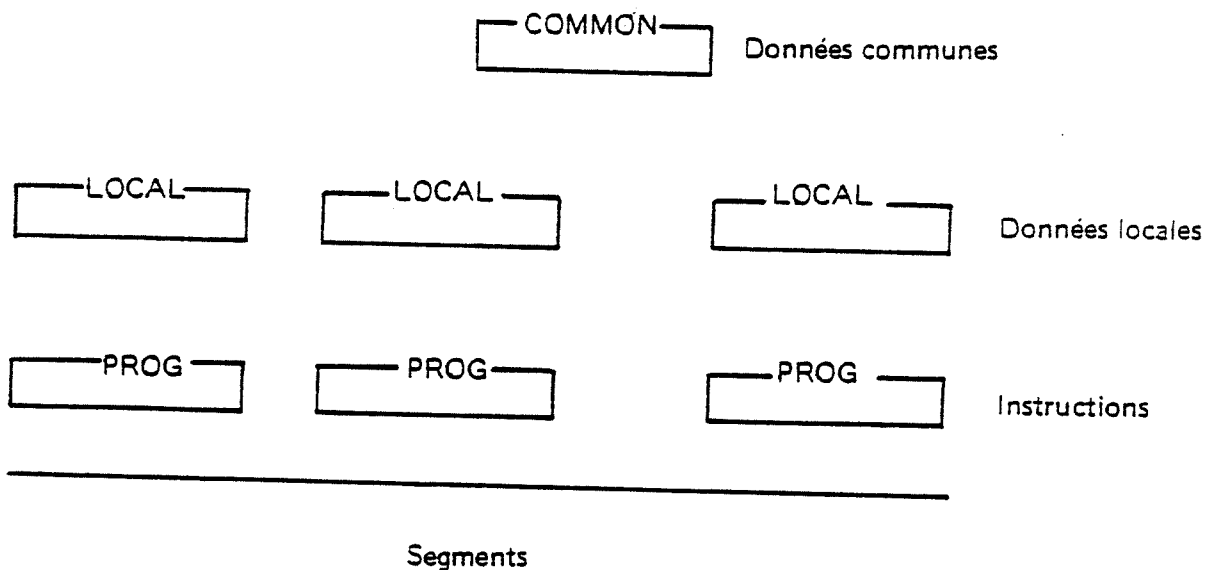
On voit que, dans une telle organisation, les modules sont caractérisés à la fois par les instructions qu'ils renferment et par les zones de données auxquelles ils ont accès (de même qu'un module hardware est caractérisé par les opérations qu'il peut effectuer et par les connections qui sont possibles entre lui-même et le reste d'un système).

### 13.3 - SEGMENTS ET ZONE DE DONNEES COMMUNES

Dans les programmes SOLAR 16 on distingue d'une part des segments, d'autre part une zone de données communes.

Chaque segment comporte des instructions et les données auxquelles les instructions du segment sont les seules à faire référence.

Les données auxquelles plusieurs segments font référence sont au contraire placées dans la zone de données communes.



### 13.4 - ADRESSAGE DES OPERANDES EN MEMOIRE

#### 13.4.1 - Utilisation des registres de base

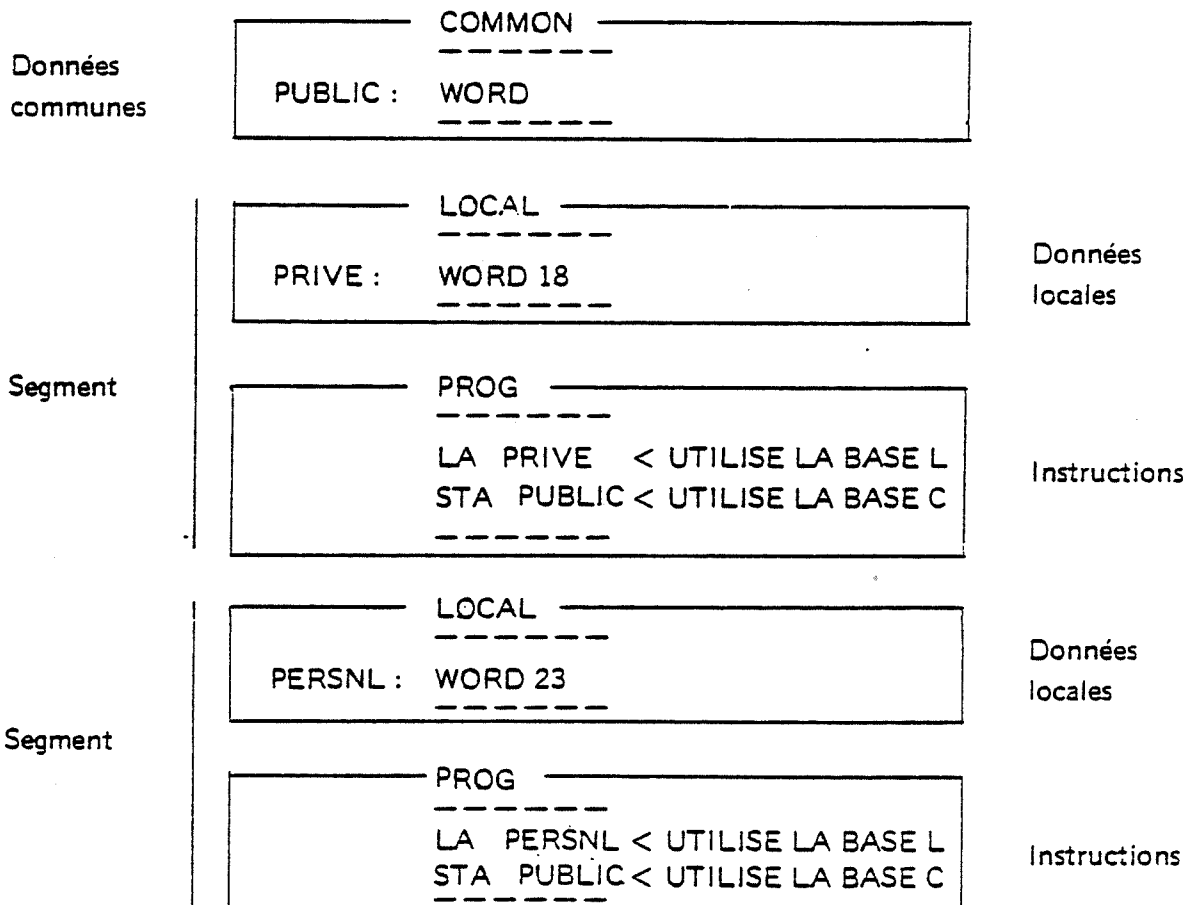
L'organisation des programmes décrite précédemment correspond à l'utilisation de registres de base dans le dispositif d'adressage SOLAR 16, cette organisation présentant de nombreux avantages (facilité de programmation - efficacité des programmes).

- Les instructions telles que les chargements, les rangements, les opérations arithmétiques et logiques, les comparaisons, utilisent un des registres de base quand elles adressent une mémoire. Un registre de base, une fois chargé par une valeur donnée, permet l'accès à 256 mots mémoire consécutifs.

Pour adresser une des zones de données, on utilise un de ces registres, après l'avoir chargé par la valeur appropriée. L'assembleur utilise la base C pour le COMMON du programme et la base L pour le LOCAL du segment en cours, chacune de ces zones comportant au maximum 256 mots. De cette manière il est possible à toutes les instructions du programme d'adresser la totalité du COMMON, et à toutes les instructions d'un segment d'adresser la totalité du LOCAL de ce segment.

Cette utilisation systématique des bases C et L, en définissant d'une manière claire et précise les zones de mémoire qui sont adressables par chaque segment, a l'avantage de faciliter l'écriture, la mise au point, la lecture des programmes.

Exemple :





## 13.4.2 - Initialisation des bases C et L

L'utilisation des bases C et L décrite ci-dessus a aussi l'avantage de simplifier la gestion de ces bases.

La base C doit être initialisée à chaque entrée dans le programme, sa valeur restant constante pendant tout le déroulement du programme.

Exemple :

	COMMON		
MEMOR :	EQU	S + 128	
	-----		

	PROG		
DEPART :	LRM	C	< INITIALISATION DE LA BASE C
	WORD	MEMOR	< CONSTANTE ADRESSE

La base L doit être initialisée par programme à chaque entrée dans un segment.

Il est préférable de définir par équivalence un symbole associé à la 128<sup>ème</sup> mémoire du LOCAL ou du COMMON (quelles que soient les longueurs de ces sections).

Exemple :

	LOCAL		
LOCAL :	EQU	S + 128	< ETIQUETTE UTILISEE DANS
	-----		< LA CONSTANTE ADRESSE

	PROG		
ENTREE :	LRM	L	< INTIALISATION DE LA BASE L
	WORD	LOCAL	< CONSTANTE ADRESSE

## 13.5 - ADRESSAGE DES TABLES

L'adressage et le code d'ordre SOLAR 16 permettent d'adresser les tables de données selon trois méthodes différentes décrites ci-dessous.

Remarque :

Dans ce qui suit on appelle table aussi bien une table initialisée à zéro à l'assemblage par une directive DZS qu'une table de constantes.



## 13.5.1 - Adressage indirect d'une table

On utilise un relais d'indirection qui contient l'adresse d'un des mots de la table et joue le rôle de pointeur. On peut faire évoluer ce pointeur en utilisant les instructions IC et/ou DC.

Exemple :

DEBTAB :	WORD	TABLO	< CONSTANTE ADRESSE
ATABLO :	WORD		< POINTEUR
TABLO :	DZS	50	< TABLE DE 50 MOTS
LY	DEBTAB		< INITIALISATION DU POINTEUR
STY	ATABLO		
LA	&ATABLO		< ACCES A LA TABLE PAR INDIRECTION
IC	ATABLO		< EVOLUTION DU POINTEUR

## 13.5.2 - Adressage indirect post-indexé d'une table

On utilise un relais d'indirection qui contient l'adresse du mot précédant le premier mot de la table, dans le cas d'un balayage par indice décroissant (ou suivant le dernier mot de la table dans le cas d'un balayage par indice croissant) et le bit 0 à 1 pour indiquer l'utilisation de l'adressage indexé.

Pour avoir accès à l'un des mots de la table on initialise l'index par la valeur correcte. On peut faire évoluer l'index à l'aide des instructions JIX (indice croissant), JDX (indice décroissant), ADRI, etc. . . .

Exemple :

ATABLO :	WORD	TABLO + 50,X	< CONSTANTE POINTANT SUR LE < PREMIER MOT APRES LA TABLE
TABLO :	DZS	50	< TABLE DE 50 MOTS
LXI	- 50		< INITIALISATION DE L'INDEX < (NEGATIF)
BOUCLE :	LA	&ATABLO	< ACCES A LA TABLE PAR ADRESSAGE < INDIRECT POST-INDEXE
JIX	BOUCLE		< EVOLUTION DE L'INDEX



### 13.5.3 - Adressage basé d'une table

On utilise une mémoire, qui contient l'adresse du premier mot de la table, pour charger le registre de base W. On peut ainsi adresser directement la table.

Cette méthode, en plus de sa rapidité (elle ne nécessite pas de cycle mémoire d'adressage indirect), est particulièrement intéressante car elle permet d'avoir accès à plusieurs mémoires de la table sans faire évoluer le registre de base : dans le cas où la table se subdivise en éléments de plusieurs mots, on a accès simultanément à tous les mots d'un élément, et on ne fait évoluer la base W (par une instruction ADRI) que lorsque l'on désire changer d'élément.

De plus, si la table contient des adresses de données, il est possible d'accéder à ces données par adressage indirect.

#### Exemple :

Dans cet exemple on suppose qu'il s'agit d'une table comportant des éléments de 3 mots, chaque élément correspondant à une mesure. Le premier mot donne l'adresse de la mémoire où se trouve la mesure, les deux mots suivant des seuils auxquels on veut comparer la mesure.

ATABLE :	WORD	TABMES	< CONSTANTE ADRESSE
TABMES :	EQU	§	< TABLE CONSTITUEE D'ELEMENTS < DE 3 MOTS
	WORD	M1	< PREMIER ELEMENT
	WORD	583	
	WORD	579	
	WORD	M2	< SECOND ELEMENT
	WORD	520	
	WORD	510	
	LY	ATABLE	< INITIALISATION DE LA BASE W
	LR	Y,W	
	LA	&0,W	< ACCES A LA MESURE
	CP	1,W	< COMPARAISON AU 1ER SEUIL
	CP	2,W	< COMPARAISON AU 2EME SEUIL
	ADRI	3,W	< PASSAGE A L'ELEMENT SUIVANT

Pour décrire la composition d'un élément de cette table on pourrait utiliser une DSEC, ce qui permettrait d'écrire les instructions d'une manière plus lisible :

ADMESU :	WORD		
SEUILH :	WORD		
SEUILB :	WORD		
-----			
ATABLE :	WORD	TABMES	< CONSTANCE ADRESSE
-----			
TABMES :	EQU	\$	< TABLE CONSTITUEE D'ELEMENTS < DE 3 MOTS
	WORD	M1	< PREMIER ELEMENT
	WORD	583	
	WORD	579	
	WORD	M2	< SECOND ELEMENT
	WORD	520	
	WORD	510	
-----			
	LY	ATABLE	< INITIALISATION DE LA BASE W
	LR	Y,W	
	USE	W,ADMESU	< ACCES A LA MESURE
-----			
	LA	&ADMESU	< ACCES A LA MESURE
	CP	SEUILH	< COMPARAISON AU 1ER SEUIL
-----			
	CP	SEUILB	< COMPARAISON AU 2EME SEUIL
-----			
	ADRI	3,W	< PASSAGE A L'ELEMENT SUIVANT
-----			

#### 13.5.4 - Réserve des tables

Le système d'adressage SOLAR 16, comme le montre d'ailleurs les 3 méthodes ci-dessus, nécessite dans tous les cas l'utilisation d'une constante adresse pour l'adressage d'une table.

Pour avoir accès à une table il faut donc placer la constante adresse correspondante soit dans le COMMON, soit dans un LOCAL.

Par contre, il n'est pas nécessaire de placer la table elle-même dans le COMMON ou dans un LOCAL : la directive TABLE de l'assembleur permet la définition de sections réservées aux tables. Ces sections comportent aussi bien les tables de constantes (directives WORD, BYTE, etc. . . ) que les tables de travail (directive DZS).



Exemple :

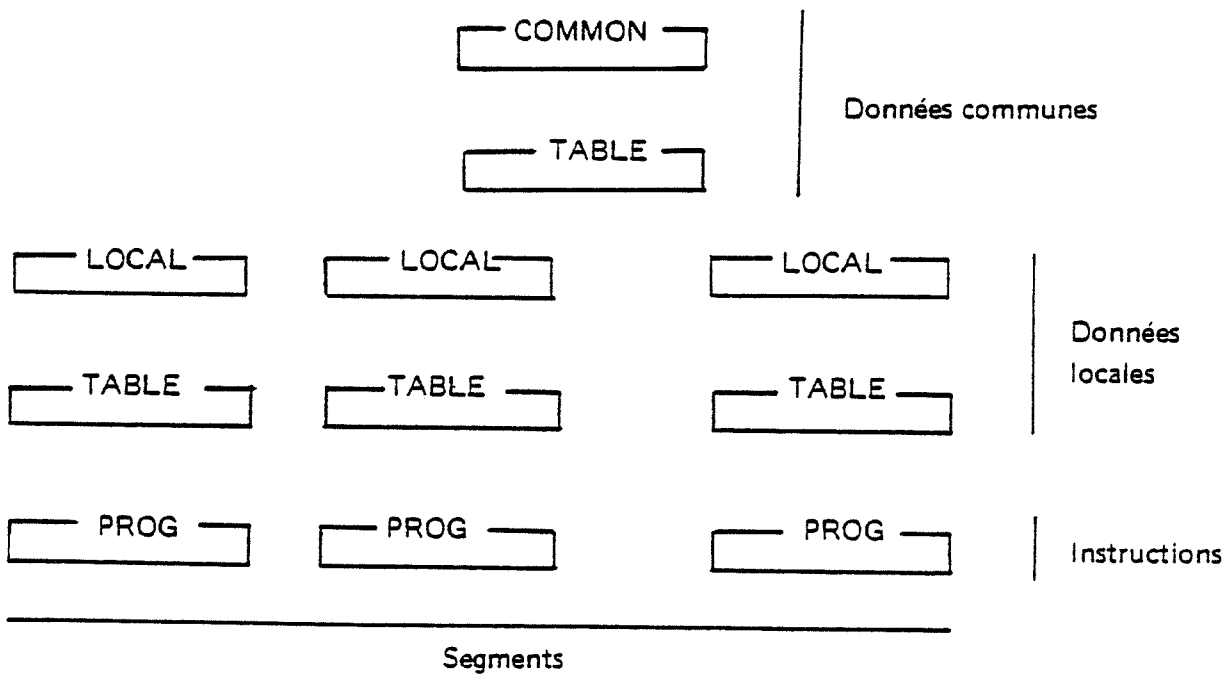
	LOCAL		
ACT :	WORD	TC, X	< CONSTANTES ADRESSES PERMETTANT < L'ACCES
	WORD	TT, X	< AUX TABLES TC ET TT

	TABLE		
TC :	WORD	1	< TABLE TC (TABLE DE CONSTANTES)
	WORD	3	
	WORD	5	
	WORD	7	
TT :	DZS	50	< TABLE TT (TABLE DE TRAVAIL)

### 13.5.5 - Tables communes et tables locales

La distinction entre les tables de données communes qui doivent être adressables par tous les segments et les tables de données locales qui ne sont adressées que dans un segment, se traduit dans le programme par le fait que la constante adresse qui permet de les adresser est placée soit dans le COMMON, soit dans le LOCAL.

On peut compléter le schéma du paragraphe 13.3 de la manière suivante :



### 13.6 - ADRESSAGE LORS DES RUPTURES DE SEQUENCE

On a étudié dans les chapitres précédents l'adressage des opérandes mémoires pour les instructions arithmétiques et logiques, les comparaisons, les chargements et les rangements, etc. . . . Le présent chapitre expose l'utilisation de l'adressage dans les instructions de rupture inconditionnelle (BR, JMP) et conditionnelle (JL, JAL, etc. . . ).

#### 13.6.1 - Ruptures de séquence entre segments

Pour les ruptures de séquence d'un segment à un autre on utilise l'instruction BR et une constante adresse. Pour éviter la duplication de cette constante adresse dans des LOCAL différents on aura en général intérêt à la placer dans le COMMON. L'utilisation de JMP pour des ruptures de séquence entre segments, d'ailleurs le plus souvent impossible, est à déconseiller à cause des contraintes qu'elle imposerait à l'implantation des segments en mémoire. Elle n'est pas acceptée par l'assembleur.

Exemple :

	COMMON		
ATOTO:	WORD	TOTO	< CONSTANTE ADRESSE
	PROG		
TOTO	EQU	\$	< BUT DE LA RUPTURE
	PROG		
	BR	ATOTO	< INSTRUCTION DE RUPTURE

#### 13.6.2 - Ruptures de séquence à l'intérieur d'un même segment

Pour les ruptures de séquence conditionnelles ou inconditionnelles à l'intérieur d'un même segment on utilise de préférence les instructions du type "jump". L'adressage par rapport à P qu'elles utilisent permet d'avoir accès directement aux 128 instructions qui précèdent et aux 127 qui suivent.

Dans le cas de portée insuffisante du "jump", la rupture se fera à l'aide d'une instruction BR et d'une constante adresse placée dans le LOCAL du segment ; si ce "jump" est conditionnel, on fera précéder le BR d'un "jump" \$ + 2 sur la condition inverse.



Exemple :

ATOTO :	LOCAL	
	WORD	TOTO
	-----	-----

	PROG		
TOTO	EQU	\$	
	-----	-----	
	JMP	TOTO	< JMP POSSIBLE
	-----	-----	
	JC	TOTO	< JC POSSIBLE
	-----	-----	
	BR	ATOTO	< JMP IMPOSSIBLE
	-----	-----	
	JNC	S + 2	< JC IMPOSSIBLE
	BR	ATOTO	
	-----	-----	

### 13.7 - SOUS-PROGRAMMES

#### 13.7.1 - Zone de rangement commune

La facilité d'écriture des sous-programmes sur SOLAR 16 repose sur l'utilisation de la zone de rangement commune qui est pointée en permanence par le registre K. Cette zone, partagée entre les différents segments, est réservée dans un programme sous la forme d'une table rattachée aux données communes.

Cette zone est utilisée :

- par l'instruction d'appel de sous-programme BSR pour ranger l'adresse de retour, et par l'instruction de fin de sous-programme RSR pour retrouver cette adresse,
- par l'instruction de rangement multiple de registres PSR pour sauvegarder le contenu de un ou plusieurs registres, et par l'instruction de chargement multiple PLR pour restaurer le contenu de ces registres.

Au lancement du programme le registre K doit être initialisé pour pointer immédiatement avant le premier mot de la zone. Les instructions y stockent des informations (BSR, PSR et SVC), effectuent simultanément l'incrément de K, et celles qui y reprennent ces informations (RSR, PLR et RSV) effectuent simultanément la décrémentation de K. Ce fonctionnement permet de les utiliser dans le cas de sous-programmes imbriqués. Dans ce cas le nombre de mémoires réservées pour cette zone doit correspondre au nombre de mémoires utilisées par la chaîne d'appels imbriqués la plus longue ou par celle qui utilise le plus de mémoires.

### 13.7.2 - Sous-programmes communs

On appelle sous-programme commun un sous-programme utilisé par plusieurs segments.

La constante adresse permettant les BSR vers un sous-programme commun est placée dans le COMMON.

Un sous-programme commun a son propre LOCAL et constitue un segment. Il a obligatoirement à sauvegarder et restaurer la base L, et éventuellement la base C s'il l'utilise comme registre ou base de travail (dans ce cas si ce sous-programme en appelle un autre on doit aussi restaurer la base C avant cet appel).

Exemple :

	COMMON		
	-----	-----	
ASP :	WORD	SP	< CONSTANTE ADRESSE
	-----	-----	

	PROG		
	-----	-----	
	BSR	ASP	< APPEL DU SOUS-PROGRAMME
	-----	-----	

	PROG		
	-----	-----	
SP :	PSR	L	< POINT D'ENTREE - SAUVEGARDE DE L
	-----	-----	
	PLR	L	< RESTAURATION DE L
	RSR		< RETOUR A L'APPELANT

Un même LOCAL peut être partagé par plusieurs sous-programmes communs, ces sous-programmes constituant ensemble un segment.

### 13.7.3 - Transmission des paramètres : méthode générale

La méthode la plus générale de transmission des paramètres à un sous-programme utilise une "table de paramètres", c'est-à-dire une suite de mémoires consécutives contenant des valeurs ou des adresses. Cette table se trouve (indifféremment) dans un LOCAL ou dans le COMMON ; c'est l'adresse de cette table que l'on transmet au sous-programme. L'appelant charge cette adresse dans A en utilisant l'instruction LAD ; le sous-programme transfère cette adresse dans la base W et peut ainsi adresser directement la table de paramètres.



Exemple :

COMMON			
SOMME :	WORD	SPSOM	< CONSTANTE ADRESSE POUR BSR

LOCAL			
PARAMS :	WORD	1000	< PARAMETRES : LE 1ER EST UNE < VALEUR
	WORD	TOTO	< LE 2EME EST UNE ADRESSE

PROG			
LAD		PARAMS	< SEQUENCE D'APPEL
BSR		SOMME	

PROG			
SPSOM :	EQU	\$	< ENTREE DU SOUS-PROGRAMME
	LR	A,W	< TRANSFERT DE L'ADRESSE PARAMS < DANS W
	LA	O,W	< ACCES AU 1ER PARAMETRE
	AD	&1,W	< ACCES AU 2EME PARAMETRE

En utilisant une DSEC, on peut adresser les paramètres dans le sous-programme d'une manière plus lisible :

DSEC			
P1 :	WORD		< DSEC DECRIVANT
P2 :	WORD		< LA TABLE DE PARAMETRES

PROG			
LR		A,W	< TRANSFERT DE L'ADRESSE PARAMS < DANS W
USE		W,P1	
LA		P1	< ACCES AU 1ER PARAMETRE
AD		&P2	< ACCES AU 2EME PARAMETRE



### 13.7.4 : Transmission des paramètres : méthode utilisant les registres

Quand le nombre de paramètres est peu élevé, on peut avoir intérêt à transmettre les paramètres directement dans les registres tels que A, B, X, Y, W.

Il est évident que cette méthode est la meilleure quand il n'y a qu'un seul paramètre.

Exemple :

```

-----
LA      ANGLE
BSR     SINUS
-----

```

### 13.7.5 - Transmission des paramètres : paramètres placés derrière BSR

Dans le cas où il est utile de placer les paramètres derrière l'instruction BSR on peut utiliser la méthode ci-dessous. Cette méthode n'est valable que si les paramètres placés derrière le BSR sont des valeurs ou des adresses qui ne sont pas modifiées pendant le déroulement du programme.

Exemple :

```

-----
BSR     SOMME      < APPEL SOUS-PROGRAMME
WORD    1000       < 1ER PARAMETRE
WORD    TOTO       < 2EME PARAMETRE
-----
SPSOM :
-----
PLR     W          < W = ADRESSE DU 1ER PARAMETRE
LA      O,W       < ACCES AU 1ER PARAMETRE
AD      &1,W      < ACCES AU 2EME PARAMETRE
-----
ADRI    2,W       < ADRESSE DE RETOUR
PSR     W
RSR                    < RETOUR

```

### 13.7.6 - Utilisation de la zone de rangement pour des opérandes (sous-programmes réentrants)

En général un sous-programme utilise un certain nombre de mémoires de travail qui jouent un rôle temporaire, c'est-à-dire qui ne conservent aucune information entre un appel et l'appel suivant du sous-programme.

D'une manière analogue à l'utilisation de la zone de rangement commune pour la sauvegarde du point de retour et la sauvegarde des registres, on peut utiliser cette même zone pour les mémoires de travail en procédant de la manière suivante :

- a) On transfère le registre K dans une base de manière à pouvoir adresser les mémoires ainsi réservées (on utilise W ou C, L étant utilisée pour adresser les constantes du sous-programme).



A l'entrée dans le sous-programme on "réserve" les n mémoires utilisées par le sous-programme en faisant progresser de n le registre K.

c) Avant de sortir, on libère les n mémoires utilisées en diminuant de n le registre K.

Une telle organisation des sous-programmes, utilisable avec des appels de sous-programmes imbriqués, offre les avantages suivants :

- elle permet une diminution du nombre de mémoires de travail pour l'ensemble du programme grâce à l'utilisation commune de la zone de rangement ;
- si toutes les mémoires de travail sont réservées dans la zone commune, le sous-programme est réentrant.

Exemple :

```

-----
ENTREE :  PSR      L      < SAUVEGARDE DE L
          LR       K, W   < W EST UTILISE COMME BASE
          ADRI     3,K    < RESERVATION DE 3 MEMOIRES DE TRAVAIL
-----
          LA      1,W    < ADRESSAGE 1ERE MEMOIRE
-----
          LA      2,W    < ADRESSAGE 2EME MEMOIRE
-----
          LA      3,W    < ADRESSAGE 3EME MEMOIRE
-----
          ADRI    -3,K   < LIBERATION DES MEMOIRES DE TRAVAIL
          PLR     L      < RESTAURATION DE L
          RSR                      < RETOUR
    
```

En utilisant une DSEC, on peut adresser les mémoires de travail d'une manière plus lisible :

	DSEC	
MEM1 :	WORD	< DSEC DECRIVANT
MEM2 :	WORD	< LES MEMOIRES UTILISEES
MEM3 :	WORD	

	-----	
	USE	W, MEM1-1
	-----	
	LA	MEM1 < ADRESSAGE 1ERE MEMOIRE
	-----	
	LA	MEM2 < ADRESSAGE 2EME MEMOIRE
	-----	
	LA	MEM3 < ADRESSAGE 3EME MEMOIRE
	-----	

## 14 • UTILISATION DE L'ASSEMBLEUR

## 14.1 - INTRODUCTION

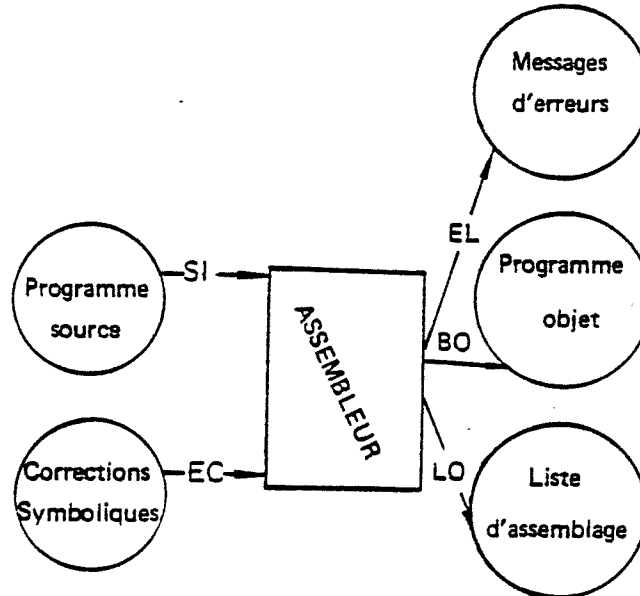
Les assembleurs ASM et ASM-E s'exécutent sous le contrôle des systèmes d'exploitation BOS-A, BOS-B et BOS-C.

L'assembleur ASM-D s'exécute sous le contrôle de BOS-D et BACKM.

Ils utilisent un certain nombre d'unités symboliques pour leurs demandes d'opérations d'entrée-sortie.

Ce sont les suivantes :

SI (Symbolic Input)	Entrée du programme source
BO (Binary Output)	Sortie du programme objet
LO (List Output)	Sortie de la liste d'assemblage
EL (Error Listing)	Sortie des messages d'erreurs
EC (Error Correction)	Entrées des corrections symboliques.



Avant d'activer l'assembleur, l'utilisateur peut associer à chaque unité symbolique une unité fonctionnelle (ou un fichier disque sous les systèmes d'exploitation disque) dont il dispose dans sa configuration.

Les commandes d'association sont décrites dans les manuels de référence des différents systèmes.

**Bull**  remarques :

- l'édition des symboles définis dans le programme, qui est demandée par la directive EST, est réalisée sur l'unité symbolique LO.
- l'édition des symboles non définis, qui est demandée par la directive NDS, est réalisée sur l'unité symbolique EL.

## 14.2 - ACTIVATION DE L'ASSEMBLEUR

L'assembleur possède trois points d'entrée. A chacun d'eux correspond une commande qui est reconnue par le superviseur.

### 14.2.1 - Commande IASM

La commande IASM donne le contrôle à l'assembleur ASM qui, après s'être réinitialisé (remise à zéro de tous les symboles — translatables et absolus —, de tous les noms de section, etc...) lance l'assemblage à partir de l'unité symbolique SI.

Le programme objet est produit sur l'unité symbolique BO et la liste d'assemblage (incomplète) sur l'unité symbolique LO:

Lorsque l'assembleur trouve l'une des deux directives EOT et END, il rend le contrôle au superviseur.

#### Exemple :

* SI	CR	Affectation à SI du lecteur de carte
* BO	HP	Affectation à BO du perforateur rapide
* LO	LP	Affectation à LO de l'imprimante rapide
* IASM		Activation de l'assembleur avec demande de réinitialisation
EOT		Retour après la directive EOT
*		sous le contrôle du superviseur.

Une liste d'assemblage complète peut être obtenue dans un second passage. L'association à l'unité symbolique LO de ZE permet alors la suppression de la liste incomplète pour toute la durée du premier passage.

Exemple :

```
* SI      HR
* BO      HP
* LO      ZE
* IASM
END      Retour après la directive END
*      sous le contrôle du superviseur
```

#### 14.2.2 - Commande LASM E

La commande LASM donne le contrôle à l'assembleur ASM pour le second passage donnant lieu à une liste d'assemblage complète.

On peut ainsi compléter l'exemple précédent :

```
* SI      HR
* BO      HP
* LO      ZE      Suppression de la liste incomplète
* IASM
END

* LO      LP      Affectation à LO de l'imprimante rapide
* LASM
END
*
```

#### 14.2.3 - Commande CASM

Tout programme symbolique écrit en langage assembleur peut être fractionné par l'intermédiaire de la directive EOT.

- a) Lorsque le support utilisé est le ruban papier, la directive EOT permet d'avoir un programme symbolique en plusieurs bandes (voir chapitre 10).

En cours d'assemblage (au premier ou au second passage) lorsque cette directive est reconnue par l'assembleur le contrôle est donné au superviseur. L'utilisateur peut ainsi placer la suite du programme sur l'unité de lecture, avant de continuer l'assemblage.

La commande CASM permet alors de réactiver l'assembleur.

**Bull**

Exemple :

```

-----
* LO   ZE
* IASM
EOT    Premier passage
* CASM
END
* LO   LP
* LASM
EOT    Second passage
* CASM
END
*
    
```

- b) Lorsque la directive END a été reconnue et acceptée par l'assembleur il est encore possible de compléter un programme assemblé.

La commande CASM permet alors de redonner le contrôle à l'assembleur sans réinitialisation.

Ce point sera repris dans le paragraphe 14.5.

### 14.3 - CORRECTION DES ERREURS

Lorsqu'une phrase du programme source comporte une erreur, l'assembleur imprime sur l'unité symbolique EL un message comportant :

- le repérage symbolique de cette phrase dans le programme,
- le numéro de l'erreur (la liste des numéros d'erreurs est donnée en Annexe),
- la phrase erronée.

Il y a alors commutation sur l'unité symbolique EC.

La correction apportée doit comporter la phrase correcte (éventuellement plusieurs, ou aucune) et se terminer par **#** **CR** qui déclenche la poursuite de l'assemblage sur l'unité symbolique SI.

Exemple :

```

SYMBOL + 12      Repérage symbolique
ERA  05         Numéro de l'erreur
LA  ADRES      Phrase erronée
ADRES : EQU  ADDRESS |
LA  ADRES      Correction
#              Fin de correction
    
```

- a) Les directives WORD et BYTE peuvent comporter plusieurs opérandes.

Lorsque l'un d'eux est rejeté par l'assembleur, les règles suivantes sont adoptées :

- avec WORD tous les opérandes qui précèdent l'erreur sont assemblés.

Phrase source : ETIQ:WORD 1;2;AO;4

Message d'erreur : ETIQ + 2  
ERA 01  
ETIQ : WORD 1;2;AO;4  
accepté rejeté

Correction : WORD 'AO;4  
#

- avec BYTE l'opérande qui précède l'erreur est refusé s'il appartient au même mot.

Phrase source : ETIQ1:BYTE 1;2;3;X;5

Message d'erreur : ETIQ1 + 03  
ERA 01  
ETIQ1 : BYTE 1;2;3;X;5  
accepté rejeté

Correction : BYTE 3;'F;5  
#

- l'étiquette éventuelle est introduite dans la table des symboles dès que le premier opérande est assemblé.

Phrase source : ETIQ2 : BYTE 1;'G;3

Message d'erreur : ETIQ2 + 1  
ERA 01  
ETIQ2 : BYTE 1;'G;3

Correction : BYTE 1;'G;3

- b) Une directive de sectionnement peut provoquer une erreur de numéro 15 lorsqu'elle clot une section PROG considérée comme étant incomplète.

Deux cas peuvent se produire :

- il existe des instructions de saut en avant hors des limites de la section.

Programme source :  
-----  
                  PROG  
                  -----  
ETIQ : JMP      S + 1  
                  LOCAL

Message d'erreur : ETIQ + 01  
ERA 15  
                  LOCAL      (non assemblé)

**Bull** il existe des instructions de saut en avant non résolues

Programme source :

-----  
PROG  
-----

JMP            AVANT  
-----

JMP            AVANT1 - 3  
-----

ETIQ1 : LA            ADRES  
         LOCAL

Message d'erreur : AVANT  
                      AVANT1  
                      ETIQ1 + 01  
                      ERA 15  
                      LOCAL

Dans ce dernier cas le message d'erreurs comporte la liste des symboles qui n'ont pas été définis dans la section PROG.

#### Remarque :

En début d'assemblage avant la lecture de la première étiquette, le repérage symbolique ne comporte qu'un déplacement.

#### 14.4 - PASSAGE EN ASSEMBLAGE CLAVIER

Lorsque l'assembleur effectue la lecture du programme source sur l'unité symbolique SI, la détection d'une phrase commençant par le caractère  $\#$  a pour effet :

- l'impression de la phrase sur l'unité symbolique EL,
- la commutation de la lecture sur l'unité symbolique EC.

Ceci permet en particulier une écriture aisée de programmes généraux pouvant être adaptés à une utilisation particulière.

Avec un programme source comportant la séquence suivante :

```
 $\#$  DEFINIR NPERIF = NOMBRE DE PERIPHERIQUES  
IF NPERIF ,, SUITE  
 $\#$  NPERIF INCORRECT (NEGATIF OU NUL)  
SUITE: EQU $
```



le dialogue peut être le suivant :

## DEFINIR NPERIF = NOMBRE DE PERIPHERIQUES

NPERIF: VAL 0

##

## NPERIF INCORRECT (NEGATIF OU NUL)

(les phrases émises par l'assembleur sont soulignées).

Dans cet exemple l'utilisateur définit en cours d'assemblage le paramètre NPERIF. La valeur qui lui est associée est contrôlée par le programme source. Celui-ci demande l'impression d'un message d'erreur dans le cas où la valeur n'est pas conforme à ce qu'il attend.

#### 14.5 - CORRECTION D'UN PROGRAMME PAR SURCHARGE

Lorsque la directive END a été reconnue et acceptée par l'assembleur, il est encore possible de modifier un programme assemblé.

La commande CASM permet de redonner le contrôle à l'assembleur sans réinitialisation. Le second programme objet généré viendra, au moment de son chargement en mémoire, surcharger le premier.

Si, par exemple, le programme principal comporte la séquence suivante :

```
-----  
ETIQ:  LA   OPER  
-----  
ETIQ1: LR   Y,C  
       STA  OPER1  
-----
```

un programme "correction" peut être du type :

```
$ EQU  ETIQ  
  LA   OPER + 4  
$ EQU  ETIQ+ 1  
  STA  OPER1 - 3  
  END
```

L'adresse de chargement des deux programmes objets devra être identique.



#### 14.6 - INTERRUPTION D'UN ASSEMBLAGE EN COURS

L'opérateur dispose sur le périphérique de dialogue du bouton BREAK permettant l'appel du superviseur qui peut être utilisé pour interrompre un assemblage en cours.

Dans ce cas l'assembleur rend le contrôle au superviseur.

Il pourra d'ailleurs être réactivé ultérieurement car le test des appels superviseur est fait en des points où l'assembleur est interruptible.

#### 14.7 - UTILISATION DES ASSEMBLEURS ASM et ASM-E

Les assembleurs ASM et ASM-E s'exécutent sous le contrôle des superviseurs BOS-A, BOS-B et BOS-C.

Exploités par l'un des deux premiers systèmes ils se présentent sous la forme de bandes binaires translatables. La suite de ce paragraphe ne concerne donc que BOS A et BOS B.

La séquence de programme qui est activée lors du lancement automatique par le chargeur se termine par une requête SVC CAMO transmettant au système les commandes introduites en 14.2.

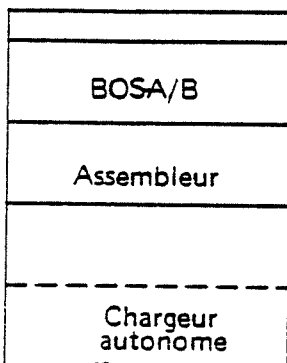
Il est donc nécessaire, au moment du chargement des assembleurs, que le superviseur soit déjà en mémoire.

D'autre part, la zone allouée à la table des symboles est déterminée par le contenu de la mémoire FIRSTM ('B) au moment de l'activation par la commande IASM.

Cette mémoire, gérée par le chargeur traducteur, pointe sur la première mémoire libre, ce qui interdit à la table de venir recouvrir un processeur chargé postérieurement.

##### 14.7.1 - Chargement par l'intermédiaire du chargeur autonome

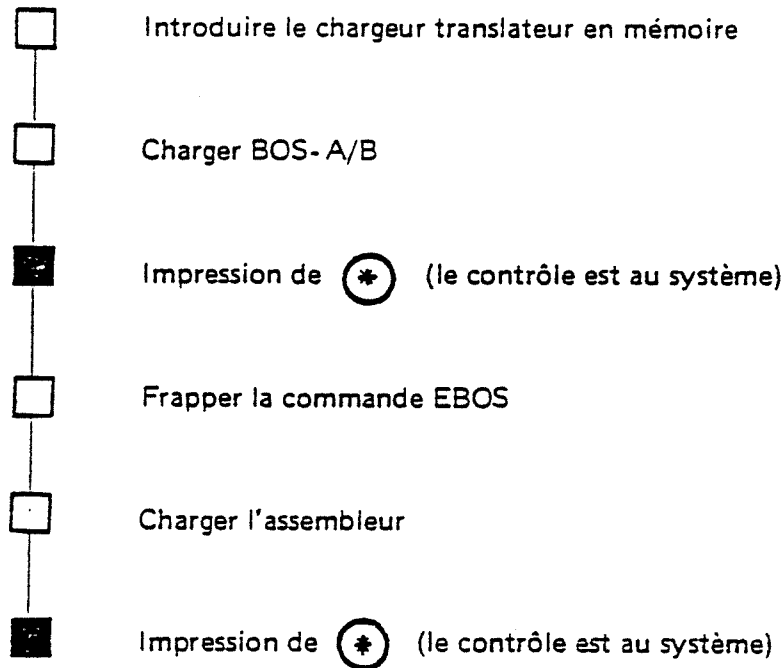
Une implantation peut être la suivante :



Place disponible pour la table des symboles  
qui est gérée par l'assembleur



La suite des opérations à effectuer est :



Le système est alors prêt à accepter les commandes d'activation de l'assembleur.

#### 14.7.2 - Chargement par l'intermédiaire du chargeur sous système

Lorsque le système ainsi que le chargeur traducteur sous système sont en mémoire, l'implantation est la suivante :

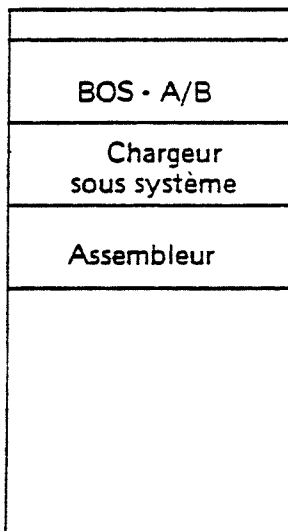
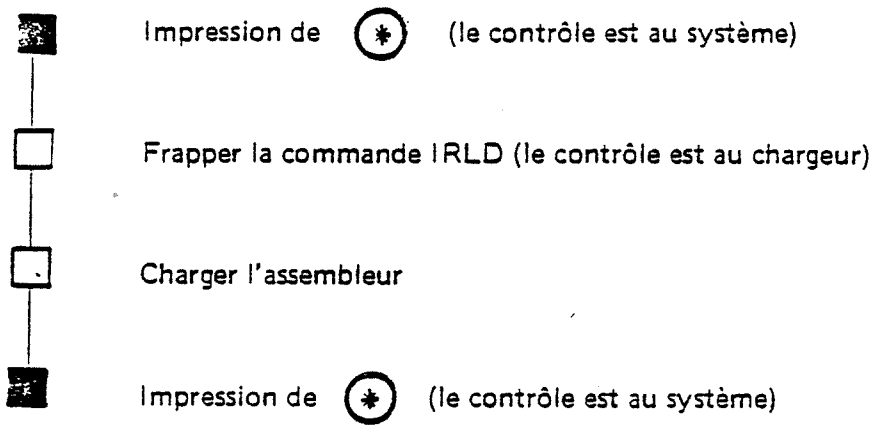


Table des symboles

**Bull** La suite des opérations à effectuer devient :



### 14.7.3 - Particularité concernant l'assembleur ASM-E

L'assemblage de la directive `FLOAT` permettant la génération des nombres en virgule flottante nécessite :

- soit l'opérateur flottant câblé,
- soit la présence en mémoire de l'arithmétique flottante programmée.

L'implantation mémoire devient donc la suivante :

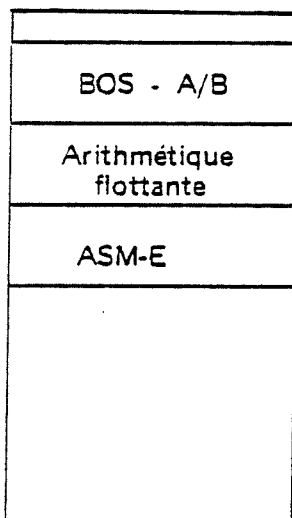


Table des symboles

#### 14.8 - UTILISATION DE L'ASSEMBLEUR ASM-D

L'assembleur ASM-D s'exécute sous le contrôle de BOS-D ou BACKM, moniteur Background sous RTES-C/D.

Ces deux systèmes permettent deux modes de fonctionnement : conversationnel ou traitement par lots (BATCH).

Dans le mode conversationnel, l'utilisation des unités symboliques, les corrections des erreurs se déroulent comme cela est décrit précédemment.

Dans le mode train de travaux toute erreur détectée par ASM-D est imprimée mais l'assemblage se poursuit à partir de l'unité symbolique SI (aucune commutation sur EC) ; l'assemblage s'interrompt lorsqu'est rencontrée la directive EOT ou END, avec impression éventuelle de la liste des symboles non définis dans le second cas.

De même qu'avec l'assembleur ASM-E, l'assemblage de la directive FLOAT nécessite :

- soit l'opérateur flottant câblé,
- soit la présence en mémoire de l'arithmétique flottante programmée (l'alimentation est décrite dans le manuel de référence de BOS-D).

## 15 - ANNEXES

## 15.1 - LISTE DES DIRECTIVES

	COMMON	[Nom de section]
	LOCAL	[Nom de section]
	TABLE	[Nom de section]
	PROG	[Nom de section]
	DSEC	[Nom de section]
[Etiquette]	WORD	Expression [; Expression. . .] avec : Expression ::= Expression absolue   Expression translatable [,X]
[Etiquette]	BYTE	Expression absolue [;Expression absolue...]
[Etiquette]	FLOAT (E)	Constante flottante
[Etiquette]	ASCII (E)	Chaine de caractères
[Etiquette]	DZS	Expression absolue
Symbole :	EQU	Expression translatable définie
Symbole :	VAL	Expression absolue
	USE	Base [,Expression translatable définie]
	IF (E)	Expression absolue, [Symbole] , [Symbole], [Symbole]
[Etiquette]	DO (E)	Expression absolue
	PSTS	Expression absolue
	PSTH	Expression absolue
	EXT	Symbole [, Symbole. . .]

Bull 

	ENT		Symbole [,Symbole. . .]
	EOT		
	END		[Expression translatable définie]
	IDP	ⓔ	Chaine de caractères
	EST	ⓔ	[Nom de section]
	NDS		
	DST		Nom de section

ⓔ

Directive non admise par l'assembleur ASM

## 15.2 - LISTE DES NUMEROS D'ERREUR

- ERA 00 Erreur de parité
- ERA 01 Erreur d'écriture
- . Erreur dans l'écriture d'un nombre, d'une constante flottante ou d'une chaîne de caractères
  - . Symbole comportant plus de 6 caractères
  - . Utilisation d'un caractère non reconnu par l'assembleur
  - . Pas de code instruction ou de directive dans la phrase
  - . Erreur de syntaxe
- 
- ERA 02 Expression incorrecte
- . Utilisation d'un nom de section interdit
  - . Valeur non représentable sur 16 bits
  - . Position de l'élément translatable incorrect (dans une expression translatable)
  - . Multiplicande ou multiplicateur non absolu
  - . Dividende ou diviseur non absolu
  - . Type de l'expression non reconnu
- 
- ERA 03 Symbole interdit
- . Avec une directive de sectionnement
  - . Avec les directives EXT, ENT, USE, DST, EST, NDS, EOT, END, PSTS, PSTH
  - . Dans la phrase suivant une directive DO
  - . Avant la première directive de sectionnement
- Symbole manquant
- . Avec la directive EQU
  - . Avec la directive VAL
- ERA 04 Symbole incorrect
- . Définition d'un symbole après qu'il ait été déclaré externe
  - . Symbole absolu non suivi de la directive VAL
  - . Tentative de redéfinition par la directive VAL d'un symbole non absolu
- ERA 05 Expression non définie
- . Dans une instruction avec référence mémoire
  - . Avec les directives USE, EQU, END.
-





ERA 06

**Expression interdite**

- . Directive USE indiquant un positionnement sur une section PROG
- . Directive END avec une adresse de lancement n'appartenant pas à une section PROG
- . Déclaration par la directive EXT ou ENT d'un symbole alors qu'il est déjà apparu dans le programme
- . Directive EST ou NDS non suivie d'un nom de section

ERA 07

**Valeur d'une expression absolue incorrecte**

Non comprise entre

- . 0 et 15 avec PSTH
- . 0 et 128 avec PSTS
- . - 128 et 127 avec ADRI

ERA 08

**Symbole déjà défini ou utilisé**

- . Symbole translatable déjà défini
- . Nom de section déjà utilisé

ERA 09

**Saturation des tables de l'assembleur**

- . Plus de place dans la table des symboles
- . Nombre de sections dans le programme supérieur à 127

ERA 10

**Symbole mal utilisé**

- . Dans une section DSEC définition d'un symbole déjà utilisé à l'extérieur
- . Constante contenant l'adresse d'un mot d'une DSEC

ERA 11

**Opérande inaccessible**

- . Appartient à une section sur laquelle aucune base n'est positionnée (dans une instruction avec référence mémoire)
- . Conduit à un déplacement inférieur à - 128 ou supérieur à 127
- . Instruction de saut hors de la section PROG

ERA 12

**Passage du compteur d'assemblage au-delà de 32 K**

ERA 13

**Contexte incorrect**

- . Instruction hors d'une section PROG
- . Directive USE hors d'une section PROG
- . Directive PSTS ou PSTH dans une section DSEC

ERA 14

**Adresse opérande supérieure à 32 K**

- ERA 15    **Limite de la section PROG non atteinte**
- . Section contenant encore des instructions de saut en avant non résolues
  - . Adresse finale de la section conduisant à des instructions de saut en dehors de la section
- ERA 16    **Enchaînement des sections incorrect**
- . Programme comportant déjà une section COMMON
  - . Directive COMMON apparaissant dans le programme après une directive LOCAL ou PROG
  - . Segment comportant plus d'une directive LOCAL
- ERA 17    **Incomplet**
- Directive END apparaissant alors que des symboles sont encore non définis.

**Bull** 5.3 - CODES INSTRUCTIONS

a) Instructions avec référence mémoire

[ Etiquette ]	Code instruction	[&] { Expression translatable { (-128 ≤ Expression absolue ≤ 127), Base }
---------------	------------------	--

ACT	Réactivation d'une tâche software
AD	Addition à A
AND	Intersection avec A
BR	Branchement
BSR	Branchement à un sous-programme
CP	Comparaison de A
CPBY	Comparaison de A à un octet
CPZ	Comparaison à 0
DC	Décrémentation mémoire
DV	Division
EOR	Disjonction avec A
IC	Incrémentation mémoire
LA	Chargement de A
LAD	Chargement de l'adresse effective dans A
LB	Chargement de B
LBY	Chargement d'un octet dans A
LX	Chargement de X
LY	Chargement de Y
MP	Multiplication
OR	Union avec A
RLSE	Libération d'un accès à une ressource
RQST	Demande d'un accès à une ressource
SB	Soustraction à A
SIO	Lancement d'entrée-sortie
STA	Rangement de A
STB	Rangement de B
STBY	Rangement de l'octet droit de A
STX	Rangement de X
STY	Rangement de Y
STZ	Remise à 0 d'une mémoire
WAIT	Mise en attente d'une tâche software
XM	Échange mémoire avec A

## Solar 16

## b) Instructions avec opérande immédiat

Opérande immédiat 8 bits

[ Etiquette ]	Code instruction	(-128 ≤ Expression absolue ≤ 127), Registre
---------------	------------------	---

ADRI          Addition immédiate à un registre

Opérande immédiat 9 bits

[ Etiquette ]	Code instruction	(-256 ≤ Expression absolue ≤ 255)
---------------	------------------	-----------------------------------

ANDI	Intersection immédiate avec A
CPI	Comparaison immédiate de A
EORI	Disjonction immédiate avec A
LAI	Chargement immédiat de A
LBI	Chargement immédiat de B
LXI	Chargement immédiat de X
LYI	Chargement immédiat de Y
ORI	Union immédiate avec A

## c) Instructions de saut

[ Etiquette ]	Code instruction	Expression translatable
---------------	------------------	-------------------------

JAE	Saut si A = 0
JAG	Saut si A > 0
JAGE	Saut si A ≥ 0
JAL	Saut si A < 0
JALE	Saut si A ≤ 0
JANE	Saut si A ≠ 0
JC	Saut si report
JCV	Saut si report ou débordement
JDX	Décrémentation de X et saut conditionnel
JE	Saut si égal
JG	Saut si supérieur
JGE	Saut si supérieur ou égal
JIX	Incrémentation de X et saut conditionnel
JL	Saut si inférieur

<b>Bull</b>	<b>LE</b>	Saut si inférieur ou égal
	JMP	Saut inconditionnel
	JNC	Saut si pas de report
	JNCV	Saut si pas de report et pas de débordement
	JNE	Saut si différent
	JNV	Saut si pas de débordement
	JV	Saut si débordement

d) Instructions de décalage et de traitement de bits

[ Etiquette ]	Code instruction	( $0 \leq \text{Expression absolue} \leq 31$ ) [ , X ]
---------------	------------------	--

SARD	Décalage arithmétique à droite de A-B
SARS	Décalage arithmétique à droite de A
SCLD	Décalage circulaire à gauche de A-B
SCLS	Décalage circulaire à gauche de A
SCRD	Décalage circulaire à droite de A-B
SCRS	Décalage circulaire à droite de A
SLLD	Décalage logique à gauche de A-B
LLS	Décalage logique à gauche de A
SLRD	Décalage logique à droite de A-B
SLRS	Décalage logique à droite de A
IBT	Inversion d'un bit de A-B
RBT	Mise à 0 d'un bit de A-B
SBT	Mise à 1 d'un bit de A-B
TBT	Test d'un bit de A-B

e) Instructions sur registres

[ Etiquette ]	Code instruction	Registre [ , Registre ]
---------------	------------------	-------------------------

ADCR	Addition du report à un registre
ADR	Addition de registres
ADRP	Addition d'un registre à P
ANDR	Intersection entre registres
CLSR	Remise à zéro sélective d'un registre
CMR	Complémentation d'un registre
CPR	Comparaison entre registres
CPZR	Comparaison d'un registre à 0

EORR	Disjonction entre registres
LR	Chargement d'un registre par un autre registre
LRP	Chargement de P dans un registre
NGR	Opposé d'un registre
ORR	Union entre registres
SBCR	Soustraction du report à un registre
SBR	Soustraction entre registres
SWBR	Echange des octets de registres
XIMR	Echange d'un registre avec IM
XR	Echange entre registres

## f) Instructions d'arithmétique flottante

E

## Instructions avec référence mémoire

[ Etiquette ]	Code instruction	Expression translatable
---------------	------------------	-------------------------

FAD	Addition flottante
FCAM	Comparaison à A-B
FCMZ	Comparaison à 0
FDV	Division flottante
FLD	Chargement double de A-B
FMP	Multiplication flottante
FSB	Soustraction flottante
FST	Rangement double de A-B

## Instructions sur A-B

[ Etiquette ]	Code instruction	
---------------	------------------	--

FABS	Valeur absolue de A-B
FCAZ	Comparaison à 0 de A-B
FIX	Partie entière de A-B
FLT	Conversion entier flottant
FNEG	Opposition flottante
NORM	Normalisation

Instructions sans opérande

[ Etiquette ]	Code instruction	
---------------	------------------	--

ACK	Reconnaissance de sous-niveau d'interruption
ACQ	Acquittement d'une interruption
DBP	Recherche d'un point d'arrêt
DBT	Recherche du premier bit à 1 de A-B
DIT	Inhibition des interruptions
EIT	Validation des interruptions
HALT	Attente d'interruption
IPI	Interruption interprocesseurs
LAR	Chargement relatif de A
MOVE	Transfert d'une zone mémoire
MVTM	Transfert vers une zone maitre
MVTS	Transfert vers une zone esclave
NOP	Passage en séquence
PTY	Calcul de la parité de l'octet droit de A
PULL	Dépilement de A
PUSH	Empilement de A
QUIT	Fin de tâche software
RBP	Effacement d'un point d'arrêt
RDHV	Chargement dans A de HV
RDOE	Lecture de SLO-SLE
RDSI	Identification du système
ROMB	Branchement en mémoire permanente
RSR	Retour de sous-programme
RST	Mise à 0 sélective de ST
RSV	Retour du superviseur
SBP	Positionnement d'un point d'arrêt
SBS	Recherche d'un octet dans une zone
SCY	Mise à 1 de l'indicateur C
SST	Mise à 1 sélective de ST
STAR	Rangement relatif de A
STEP	Exécution en pas à pas
WOE	Chargement de SLO-SLE

Solar 16

Instructions sur registres

[ Etiquette ]	Code instruction	Liste de registres
---------------	------------------	--------------------

- LRM            Chargement multiple de registres  
PLR            Dépilement de registres par rapport à K  
PSR            Empilement de registres par rapport à K

Instructions avec opérande immédiat

[ Etiquette ]	Code instruction	Expression absolue
---------------	------------------	--------------------

- ACTD          Activation de la tâche alarme ( $0 \leq \text{Expression absolue} \leq 7$ )  
SVC          Appel du superviseur ( $0 \leq \text{Expression absolue} \leq 255$ )

[ Etiquette ]	Code instruction	$(0 \leq \text{Expression absolue} \leq 127)$ [ , X ]
---------------	------------------	---

- ARM            Demande d'activation d'une tâche software

Instructions optionnelles sans opérande

(E)

[ Etiquette ]	Code instruction	
---------------	------------------	--

- DRBM          Recherche et mise à 0 d'un bit  
INSQ          Insertion dans une liste  
RCDA          Lecture en CDA  
SFQ          Suppression en tête de liste  
SLQ          Suppression en queue de liste  
SUPQ          Suppression dans une liste  
WCDA          Ecriture en CDA

Instructions optionnelles avec opérande

(E)

[ Etiquette ]	Code instruction	$(0 \leq \text{Expression absolue} \leq 32\ 767)$ [ , X ]
---------------	------------------	---

- RBTM          Mise à 0 d'un bit  
SBTM          Mise à 1 d'un bit



**ASM16**

**ADDITIF**

**SEPTEMBRE 1984**

**Additif n° 1 164 000 00 036 02**  
**concernant la notice n° 1 164 000 00 036 01**

ADDITIF A LA NOTICE  
**ASM16**  
concernant les particularités de  
l'IE 05

## PRESENTATION

La nouvelle version de l'assembleur ASM16 permet de définir et d'utiliser des macro-instructions et en fait donc un macro-assembleur.

On utilise des macro-instructions lorsque l'on veut, par un appel unique, générer une suite de phrases reconnues par l'assembleur. Cette génération peut être conditionnée par des paramètres passés en entrée de la macro-instruction.

Une macro-instruction est repérée par un en-tête fourni lors de la définition et servant à en demander l'expansion en phrases assembleur lors de l'appel de la macro.

Pour appeler une macro-instruction, il suffit de citer une structure compatible avec son en-tête. On peut la faire précéder du caractère "%" qui indique explicitement qu'il s'agit d'une macro-instruction et provoque une erreur si la macro n'a pas été définie préalablement.

## DEFINITION D'UNE MACRO-INSTRUCTION

Une macro-instruction doit toujours être définie avant d'être utilisée. Une macro définition commence toujours par une directive

\*DEF en-tête

et se termine par une directive \*ENDEF. Entre ces deux directives figure le corps de la macro.

L'en-tête contient la structure à respecter lors de l'appel. Cet appel utilisera simplement un nom ou associera des caractères à des paramètres formels qui seront référencés dans le corps de la macro et remplacés au moment de l'expansion par des paramètres effectifs.

Ex : \*DEFMAC ? ? ?

Cette déclaration définit une macro-instruction MAC avec 3 paramètres. On note que l'en-tête est directement accolé à la directive \*DEF sans espace.

La directive \*ENDEF clôt la définition d'une macro-instruction. On peut alors continuer par d'autres macro-définitions, enchaîner sur des phrases assembleur, ou retourner au système par une directive

\*EOT

## CORPS D'UNE MACRO-INSTRUCTION

Le corps d'une macro-instruction peut contenir :

- des phrases assembleur
- des appels à d'autres macro-instructions ou à la macro-instruction en cours
- des instructions de macro commençant par le caractère "\*".

Les paramètres sont repérés par "Pi" avec i = rang du paramètre lors de la définition :

- ?Pi demande le remplacement du paramètre formel par le paramètre effectif de rang i
- ?Li demande le remplacement du paramètre formel par la longueur du paramètre effectif de rang i exprimée en nombre de caractères
- ?Ti demande le remplacement du paramètre formel par le type du paramètre effectif de rang i avec les conventions suivantes :

```
type = 1 : nombre hexadécimal
      = 2 : nombre décimal signé ou non
      = 3 : chaîne de caractères encadrée par des
            doubles apostrophes
      = 4 : symbole
      = 5 : autre
```

## DEFINITION DE VARIABLE : INSTRUCTION \*V

On peut définir dans le corps d'une macro-instruction des variables, utilisables dans toute autre macro.

Une variable est identifiée par un nom de la forme

Vxy avec xy variant de 01 à 99

Pour attribuer une valeur et un type à une variable, on utilise l'instruction

```
*Vxy = expression
```

Si expression = expression arithmétique, alors la variable est de type entier et a pour valeur l'évaluation de l'expression arithmétique.

Si expression = expression chaîne, alors la variable est de type chaîne et contient l'expression chaîne.

Une variable peut être définie plusieurs fois : c'est la dernière définition qui prévaut en type et en valeur.

### UTILISATION DES VARIABLES

L'utilisation d'une variable se fait par la syntaxe :

Vxy

Ex : \*V01 = 123

V01 contient la valeur 123

### EXPRESSIONS ARITHMETIQUES

Une expression arithmétique est un assemblage de variables entières et de nombres décimaux signés reliés par les opérateurs arithmétiques :

+ - \* /

Ces opérateurs ont la même priorité. L'évaluation se fait de la gauche vers la droite.

### EXPRESSION CHAÎNE

Une expression chaîne est un assemblage de chaînes de caractères et de variables chaînes.

Une expression arithmétique peut être considérée comme une variable chaîne. La réciproque n'est pas vraie.

### RUPTURES DE SEQUENCE

Dans le corps d'une macro-instruction, on peut utiliser des instructions de rupture de séquence permettant de modifier l'expansion de la macro-instruction en fonction des paramètres de la macro.

Il s'agit des instructions :

a) \*SKIP expression arithmétique

Si n est la valeur de l'expression arithmétique (positive ou négative) et p le numéro de la ligne contenant l'instruction \*SKIP, l'instruction \*SKIP provoque un déroutement vers la ligne p+n.

Si l'expression arithmétique vaut zéro, il y a abandon de la macro en cours.

b) \*IF relation SKIP expression arithmétique

Cette instruction se comporte :

- comme une instruction \*SKIP expression si la relation est vérifiée

- comme une instruction \*SKIP 1 si la relation est non vérifiée.

La relation peut être de deux types :

- relation arithmétique : de la forme

expression arithmétique  $\left\{ \begin{array}{l} < \\ = \\ > \end{array} \right\}$  expression arithmétique

- relation logique : de la forme

expression chaîne  $\left\{ \begin{array}{l} \text{NE} \\ \text{EQ} \end{array} \right\}$  expression chaîne

avec EQ : équivalent à  
NE : non équivalent à

c) \*KILL expression chaîne

Cette instruction provoque :

- l'abandon de la génération de la macro-instruction

- l'impression de l'expression chaîne

- le retour à l'assembleur qui poursuit l'assemblage en cours

### INSTRUCTIONS DE BOUCLE

Ces directives sont utilisées dans le corps d'une macro-instruction lorsque l'on désire répéter une séquence un certain nombre de fois.

Elles ont la forme suivante :

```
*DO  expression arithmétique positive ou nulle  
      séquence à répéter
```

```
*ENDO
```

La séquence encadrée par les 2 directives \*DO et \*ENDO est traitée autant de fois que la valeur de l'expression arithmétique avec un maximum autorisé de 252 fois.

A chaque directive \*DO est associée une directive \*ENDO.

Les boucles DO peuvent être imbriquées : on autorise jusqu'à 10 boucles imbriquées. Dans ce cas, il doit y avoir autant de \*DO que de \*ENDO.

On peut sortir d'une boucle DO avant la fin par une directive \*IF ou \*SKIP.

On peut rentrer de l'extérieur dans une boucle DO sans passer par la directive \*DO. Dans ce cas, sur la directive \*ENDO, on reboucle au début ou on passe en séquence selon l'état dans lequel se trouve la boucle vers laquelle on se branche (encore active ou abandonnée).

### INSTRUCTION \*EOT

Cette instruction indique la fin du fichier source en cours de traitement. Le contrôle est rendu au système. On peut continuer l'assemblage en cours par une commande CASM. Cette instruction est donc équivalente à la directive EOT de l'assembleur.

### REGLES DE RECONNAISSANCE DE L'EN-TETE

Afin de définir d'une manière unique l'en-tête correspondant à une ligne donnée, les règles suivantes sont utilisées :

- Recherche de la gauche vers la droite
- Au moins un séparateur entre deux indicateurs de paramètre ("?")
- Entre un indicateur de paramètre et un séparateur, c'est le second qui est toujours choisi
- En cas d'ambiguïté dans le choix des paramètres, un groupement des caractères à droite est effectué
- Dans un en-tête, le nombre d'espaces d'une suite d'espaces n'est pas significatif.

## NOUVELLES INSTRUCTIONS RECONNUES PAR ASM16

La nouvelle version de ASM16 sait traiter les nouvelles instructions privilégiées connues sous le nom ISP16. Ces nouvelles instructions se classent dans les catégories suivantes :

a) Instructions basées avec deux références mémoire ou deux registres

[Etiquette]	Code opér.	[&] Expres. transl.	, [&] Expres. transl.	[, X]
-------------	------------	---------------------	-----------------------	-------

ou

[Etiquette]	Code opér.	[&]Registre1,	[&]Registre2	[, X]
-------------	------------	---------------	--------------	-------

Ce sont les instructions :

- \* BDBTM : Recherche du premier bit à 1 dans une file de bits
- BDDL : Chargement double de A et B
- BDST : Rangement double de A et B
- BLA : Chargement de A
- \* BLBY : Chargement octet droit du registre A
- \* BMOVE : Transfert à une zone mémoire
- \* BRBTM : Mise à zéro d'un bit dans une file de bits
- \* BSBTM : Mise à un d'un bit dans une file de bits
- BSTA : Rangement de A
- \* BSTBY : Rangement de l'octet droit de A
- BXM : Echange de A et d'un mot mémoire

Les instructions repérées par le caractère "\*" ne peuvent comporter d'indexation ("X"), car elles utilisent le registre X en entrée.

b) Instructions d'accès à la CDA

Ce sont des instructions avec une référence mémoire

[Etiquette]	Code opération	[&] Expression translatable	[, X]
-------------	----------------	-----------------------------	-------

- CDLD : Chargement double A et B
- CDST : Rangement double A et B
- CLA : Chargement de A
- \*CLBY : Chargement octet droit de A
- CSTA : Rangement de A
- \*CSTBY : Rangement octet droit de A
- CXM : Echange de A avec une mémoire

Les instructions repérées par un "\*" ne peuvent être indexées (registre X utilisé en entrée).



c) Instruction de changement de contexte

Cette instruction est sans opérande

[Etiquette]	XCTX	
-------------	------	--

### ACTIVATION D'ASM16

La commande IASM d'activation de l'assembleur peut être suivie d'une liste d'options séparées par des virgules :

IASM,G,n      (0<n<10)

- le caractère G indique que l'on désire la liste symbolique de l'expansion des macro-instructions. En l'absence de cette option, seul le binaire généré par ASM16 lors de l'utilisation d'une macro-instruction est listé. Le symbolique expansé est précédé du caractère "+" pour signaler qu'il s'agit d'une macro-instruction.

- le chiffre n représente le nombre de dixièmes de la mémoire libre que l'on désire allouer à la table des macro-définitions, l'autre partie de la mémoire libre étant utilisée par la gestion de la table des symboles de l'assembleur. En standard, la répartition est faite de façon à optimiser les performances de l'assembleur. Cette option n'est à utiliser que si l'une ou l'autre table arrive à saturation.

Par défaut, n vaut 4.

### NOUVEAUX MESSAGES D'ERREUR

ERA 18	Commande d'activation incorrecte
ERM 00	Erreur de parité
ERM 01	Erreur d'écriture . erreur dans l'écriture d'une instruction de macro . numéro de variable supérieur à 99
ERM 02	Expression arithmétique incorrecte
ERM 03	Contexte incorrect ENDEF KILL SKIP IF hors d'une macro-définition
ERM 04	Élément non défini : variable ou paramètre
ERM 05	Appel de macro incorrect ou définition non intégrée

- ERM 06 Définitions imbriquées
- ERM 09 Saut à l'extérieur d'une macro
- ERM 11 Niveau d'imbrication d'une macro supérieur à 10
- ERM 12 Saturation des tables
- ERM 14 Ligne générée trop longue
- ERM 15 Saturation de l'espace alloué aux variables

REMARQUE :

Le symbole OFF ne peut être utilisé en tant qu'étiquette. Il est réservé à l'usage du macro-assembleur.