# MASM : Directives et pseudo-opcodes Chapitre 8

Des instructions comme mov ax, 0 et add ax, bx sont sans signification pour le microprocesseur. Malgré leur aspect ésotériques, elles constituent toutefois des versions humainement lisibles des instructions 80x86. Mais le processeur, par contre, ne répond qu'à des commandes telles que B80000 et 03C3. Un assembleur, est un programme capable de convertir des chaînes comme mov ax, 0 en instructions machine comme "B80000". Un programme écrit en assembleur, est donc composé d'instructions telles que mov ax, 0. Un assembleur convertit ces instructions à partir d'un fichier source en code machine, qui est l'équivalent binaire de ces instructions. En ce sens, un assembleur est très semblable à un compilateur, il lit un fichier ASCII à partir d'un disque et produit en sortie un programme en langage machine. La seule différence entre un compilateur de haut niveau, comme Pascal et un assembleur, est que le compilateur peut produire, à partir d'une instruction Pascal, plusieurs instructions machine, alors qu'un assembleur émet, pour chaque instruction assembleur, une seule instruction machine.

Essayer d'écrire des programmes directement en code exécutable (c.-à.-d. en binaire) ne serait pas particulièrement intelligent. Ce procédé est très ennuyeux, enclin aux erreurs et il n'offre pratiquement aucun avantage sur la programmation en assembleur, sauf peut-être qu'il faut d'abord assembler et éditer les liens du programme avant de pouvoir l'exécuter. Cependant, faire cela à la main prendrait plus que le peu de temps qu'un assembleur prend pour effectuer la conversion.

Apprendre l'assembleur donne aussi un autre avantage : un programme comme MASM (Microsoft's Macro Assembler), fournit un grand nombre de fonctionnalités pour le programmeur. Bien que leur apprentissage demande beaucoup de temps et d'application, c'est tellement utile qu'il vaut définitivement la peine de s'y mettre.

## 8.0 Vue d'ensemble du chapitre

Comme pour le chapitre 6, beaucoup des informations qu'on trouvera dans ce chapitre sont à titre de référence et comme dans toute référence, certaines connaissances sont essentielles, d'autres sont utiles, mais optionnelles, et d'autres encore peuvent ne jamais être utilisées pour écrire des programmes. La liste qui suit donne le résumé général de ce qui fera l'objet de ce chapitre. Un symbole "•", indique les matières essentielles, alors qu'un "□" indique les sujets optionnels et moins utilisés.

- • Le format des instructions assembleur dans un fichier source
- □ Le compteur d'emplacement
- • Les symboles et les identificateurs
- • Les constantes
- • Les déclarations de procédures
- □ Les segments des programmes
- • Les variables
- • Les types symboliques
- • Les expressions d'adresse (dont les dernières sections contiennent du matériel avancé)
- □ L'assemblage conditionnel
- □ Les macros
- □ Les directives des listings
- □ L'assemblage séparé

## 8.1 Les instructions de l'assembleur

Dans un fichier source, les instructions de l'assembleur ont le format suivant :

```
{Etiquette}     {Mot-Clé        {Opérande}}     {;Commentaire}
```

Chacune de ces entités est un *champ*, lequel peut représenter une étiquette, un mot-clé, une opérande ou un commentaire.

Un *champ étiquette* est (habituellement), un champ optionnel, contenant une étiquette symbolique pour l'instruction courante. Les étiquettes ont la même utilisation en assembleur que dans les langages de haut niveau, elles servent à marquer des lignes comme cibles pour des instructions telles que GOTO (sauts). Vous pouvez également spéficier des noms de variables, des noms de procédures et d'autres entités à l'aide des étiquettes symboliques. La plupart du temps ce champ est optionnel, ce qui veut dire qu'une étiquette doit être présente seulement quand vous avez besoin de l'insérer dans une ligne donnée. Certains mots-clés cependant, requièrent des étiquettes, d'autres ne les permettent pas. En général, on les place à la première colonne d'une ligne (elles sont les premières entités qui figurent, de sorte qu'un un programme sera plus facile à lire).

Un *mot-clé* est un nom d'instruction (par exemple, mov, add, etc.). Ce terme en anglais s'appelle *mnemonic* qui veut dire "aide mémoire", car, par exemple, "mov" est beaucoup plus facile à mémoriser que son équivalent binaire ! Les accolades indiquent que ce champ est optionnel. Notez pourtant qu'on ne peut pas avoir des opérandes sans mot-clé.

Le *champ mot-clé* contient donc une instruction. Ces dernières sont divisées en trois classes : instructions machine 80x86, directives de l'assembleur et pseudo-opcodes. La première classe, évidemment, est constituée par les mots-clés que nous avons présentés au chapitre 6.

Les directives assembleur sont des instructions spéciales qui fournissent des informations à l'assembleur mais qui ne génèrent aucun code. Des exemples incluent la directive *segment*, *equ*, *assume* ou *end*. Ces mots-clés ne sont pas des instructions valides pour le processeur. Elles sont simplement des messages pour le logiciel assembleur, rien de plus.

Un pseudo-opcode est un message pour l'assembleur, tout comme une directive, mais il émet des octets de code objet. Des exemples de pseudo-opcodes comprennent les termes byte, word, dword, qword et tbyte. Ces instructions produisent les octets des données spécifiées par leurs opérandes, mais elles ne sont pas de vraies instructions pour le processeur 80x86.

Le *champ opérande* contient les opérandes, ou paramètres, de l'instruction spécifiée par le champ *mot-clé*. Elles n'apparaissent pas dans une ligne de code sans être précédées donc par un mot-clé. Leur type et leur nombre (zéro, un, deux ou plus) dépend entièrement de l'instruction spécifique qui les précède.

Le *champ commentaire* vous permet de documenter chaque ligne du code source. Notez qu'un commentaire commence toujours par un point-virgule. Lorsque l'assembleur traite une ligne de texte, il ignore complètement tout ce qui suit le point-virgule[1].

Chaque instruction assembleur apparaît dans sa propre ligne du fichier source. Vous ne pouvez pas avoir d'instructions multiples dans la même ligne. D'autre part, tous les champs d'une ligne étant optionnels, des lignes vides sont permises. Vous pouvez en placer n'importe où dans votre code source. Elles sont utiles pour espacer certaines sections de code afin de les rendre plus faciles à lire.

MASM est libre de toute contrainte de rédaction. Les divers champs d'une instruction peuvent apparaître dans n'importe quelle colonne (tant qu'elles sont écrites dans un ordre correct). N'importe quel nombre d'espaces ou de tabulations peut séparer les divers champs d'une instruction. Pour l'assembleur, les deux séquences qui suivent sont identiques :

```
                mov     ax, 0
                mov     bx, ax
                add     ax, dx
                mov     cx, ax
```

```
mov             ax,                             0
        mov bx,                                 ax
add                     ax,     dx
                        mov             cx, ax
```

Bien sûr, la première séquence est plus lisible (si vous ne le croyez pas, il vous faudra peut-être consulter un psychiatre !). Donc, pour amour à la lisibilité, un usage judicieux des espacements et des tabulations fera toute la différence.

[1] Sauf, naturellement, le point-virgule apparaissant dans une chaîne constante.

En plaçant les étiquettes dans la première colonne, les mots-clés à la colonne 17 (c'est-à-dire à la seconde tabulation), le champ de l'opérande à la colonne 25 (troisième tabulation) et les commentaires vers la colonne 41 ou 49 (cinq ou six tabulations), on peut produire les meilleurs listings. Les programmes assembleur sont déjà, par leur nature même, suffisamment difficiles à lire, donc présenter votre listing de façon à le rendre plus clair produira un code beaucoup plus facile à maintenir.

Vous pouvez également écrire des commentaires qui portent sur une ligne tout entière. Dans de tels cas, placez un point-virgule dans la première colonne et utilisez toute la ligne pour le commentaire. Des exemples :

```
; La section de code suivante place le curseur au coin supérieur gauche de
; l'écran :
                mov     x, 0
                add     y, 0

; Maintenant, faire courir le curseur à la fin de l'affichage afin de nettoyer
; l'affichage vidéo :

;               etc.
```

## 8.2 Le compteur d'emplacement

Souvenez-vous que toutes les adresses dans l'espace mémoire des systèmes 80x86 sont constituées d'une adresse de *segment* et d'un offset dans ce segment. Pendant la conversion d'un fichier source en code objet, l'assembleur doit garder trace des offsets à l'intérieur du segment courant. Le *compteur d'emplacement* est une variable réservée à ce but.

Lorsqu'on crée un segment dans le fichier source (voir la section des segments plus loin dans ce chapitre), l'assembleur associe la valeur courante du compteur d'emplacement avec ce segment. Le compteur d'emplacement contient l'offset courant dans le segment. Au début (lorsque l'assembleur trouve pour la première fois un segment), ce compteur est mis à zéro. En trouvant des instructions ou des pseudo-opcodes, MASM incrémentera le compteur d'emplacement pour chaque octet écrit dans le fichier du code objet. Par exemple, il sera incrémenté de 2, après l'apparition d'une instruction comme mov ax, bx, car elle a deux octets de long.

La valeur du compteur d'emplacement varie sans arrêt pendant le processus d'assemblage. Elle change pour chaque ligne de votre code émettant du code objet. On utilisera le terme *compteur d'emplacement* pour signifier la valeur de cette variable à l'occurrence d'une instruction donnée avant que celle-ci génère tout code. Considérez les instructions suivantes :

```
0 :             or      ah, 9
3 :             and     ah, 0c9h
6 :             xor     ah, 40h
9 :             pop     cx
A :             mov     al, cl
C :             pop     bp
D :             pop     cx
E :             pop     dx
F :             pop     ds
10:             ret
```

Les instructions or, and et xor ont toutes trois octets de long ; l'instruction mov a une longueur de deux octets ; les autres instructions ont toutes une longueur d'un octet. Si ces instructions apparaissent au début d'un segment, le compteur d'emplacement sera le même que le nombre apparaissant à la gauche du listing ci-dessus. Par exemple, l'instruction or de cet exemple commence à l'offset 0. Puisque or a trois octets de long, la prochaine instruction and suit à l'offset 3. De même, and a trois octets de long, donc xor suit à l'offset 6 et ainsi de suite.

## 8.3 Symboles

Considérez par exemple l'instruction jmp. Cette instruction prend la forme :

```
                jmp     cible
```

Où *cible* est l'adresse de destination. Imaginez comme il serait frustrant s'il fallait spécifier cette adresse en format numérique. Si vous avez jamais programmé en BASIC (où les numéros de lignes sont la même chose que les étiquettes) vous avez probablement expérimenté à peu-près 10% des problèmes que vous auriez en assembleur si vous aviez à spécifier la cible de jmp par une adresse de mémoire.

En guise d'exemple, imaginez d'avoir à sauter à un certain groupe d'instructions que vous n'avez pas encore écrites. Quelle serait l'adresse de l'instruction cible ? Et comment pourriez-vous la prévoir avant d'avoir écrit chaque instruction qui précède cette adresse ? Et qu'arrive-t-il si vous modifiez le programme (gardez à l'esprit qu'ajouter ou effacer des instructions implique le changement du compteur d'emplacement pour toutes les instructions qui suivent la modification). Heureusement, tous ces problèmes ne se présentent qu'à ceux qui programment directement en langage machine. Qui programme en assembleur peut envisager le problème d'une manière beaucoup plus raisonnable : en utilisant justement les adresses symboliques.

Un *symbole*, un *identificateur* ou une *étiquette* est un nom associé à une valeur particulière. Cette valeur peut être un offset de segment, une constante, une chaîne, une adresse, un offset d'enregistrement ou même une opérande d'instruction. Dans tous les cas, un identificateur permet de représenter une certaine valeur (autrement incompréhensible) par un nom familier et facile à retrouver.

Un nom symbolique consiste en une séquence de lettres, nombres ou caractères spéciaux, soumise aux restrictions suivantes :

- Un symbole ne peut pas commencer par un chiffre numérique.
- Un nom peut avoir toute combinaison de casse de caractères alphabétiques. L'assembleur n'est pas sensible à la casse, donc, il ne distingue pas les majuscules des minuscules.
- Un symbole peut comporter n'importe quel nombre de caractères, cependant seulement les 31 premiers sont significatifs ; tous les autres sont ignorés.
- Les caractères _, $, ? et @ peuvent apparaître n'importe où dans un symbole ; cependant $ et ? sont des symboles spéciaux et vous ne pouvez pas définir un symbole avec juste ces caractères.
- Un symbole ne peut pas correspondre à un mot réservé. Les symboles suivants sont réservés :

```
%out                .186                .286                .286P
.287                .386                .386P               .387
.486                .486P               .8086               .8087
.ALPHA              .BREAK              .CODE               .CONST
.CREF               .DATA               .DATA?              .DOSSEG
.ELSE               .ELSEIF             .ENDIF              .ENDW
.ERR                .ERR1               .ERR2               .ERRB
.ERRDEF             .ERRDIF             .ERRDIFI            .ERRE
.ERRIDN             .ERRIDNI            .ERRNB              .ERRNDEF
.ERRNZ              .EXIT               .FARDATA            .FARDATA?
.IF                 .LALL               .LFCOND             .LIST
.LISTALL            .LISTIF             .LISTMACRO          .LISTMACROALL
.MODEL              .MSFLOAT            .NO87               .NOCREF
.NOLIST             .NOLISTIF           .NOLISTMACRO        .RADIX
.REPEAT             .UNTIL              .SALL               .SEQ
.SFCOND             .STACK              .STARTUP            .TFCOND
.UNTIL              .UNTILCXZ           .WHILE              .XALL
.XCREF              .XLIST              ALIGN               ASSUME
BYTE                CATSTR              COMM                COMMENT
DB                  DD                  DF                  DOSSEG
DQ                  DT                  DW                  DWORD
ECHO                ELSE                ELSEIF              ELSEIF1
ELSEIF2             ELSEIFB             ELSEIFDEF           ELSEIFDEF
ELSEIFE             ELSEIFIDN           ELSEIFNB            ELSEIFNDEF
END                 ENDIF               ENDM                ENDP
ENDS                EQU                 EVEN                EXITM
EXTERN              EXTRN               EXTERNDEF           FOR
FORC                FWORD               GOTO                GROUP
IF                  IF1                 IF2                 IFB
IFDEF               IFDIF               IFDIFI              IFE
IFIDN               IFIDNI              IFNB                IFNDEF
```

```
INCLUDE             INCLUDELIB          INSTR               INVOKE
IRP                 IRPC                LABEL               LOCAL
MACRO               NAME                OPTION              ORG
PAGE                POPCONTEXT          PROC                PROTO
PUBLIC              PURGE               PUSHCONTEXT         QWORD
REAL4               REAL8               REAL10              RECORD
REPEAT              REPT                SBYTE               SDWORD
SEGMENT             SIZESTR             STRUC               STRUCT
SUBSTR              SUBTITLE            SUBTTL              SWORD
TBYTE               TEXTEQU             TITLE               TYPEDEF
UNION               WHILE               WORD
```

De plus, tout nom de registre et tout nom d'instruction est également réservé. Notez que cette liste s'applique à la version 6.0 de MASM. Les versions antérieures ont moins de mots réservés. Des versions ultérieures en auront probablement plus.

Voici certains exemples de symboles valides :

```
L1                  Bletch              RightHere
Right_Here          Item1               __Special
$1234               @Home               $_@1
Dollar$             WhereAmI?           @1234
```

$1234 et @1234 sont parfaitement valides, aussi étranges qu'ils peuvent paraître.

Certains exemples de symboles incorrects incluent :

```
1TooMany        - Commence par un nombre.
Hello.There     - Contient un point au milieu du symbole.
$               - Un symbole ne peut pas être constitué uniquement par $ et ?
LABEL           - Mot réservé.
Right Here      - Un symbole ne peut pas contenir des espaces.
Hi,There        - Ou d'autres symboles spéciaux sauf _, ?, $ et @.
```

Comme on a mentionné ci-dessus les symboles peuvent représenter des valeurs numériques (telles que des compteurs d'emplacement), des chaînes ou même des opérandes complètes. De plus, l'assembleur assigne un type pour chaque symbole. Des exemples de types incluent near, far, byte, word, double word, quad word, et également des types texte et chaîne. La façon de déclarer des identificateurs d'un certain type est le sujet d'une grande partie du reste de ce chapitre. Pour l'instant, notez simplement que l'assembleur associe toujours un certain type à un identificateur et il a une tendance spéciale à se plaindre si on utilise un identificateur à un certain endroit où son type ne serait pas permis.

## 8.4 Constantes littérales

MASM est capable de travailler avec cinq différents types de constantes : entiers, entiers binaires codés décimaux compactés, nombres réels, chaînes et texte. Dans ce chapitre, nous considérerons seulement les entiers, les réels, les chaînes et le texte. Pour en savoir plus sur les nombres BCD compactés, veuillez consulter le Guide du Programmeur de MASM[2].

Une constante littérale est une constante dont la valeur se déduit à partir des caractères mêmes qui la constituent. Des exemples de ces constantes comprennent :

- 123
- 3.14159
- "Je suis une constante de caractères"
- 0FABCh
- 'A'
- <Constante Textuelle>

[2] Microsoft Macro Assembler Programmer's Guide, n.d.t.

À part la dernière constante, toutes les autres devraient vous sembler familières si vous avez déjà programmé dans un langage de haut niveau. Les constantes textuelles sont des versions spéciales de chaînes qui permettent la substitution textuelle pendant l'assemblage.

La représentation d'une constante littérale correspond exactement à ce qu'on attend de la valeur réelle d'un symbole. C'est pourquoi justement on les nomme aussi "constantes non symboliques", car elles se manifestent directement par la valeur qu'elles représentent et non par un nom symbolique ou identificateur. MASM permet évidemment de définir aussi des constantes symboliques ou "manifestes", mais nous parlerons de cela un peu plus loin.

### 8.4.1 Constantes entières

Une constante entière est une valeur numérique qui peut être spécifiée en format binaire, décimal ou hexadécimal[3]. Le choix de la base vous incombe. La table suivante illustre les caractères permis pour chaque base :

Tableau 35: Plage de chiffres pour chaque base

| Nom | Base | Caractères valides |
|---|---|---|
| Binaire | 2 | 0 1 |
| Décimale | 10 | 0 1 2 3 4 5 6 7 8 9 |
| Hexadécimale | 16 | 0 1 2 3 4 5 6 7 8 9 A B C D E F |

Pour faire la différence entre les nombres de chaque base, on utilise un caractère de suffixe. Si vous faites terminer un nombre par "b" ou "B", alors MASM comprendra qu'il s'agit d'un nombre binaire. Si un tel nombre contient d'autres chiffres à part le 1 ou le 0, MASM émet une erreur. Si le suffixe est "t", "T", "d" ou "D", MASM supposera qu'il s'agit d'un nombre décimal. Et, finalement, un suffixe "h" ou "H" représente une base hexadécimale.

Toutes les constantes entières doivent commencer par un nombre décimal, y compris les constantes hexadécimales. Pour représenter la valeur "FDED" il vous faudra spécifier 0FDEDh. Le premier chiffre est requis par l'assembleur afin qu'il puisse différencier entre les constantes numériques et les autres symboles. Gardez à l'esprit que "FDEDh" pourrait parfaitement être un nom de symbole.

Exemples :

```
0F000h          12345d          0110010100b
1234h           100h            08h
```

Si vous ne spécifiez pas un suffixe après la constante numérique, l'assembleur utilisera la base par défaut courante. Initialement, la base par défaut est la décimale. Par conséquent, vous pouvez normalement spécifier des valeurs décimales sans le suffixe. La directive radix de l'assembleur vous permet de changer la base par défaut. L'instruction .radix a la syntaxe suivante :

```
.radix          base            ;Commentaire optionnel
```

Où *base* est une valeur décimale entre 2 et 16.

L'instruction .radix est prise en compte dès qu'elle est rencontrée dans un fichier source. Toutes les instructions avant .radix utiliseront par conséquent la base par défaut avant le changement. En posant de multiples .radix dans votre fichier source vous pouvez passer continuellement d'une base par défaut à l'autre, selon votre convenance contextuelle.

Normalement, la base décimale est la plus susceptible d'être utilisée, par conséquent l'instruction .radix n'a pas normalement un grand usage. Cependant, s'il vous arrive de devoir entrer une énorme table de chiffres hexadécimaux, vous pouvez gagner beaucoup de temps en posant la base 16 comme base par défaut. Ensuite, après avoir terminé la table, vous pourrez passer de nouveau à la base 10 par défaut. Note : si la base par

[3]Vraiment, vous pouvez spécifier aussi la base octale. Mais on ne l'utilisera pas ici.

défaut est la hexadécimale, vous aurez à utiliser le suffixe "T" pour indiquer des valeurs décimales occasionnelles, car MASM pourrait confondre entre le suffixe "D" et le chiffre hexadécimal "D"[4].

### 8.4.2 Constantes chaînes

Une constante chaîne est une séquence de caractères entourée par des apostrophes ou des guillemets.

Des exemples :

```
"Voici un exemple"
'en voici un autre'
```

Vous pouvez insérer des apostrophes dans des chaînes entourées par des guillemets et vice-versa. Mais, si vous voulez placer un apostrophe à l'intérieur d'une chaîne délimitée par des apostrophes, il vous faudra placer des doubles apostrophes comme

```
'Ceci n''est-il pas bizarre ?'
```

Et il en va de même pour les guillements :

```
"Microsoft proclame : ""Nos logiciels sont très rapides."" Y croyez-vous ?"
```

Bien qu'on peut doubler des apostrophes ou des guillemets comme montré dans ces exemples, la façon la plus commode d'utiliser ces caractères à l'intérieur d'une constante chaîne est d'utiliser l'autre caractère comme délimitateur de chaîne :

```
"Ceci n'est-il pas bizarre ?"
'Microsoft proclame : "Nos logiciels sont très rapides." Y croyez-vous ?'
```

La seule occasion où il vous sera absolument nécessaire de doubler les guillemets ou les apostrophes, sera quand la chaîne contiendra *les deux* caractères en même temps. Mais ceci arrive assez rarement dans les programmes réels.

Comme dans les langages C et C++, il y a une différence subtile entre une valeur caractère et une valeur chaîne. Un seul caractère (qui est une chaîne de longueur 1) peut apparaître n'importe où MASM est susceptible de permettre une constante entière ou une chaîne. Si vous spécifiez une constante caractère là où MASM s'attend à trouver une constante entière, MASM utilisera le code ASCII de ce caractère comme constante entière. Les chaînes (dont la longueur est supérieure à 1) sont permises seulement dans des contextes déterminés.

### 8.4.3 Constantes réelles

Dans certains contextes, on peut utiliser des constantes à virgule flottante. MASM vous permet d'exprimer ces constantes de deux façons différentes : notation décimale ou notation scientifique. Ces versions sont très semblables à leurs représentations en Pascal, en C ou en d'autres langages de haut niveau.

La version décimale est simplement une séquence de chiffres contenant un point décimal à une position quelconque du nombre :

```
1.0             3.14159         625.25          -128.0          0.5
```

La notation scientifique aussi est pratiquement égale à celle qu'on utilise dans les divers langages de haut niveau :

```
1e5             1.567e-2        -6.02e-10       5.34e+12
```

L'étendue de la plage de précision des nombres dépend de votre package de nombres à virgule flottante. Cependant, pour les constantes mentionnées ci-dessus, MASM émet généralement des données binaires compatibles avec les coprocesseurs numériques 80x87. Ce format correspond au format numérique spécifié par le standard IEEE pour les valeurs à virgule flottante. Notez en particulier que la constante 1.0 ne correspond pas à l'équivalent binaire du nombre 1.

[4]Rappellez-vous que MASM n'est pas sensible à la casse, n.d.t.

### 8.4.4 Constantes textuelles

Les constantes textuelles ne sont pas équivalentes aux constantes chaînes. Une constate textuelle effectue une substitution mot par mot pendant le processus d'assemblage. Par exemple, les caractères 5[bx] pourraient être une constante textuelle associée au symbole VAR1. Pendant l'assemblage, une instruction comme mov ax, VAR1 serait convertie en mov ax, 5[bx].

Les associations textuelles sont très utiles dans MASM parce que cet assembleur aime souvent utiliser de longues chaînes de texte pour représenter de simples opérandes. Utiliser des associations textuelles permet de simplifier ces opérandes en substituant une chaîne donnée par un seul identifiant dans une instruction.

Une constante textuelle consiste en une séquence de caractères entourés par les symboles "<" et ">". Par exemple, la constante textuelle 5[bx] serait écrite <5[bx]>. Lors de la substitution, MASM enlève les caractères "<" et ">".

## 8.5 Déclarer des constantes manifestes à l'aide des constantes *equates*

Une constante manifeste est un nom de symbole représentant une quantité établie pendant le processus d'assemblage. Autrement dit, c'est un nom symbolique qui représente une certaine valeur. Les equates sont le mécanisme que MASM utilise pour déclarer des constantes symboliques. Il y a trois syntaxes de base :

```
symbole         equ             expression
symbole         =               expression
symbole         textequ         expression
```

L'opérande *expression* est normalement une expression numérique, ou bien une chaîne textuelle. Tandis que *symbole* se voit attribuer la valeur et le type de l'expression. Les directives equ et "=" ont été employées par MASM dès le début. Microsoft a ensuite ajouté la directive textequ avec la version 6.0.

La finalité de la directive "=" est de définir des symboles ayant une quantité entière (ou comportant un seul caractère) associée à eux. Cette directive ne permet pas d'opérandes de type réel, chaîne ou texte. C'est la directive primaire à utiliser en créant des constantes symboliques numériques. Voici des exemples :

```
NumElements             =       16
                        .
                        .
                        .
Array                   byte    NumElements dup (?)
                        .
                        .
                        .
                        mov     cx, NumElements
                        mov     bx, 0
ClrLoop:                mov     Array[bx], 0
                        inc     bx
                        loop    ClrLoop
```

La directive textequ définit un symbole de substitution de texte. L'expression dans l'opérande doit être une constante textuelle délimitée par les symboles "<" et ">". Quand MASM trouve ce symbole dans une instruction, il substitue le texte associé avec ce symbole. Les programmeurs utilisent habituellement cet equate pour économiser des frappes ou pour rendre un code plus lisible :

```
Count                   textequ <6[bp]>
DataPtr                 textequ <8[bp]>
                        .
                        .
                        .
                        les     bx, DataPtr     ;Même chose que les bx, 8[bp]
                        mov     cx, Count       ;Même chose que mov cx, 6[bp]
                        mov     al, 0
ClrLp:                  mov     es:[bx], al
                        inc     bx
                        loop    ClrLp
```

Notez que c'est parfaitement correct d'associer un symbole avec une opérande vide, comme dans l'exemple qui suit:

```
BlankEqu                textequ <>
```

L'utilité d'un tel equate sera éclaircie dans les sections dédiées à l'assemblage conditionnel et aux macros.

La directive equ constitue plutôt un surensemble des possibilités des directives "=" et textequ. Elle permet des opérandes à la fois numériques, textuelles ou chaînes littérales. Les directives suivantes sont toutes correctes :

```
One                     equ     1
Minus1                  equ     -1
TryAgain                equ     'Y'
StringEqu               equ     "Bonjour à tous"
TxtEqu                  equ     <4[si]>
                        .
                        .
                        .
HTString                byte    StringEqu       ;Même chose que "Bonjour à tous"
                        .
                        .
                        .
                        mov     ax, TxtEqu      ;Même chose que mov ax, 4[si]
                        .
                        .
                        .
                        mov     b1, One         ;Même chose que mov b1, 1
                        cmp     al, TryAgain    ;Même chose que cmp al, 'Y'
```

Les constantes manifestes que vous déclarez avec des equates vous aident à *normaliser* un programme. Si vous utilisez la même valeur, chaîne ou texte plusieurs fois dans votre listing, une association symbolique vous permettra de modifier très facilement ces données lors des développements futurs de votre programme. Considérez l'exemple suivant :

```
Array                   byte    16 dup(?)
                        .
                        .
                        .
                        mov     cx, 16
                        mov     bx, 0
ClrLoop:                mov     Array[bx], 0
                        inc     bx
                        loop    ClrLoop
```

Si vous décidez qu'*Array* doit avoir 32 éléments au lieu de 16, il vous faudra parcourir tout votre programme, localiser chaque référence à cette donnée et en ajuster la constante littérale en conséquence, ce qui peut très facilement causer des bogues, car on peut oublier des ajouts. Au contraire, si vous utilisez la constante symbolique NumElements que nous venons de montrer, vous n'aurez à modifier qu'une seule instruction dans votre programme et il ne vous restera qu'à l'assembler de nouveau et à l'exécuter. MASM mettra à jour tout le reste automatiquement en utilisant NumElements.

Vous pouvez redéfinir des symboles avec la directive "=". Autrement dit, ce qui suit est parfaitement correct :

```
Symbole         =       0
                .
                .
                .
Symbole         =       1
```

Puisqu'on peut changer la valeur d'une constante, la *portée* du symbole (quand celui-ci a une valeur particulière) devient importante. Si on ne pouvait pas redéfinir un symbole, on s'attendrait à ce que la valeur constante du symbole soit toujours la même dans toute l'étendue du programme. Puisqu'au contraire vous pouvez redéfinir une constante dans un programme, le symbole ne peut pas avoir une portée de programme. La solution que MASM utilise est évidente : la portée d'une constante manifeste part du point où une valeur a été définie jusqu'au point où elle sera redéfinie. Ceci amène à une conséquence importante : *vous devez déclarer toutes les constantes manifestes avec la directive "=" avant d'utiliser la constante.* Bien sûr, une fois qu'une constante symbolique a été redéfinie, la valeur que cette constante avait précédemment sera perdue. Notez aussi qu'on ne peut pas redéfinir des symboles qui ont été déclarés avec les directives textequ ou equ.

## 8.6 Directives du processeur

Par défaut, MASM assemble seulement les instructions disponibles dans *tous* les membres de la famille 80x86. En particulier, cela veut dire qu'il *n'assemblera pas* les instructions qui ne sont pas disponibles sur les processeurs 8086 et 8088. En générant une erreur pour les instructions non-8086, MASM prévient l'usage accidentel d'instructions qui ne sont pas compatibles avec tous les processeurs. Ceci est bien, sauf si vous *voulez* utiliser les instructions disponibles au-delà des 8086 et 8088. Les directives du processeur permettent de spécifier au microprocesseur l'étendue des instructions à utiliser.

Les directives du processeur sont :

| | | | | |
|---|---|---|---|---|
| .8086 | .8087 | .186 | .286 | .287 |
| .286P | .386 | .387 | .386P | .486 |
| .486P | .586 | .586P | | |

Aucune de ces directives n'accepte d'opérandes.

Ces directives permettent toutes les instructions disponibles sur un processeur donné. Puisque les membres de la famille 80x86 sont conçus en fonction de la compatibilité avec des membres ultérieurs, le fait de spécifier une directive de processeur particulière permet l'usage des instructions de tous les processeurs qui le précèdent.

Les directives .8087, .287 et .387 activent le jeu d'instructions à virgule flottante correspondant aux coprocesseurs mathématiques respectifs. Néanmoins, la directive .8086 active aussi l'utilisation du jeu 8087 ; de même, .286 active le jeu 80287 et .386 permet le jeu 80387. La seule utilité des directives FPU (*floating point unit*), est de permettre des instructions 80287, avec un jeu d'instructions 8086 ou 80186, ou bien permettre un jeu mathématique 80387 avec un jeu 8086, 80186 ou 80286.

Les directives terminant avec "P" permettent des instructions en *mode privilégié*. De telles instructions sont utiles seulement à ceux qui écrivent des systèmes d'exploitation, certains pilotes de périphériques et d'autres routines système avancées. Puisque ce livre ne traite pas les instructions en mode privilégié, nous n'avons pas besoin de décrire ce mode ultérieurement.

Les processeurs 80386 et ultérieurs, supportent deux types de segments lors du fonctionnement en mode protégé : des segments de 16 bits et des segments de 32 bits. En mode réel, ces processeurs supportent seulement de segments de 16 bits, L'assembleur doit générer des opcodes subtilement différents pour les segments de 16 et de 32 bits. Si on a spécifié un processeur de 32 bits en utilisant les directives .386, .486 ou .586, MASM génère des instructions pour des segments de 32 bits par défaut. Si vous tentez d'exécuter un tel code en mode réel sous DOS, vous planterez probablement le système. À ce problème, il y a deux solutions. La première est de spécifier use16 comme opérande pour chaque segment que vous créez dans votre programme. L'autre solution est légèrement plus pratique : ajoutez simplement l'instruction suivante après la directive du processeur à 32 bits :

```
option  segment:use16
```

Cette directive indique à MASM de générer des segments de 16 bits et non de 32 bits par défaut. Notez que MASM ne requiert pas un processeur 80486 ou Pentium si vous spécifiez les directives .486 ou .586. L'assembleur lui-même est écrit en code 80386[5], donc, tout ce dont vous avez besoin est un processeur 80386 pour assembler tout programme avec MASM. Sans doute, si vous utilisez des instructions qui sont spécifiques aux processeurs 80486 ou Pentium, vous aurez besoin de ces processeurs pour les exécuter.

Vous pouvez sélectivement activer ou désactiver divers jeux d'instructions dans votre programme. Par exemple, vous pouvez activer un jeu 80386 pour plusieurs lignes de code et ensuite retourner en mode 8086. La séquence de code suivante en donne un exemple :

```
.386            ;Début des instructions 80386
 .              ;ce code peut se servir d'un jeu 80386
 .
 .
.8086           ;Retour aux instructions 8086
 .              ; Ce code ne peut avoir que
 .              ; des instructions 8086
 .
```

[5] A partir de la version 6.1.

C'est également possible d'écrire une routine capable de détecter, pendant l'exécution, sous quel processeur un programme est en train de courir. Par conséquent, on peut détecter un 80386 et utiliser son jeu d'instructions. Si vous ne détectez pas un tel processeur vous pouvez en tenir aux instructions 8086. En activant les instructions 80386 de façon sélective, de sorte qu'elles s'exécutent seulement sous un processeur pouvant les admettre, vous pouvez tirer avantage des instructions les plus récentes. De même, le fait de désactiver les instructions 80386 dans d'autres sections de votre programme, vous permet d'en prévenir l'usage accidentel où celui-ci n'est pas prévu.

## 8.7 Procédures

Contrairement aux langages de haut niveau, MASM n'a pas établi de règles strictes pour ce qui concerne une procédure[6]. Vous pouvez appeler une procédure à toute adresse de mémoire. La première instruction ret trouvée à l'intérieur du sous-programme, met fin à la procédure. Une telle liberté d'expression provoque cependant souvent des abus aboutissant ainsi à des programmes vraiment difficiles à lire et à maintenir. Par conséquent, MASM a fourni des facilités pour mieux déclarer des procédures à l'intérieur du code. Le mécanisme de base est :

```
procname        proc    {NEAR or FAR}
                <instructions>
procname        endp
```

Comme vous pouvez le voir, la définition d'une procédure est similaire à celle d'un segment. La différence est que *procname* (qui est le nom de la procédure que vous définissez), doit être un identificateur unique dans toute l'étendue de votre programme. Votre code appellera la procédure en utilisant son nom, c'est pourquoi on ne peut pas déclarer une autre procédure en se servant du même nom. Autrement, comment votre programme déterminerait-il quelle routine appeler ?

*proc* permet plusieurs opérandes différentes ; nous n'en considérerons que trois : *near*, *far* et sans spécification. MASM utilise ces opérandes pour déterminer si vous appelez une procédure avec la version near ou far de l'instruction call. Elles déterminent aussi quel type d'instruction ret devra être émis à l'intérieur de la procédure. Considérez les deux procédures suivantes :

```
NProc           proc    near
                mov     ax, 0
                ret
NProc           endp

FProc           proc    far
                mov     ax, 0FFFFh
                ret
FProc           endp
```

et

```
                call    NPROC
                call    FPROC
```

L'assembleur produira automatiquement un appel de trois octets (near) pour la première instruction, car il sait d'avance que cette procédure est de type *near*. Il générera également un appel de cinq octets (far) pour le second call, car FProc est effectivement de type *far*. A l'intérieur des procédures elles-mêmes, MASM produit automatiquement des retours *near* ou *far* selon les circonstances, pour toutes les instructions *ret* trouvées.

Notez que si vous ne faites pas terminer une procédure avec une instruction ret ou tout équivalent capable de faire revenir le contrôle au programme appelant, l'exécution continuera avec la prochaîne instruction exécutable trouvée après le endp. Considérez l'exemple suivant :

```
Proc1           proc
                mov     ax, 0
Proc1           endp

Proc2           proc
```

[6]Dans ce livre, "procedure" signifie toute unité de programme pouvant être désignée sous les termes "procedure", "sous-routine", "sous-programme", "fonction", "opérateur", etc.

```
                mov     bx, 0FFFFh
                ret
Proc2           endp
```

Par conséquent, si vous appelez Proc1, le contrôle tombera aussi sur Proc2 à partir de l'instruction mov bx, 0FFFFh. Contrairement aux procédures des langages de haut niveau, une fonction en assembleur ne contient pas un retour implicite avant la directive endp. Donc soyez toujours conscient de ce que vous faites avec la logique de vos proc/endp !

En ce qui concerne les déclarations des procédures il n'y a rien de spécial. Ce n'est qu'un moyen commode fourni par l'assembleur, rien de plus. Vous pouvez parfaitement écrire des programmes en assembleur pendant le reste de votre vie sans jamais utiliser de directives proc et endp. Cependant, ce serait une mauvaise pratique de programmation, car proc et endp sont des merveilleux outils pour documenter vos programmes, les rendre plus faciles à lire et à maintenir.

MASM, à partir de la version 6.0, considère toutes les étiquettes à l'intérieur d'une procédure comme locales. Ce qui veut dire que vous ne pouvez pas faire directement référence à ces symboles à l'extérieur de la procédure où ils sont définis. Pour plus de détails, voir le paragraphe 8.11.1.

## 8.8 Segments

Tous les programmes se constituent d'un ou de plusieurs segments. Certes, pendant l'exécution, les registres de segment pointent sur les segments couramment actifs. Sur les microprocesseurs 80286 et antérieurs, on peut avoir jusqu'à quatre segments actifs (*code*, *data*, *extra* et *stack*). Sur les microprocesseurs 80386 et supérieurs on peut compter sur deux segments additionnels, fs et gs. Bien qu'on ne puisse pas accéder à des données dans plus de quatre ou six segments à la fois, on peut toutefois les modifier de sorte à les faire pointer à des endroits différents pendant le déroulement du flux d'un programme. Ce qui équivaut en fait, à pouvoir accéder à plus de quatre ou six segments. La question la plus naturelle à se poser à ce point serait : « comment créer ces différents segments dans un programme et comment y accéder, dynamiquement, pendant l'exécution » ?

Les segments, dans un programme source en assembleur, sont définis avec les directives *segment* et *ends*. Vous pouvez définir autant de segments qu'il vous chante dans vos programmes. Certes, vous êtes en réalité limité à ne pas dépasser les 65536 segments, du moins pour ce que les processeurs 80x86 vous permettent, et MASM probablement sera encore plus restrictif sur cette limite, mais en pratique, vous ne dépasserez jamais le nombre de segments que MASM vous permet d'utiliser dans un programme.

Quand MS-DOS lance l'exécution de vos programmes, il initialise deux registres de segment. Il fait pointer cs sur le segment où se trouve votre programme principal et fait pointer ss sur le segment de la pile matérielle. A partir de là, vous êtes entièrement responsable de ce que vous faites avec vos segments.

Pour accéder aux données dans un segment particulier, le registre de segment doit contenir l'adresse du segment en question. Si vous accédez à des données éparses entre différents segments, votre programme devra charger le registre de segment avec l'adresse appropriée avant d'avoir accès à chaque nouvelle zone. Si vous recourez à cette pratique fréquemment, vous perdrez un temps considérable pour recharger les registres de segment. Heureusement, la plupart des programmes font preuve de localité référentielle pour accéder à leurs données. Ce qui veut dire qu'un fragment de code accédera probablement au même groupe de variables plusieurs fois pendant une période de temps donnée. Et ce n'est pas difficile d'organiser vos programmes de façon à faire apparaître dans le même segment les variables que vous utilisez le plus souvent. En arrangeant vos programmes de cette façon, vous minimiserez le nombre de rechargements des registres de segment. En le considérant de cette façon, un segment n'est rien de plus qu'un cache conservant les données les plus fréquemment utilisées.

En mode réel, un segment peut avoir une taille allant jusqu'à 64 Ko. La plupart des programmes en assembleur "pur" utilisent moins de 64 Ko pour le code, moins de 64 Ko pour les données globales et moins de 64 Ko pour l'espace pile. Par conséquent, dans la plupart des circonstances trois ou quatre segments seront parfaitement suffisants. En fait, le fichier SHELL.ASM (votre fichier de modèle pour commencer vos programmes), ne définit que quatre segments et généralement vous n'en utiliserez que trois. Si vous vous servez de SHELL.ASM comme base, vous n'aurez que rarement à vous préoccuper de la segmentation. D'autre part, si vous voulez écrire des programmes complexes sur les plateformes 80x86, alors il vous comprendre le principe de segmentation.

Un segment dans vos fichiers doit respecter la syntaxe suivante :[7]

```
segname         segment         {READONLY} {align} {combine} {use} {'class'}
                instructions
segname         ends
```

Les sous-paragraphes qui suivent décrivent chacune des opérandes de la directive *segment*.

A noter : la segmentation est un concept que beaucoup de débutants trouvent difficile à comprendre. Vous n'avez pas besoin de comprendre profondément la notion de segmentation pour commencer à écrire des programmes. Si vous faites une copie du fichier SHELL.ASM pour chacun des programmes que vous écrivez, vous pouvez effectivement ignorer les matières relatives à la segmentation. Le but principal de SHELL.ASM est justement de se prendre en charge, à votre place, les détails de la segmentation. Tant que vous n'écrirez de gros et complexes programmes ou que vous n'utiliserez un gros volume de données, vous serez parfaitement en mesure de vous servir de SHELL.ASM et d'ignorer tout simplement la segmentation. Néanmoins, vous finirez par écrire de tels programmes à l'avenir, ou bien vous écrirez des sous-routines pour des langages de haut niveau comme Pascal ou C++. Dans une telle éventualité, il vaut définitivement la peine d'en savoir plus à ce sujet. Le point est donc que, bien que vous pouvez ne pas avoir besoin de la segmentation immédiatement, elle se révélera tôt ou tard nécessaire dès que vous décidez d'aller un peu plus loin.

### 8.8.1 Les noms des segments

La directive *segment* requiert une étiquette dans le champ étiquette. Cette étiquette correspond au nom du segment. MASM se sert des noms de segment pour trois raisons : pour combiner des segments, pour déterminer si un préfixe de surcharge de segment est nécessaire et pour obtenir l'adresse d'un segment. La directive ends qui termine le segment doit, elle aussi, être précédée du nom du segment.

Si le nom du segment n'est pas unique (en d'autres termes, si vous l'avez déjà défini ailleurs dans le programme), toutes les autres utilisations de ce nom doivent être des définitions de segment. S'il y a un autre segment défini avec le même nom, MASM le traitera simplement comme une continuation du segment du même nom déjà rencontré. Chaque segment a son propre compteur d'emplacement qui lui est associé. Quand on commence un nouveau segment (dont le nom n'a pas déjà été trouvé au cours du programme), MASM crée pour ce segment une nouvelle variable de compteur d'emplacement initialisée à zéro. Si au contraire MASM trouve une définition de segment qui correspond à un nom déjà rencontré précédemment, il utilisera simplement la dernière valeur atteinte par le compteur d'emplacement pour ce segment. Par exemple,

```
CSEG            segment
                mov     ax, bx
                ret
CSEG            ends

DSEG            segment
Item1           byte    0
Item2           word    0
DSEG            ends

CSEG            segment
                mov     ax, 10
                add     ax, Item1
                ret
CSEG            ends
                end
```

Le premier segment (CSEG) commence avec un compteur d'emplacement initialisé à 0. L'instruction mov ax, bx a une taille de deux octets et l'instruction ret a une longueur d'un octet, donc, le compteur d'emplacement a une valeur de trois à la fin de ce segment. DSEG est un autre segment de trois octets, donc le compteur d'emplacement qui lui est associé atteint également une valeur de trois. Le troisième segment a le même nom que le premier (à savoir CSEG), par conséquent, l'assembleur considérera le troisième segment comme la

[7] A partir de MASM 5.0 on peut bénéficier de directives de segment simplifiées. Puisque Microsoft les a de plus en plus améliorées, aujourd'hui elles ont acquis une certaine complexité. Bien qu'elles puissent rendre plus simples les interfaces entre l'assembleur et les langages de haut niveau, nous les ingorerons ici.

continuation du premier. Donc, le code placé dans le dernier segment sera assemblé à partir de l'offset trois, qui correspond à la dernière valeur prise par le compteur d'emplacement après l'instruction ret du premier CSEG.

Toutes les fois que vous spécifiez un nom de segment comme opérande d'une instruction, MASM utilisera le mode d'adressage direct en remplaçant ce nom par l'adresse de ce segment. Puisque vous ne pouvez pas charger une valeur immédiate dans un registre de segment avec une seule instruction, on effectue normalement cette opération à l'aide de deux instructions. Par exemple, les trois instructions suivantes apparaissent au début du fichier SHELL.ASM, elles initialisent les registres ds et es et pointent donc sur le segment dseg :

```
        mov     ax, dseg        ;chargement de ax avec l'adresse de dseg
        mov     ds, ax          ;ds pointe sur dseg
        mov     es, ax          ;es pointe sur dseg
```

L'autre fonction d'un nom de segment est de fournir un moyen pratique pour indiquer le composant segment d'un nom de variable. Souvenez-vous que les adresses 80x86 sont constituées de deux parties : un segment et un offset. Puisque le matériel 80x86 fait pointer par défaut sur le segment des données la plupart des références aux données, c'est une habitude courante chez les programmeurs de faire de même : c'est-à-dire ne pas mentionner de nom de segment pour accéder à des variables dans le segment de données. En effet, une référence complète à une variable consiste en un nom de segment, un ":" et un nom d'offset :

```
        mov     ax, dseg:Item1
        mov     dseg:Item2, ax
```

Techniquement, il faudrait préfixer toutes les variables avec le nom de segment de cette façon. Cependant, la plupart du temps, on peut ignorer cette syntaxe stricte, sauf dans les quelques cas où le fait de spécifier le nom du segment est vraiment nécessaire. Heureusement, ces situations sont rares et ont lieu seulement dans le contexte de programmes très complexes, pas le genre de choses que vous aurez à faire pour l'instant.

C'est également important garder à l'esprit que spécifier un nom de segment avant un nom de variable ne veut pas dire que vous pouvez accéder aux données d'un segment sans avoir un registre de segment qui pointe sur lui. Sauf pour les instructions *jump* et *call*, il n'y a pas d'instruction 80x86 permettant de spécifier une adresse segmentée directe de 32 bits. Toutes les autres références à la mémoire utilisent un registre de segment pour suppléer la composante segment d'une adresse.

### 8.8.2 Ordre de chargement des segments

Normalement, les segments sont chargés en mémoire selon l'ordre dans lequel ils apparaissent dans votre fichier source. Dans l'exemple qu'on a vu au paragraphe précédent, MS-DOS charge le segment CSEG en mémoire *avant* le segment DSEG. Même si dans ce cas le CSEG apparaît en deux parties, avant et après DSEG, la déclaration de CSEG avant toute occurrence de DSEG demande au DOS de charger en mémoire l'intégralité du segment CSEG avant le segment DSEG. Pour charger DSEG avant CSEG, vous pourriez utiliser le programme suivant :

```
DSEG            segment         public
DSEG            ends

CSEG            segment         public
                mov     ax, bx
                ret
CSEG            ends

DSEG            segment         public
Item1           byte            0
Item2           word            0
DSEG            ends

CSEG            segment         public
                mov     ax, 10
                add     ax, Item1
                ret
CSEG            ends
end
```

La déclaration DSEG est vide et n'émet aucun code. Le compteur d'adresse pour DSEG est à zéro à la fin de cette définition. Par conséquent, il est à zéro aussi au commencement du prochain DSEG, exactement comme il était dans la version précédente de ce programme. Cependant, puisque la déclaration de DSEG apparaît en premier ici, MS-DOS le charge en premier.

L'ordre d'apparition n'est qu'un des facteurs qui déterminent l'ordre de chargement. Par exemple, si vous utilisez la directive ".alpha", MASM organisera les segments par ordre alphabétique et non par ordre d'apparition. Les opérandes optionnelles dans les directives de segment contrôlent aussi l'ordre de chargement des segments. Ces opérandes sont le sujet de la prochaine section.

### 8.8.3 Opérandes de segment

La directive *segment* permet six opérandes différentes : align, combine, class, readonly, "use*nn*" et size. Trois de ces opérandes contrôlent la façon dont DOS charge les segments en mémoire et les autres trois contrôlent la génération de code.

#### 8.8.3.1 Le type ALIGN

Le paramètre align est l'un des termes suivants : byte, word, dword, para ou page. Ces mots-clés indiquent à l'assembleur, à l'éditeur de liens et au DOS de charger le segment à une adresse alignée sur un caractère, un mot, un double-mot, un paragraphe ou une page. Ce paramètre est optionnel. Si aucun des mots-clé qu'on vient de voir n'apparaît comme paramètre dans la directive *segment*, l'alignement par défaut est le paragraphe (qui est un multiple de 16 octets).

Aligner un segment sur un octet a pour effet de charger ce segment en mémoire à partir du premier octet disponible après le dernier segment. Aligner un segment sur un mot, fera commencer le segment au premier octet ayant une adresse paire après le dernier segment. De même, aligner sur un double-mot veut dire que le segment courant sera localisé à la première adresse multiple de 4 après le dernier segment. Et ainsi de suite.

Par exemple, si le segment #1 est déclaré en premier dans votre fichier source et segment #2 le suit immédiatement et qu'il est soumis à un alignement d'octets, alors les deux segments seront stockés en mémoire de la façon suivante (voir figure 8.1)

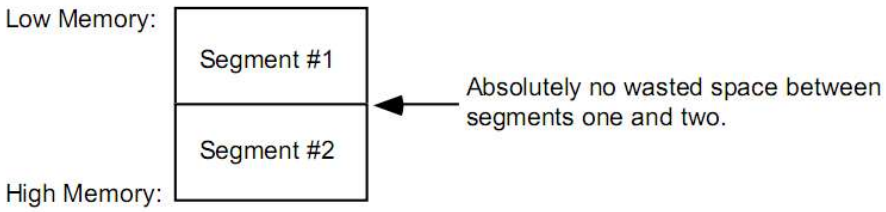


Figure 8.1 Un segment aligné sur un octet

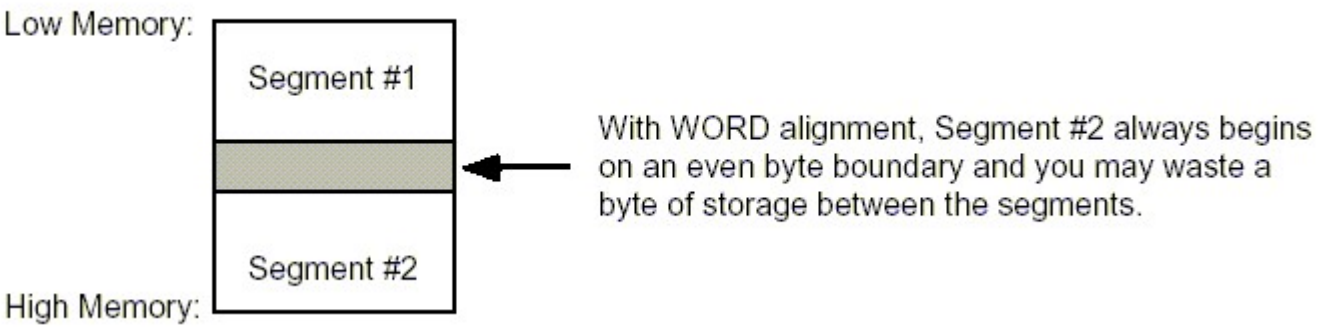

Figure 8.2 Un segment aligné sur un mot

```
seg1            segment
                .
                .
                .
seg1            ends

seg2            segment byte
                .
                .
                .
seg2            ends
```

Si les segments sont déclarés comme ci-dessus et le segment #2 est aligné sur un mot, le segment apparaît en mémoire tel que montré à la figure 8.2.

```
seg1            segment
                .
                .
                .
seg1            ends
seg2            segment word
                .
                .
                .
seg2            ends
```

Un autre exemple : si les segments 1 et 2 sont déclarés comme ci-dessus et le segment #2 est aligné sur un double-mot, alors ils seront stockés en mémoire tel que montré à la figure 8.3.

```
seg1            segment
                .
                .
                .
seg1            ends

seg2            segment dword
                .
                .
                .
seg2            ends
```

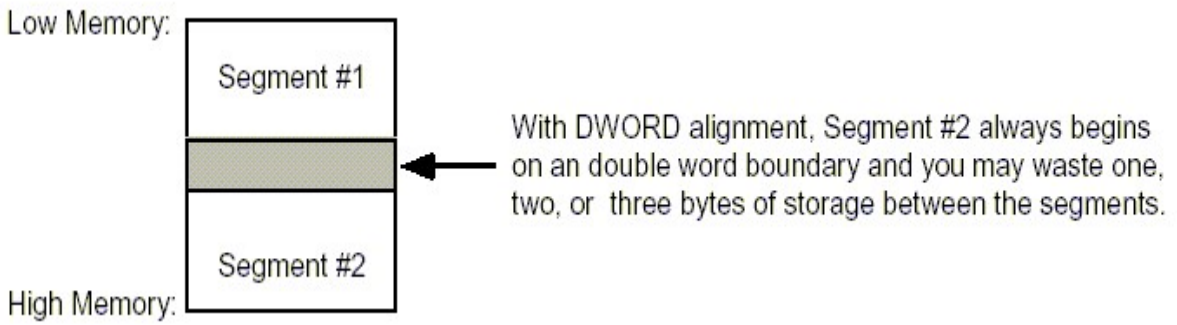


Figure 8.3 Les segments alignés sur un double-mot

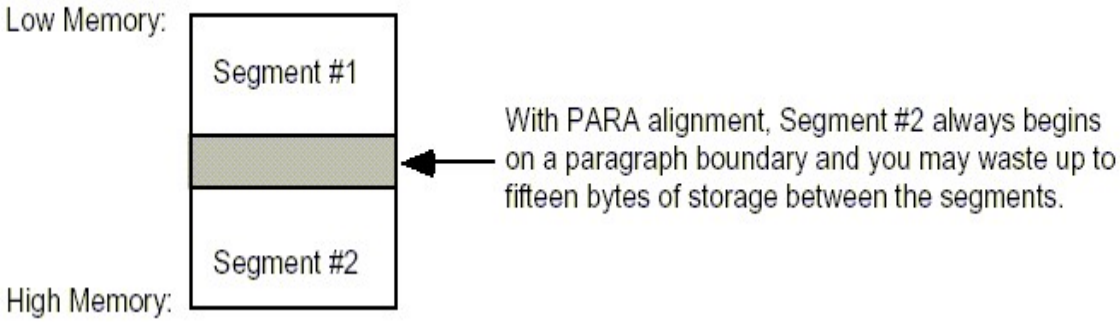


Figure 8.4 Les segments alignés sur un paragraphe

Puisque les registres de segment 80x86 pointent toujours sur une adresse de paragraphe, la plupart des segments sont alignés sur une limite de paragraphe (para). Par conséquent, à moins d'avoir une bonne raison pour faire différemment, il vaut mieux de toujours laisser vos segments alignés sur une limite de paragraphe.

Par exemple, si les segments 1 et 2 sont déclarés comme dans l'exemple suivant et le segment #2 est aligné sur un paragraphe, MS-DOS stockera les segments en mémoire tel que montré par la figure 8.4.

```
seg1            segment
                .
                .
                .
seg1            ends

seg2            segment para
                .
                .
                .
seg2            ends
```

Un alignement par page oblige le segment à commencer à une adresse multiple de 256 octets. Certains tampons de données peuvent requérir des alignements sur des multiples de 256 (ou même 512) octets. L'option d'aligner les segments par page peut donc être utile dans ces situations.

Par exemple, si les segments 1 et 2 sont déclarés tels que ci-dessous, ils seront chargés en mémoire comme il est montré à la figure 8.5.

```
seg1            segment
                .
                .
                .
seg1            ends

seg2            segment page
                .
                .
                .
seg2            ends
```

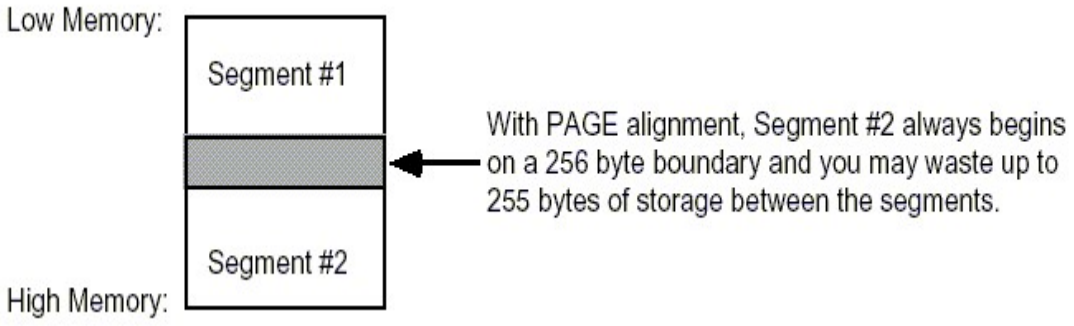


Figure 8.5 Segments alignés par page

Si vous choisissez tout autre alignement que byte, l'assembleur, l'éditeur de liens et le DOS peuvent insérer certains octets factices entre les deux segments de sorte à bien assurer leur alignement. Puisque les registres de segment 80x86 doivent toujours pointer à une adresse paragraphe (donc ils doivent être alignés sur un paragraphe), vous pouvez vous demander comment le processeur peut adresser un segment aligné sur un octet, mot ou double-mot. C'est facile. Quand vous spécifiez un alignement par segment obligeant un segment à commencer à une adresse qui ne coïncide pas avec un paragraphe, l'assembleur/éditeur de liens suppose que le registre de segment pointe sur le paragraphe précédent et le compteur d'emplacement commencera avec une valeur correspondant à un certain offset dans ce segment, une valeur différente de zéro. Par exemple, supposez que le segment #1 ci-dessus termine à l'adresse physique 10F87h et que le segment #2 est aligné par octets. Le code pour le segment #2 commencera à l'adresse 10F80h. Cependant, le segment #2 se superposera au segment #1 sur une plage de 8 octets. Pour résoudre ce problème, le compteur d'adresse du segment #2 commencera à l'offset 8, donc le segment #2 sera chargé en mémoire juste après le segment #1.

Si le segment #2 est aligné par octets et le segment #1 ne termine pas à une adresse exacte de paragraphe, MASM ajustera le compteur d'adresse de départ pour le segment #2 de sorte à lui faire utiliser l'adresse du paragraphe précédent (voir la Figure 8.6).

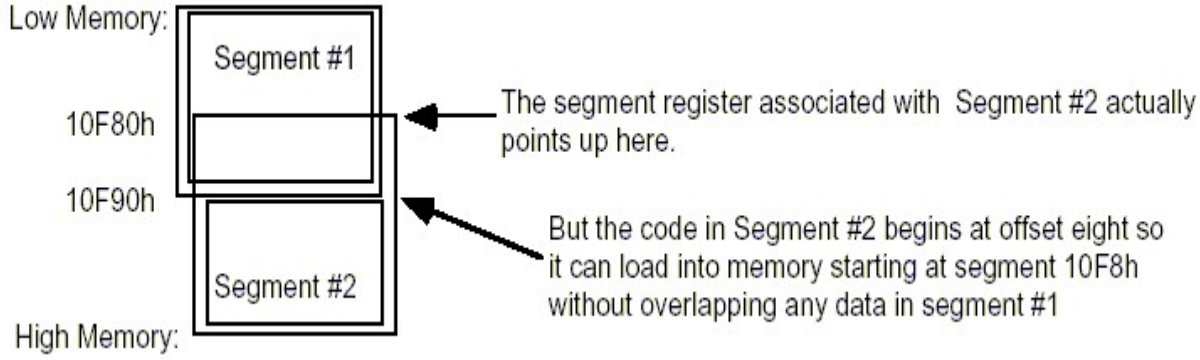


Figure 8.6 Des segments alignés par paragraphe

Puisque l'architecture 80x86 impose que tous les segment commencent à une adresse alignée sur un paragraphe, MASM supposera par défaut que vous voulez ce type d'alignement pour vos segments. La définition suivante de segment respectera toujours un alignement sur un paragraphe.

```
CSEG            segment
        mov     ax, bx
                ret
CSEG            ends
end
```

### 8.8.3.2 Le type COMBINE

Le type *combine* contrôle l'ordre dans lequel les segments *ayant le même nom* sont écrits dans le fichier code objet que l'assembleur produit. Pour spécifier le type de *combine*, on utilise l'un des mots-clés *public*, *stack*, *common*, *memory* ou *at*. Alors que *memory* est un synonyme de *public* fourni pour des raisons de compatibilité ; il faudrait toujours utiliser *public* au lieu de *memory*. *common* et *at* sont des types avancés qui ne seront pas abordés dans ce livre. Le type *stack* doit être utilisé avec vos segments de pile (pour voir un exemple, relisez le paragraphe 4.9.3). Le type *public* est utilisé pour toute autre situation.

Les types *public* et *stack* effectuent essentiellement la même opération. Ils concatènent des segments ayant le même nom dans un seul segment contigu, comme il a été décrit précédemment. La différence entre les deux est la façon dont le DOS traite l'initialisation du segment de pile et des registres pointeurs de pile. Tous les programmes doivent avoir au moins un segment de type pile (sinon l'éditeur de liens générera des avertissements) ; tout le reste doit être *public*. Au chargement du programme en mémoire, le DOS fera pointer automatiquement le registre de segment de pile sur le segment que vous déclarez avec *stack*.

Si vous ne spécifiez pas un type *combine*, l'assembleur ne concaténera pas les segments quand il produira le fichier de code objet. En effet, l'absence de mot-clé de type *combine* produit par défaut un type de concaténation des segments *private.* Sauf si les classes de types sont les mêmes (voir section suivante) chaque segment sera émis selon l'ordre dans lequel il sera trouvé par MASM dans le fichier source. Par exemple, considérez le programme suivant :

```
CSEG            segment         public
                mov             ax, 0
                mov             VAR1, ax
CSEG            ends

DSEG            segment         public
I               word            ?
DSEG            ends

CSEG            segment         public
                mov             bx, ax
                ret
CSEG            ends

DSEG            segment         public
J               word            ?
DSEG            ends
                end
```

Ce programme produit le même code que :

```
CSEG            segment         public
                mov             ax, 0
                mov             VAR1, ax
                mov             bx, ax
                ret
CSEG            ends

DSEG            segment         public
I               word            ?
J               word            ?
DSEG            ends
                end
```

L'assembleur fusionne automatiquement tous les segments qui ont le même nom et sont publics. S'il vous permet de séparer les segments de cette façon, c'est uniquement pour votre convenance. Supposez que vous avez différentes procédures, chacune desquelles requiert certaines variables. Vous pourriez déclarer toutes les variables dans un segment quelque part, mais ceci est souvent obscur. Beaucoup de programmeurs préfèrent déclarer leurs variables juste avant la procédure qui les utilise. En utilisant la le type combine *public* dans la déclaration des segments, vous pouvez déclarer vos variables à votre convenance et ensuite l'assembleur les déplacera dans le segment approprié pendant l'assemblage. Par exemple,

```
CSEG            segment         public
;ici la première procédure
DSEG            segment         public

; variables locales pour la procédure 1
VAR1            word            ?
DSEG            ends
                mov             ax, 0
                mov             var1, ax
                mov             bx, ax
                ret
; ici la deuxième procédure

DSEG            segment         public
I               word            ?
J               word            ?
DSEG            ends

                mov             ax, I
                mov             ax, J
                ret
CSEG            ends
                end
```

Notez que vous pouvez imbriquer les segments de toutes les façons que vous désirez. Malheureusement, les règles de portée de MASM ne fonctionnent pas de la même façon que chez les langages de haut niveau comme Pascal. Une fois que vous avez défini un symbole à l'intérieur d'un programme, il est visible partout ailleurs dans ce programme[8].

### 8.8.4 Le type CLASS

La dernière opérande de la directive segment est normalement le type *class*. Son rôle est de spécifier l'ordre des segments qui n'ont pas le même nom. L'opérande consiste en un symbole entouré par des apostrophes (les guillemets ne sont pas permis dans ce contexte). Généralement, vous devrez utiliser les noms suivants : CODE (pour les segments contenant le code du programme), DATA (pour les segments contenant les variables, les données constantes et les tables), CONST (pour des segments ne contenant que des données constantes et des tables), et STACK (pour le segment de pile). Le programme suivant illustre cet usage :

```
CSEG            segment         public 'CODE'
                mov             ax, bx
                ret
CSEG            ends

DSEG            segment         public 'DATA'
Item1           byte            0
Item2           word            0
DSEG            ends

CSEG            segment         public 'CODE'
                mov             ax, 10
                add             ax, Item1
                ret
CSEG            ends

SSEG            segment         stack 'STACK'
STK             word            4000 dup(?)
SSEG            ends

C2SEG           segment         public 'CODE'
                ret
C2SEG           ends
```

[8] La plus grosse exception à cette règle, sont les étiquettes d'instruction à l'intérieur d'une procédure.

```
end
```

La procédure de chargement effectue ce qui suit. L'assembleur localise le premier segment dans le fichier. Puisuq'il s'agit d'un type public, tous les autres segments CSEG seront concaténés à la suite de celui-ci. Finalement, puisqu'il s'agit de la classe 'CODE', MASM y ajoute tous les segments (C2SEG) qui relèvent de la même classe. Après avoir traité ces segments, MASM recherche dans le fichier source le prochain segment ne faisant partie de la combinaison et répète le même processus. Dans l'exemple qu'on vient de voir, les segments seront chargés en mémoire selon l'ordre suivant : CSEG, CSEG (la seconde occurrence), C2SEG, DSEG et finalement SSEG. Les règles générales concernant la manière dont vos fichiers seront chargés en mémoire sont les suivantes :

- (1) L'assembleur combine tous les segments publics qui ont le même nom.
- (2) Une fois combinés, les segments sont émis sur le fichier code objet selon leur ordre d'apparition dans le fichier source. Si un nom de segment apparaît deux fois dans un même fichier source (qui est public), alors le segment combiné sera produit sur le fichier du code objet à la position reflétant la première occurrence du segment à l'intérieur du fichier source.
- (3) L'éditeur de liens lit le fichier code objet produit par l'assembleur et réarrange les segments lors du processus de création du fichier exécutable. L'éditeur de liens commence par écrire le premier segment trouvé dans le fichier code objet dans le fichier .EXE. Puis, il cherche à travers le fichier de code objet tout autre segment ayant le même nom. De tels segments seront écrits séquentiellement dans le fichier .EXE.
- (4) Une fois que tous les segments qui partagent le même nom de classe que le premier segment sont écrits dans le fichier .EXE, l'éditeur de liens parcourt encore le fichier du code objet à la recherche du prochain segments qui n'appartient pas au même nom de classe que le(s) précédent(s). Il écrit donc ce segment dans le fichier .EXE et répète l'étape (3) pour chaque autre segment correspondant éventuellement à cette nouvelle classe.
- (5) Finalement, l'éditeur de liens répétera l'étape (4) jusqu'à ce qu'il liera tous les segments du fichier code objet.

### 8.8.5 L'opérande Read-Only

Si *readonly* est la première opérande de la directive *segment*, l'assembleur générera une erreur chaque fois qu'une instruction tentera d'écrire dans ce segment. Ceci peut être utile aussi bien pour les segments de code que pour les segments de données. Cette option ne vous empêche pas d'écrire dans ces segments pendant l'exécution. C'est très facile de tromper l'assembleur en ce sens. L'utilité de *readonly* consiste à pouvoir détecter certaines erreurs de programmation que vous pourriez manquer autrement. Puisque vous ne placez que rarement des variables dans votre segment de code, c'est probablement une bonne idée de définir ce segment en lecture seule.

Voici un exemple :

```
seg1            segment         readonly para public 'DATA'
                .
                .
                .
seg1            ens
```

### 8.8.6 Les options USE16, USE32 et FLAT

En travaillant avec des processeurs 80386 ou ultérieurs, MASM génère des codes différents selon que vos segments soient de 16 ou de 32 bits. Mais en écrivant des programmes qui fonctionnent en mode réel sous le DOS, il faudra toujours utiliser des segments de 16 bits. Les segments de 32 bits sont applicables seulement avec des programmes qui sont écrits pour le mode protégé. Malheureusement, si on sélectionne un processeur 80386 ou supérieur en utilisant des directives comme .386, .486 ou .586, MASM définit le mode 32 bits par défaut. Donc si vous voulez utiliser les instructions 32 bits, il vous faudra indiquer explicitement à MASM d'utiliser des segments de 16 bits. Les opérandes de la directive segment use16, use32 et flat vous permettent de spécifier la taille des segments.

Pour la plupart des programmes DOS, vous devrez toujours utiliser l'opérande use16, indiquant à MASM qu'il devra traiter avec des segments de 16 bits et qu'il faudra les assembler en conséquence. *Si vous utilisez l'une des directives qui activent un jeu d'instructions 80386 ou supérieur, alors vous devrez aussi spécifier use16 dans tous vos segments de code, sinon MASM générera un mauvais résultat.*

Un exemple :

```
seg1            segment         readonly para public use16 'data'
                .
                .
                .
seg1            ends
```

Les opérandes use32 et flat indiquent à MASM de générer un code pour les segments de 32 bits. Puisque la programmation en mode protégé n'est pas l'objet de ce livre[9], nous ne considérerons pas ces options (voir *The MASM Programmer's Guide*, pour plus de détails).

Si vous voulez imposer l'option use16 par défaut dans un programme qui se servira d'un jeu d'instructions 80386 ou supérieur, placez simplement la directive suivante dans votre programme, avant toute déclaration de segment :

```
                .option         segment:use16
```

### 8.8.7 Définitions typiques de segments

L'explication qu'on vient de faire vous a laissé totalement confus ? Ne vous inquiétez pas. Tant que vous n'écrirez des programmes extrêmement grands, vous n'aurez pas à vous soucier de toutes les opérandes associées avec la directive segment. Pour la plupart des programmes, les trois segments suivants sont largement suffisants :

```
DSEG            segment         para public 'DATA'
; Insérez vos variables ici.
DSEG            ends

CSEG            segment         para public use16 'CODE'
; Insérez vos instructions ici.
CSEG            ends

SSEG            segment         para stack 'STACK'
stk             word            1000h dup(?)
EndStk          equ             this word
SSEG            ends

                end
```

Le fichier SHELL.ASM déclare automatiquement ces segments à votre place. Si vous faites toujours une copie de ce fichier à chaque nouveau programme, vous n'aurez pas en général à vous inquiéter à propos de la façon appropriée de déclarer des segments.

### 8.8.8 Pourquoi contrôler l'ordre de chargement

Certains appels du DOS requièrent que vous passiez la longueur de votre programme comme paramètre. Malheureusement, calculer la longueur d'un programme qui comporte plusieurs segments est une tâche très difficile. Cependant, quand le DOS charge vos programmes en mémoire, il les charge entièrement dans un bloc contigu de RAM. Donc, pour connaître la longueur de votre programme, vous n'aurez qu'à connaître son adresse de départ et de de fin. En effectuant une simple différence entre ces deux valeurs, vous pourrez déterminer la longueur du programme.

Dans un programme contenant de multiples segments, il faut savoir quel segment a été chargé en premier et lequel a été chargé en dernier, sinon vous ne pourrez pas en déterminer la longueur. Il s'avère que le DOS

[9]Vous pouvez trouver, du même auteur, des livres écrits pour la programmation 32 bits et Windows dans son site web : https://www.plantation-productions.com/Webster/, n.d.t.

charge toujours le préfixe de segment de programme ou PSP (*Program Segment Prefix*), en mémoire avant le chargement du tout premier segment du programme. Donc, pour calculer la longueur de votre code, vous devez d'abord prendre en compte la longueur du PSP. MS-DOS passe l'adresse de segment du PSP dans le registre ds. Donc, calculer la différence entre le dernier octet de votre programme et le PSP, équivaut à produire sa longueur. Le segment de code suivant calcule la longueur, en paragraphes, d'un programme.

```
CSEG            segment public  'CODE'
                mov     ax, ds          ;Obtenir l'adresse du PSP
                sub     ax, seg LASTSEG ;Calcul de la différence

;AX contient maintenant la longueur de ce programme (en paragraphes)
                .
                .
                .
CSEG            ends

; Insérez tous vos autres segments ici

LASTSEG         segment para    public 'LASTSEG'
LASTSEG         ends
                end
```

### 8.8.9 Préfixes de segment

Quand les processeurs 80x86 font référence à une opérande de mémoire (une variable), ils font référence à un emplacement dans le segment de données courant[10]. Cependant, vous pouvez indiquer au microprocesseur de référencer des données dans un autre segment en utilisant un préfixe de segment avant une expression d'adresse.

Un préfixe de segment est soit ds:, cs:, ss:, es:, fs: ou gs:. Quand il est utilisé avant une expression d'adresse, un tel préfixe indique au processeur d'aller chercher l'opérande mémoire dans le segment spécifié et non dans le segment de données par défaut. Par exemple, mov ax, cs:L[bx] (où L est une variable), charge l'accumulateur à partir de de l'adresse L+bx dans le *segment de code courant*. Si le préfixe cs: était absent, l'instruction aurait chargé la donnée depuis ds. De même, mov ds:[bp], ax, stocke la valeur de ax dans l'emplacement de mémoire pointé par le registre bp dans le segment de données courant (souvenez-vous que quand vous utilisez bp comme registre de base, il pointe normalement sur le registre de pile et non sur le registre de données).

Les préfixes de segments sont des opcodes d'instructions. Par conséquent, quand vous utilisez un préfixe de segment, vous augmentez la longueur (en diminuant la vitesse aussi), de l'instruction utilisant ce préfixe. Par conséquent, si vous n'avez pas une bonne raison de le faire, n'utilisez pas les préfixes de segment.

### 8.8.10 Contrôler les segments à l'aide de la directive ASSUME

L'architecture 80x86 fait généralement référence aux éléments de données dans le registre ds (ou bien dans le registre ss). De même, toutes les références de code (sauts, appels, etc.) sont toujours relatives au segment de code courant. Il y a seulement une question à se poser : comment l'assembleur connaît-il quel segment est le segment de données et quel segment est le segment de code ? La directive *segment* n'indique pas à quel segment une partie programme correspond. Rappellez-vous qu'un segment de données est tel *parce que le registre ds y pointe dessus*. Puisque le registre ds peut être changé pendant l'exécution d'un programme (par exemple, par une instruction comme mov ds, ax), tout segment peut être le segment de données. Et ceci a certaines conséquences intéressantes pour l'assembleur. Quand on spécifie un segment dans un programme, il ne faut pas seulement indiquer au CPU qu'il s'agit, par exemple, du segment de données[11], mais il faut aussi lui indiquer où et quand ce segment est un segment de données (ou de code/pile/extra/F/G). C'est la directive *assume* qui fournit cette information à l'assembleur. Elle a la syntaxe suivante :

```
assume {CS:seg} {DS:seg} {ES:seg} {FS:seg} {GS:seg} {SS:seg}
```

[10] Les exceptions à cette règle, sont assurément les instructions et les modes d'adressage qui utilisent le segment de pile par défaut (par ex. push/pop et les modes d'adresssage qui utilisernt bp ou sp).

[11] En chargenant ds par l'adresse de ce segment, bien sûr.

Les accolades entourent des éléments optionnels, elles ne font pas partie de la syntaxe. Notez qu'il faut spécifier au moins une opérande. *seg* peut être soit le nom d'un segment (défini par la directive *segment*) ou le mot réservé *nothing*. Des opérandes multiples de la directive *assume* doivent être séparées par une virgule. Voici des exemples de directives valides :

```
        assume          DS:DSEG
        assume          CS:CSEG, DS:DSEG, ES:DSEG, SS:SSEG
        assume          CS:CSEG, DS:NOTHING
```

La directive assume indique à l'assembleur que vous avez chargé les registres de segment avec l'adresse de segment de la valeur spécifiée. **Notez que cette directive ne modifie aucun des registres de segment, elle indique simplement à l'assembleur de supposer que les registres de segment pointent sur certains segments du programme.** Tout comme les directives equate ou de sélection du processeur, la directive *assume* modifie le comportement de l'assembleur à partir de la première apparition de celle-ci, jusqu'à une autre directive *assume* qui produirait alors un autre comportement.

Considérez le programme suivant :

```
DSEG1           segment para public 'DATA'
var1            word    ?
DSEG1           ends

DSEG2           segment para public 'DATA'
var2            word    ?
DSEG2           ends

CSEG            segment para public 'CODE'
                assume  CS:CSEG, DS:DSEG1, ES:DSEG2
                mov     ax, seg DSEG1
                mov     ds, ax
                mov     ax, seg DSEG2
                mov     es, ax

                mov     var1, 0
                mov     var2, 0
                .
                .
                .
                assume  DS:DSEG2
                mov     ax, seg DSEG2
                mov     ds, ax
                mov     var2, 0
                .
                .
                .
CSEG            ends
                end
```

Quand l'assembleur trouve un nom symbolique, il vérifie quel segment contient ce symbole. Dans le programme qu'on vient de voir, *var1* apparaît dans le segment DSEG1 et *var2* dans le segment DSEG2. Souvenez-vous que le microprocesseur ne connaît pas les segments déclarés dans votre programme, il peut seulement accéder aux données pointées par des registres de segment tels que cs, ds, es, ss, fs et gs. Dans ce programme, l'instruction *assume* indique à l'assembleur que le registre ds pointe sur DSEG1 pendant la première partie du programme et sur DSEG2 pendant la seconde.

Quand l'assembleur trouve une instruction de la forme mov var1, 0, la première chose qu'il fait est déterminer le segment de cette variable. Ensuite il compare le segment avec la liste des assomptions que l'assembleur fait pour l'usage des registres de segments. Si vous ne déclarez pas var1 dans l'un de ces segments, l'assembleur génère une erreur en indiquant que le programme ne peut pas accéder à cette variable. Si le symbole - *var1* dans notre exemple - apparaît dans la liste des segments couramment assumés, alors l'assembleur vérifie si le segment en question est le segment de données. Si c'est le cas, alors l'instruction est assemblée, selon ce qui est décrit dans les appendices. Si le symbole apparaît dans un segment autre que celui sur lequel ds est supposé de pointer, alors l'assembleur émet un *segment override prefix byte* (octet de préfixe de surcharge de segment), en spécifiant le segment qui contient réellement la donnée.

Dans l'exemple ci-dessus, MASM assemble mov var1, 0 sans l'octet de préfixe de segment. MASM assemble la première occurrence de l'instruction mov var2, 0 avec es: comme préfixe, car l'assembleur suppose que c'est es et non ds qui pointe sur DSSEG2. MASM assemble ensuite la seconde occurrence de cette instruction sans le préfixe es:, car, à ce point du fichier source, il présume que ds pointe sur DSEG2. Gardez à l'esprit que c'est très facile de tromper l'assembleur. Considérez le code suivant :

```
CSEG            segment para public 'CODE'
                assume  CS:CSEG, DS:DSEG1, ES:DSEG2
                mov     ax, seg DSEG1
                mov     ds, ax
                .
                .
                .
                jmp     SkipFixDS

                assume  DS:DSEG2

FixDS:          mov     ax, seg DSEG2
                mov     ds, ax

SkipFixDS:
                .
                .
                .
CSEG            ends
                end
```

Remarquez que le programme saute par-dessus le code qui charge le registre ds avec une valeur de segment pour DSEG2. Ceci veut dire qu'à l'étiquette SkipFixDS le registre DS contient un pointeur sur DSEG1 et pas sur DSEG2. Cependant l'assembleur n'est pas assez malin pour réaliser qu'il vient de se produire un problème, donc il suppose tout bonnement que ds pointe sur DSEG2 et non sur DSEG1. Ce qui équivaut à un désastre imminent. Puisqu'il croit que vous accédez aux variables sur DSEG2 quand en réalité le registre ds pointe sur DSEG1, de tels accès feront référence à des emplacements de mémoire dans DSEG1 au même offset que les variables accédées dans DSEG2. Ce qui va modifier les données dans DSEG1 (ou provoquera la lecture de valeurs incorrectes pour les variables censées de se trouver dans DSEG2).

Pour les programmeurs débutants, la meilleure solution à ce problème est éviter d'utiliser des segments (de données) multiples autant que possible. Remettez-les au jour où vous serez en mesure d'envisager des problèmes de ce genre. En tant que débutant, utilisez seulement un segment de code, un segment de données et un segment de pile en laissant simplement les registres de segments correspondants pointer sur chacun d'eux. La directive *assume* est assez complexe et peut vous mener à des problèmes considérables si vous en abusez. Mieux vaut ne pas s'adonner à des usages extravagants tant que vous ne serez tout à fait à votre aise avec la vision complète de la programmation assembleur et de la segmentation de l'architecture 80x86.

Le mot réservé *nothing* indique à l'assembleur que vous n'avez pas la moindre idée de l'endroit où un registre de segment est en train de pointer. Il lui indique aussi que vous n'aurez accès à aucune donnée relative à ce segment en question, sauf si vous fournissez explicitement un préfixe de segment à une adresse donnée. Une convention de programmation courante est de placer les directives assume avant toute procédure de programme. Puisque dans un programme les pointeurs de segment ne changent que rarement, sauf pour l'entrée et la sortie d'une procédure, voici l'endroit idéal où placer les directives assume :

```
                assume  ds:P1Dseg, cs:cseg, es:nothing
Procedure1      proc    near
                push    ds              ; Préserve ds
                push    ax              ; Préserve ax
                mov     ax, P1Dseg      ; Obtient un pointeur sur P1Dseg dans
                mov     ds, ax          ; le registre ds.
                .
                .
                .
                pop     ax              ; Récupère la valeur de ax
                pop     ds              ; Récupère la valeur de ds
                ret
Procedure1      endp
```

Le seul problème avec ce code est que MASM suppose encore que ds pointe sur P1Dseg dans la portion de code après Procedure1. La meilleure solution est de placer une seconde directive assume après la directive endp, afin d'indiquer à MASM qu'il ne doit rien connaître à propos de la valeur du registre ds :

```
                .
                .
                .
                ret
Procedure1      endp
                assume  ds:nothing
```

Bien que les instructions au-delà de la procédure contiendront probablement une nouvelle directive *assume* indiquant à l'assembleur le nouveau comportement de ds (par exemple au début de la procédure qui suit le code ci-dessus), c'est encore une bonne idée d'adopter cette convention. Si vous ne placez pas une directive assume avant la prochaine procédure de votre fichier source, l'instruction assume ds:nothing empêchera l'assembleur de considérer l'accès à d'autres variables dans P1Dseg.

Les préfixes de surcharge de segment écrasent toujours toute supposition faite par l'assembleur. mov ax, cs:var1 charge toujours le registre ax avec le mot de l'offset var1 dans le segment de code courant, peu importe dans quelles circonstances vos avez défini var1. Le but principal derrière les préfixes de surcharge de segment est le traitement des références indirectes. Si vous avez une instruction de la forme mov ax, [bx], l'assembleur supposera que bx pointe sur le segment de données. Si vous voulez en fait accéder aux données d'un autre segment, vous pouvez utiliser un préfixe de surcharge de segment, par exemple, mov ax, es:[bx].

En général si vous utilisez des segments de données multiples dans votre programme, il vous faudra utiliser la syntaxe complète *segment:offset* pour toutes vos variables. Par exemple, mov ax, DSEG1:I et mov bx, DSEG2:J. Ceci n'élimine pas le besoin de charger les registres de segment ou de faire un bon usage des directives assume, mais rend un programme assurément plus facile à lire et aide aussi MASM à mieux signaler les erreurs.

La directive assume est utile également pour des usages autres que spécifier le segment par défaut. Vous verrez quelques autres applications un peu plus tard dans ce chapitre.

### 8.8.11 Combiner les segments : la directive GROUP

La plupart des segments dans un programme ont une taille inférieure à 64 ko. En réalité c'est même assez difficile d'atteindre cette limite. Quand MS-DOS charge les segments en mémoire, une seule région de 64 ko peut contenir plusieurs segments. Autrement dit, on peut combiner ces segments dans la mémoire en un seul segment. Ceci peut améliorer l'efficacité de votre code s'il permet d'économiser le rechargement des registres de segment pendant l'exécution d'un programme.

Donc, pourquoi pas ne pas le faire ? Comme la prochaine section le montrera, maintenir des segments séparés aide à mieux structurer un programme et à le rendre plus modulaire. Cette modularité est très importante dans les programmes à mesure qu'ils deviennent plus complexes. Comme d'habitude, améliorer la structure et la modularité des programmes peut les rendre moins efficaces. Heureusement, MASM dispose d'une directive, *group*, vous permettant de traiter deux segments comme s'ils étaient un même segment sans devoir renoncer à la structure et à la modularité.

La directive *group* permet de créer un nouveau nom de segment contenant les segments qu'il regroupe ensemble. Par exemple, si vous avez deux segments nommés "Module1Data" et "Module2Data" que vous voulez combiner dans un seul segment physique, utilisez la directive *group* comme suit :

```
moduleData      group   Module1Data,    Module2Data
```

La seule restriction est que la fin du deuxième module ne peut pas être à plus de 64 Ko du début du premier module dans la mémoire. MASM et l'éditeur de liens ne vont *pas* combiner automatiquement ces segments et les placer ensemble dans la mémoire. S'il y a d'autres segments entre ces deux-ci en mémoire, alors le total de tous ces segments ne peut pas dépasser 64 Ko. Pour minimiser ce problème, on peut utiliser l'opérande *class* de la directive segment, afin d'indiquer à l'éditeur de liens de combiner les deux segments en mémoire en utilisant le même nom de classe :

```
ModuleData      group   Module1Data,    Module2Data

Module1Data     segment para public     'MODULES'
```

```
                .
                .
                .
Module1Data     ends
                .
                .
                .
Module2Data     segment byte public     'MODULES'
                .
                .
                .
Module2Data     ends
```

Avec de telles déclarations, vous pouvez utiliser "ModuleData" partout où MASM peut permettre des segments de données, en tant qu'opérande d'une instruction mov, ou en tant qu'opérande d'une directive *assume*, etc. L'exemple suivant illustre cet usage :

```
                assume  ds:ModuleData
Module1Proc     proc    near
                push    ds              ; Préserve la valeur de ds
                push    ax              ; Idem avec ax
                mov     ax, ModuleData  ; Charge ds avec l'adresse du segment
                .
                .
                .
                pop     ax              ;Restauration des valeurs de ax et ds
                pop     ds
                ret
Module1Proc     endp
                assume  ds:nothing
```

Certes, utiliser la directive group de cette façon n'a pas vraiment amélioré le code. Au contraire, en utilisant un nom différent pour le segment de données, on pourrait dire que le code a même été obscurci. Cependant, supposez que vous aviez une séquence devant accéder à des variables dans les deux segments Module1Data et Module2Data. Si ces segments étaient physiquement et logiquement séparés, vous auriez dû charger deux registres de segment avec les adresses de ces deux segments de façon à pouvoir accéder correctement à leurs données respectives. Et ceci vous coûte un préfixe de surcharge de segment dans toutes les instructions accédant à ces segments. Et si vous ne disposiez pas d'un registre de segment en réserve, la situation serait même pire, vous auriez à charger continuellement de nouvelles valeurs dans un seul registre de segment à mesure que vous accédez à des données dans ces deux segments. Vous pouvez éviter cette surcharge en combinant deux segments logiques dans un seul segment physique et y accéder à travers leur groupe au lieu de leurs noms de segments correspondants.

Si vous groupez deux segments ou plus, vous ne faites que créer un pseudo-segment qui contient les segments apparaissant dans les champs des opérandes de la directive group. Grouper des segments ne vous empêche pas d'accéder aux segments individuels de la liste de regroupement. Le code suivant est parfaitement autorisé :

```
                assume  ds:Module1Data
                mov     ax, Module1Data
                mov     ds, ax
                .
<code qui accède aux données de Module1Data>
                .
                assume  ds:Module2Data
                mov ax, Module2Data
                mov ds, ax
                .
<code qui accède aux données de Module2Data>
                .
                assume  ds:ModuleData
                mov     ax, ModuleData
                mov     ds, ax
<code qui accède aux données à la fois dans Module1Data et Module2Data>

                .
                .
```

.

Pendant que l'assembleur traite des segments, il commence à initialiser, pour un segment donné, la valeur du compteur d'emplacement à zéro. Une fois que vous groupez un ensemble de segments, une ambiguïté se produit ; grouper deux segments fait concaténer à MASM et à l'éditeur des liens les variables d'un ou de plusieurs segments à la fin du premier segment de la liste de regroupement. Et ils le font en ajustant les offsets de tous les symboles des segments concaténés comme s'ils étaient tous des symboles du même segment. Et cette ambiguïté existe parce que MASM vous permet de référencer un symbole aussi bien dans son segment que dans le segment groupé. Le symbole a un offset différent selon le choix du segment. Pour résoudre l'ambiguïté, MASM se sert de l'algorithme suivant :

- Si MASM ne sait pas qu'un registre de segment pointe sur le segment du symbole ou sur un groupe qui contient ce segment, alors il génère une erreur.
- Si une directive *assume* associe le nom du segment avec un registre de segment, mais n'associe pas de registre de segment avec le nom du groupe, alors MASM utilise l'offset du symbole à l'intérieur de son segment.
- Si une directive *assume* associe le nom du groupe avec un registre de segment, mais n'associe pas un registre de segment avec le nom de segment du symbole, MASM utilise l'offset du symbole dans le groupe.
- Si une directive *assume* fournit une association de registre de segment à la fois avec le segment du symbole et son groupe, alors MASM sélectionne l'offset qui ne demandera pas un préfixe de surcharge de segment. Par exemple, si la directive *assume* spécifie que ds pointe sur le nom du groupe et que es pointe sur le nom du segment, MASM utilisera l'offset du groupe si le registre de segment par défaut est ds, car ceci lui éviterait d'émettre un opcode de préfixe de surcharge de segment. Si les deux options entrainent l'émission d'un opcode de préfixe de surcharge, alors MASM choisira l'offset (et le préfixe de surcharge de segment), associé avec le segment du symbole.

MASM utilise cet algorithme si on spécifie un nom de variable sans un préfixe de segment. Si vous spécifiez un préfixe de surcharge de registre de segment, alors MASM pourra opter pour un offset arbitraire. Souvent, il s'agit de l'offset du groupe. Par conséquent, la séquence d'instructions qui suit, sans directive assume qui indique à MASM que le symbole BadOffset est dans seg1, peut produire un mauvais code objet :

```
DataSegs        group   Data1, Data2, Data3
                .
                .
                .
Data2           segment
                .
                .
                .
BadOffset       word    ?
                .
                .
                .
Data2           ends
                .
                .
                .
                assume  ds:nothing, es:nothing, fs:nothing, gs:nothing
                mov     ax, Data2       ;Oblige ds à pointer sur data2 en dépit
                mov     ds, ax          ;de la directive assume ci-dessus
                mov     ax, ds:BadOffset ;Pourrait utiliser l'offset de
                                        ;DataSegs au lieu de Data2!
```

Si vous voulez imposer l'offset correct, utilisez le nom de variable contenant la forme d'adressage complète *segment:offset* :

```
; Pour imposer l'usage de l'offset à l'intérieur du groupe DataSegs utilisez
; une instruction comme celle qui suit :

                mov     ax, DataSegs:BadOffset

; Pour imposer l'usage de l'offset dans Data2, utilisez :

                mov     ax, Data2:BadOffset
```

Vous devez faire preuve d'une attention spéciale quand vous travaillez avec des groupes. Si vous obligez MASM à utiliser un offset à l'intérieur d'un segment (ou groupe) particulier et que le segment de registre ne pointe pas sur ce segment ou groupe en question, MASM peut ne pas émettre de message d'erreur et le programme ne s'exécuterait pas correctement. Et lire les offsets que MASM produit sur les listings d'assemblage ne vous aidera pas à trouver l'erreur. MASM affichera *toujours* les offsets à l'intérieur du segment du symbole dans son listing. La seule manière de vraiment détecter que MASM et l'éditeur de liens sont en train d'utiliser des offsets incorrects, et de recourir à un débogueur comme CodeView et de regarder les offsets que l'éditeur de liens et le module de chargement ont produits.

### 8.8.12 Pourquoi se tracasser avec les segments ?

Après avoir lu les paragraphes précédents vous vous demanderez qu'est-ce qu'il peut y avoir de positif avec l'usage des segments. Pour être parfaitement francs, si vous utilisez le ficher SHELL.ASM comme squelette de vos programmes, vous pouvez parfaitement procéder sans vous préoccuper de segments, de groupes, de préfixes de surcharge et de noms complets *segment:offset*. En tant que débutant en programmation assembleur, c'est probablement une bonne idée d'ignorer une grande partie de toute la discussion qui précède sur la segmentation, et de laisser cette matière pour le jour où vous aurez beaucoup plus d'expertise. Il y a cependant trois raisons pour lesquelles il vaut bien la peine d'approfondir cette matière : la limitation des segments de 64 Ko, la programmation modulaire et l'interface avec les langages de haut niveau.

En fonctionnant en mode réel, les segments peuvent avoir une longueur maximale de 64 Ko. Si on veut avoir accès à plus de 64 Ko de données dans les programmes, on aura besoin d'utiliser plus d'un segment. Ce fait, plus que toute autre raison, a porté les programmeurs (malgré leurs réticences et leurs récriminations) dans le monde de la segmentation. Malheureusement, c'est tout ce que beaucoup de programmeurs apprennent sur la segmentation. Ils n'apprennent que rarement quelque chose de plus de ce qui est à peine suffisant à pouvoir accéder à plus de 64 Ko de données. Comme résultat, à l'occurrence du premier problème de segmentation, - dû à leur manque de connaissance de la segmentation, ils condamnent la segmentation à cause de leurs problèmes et ils essayent de l'éviter autant que possible.

C'est vraiment dommage, parce que la segmentation est un moyen puissant de gérer la mémoire, permettant d'organiser un programme en entités logiques (segments) qui sont, en théorie, indépendantes les unes des autres. Le domaine de l'ingénierie informatique étudie justement comment écrire correctement de gros programmes. La modularité et l'indépendance entre les entités logiques d'un programme constituent deux des principaux outils que les ingénieurs informatiques utilisent pour mettre en place de gros programmes qui sont corrects et faciles à maintenir. La famille 80x86 fournit, du point de vue matériel, les moyens d'implémenter la segmentation. Sur d'autres types de processeur, la segmentation est surtout réalisée par le côté logiciel. Il en résulte donc qu'il est plus facile de travailler avec les segments de l'architecture 80x86.

Bien que ce livre ne s'attarde pas sur la programmation en mode protégé, il vaut la peine de signaler que dans ce mode le matériel peut, en fait, empêcher un module d'accéder aux données d'un autre module (en effet, le terme "mode protégé" veut dire que les segments sont protégés contre les accès illégaux). Beaucoup de débogueurs disponibles sur MS-DOS fonctionnent en mode protégé, permettant de détecter les violations des limites des tableaux et des segments. Soft-ICE et Bounds Checker de la société NuMega sont des exemples de ces produits. Beaucoup de gens qui ont travaillé avec la segmentation dans un environnement en mode protégé (par exemple, OS/2 ou Windows) apprécient les bénéfices qu'offre la segmentation.

Une autre raison d'étudier la segmentation est que vous pourriez avoir besoin d'écrire une routine qu'un programme écrit dans un langage de haut niveau peut appeler. Puisque les compilateurs de haut niveau font certaines suppositions sur l'organisation des segments en mémoire, il vous faudra connaître un petit peu de segmentation afin de pouvoir écrire de tels codes.

## 8.9 La directive END

Cette directive met fin à un programme. En plus de signaler à MASM qu'il a atteint la fin du fichier source, l'opérande optionnelle indique à MS-DOS où transférer le contrôle quand le programme commence son exécution ; c'est-à-dire, vous spécifiez le nom de la procédure principale comme opérande de la directive end. Si celle-ci n'est pas mentionnée, MS-DOS commence l'exécution à partir du premier octet du fichier .exe. Puisque

c'est souvent peu pratique de garantir que votre programme commence avec le premier octet de code objet du fichier .exe, il vaut mieux spécifier l'emplacement de départ d'un programme à l'aide de la directive end. Si vous utilisez SHELL.ASM comme squelette de vos programmes, vous remarquerez que la procédure main est spécifiée comme point de départ du programme.

Si vous utilisez des assemblages séparés et vous joignez divers codes objets différents (voir le paragraphe 8.20), un seul module peut avoir un programme principal. De même, un seul module peut être le point de départ du programme. Si vous spécifiez plus d'un emplacement, vous tromperez l'éditeur de liens, qui générera une erreur.

## 8.10 Variables

Les déclarations des variables globales utilisent les pseudo-opcodes byte/sbyte/db, word/sword/dw, dword/sdword/dd, qword/dq et tbyte/dt. Bien que vous pouvez placer vos variables dans tout segment (y compris le segment de code), il est d'usage chez les débutants de placer toutes leurs variables globales dans un seul segment de données.

Une déclaration typique de variable a la syntaxe suivante :

```
nomvariable     byte    valeur_initiale
```

où *nomvariable* est le nom de la variable que vous êtes en train de déclarer et valeur_initiale est la valeur initiale que vous voulez donner à cette variable lors du début de l'exécution du programme. "?" est une valeur initiale spéciale et veut dire que vous ne voulez pas encore donner à la variable une valeur. Quand DOS charge en mémoire un programme contenant une telle variable il ne l'initialise pas.

La déclaration ci-dessus réserve un octet de mémoire. Mais cette déclaration pourrait être aussi de tout autre type, simplement en changeant le pseudo-opcode *byte* par tout autre pseudo-opcode approprié.

Dans la plupart des circonstances, ce livre supposera que vous déclarerez toutes vos variables dans le segment de données, autrement dit, un segment pointé par le registre ds. En particulier, la plupart des programmes que nous verrons placeront toutes leurs variables dans le segment DSEG (CSEG est pour le code, DSEG pour les données et SSEG pour la pile). Regardez le programme SHELL.ASM du chapitre 4 pour plus de détails sur ces segments.

Puisque le chapitre 5 couvre en profondeur la déclaration des variables, les types de données, les structures, les tableaux et les pointeurs, ce chapitre ne s'attardera pas davantage sur ce sujet.

## 8.11 Types étiquette

Une caractéristique inusuelle des syntaxes des assembleurs d'Intel (comme MASM) est qu'ils sont *fortement typés*. Un assembleur fortement typé associe un certain type de données avec des symboles déclarés précédemment dans le code source et il génère un avertissement ou une erreur si vous utilisez un symbole donné dans un contexte qui n'admet pas son type particulier. Même si en assembleur c'est inusuel, la plupart des langages de haut niveau appliquent certaines règles de typage aux symboles déclarés dans le fichier source. Par exemple, Pascal est connu pour être un langage fortement typé. En Pascal, vous ne pouvez pas assigner une chaîne de caractères à une variable numérique ou essayer d'attribuer un entier à un nom de procédure. En concevant la syntaxe pour l'assembleur 8086, Intel a décidé que toutes les raisons qui s'appliquent à un langage fortement typé s'appliquent aussi pour l'assembleur. Par conséquent, la syntaxe standard des assembleurs 80x86 comme MASM, impose l'usage de symboles représentant des types spécifiques de données dans vos programmes en assembleur.

### 8.11.1 Comment attribuer un type particulier à un identificateur

Les identificateurs, dans un contexte 80x86, peuvent être d'un type de données parmi le huit types de base : byte, word, dword, qword, tbyte, near, far et abs (constante)[12]. Toutes les fois que vous définissez un identificateur avec l'un des pseudo-opcodes byte, word, dword, qword ou tbyte, MASM associe le type du

[12]MASM supporte aussi un type FWORD. Ce type est pour la programmation 32 bits en mode protégé.

pseudo-opcode avec l'identificateur. Par exemple, la déclaration de variable suivante créera un symbole de type byte :

```
BVar            byte    ?
```

De même, l'instruction qui suit définit une variable dword :

```
DWVar           dword   ?
```

Les types de variables ne sont pas limités aux types primitifs définis par MASM. Si vous créez vos propres types en utilisant typedef ou struct, MASM associera ces types avec toute déclaration de variables compatible avec ces types.

Vous pouvez définir des symboles near (appelés aussi étiquettes d'instruction) de deux façons différentes. D'abord, tous les symboles de procédure déclarés avec la directive proc (qui peut être accompagnée ou non du mot-clé *near*[13]) sont des symboles near. Les étiquettes d'instruction sont aussi des symboles near. Par exemple :

```
etiquette:      instr
```

instr représente une instruction[14]. Notez qu'un caractère deux-points doit suivre le symbole. Il n'en fait pas partie et sert à spécifier à MASM qu'il s'agit d'une étiquette et doit être traité comme un symbole de type *near*. Les étiquettes servent souvent comme cibles aux instructions de saut[15] ou aux boucles. Par exemple, considérez la séquence de code suivante :

```
                mov     cx, 25
Loop1:          mov     ax, cx
                call    PrintInteger
                loop    Loop1
```

L'instruction *loop* de la dernière ligne, décrémente le registre cx et transfère le contrôle à l'instruction étiquetée avec Loop1, tant que cx n'atteint pas zéro.

À l'intérieur d'une procédure, les étiquettes d'instruction sont *locales*. Autrement dit, une étiquette est visible seulement pour le code de la procédure qui la contient. Cette étiquette a alors une *portée de procédure*. Si vous voulez rendre global un identificateur d'une procédure, il vous suffira de placer deux caractères deux-points après le nom de l'étiquette. Par exemple, si dans le code ci-dessus vous aviez à faire référence à l'étiquette Loop1 en dehors de la procédure qui la contient, il vous suffirait d'écrire :

```
                mov     cx, 25
Loop1::         mov     ax, cx
                call    PrintInteger
                loop    Loop1
```

Généralement, les symboles *far* sont les cibles d'instructions jump et call. La méthode la plus commune que les programmeurs utilisent pour créer une étiquette de ce genre est de placer le mot-clé *far* comme opérande de la directive proc. Les symboles qui sont simplement des constantes, sont normalement déclarés avec la directive *equ*. Vous pouvez également déclarer des symboles avec différents types en utilisant equ et les directives *extrn/ extern/externdef*. Une explication des directives extern paraît au paragraphe 8.20.

Si vous déclarez une constante numérique à l'aide d'un "equate", MASM lui assignera le type *abs* (*absolute* ou constante). Les affectations de type constante texte et chaîne de caractère reçoivent le type *text*. Vous pouvez également associer à un symbole un type arbitraire à l'aide de la directive *equ* (voir le paragraphe 8.12.4).

---

[13] Notez que si vous utilisez la version simplifiée de la directive proc, à savoir l'omission du mot-clé *proc*, ceci ne veut pas nécessairement dire que cette procédure sera *near*. Si votre programme ne contient pas la directive ".MODEL", cependant, l'omission du mot-clé *near* implique effectivement que la procédure est de ce type.

[14] L'instruction qui suit une étiquette est optionnelle. En cas d'omission, MASM associera simplement le compteur d'emplacement de l'instruction suivante avec cette étiquette.

[15] D'ailleurs les équivalents de l'instruction goto autant critiquée, sont extrêmement communs dans la programmation assembleur. Toutes les boucles des langages de haut niveau sont construites avec jump/goto. Selon l'avis du traducteur, condamner goto n'est autre chose qu'un préjugé. Bien plus, même dans un langage de haut niveau, l'instruction goto a parfois des applications très intéressantes qui n'ôtent rien à la clarté d'un programme. Donc, c'est l'abus de goto qu'il faut condamner, pas son usage, n.d.t.

### 8.11.2 Valeurs des identificateurs

Quand vous définissez un identificateur, à l'aide d'une directive ou d'un pseudo-opcode, MASM lui attribue un type et une valeur. La valeur que normalement MASM donne à l'identificateur est généralement la valeur courante du compteur d'emplacement. Si vous définissez l'identificateur avec un (equate), l'opérande de cette affectation spécifie normalement la valeur de ce dernier. A l'occurrence d'une étiquette dans un champ d'opérande, MASM substitue la valeur de l'étiquette à celle-ci.

### 8.11.3 Conflits de type

Puisque l'architecture 80x86 supporte des symboles fortement typés, la prochaine question à se poser est : "Dans quel but sont-ils utilisés ?". En un mot, des symboles fortement typés peuvent aider à vérifier le fonctionnement correct de vos programmes. Considérez les sections de code qui suivent :

```
DSEG            segment public 'DATA'
                .
                .
                .
I               byte    ?
                .
                .
                .
DSEG            ends

CSEG            segment public 'CODE'
                .
                .
                .
                mov     ax, I
                .
                .
                .
CSEG            ends
                end
```

L'instruction mov de cet exemple essaie de charger le registre ax (16 bits) avec une variable de type octet. Certes, un microprocesseur 80x86 est parfaitement capable d'effectuer cette opération ; il chargerait le registre al avec l'emplacement de mémoire associé à la variable I et le registre ah avec l'emplacement de mémoire immédiatement successif à I (qui est probablement l'octet le moins significatif d'une autre variable donnée). Cependant, ceci ne correspond probablement pas à ce que vous attendiez. La personne qui a écrit ce code a vraisemblablement oublié que la variable I était de type byte et supposait que c'était un type word. Ce qui est définitivement une erreur dans la logique du programme.

MASM ne permet jamais qu'une telle instruction soit assemblée sans génération d'un message de diagnostic. Ceci vous aide à trouver les erreurs dans vos programmes, des erreurs qui seraient autrement difficiles à détecter. A l'occasion, des programmeurs avancés peuvent volontairement vouloir produire une instruction de la sorte et à cet effet MASM fournit des opérateurs de coercition qui permettent de se passer des mécanismes de vérification des erreurs, permettant ainsi d'effectuer en toute liberté des opérations non conformes aux règles (consulter le paragraphe 8.12.3).

## 8.12 Expressions d'adresse

Une *expression d'adresse* est une expression algébrique qui produit un résultat numérique que MASM fusionne dans le champ *déplacement* d'une instruction. Une constante entière est probablement l'exemple le plus simple d'expression d'adresse. L'assembleur n'a qu'à substituer la valeur de la constante numérique à l'opérande spécifiée. Par exemple, l'instruction suivante remplit le champ de donnée immédiate de l'instruction mov avec des zéros :

```
        mov     ax, 0
```

Un autre exemple simple de mode d'adressage est constitué par un symbole. A l'occurrence de celui-ci, MASM substitue la valeur de ce symbole. Par exemple, les deux instructions suivantes sont susceptibles d'émettre le même code objet de l'instruction qu'on vient de voir :

```
Value   equ     0
        mov     ax, Value
```

Une expression d'adresse, cependant, peut être beaucoup plus complexe que ce qu'on vient de voir. Vous pouvez vous servir de divers opérateurs arithmétiques et logiques pour modifier la valeur de base de symboles ou constantes.

Gardez à l'esprit que MASM calcule les expressions d'adresse pendant l'assemblage et non durant l'exécution. Par exemple, l'instruction suivante ne charge pas ax avec la valeur de Var avant de l'incrémenter :

```
        mov     ax, Var+1
```

Cette instruction charge le registre al avec l'octet qui se trouve dans l'adresse Var + 1 et ensuite charge ah avec l'octet qui se trouve à l'adresse de Var + 2.

Les débutants confondent souvent l'arithmétique effectuée pendant l'assemblage et l'arithmétique effectuée pendant l'exécution. Gardez à l'esprit que MASM calcule toutes les adresses des expressions pendant l'assemblage !

### 8.12.1 Types de symboles et modes d'adressage

Considérez l'instruction suivante :

```
        jmp     Emplacement
```

Selon la façon dont l'étiquette Emplacement est définie, l'instruction jmp exécute l'une des diverses opérations possibles. Si vous jetez un coup d'œil de nouveau au chapitre 6, vous allez remarquer que l'instruction jmp a différents formats. Come aide-mémoire, ils sont :

```
        jmp     label           (short)
        jmp     label           (near)
        jmp     label           (far)
        jmp     reg             (near indirect, à travers un registre)
        jmp     mem/reg         (idem, à travers la mémoire)
        jmp     mem/reg         (far indirect, par la mémoire)
```

Notez que MASM utilise le même terme (jmp) pour chaque instruction ; comment fait-il pour les distinguer ? Le secret est révélé par l'opérande. Si l'opérande est une étiquette à l'intérieur du segment courant, l'assembleur sélectionne l'un des deux premiers formats, selon la distance de l'instruction cible. Si l'opérande est une étiquette située dans un autre segment, alors l'assembleur choisira la version far. Si l'opérande qui suit l'instruction jmp est un registre, alors MASM se sert du jmp near indirect et le programme saute à l'adresse indiquée par le registre. S'il s'agit d'une variable, alors l'assembleur se comporte d'une des deux façons suivantes :

- Un saut NEAR, si la variable a été déclarée avec word/sword/dw
- Un saut FAR, si la variable a été déclarée avec dword/sdword/dd

Une erreur résulte si vous avez utilisé byte/sbyte/db, qword/dq ou tbyte/dt ou tout autre type.

Si vous avez spécifié une adresse indirecte, par exemple jmp [bx], l'assembleur lancera aussi un message d'erreur, car il ne pourra pas déterminer si [bx] est en train de pointer sur une variable word ou dword. Pour savoir comment spécifier la taille, voyez la section sur la coercition, dans ce même chapitre.

### 8.12.2 Opérateurs logiques et arithmétiques

MASM reconnaît divers opérateurs arithmétiques et logiques. Le tableau suivant en fournit une liste :

Tableau 36 : Opérateurs Arithmétiques

| Opérateur | Syntaxe | Description |
|---|---|---|
| + | *+expr* | Positif (et unaire) |
| - | *-expr* | Négatif (idem) |

| | | |
|---|---|---|
| + | *expr* + *expr* | Addition |
| - | *expr* - *expr* | Soustraction |
| * | *expr* * *expr* | Multiplication |
| / | *expr* / *expr* | Division |
| MOD | *expr* MOD *expr* | Modulo (reste) |
| [ ] | *expr[expr]* | Addition (opérateur d'index) |

Tableau 37 : Opérateurs Logiques

| Opérateur | Syntaxe | Description |
|---|---|---|
| SHR | *expr* SHR *expr* | Décalage à droite |
| SHL | *expr* SHL *expr* | Décalage à gauche |
| NOT | NOT *expr* | Négation logique binaire (bit à bit) |
| AND | *expr* AND *expr* | ET logique |
| OR | *expr* OR *expr* | OU logique |
| XOR | *expr* XOR *expr* | OU exclusif logique |

Tableau 38 : Opérateurs Relationnels

| Opérateur | Syntaxe | Description |
|---|---|---|
| EQ | *expr* EQ *expr* | Vrai (0FFh) si égal, faux (0) dans le cas contraire |
| NE | *expr* NE *expr* | Vrai (0FFh) si non égal, faux (0) sinon |
| LT | *expr* LT *expr* | Vrai (0FFh) si inférieur, faux (0) sinon |
| LE | *expr* LE *expr* | Vrai (0FFh) si inférieur ou égal, faux (0) sinon |
| GT | *expr* GT *expr* | Vrai (0FFh) si supérieur, faux (0) autrement |
| GE | *expr* GE *expr* | Vrai (0FFh) si supérieur ou égal, faux (0) sinon |

Vous ne devez pas confondre ces opérateurs avec les instructions 80x86 ! L'opérateur d'addition additionne deux valeurs et leur somme devient une opérande d'instruction. Cette addition est effectuée pendant l'assemblage du programme, non pendant son exécution. Pour effectuer une addition pendant l'exécution, vous devez utiliser des instructions comme add ou adc.

Vous vous demanderez sûrement « À quoi bon ces opérateurs ? » En vérité on ne les utilise pas beaucoup. L'opérateur d'addition est utilisé assez souvent, la soustraction parfois, la comparaison rarement et tout le reste encore moins. Puisque l'addition et la soustraction sont les seuls opérateurs que les débutants emploient régulièrement, cette exposition considérera seulement ces deux opérateurs et fera mention des autres au fur et à mesure qu'il sera nécessaire dans le reste de ce livre.

L'opérateur d'addition admet deux syntaxes : expr+expr et expr[expr]. Par exemple, l'instruction suivante charge l'accumulateur avec le contenu de COUNT mais de l'emplacement de mémoire suivant :

```
mov     al, COUNT+1
```

L'assembleur, à l'occurrence de cette instruction, calculera la somme de l'adresse de COUNT et 1. La valeur résultante, sera l'adresse mémoire de l'instruction. Comme vous vous rappelez, l'instruction mov ax, mem a une longueur de trois octets et a la structure suivante :

Opcode | Octet le moins significatif de déplacement | Octet le plus significatif de déplacement

Les deux octets de déplacement de cette instruction contiennent la somme COUNT+1.

L'opérateur d'addition de la forme expr[expr] sert à accéder à des éléments de tableaux. Si AnyData est un symbole représentant l'adresse du premier élément, AnyData[5] représente l'adresse du sixième octet de AnyData. L'expression AnyData+5 produit le même résultat et les deux versions peuvent être utilisée de façon interchangeable, cependant, dans un contexte de tableau la première version est plus parlante. Un piège à éviter : expr1[expr2][expr3], ne produit pas l'indexage automatique d'un tableau à deux dimensions, mais calcule simplement la somme expr1+expr2+expr3.

L'opérateur de soustraction fonctionne tout comme l'opérateur d'addition, sauf qu'il effectue une différence. Cet opérateur deviendra très important en matière de variables locales (comme on verra au chapitre 11).

Faites attention quand vous utilisez plusieurs symboles dans une expression d'adresse. MASM restreint les opérations d'addition et de soustraction que l'on peut effectuer sur les symboles et permet seulement les formats déterminés que voici :

| Expression : | Type résultant : |
|---|---|
| reloc + const | Déplacer à l'adresse spécifiée |
| reloc - const | Même chose |
| reloc - reloc | Constante dont la valeur est le nombre d'octets entre la première et la seconde opérande. Les deux variables doivent physiquement apparaître dans le même segment du fichier source courant. |

*reloc* veut dire "relocatable symbol or expression" (symbole ou expression déplaçable). Ceci peut être un nom de variable, une étiquette d'instruction, un nom de procédure ou tout autre symbole associé à une adresse de mémoire. Il peut aussi être une expression produisant un résultat déplaçable. MASM ne permet pas autre opération que l'addition ou la soustraction sur des expressions dont le résultat est transférable. Vous ne pouvez pas, par exemple, calculer le produit de deux symboles déplaçables.

Les deux premières formes sont très communes dans les programmes assembleur. Une expression d'adressage de ce type consistera souvent en un seul symbole, transférable, symbole et une seule constante (par exemple, "var + 1"). Vous n'utiliserez pas la troisième forme très souvent, mais elle est très utile parfois. Vous pouvez utiliser cette forme d'expression d'adresse pour calculer la distance, en octets, entre deux points de votre programme. Le symbole procsize, dans le code suivant, calcule la taille de Proc1 :

```
Proc1           proc    near
                push    ax
                push    bx
                push    cx
                mov     cx, 10
                lea     bx, SomeArray
                mov     ax, 0
ClrArray:       mov     [bx], ax
                add     bx, 2
                loop    ClrArray
                pop     cx
                pop     bx
                pop     ax
                ret
Proc1           endp

procsize        =       $ - Proc1
```

"$" est un symbole spécial que MASM utilise pour indiquer l'offset courant à l'intérieur du segment (c'est-à-dire le compteur d'emplacement). C'est un symbole déplaçable, tout comme Proc1, donc l'égalité ci-dessus calcule la différence entre l'offset du début de Proc1 et la fin de Proc1. C'est la longueur de Proc1, en octets.

Les opérandes des opérateurs autres que l'addition ou la soustraction doivent être des constantes on des expressions contenant une constante (par exemple, "$-Proc1" ci-dessus produit une valeur constante). Vous utiliserez ces autres opérateurs surtout dans des macros ou dans les directives d'assemblage conditionnel.

### 8.12.3 Coercition

Considérez le segment suivant :

```
DSEG            segment public 'DATA'
I               byte    ?
J               byte    ?
DSEG            ends

CSEG            segment
                .
                .
                .
                mov     al, I
                mov     ah, J
                .
                .
                .
CSEG            ends
```

Puisque I et J sont adjacents, il n'y a pas besoin d'utiliser deux instructions mov pour charger al et ax, un simple mov ax, I ferait le même boulot. Malheureusement l'assembleur vous arrête, parce que I est un octet et une erreur se produit si vous essayez de l'utiliser comme un mot. Comme vous voyez donc, il y a des occasions où il sera nécessaire de traiter un octet comme un mot (ou bien un mot comme un double-mot, etc.).

Le fait de changer temporairement le type d'un identificateur s'appelle *coercition de type*. Des expressions peuvent être forcées à un type différent avec l'opérateur de MASM *ptr*. Vous l'utiliserez comme suit :

```
        type PTR expression
```

*Type* peut être un octet, un mot et ainsi de suite et *expression* est toute expression générale représentant l'adresse d'un objet. L'opérateur de coercition retourne une expression ayant la même valeur que *expression*, mais avec le type spécifié par *type*. Pour résoudre le genre de problème ci-dessus, on utilise l'instruction suivante :

```
                mov     ax, word ptr I
```

Ceci invite l'assembleur à émettre un code qui charge l'adresse ax avec le mot situé à l'adresse représentée par I. Ce qui aura pour effet de charger al avec I et ah avec J.

Le code utilisant des valeurs double-mot utilise assez fréquemment l'opérateur de coercition de type. Puisque *lds* et *les* sont les seules instructions de 32 bits qu'on peut trouver sur des processeurs inférieurs au 80386, on ne peut pas (sans coercition) placer une valeur entière dans une variable de 32 bits par l'instruction mov sur ces vieux CPUs. Si vous avez déclaré la variable DBL avec le pseudo-opcode dword une, instruction comme *mov DBL, ax* ne fonctionnera pas parce que les opérandes sont de tailles différentes. Donc, placer des valeurs dans des variables de 32 bits requiert l'utilisation de l'opérateur ptr. Le code suivant montre comment charger les registres ds et bx dans la variable DBL de type dword :

```
        mov     word ptr DBL, bx
        mov     word ptr DBL+2, ds
```

Puisque beaucoup d'appels au DOS et à la bibliothèque standard retournent des valeurs de 32 bits dans une paire de registres, vous utiliserez souvent cette technique.

**Mise en garde :** si vous forcez l'instruction jmp à effectuer un saut éloigné sur une étiquette proche, votre programme - à part la perte de performance, due au fait qu'un saut éloigné emploie plus de temps à s'exécuter - *fonctionnera*. Mais, si vous forcez une instruction *call* à effectuer un appel éloigné sur une sous-routine proche, alors vous aurez des problèmes. Il ne faut pas oublier que les appels éloignés poussent le registre cs dans la pile (avec l'adresse de retour). En exécutant un *ret near*, l'ancienne valeur de cs ne sera pas récupérée de la pile. Les instructions *pop* ou *ret* qui suivent immédiatement ne fonctionneront pas correctement, car elles retireront de la pile la valeur de cs au lieu de la valeur attendue[16].

[16] La situation, quand on force un appel proche sur une sous-routine éloignée, est même pire. Voir les exercices pour plus de détails.

La coercition peut s'avérer pratique dans beaucoup d'occasions, voire parfois essentielle. Cependant, il faut pas en abuser, car la vérification des types des données est un outil de débogage puissant de MASM. En utilisant la coercition, vous annulez les moyens de protection fournis par l'assembleur. Par conséquent, soyez toujours vigilant avec l'opérateur ptr.

Un autre contexte où la coercition se révèle utile est avec l'instruction *mov mem, imm*. Considérez l'instruction suivante :

```
mov     [bx], 5
```

L'assembleur ne peut pas déterminer sur quel type bx est en train de pointer[17]. La valeur de l'opérande immédiate n'est d'aucune utilité. Même si tout d'abord 5 apparaît comme une donnée d'un octet, elle pourrait toutefois représenter aussi 0005h (word) ou 00000005h (dword). Par conséquent, si vous essayez d'assembler cette instruction, vous obtiendrez une erreur, car la taille de l'opérande est obligatoire. Vous pouvez facilement le faire avec les opérateurs byte ptr, word ptr ou dword ptr, comme suit :

```
mov     byte ptr [bx], 5        ;Pour une variable byte
mov     word ptr [bx], 5        ;Pour une variable word et
mov     dword ptr [bx], 5       ;Pour une variable dword
```

Certains programmeurs paresseux tendent à se plaindre d'avoir à écrire "word ptr" ou "far ptr" ; c'est trop long. Ne serait-il pas mieux si Intel avait choisi un seul caractère au lieu de ces longues phrases ? Bon, assez de se plaindre et rappellez-vous plutôt de la directive textequ. Avec cette directive on peut attribuer à de longues phrases comme "word ptr" un plus court symbole. Vous trouverez des affectations (*equates*) semblables à celles qui suivent dans beaucoup de programmes, dont certains de ce livre :

```
byp     textequ         <byte ptr>      ;Car "bp" est un mot réservé !
wp      textequ         <word ptr>
dp      textequ         <dword ptr>
np      textequ         <near ptr>
fp      textequ         <far ptr>
```

Pour ensuite utiliser des instructions comme :

```
mov     byp [bx], 5
mov     ax, wp I
mov     wp DBL, bx
mov     wp DBL+2, ds
```

### 8.12.4 Opérateurs de type

L'opérateur de coercition "xxxx ptr" est un exemple d'opérateur de type. Les expressions de MASM possèdent deux attributs principaux : une valeur et un type. Les opérateurs arithmétiques, logiques ou relationnels changent la valeur d'une expression. Les opérateurs de type changent son type. La section précédente a montré comment l'opérateur ptr peut changer le type d'une expression. Mais il y a d'autres opérateurs de type :

Tableau 39: Opérateurs de type

| Opérateur | Syntaxe | Description |
|---|---|---|
| PTR | byte ptr *expr*<br>word ptr *expr*<br>dword ptr *expr*<br>qword ptr *expr*<br>tbyte ptr *expr*<br>near ptr *expr*<br>far ptr *expr* | Force l'expression à pointer sur un type byte<br>Force l'expression à pointer sur un type word<br>Force l'expression à pointer sur un type dword<br>Force l'expression à pointer sur un type qword<br>Force l'expression à pointer sur un type tbyte<br>Force l'expression sur une valeur near<br>Force l'expression sur une valeur far |
| short | short *expr* | *expr* doit être à ±128 octets de l'instruction jmp |

[17]Maintenant, on peut utiliser la directive assume pour indiquer à MASM sur quel type de donnée bx doit pointer. Consultez *MASM Reference Manual* pour plus de détails.

| | | |
|---|---|---|
| | | courante. L'opérateur force l'instruction à être longue de deux octets (si possible) |
| `this` | this *type* | Retourne une expression du type spécifié dont la valeur est la valeur du compteur d'emplacement courant. |
| `seg` | seg *étiquette* | Retourne la partie segment de l'adresse d'une étiquette. |
| `offset` | offset *étiquette* | Retourne la partie offset de l'adresse d'une étiquette. |
| `.type` | type *étiquette* | Retourne un octet indiquant si le symbole est une variable, une étiquette ou un nom de structure. Obsolète, il est remplacé par opattr. |
| `opattr` | opattr *étiquette* | Retourne une valeur de 16 bits donnant des informations à propos d'un identifiant. |
| `length` | length *variable* | Retourne le nombre d'éléments d'un tableau à une dimension. S'il s'agit d'un tableau multidimensionnel, l'opérateur retourne le nombre d'éléments de la première dimension. |
| `lengthof` | lengthof *variable* | Retourne le nombre d'éléments d'une variable tableau. |
| `type` | type *symbole* | Retourne une expression dont le type est le même que *symbole* et dont la valeur est la taille, en octets, du symbole spécifié. |
| `size` | size *variable* | Retourne le nombre d'octets alloués pour une variable de tableau à une seule dimension. Obsolète, il est remplacé par sizeof. |
| `sizeof` | sizeof *variable* | Retourne la taille, en octets, d'une variable tableau. |
| `low` | low *expr* | Retourne l'octet le moins significatif d'une expression. |
| `lowword` | lowword *expr* | Retourne le mot le moins significatif d'une expression. |
| `high` | high *expr* | Retourne l'octet le plus significatif d'une expression. |
| `highword` | highword *expr* | Retourne le mot le plus significatif d'une expression. |

L'opérateur short fonctionne exclusivement avec l'instruction jmp. Rappellez-vous qu'il y a deux instructions jump near directes, la première ayant une plage de 128 octets autour du jmp et l'autre ayant une plage de 32768 octets autour du jmp. MASM génère automatiquement un jmp proche si l'adresse cible se trouve jusqu'à 128 octets de distance de l'instruction courante. Cet opérateur est présent surtout pour des questions de compatibilité avec les anciennes versions de MASM (avant la version 6.0).

L'opérateur this forme une expression du type spécifié et dont la valeur est le compteur d'emplacement courant. Par exemple, l'instruction *mov bx, this word*[18], charge le registre bx avec la valeur 8B1Eh, l'opcode de l'instruction *mov bx, mem*. L'adresse *this word* est donc l'adresse même de l'opcode de cette instruction ! Vous utilisez la plupart de temps l'opérateur this en conjonction avec la directive equ, pour donner à un symbole un type autre que constant. Par exemple, considérez l'instruction suivante :

```
HERE            equ     this near
```

[18] En pratique, cette instruction place dans bx la valeur, de type word, du compteur d'emplacement courant, n.d.t.

Cette instruction affecte la valeur courante du compteur d'emplacement au symbole HERE et lui attribue le type near. Ceci, sans doute, aurait pu se faire beaucoup plus facilement en plaçant l'étiquette HERE:. Cependant, l'opérateur this avec la directive equ a certaines applications utiles, considérez ce qui suit :

```
WArray          equ     this word
BArray          byte    200 dup (?)
```

Dans cet exemple, le symbole BArray est de type byte. Par conséquent, des instructions ayant accès à BArray doivent toutes contenir des opérandes de type byte. MASM signalerait l'instruction mov ax, BArray+8 comme une erreur. Néanmoins, utiliser le symbole WArray vous permet d'accéder exactement aux mêmes emplacements de mémoire (étant donné que WArray a la valeur du compteur d'emplacement immédiatement avant le pseudo-opcode byte) donc mov ax, WArray+8 accède à l'emplacement BArray+8. Notez que les deux instructions suivantes sont identiques :

```
        mov     ax, word ptr BArray+8
        mov     ax, WArray+8
```

L'opérateur seg fait deux choses. D'abord il extrait la portion segment de l'adresse spécifiée, ensuite, il convertit le type de l'expression spécifiée en une constante. Une instruction comme mov ax, seg symbol charge toujours l'accumulateur avec la constante correspondant à la portion segment de l'adresse de *symbol*. Si le symbole est un nom de segment, MASM remplace automatiquement ce nom par l'adresse de paragraphe du segment. Cependant, c'est parfaitement légitime d'utiliser aussi l'opérateur seg. Si dseg est le nom d'un segment, alors les deux instructions suivantes sont identiques :

```
        mov     ax, dseg
        mov     ax, seg dseg
```

L'opérateur offset fonctionne tout comme seg, avec la différence qu'il retourne la partie offset de l'expression spécifiée. Si VAR1 est une variable de type word, alors mov ax, VAR1 chargera toujours dans le registre ax les deux octets à l'adresse spécifiée par VAR1. Alors que l'instruction mov ax, offset VAR1, charge l'offset de VAR1 dans le registre ax. Notez que pour charger l'adresse d'une variable dans un registre de 16 bits, vous pouvez utiliser soit l'instruction lea, soit l'instruction mov avec l'opérateur offset. Les deux instructions suivantes chargent toutes deux bx avec l'adresse de la variable J :

```
        mov     bx, offset J
        lea     bx, J
```

L'instruction lea est plus flexible, car on peut spécifier tout mode d'adressage, alors que l'opérateur offset ne permet qu'un seul symbole (c'est-à-dire, le mode d'adressage déplacement seul). Beaucoup de programmeurs utilisent la version mov pour des variables scalaires et la version lea pour d'autres modes d'adressage. Ceci parce que sur les processeurs anciens, la version mov était plus rapide.

Un usage très commun des opérateurs seg et offset est d'initialiser un registre de segment et un registre de pointeur avec l'adresse segmentée d'un objet donné. Par exemple, pour charger es:di avec l'adresse de SomeVar, on peut utiliser le code suivant :

```
        mov     di, seg SomeVar
        mov     es, di
        mov     di, offset SomeVar
```

Puisque vous ne pouvez pas charger directement une constante dans un registre de segment, le code ci-dessus copie la portion segment de l'adresse dans di et ensuite il copie di dans es avant de copier l'offset dans di. Ce code utilise le registre di pour copier la portion segment dans es, et ainsi utiliser le moins de registres possible.

L'opérateur opattr retourne une valeur de 16 bits fournissant des informations spécifiques sur l'expression qui le suit. L'opérateur .type est une ancienne version de opattr qui retourne les huits bits les moins significatifs de cette valeur. Chaque bit a la signification suivante :

Tableau 40: Valeur de retour de l'opérateur OPATTR/.TYPE

| Bit(s) | Signification |
|---|---|
| 0 | S'il vaut 1, il s'agit d'une étiquette dans le segment de code. |

| | |
|---|---|
| 1 | S'il vaut 1, il s'agit d'une variable en mémoire ou d'un objet de données déplaçable. |
| 2 | S'il vaut 1, il s'agit d'une valeur immédiate (ou d'une constante/valeur absolue). |
| 3 | S'il vaut 1, l'expression suivante utilise un mode d'adressage direct. |
| 4 | S'il vaut 1, il s'agit d'un nom de registre. |
| 5 | S'il est à 1, l'expression qui le suit est un symbole défini et il n'y a pas d'erreur. |
| 6 | S'il vaut 1, l'expression suivante est une référence relative à SS. |
| 7 | S'il vaut 1, l'expression suivante fait référence à un nom extérieur. |
| 8-10 | 000 - Aucun type de langage<br>001 - Type de langage C/C++<br>010 - Type de langage SYSCALL<br>011 - type de langage STDCALL<br>100 - Type de langage PASCAL<br>101 - Type de langage FORTRAN<br>110 - Type de langage BASIC |

Les bits du langage sont pour les programmeurs qui écrivent du code en collaboration avec un langage de haut niveau. De tels programmes utilisent les directives de segment simplifiées est les caractéristiques de haut niveau de MASM.

Normalement, on utilise ces valeurs avec les directives conditionnelles et les macros de MASM. Ceci vous permet de générer séquences d'instructions différentes selon le type de paramètre de macro ou selon la configuration d'assembleur courante. Pour plus de détails, voir le paragraphe 8.13 (directives conditionnelles) ou le paragraphe 8.14 (macros).

Les opérateurs size, sizeof, length et lengthof calculent la taille des variables (y compris les tableaux) et retournent leur taille et leur valeur. Normalement, vous ne devriez pas utiliser size et length. Ces opérateurs sont obsolètes et ils ont été remplacés par sizeof et lengthof. Les opérateurs size et length ne retournent pas toujours des valeurs appropriées pour des opérandes arbitraires. MASM 6.x les a inclus pour garder la compatibilité avec les versions plus anciennes. Néanmoins, plus tard dans ce chapitre, vous verrez un exemple où vous pourrez utiliser ces opérateurs malgré tout.

L'opérateur sizeof retourne toujours le nombre d'octets directement alloués à la variable spécifiée. L'exemple suivant illustre ce point :

```
a1              byte    ?                       ;SIZEOF(a1) = 1
a2              word    ?                       ;SIZEOF(a2) = 2
a4              dword   ?                       ;SIZEOF(a4) = 4
a8              real8   ?                       ;SIZEOF(a8) = 8
ary0            byte    10 dup (0)              ;SIZEOF(ary0) = 10
ary1            word    10 dup (10 dup (0))     ;SIZEOF(ary1) = 200
```

Vous pouvez aussi utiliser l'opérateur sizeof pour calculer la taille, en octets, d'une structure ou d'un autre type de données. Ceci est très utile pour calculer l'index d'un tableau à l'aide de la formule vue au chapitre 4 :

```
        ElementAddress  := base_address + index*Element_Size
```

Vous pouvez obtenir la valeur de la taille d'un élément d'un tableau ou d'une structure en utilisant l'opérateur sizeof. Donc, si vous avez un tableau de structures, vous pouvez calculer un index dans le tableau comme suit :

```
        .286                    ;Permettre les instructions du 80286
s       struct
        <un certain nombre de champs>
        .
        .
        .
array   s       16 dup ({})             ;Un tableau de 16 éléments "s"
        .
```

```
        .
        .
        imul    bx, I, sizeof s         ;Calcule BX := I * elementsize
        mov     al, array[bx].fieldname
```

Vous pouvez appliquer aussi l'opérateur sizeof à d'autres types de données afin d'obtenir leur taille en octets. Par exemple, sizeof byte retourne 1, sizeof word retourne 2 et sizeof dword retourne 4. Sans doute, appliquer cet opérateur sur des types prédéfinis par MASM n'est pas très utile, car la taille de ces objets est fixée d'avance. Cependant, si vous créez vos propres types de données en utilisant typedef, c'est parfaitement compréhensible d'utiliser sizeof pour déterminer sans erreur la taille d'un certain objet. Par exemple :

```
integer         typedef word
Array           integer 16 dup (?)
                .
                .
                .
                imul    bx, bx, sizeof integer
                .
                .
                .
```

Dans ce dernier code, sizeof integer retourne 2, tout comme sizeof word. Cependant, si vous changez l'instruction typedef de sorte que *integer* est un dword au lieu d'être un word, sizeof integer retournera automatiquement la nouvelle valeur, à savoir 4, la nouvelle taille de l'opérande.

L'opérateur lengthof retourne le nombre total d'éléments d'un tableau. Pour la variable Array ci-dessus, lengthof Array retournera 16. Si vous avez un tableau à deux dimensions, lengthof retourne le nombre total des éléments de ce tableau.

Quand vous utilisez les opérateurs lengthof et sizeof avec les tableaux, vous devez garder à l'esprit qu'on peut déclarer des tableaux que MASM pourra mal interpréter. Par exemple toutes les instructions qui suivent déclarent des tableaux qui contiennent huit mots :

```
A1              word    8 dup (?)
A2              word    1, 2, 3, 4,, 5, 6, 7, 8
;Notez : le symbole "\" est un symbole de continuation de ligne. Il indique à
;MASM de concaténer la ligne suivante à la fin de la ligne courante.
A3              word    1, 2, 3, 4, \
                        5, 6, 7, 8
A4              word    1, 2, 3, 4
                word    5, 6, 7, 8
```

Appliquer sizeof et lengthof aux variables A1, A2 et A3 produit 16 (sizeof) et 8 (lengthof). Cependant, sizeof(A4) produira 8 et lengthof(A4) produira 4. Ceci arrive parce que MASM croit que les tableaux commencent et se terminent avec une seule déclaration. Bien que la déclaration de A4 initialise 8 mots consécutifs, tout comme les trois autres déclarations qui la précèdent, MASM interprète que les deux directives word déclarent deux tableaux différents et non un seul tableau. Si vous voulez initialiser les éléments d'un long tableau ou d'un tableau multidimensionnel et que vous voulez également être en mesure d'appliquer lengthof et sizeof à ce tableau, vous devrez absolument utiliser la version qu'on a utilisée pour déclarer le tableau A3 et non la version A4.

L'opérateur type retourne une constante qui représente le nombre d'octets de l'opérande spécifiée. Par exemple, type(word) retourne 2. Cette révélation en soi n'est pas particulièrement intéressante, puisque les opérateurs size et sizeof peuvent faire le même boulot. Cependant quand vous utilisez l'opérateur type avec un opérateur de comparaison (eq, ne, le, lt, gt et gr), la comparaison produit un résultat vrai seulement si les types des opérandes sont les mêmes. Considérez les définitions suivantes :

```
Integer         typedef word
J               word    ?
K               sword   ?
L               integer ?
M               word    ?

                byte    type (J) eq word                ;valeur = 0FFh
                byte    type (J) eq sword               ;valeur = 0
                byte    type (J) eq type (L)            ;valeur = 0FFh
                byte    type (J) eq type (M)            ;valeur = 0FFh
```

```
                byte    type (L) eq integer             ;valeur = 0FFh
                byte    type (K) eq dword               ;valeur = 0
```

Puisque le code ci-dessus a utilisé un typedef pour typer integer comme word, MASM traite les *integer* et les *word* comme s'ils étaient du même type. Notez que, sauf l'exception du dernier exemple, la valeur des deux côtés de l'opérateur eq est 2. Par conséquent, en utilisant les opérations de comparaison avec l'opérateur type, MASM ne se limite pas à ne comparer que la valeur. Par conséquent, type et sizeof ne sont pas des synonymes. Par exemple :

```
                byte    type (J) eq type (K)            ;valeur = 0
                byte    (sizeof J) equ (sizeof K)       ;valeur = 0FFh
```

L'opérateur type est particulièrement utile quand on utilise les directives conditionnelles. Voir le paragraphe 8.13 pour plus de détails.

Les exemples ci-dessus ont également démontré une autre caractéristique intéressante de MASM. Si vous utilisez le nom du type à l'intérieur d'une expression, MASM le traite comme si vous aviez entré "type(nom)" où *nom* est un symbole du type donné. En particulier, spécifier un nom de type retourne la taille, en octets, d'un objet de ce type. Considérez l'exemple qui suit :

```
Integer         typedef word
s               struct
d               dword   ?
w               word    ?
b               byte    ?
s               ends
                byte    word                            ;valeur = 2
                byte    sword                           ;valeur = 2
                byte    byte                            ;valeur = 1
                byte    dword                           ;valeur = 4
                byte    s                               ;valeur = 7
                byte    word eq word                    ;valeur = 0FFh
                byte    word eq sword                   ;valeur = 0
                byte    b eq dword                      ;valeur = 0
                byte    s eq byte                       ;valeur = 0
                byte    word eq integer                 ;valeur = 0FFh
```

Les opérateurs high et low, tout comme seg et offset, changent le type d'une expression en constante. Ces opérateurs affectent aussi la valeur de l'expression - ils la décomposent en un byte le plus significatif et un byte le moins significatif, l'opérateur high extrait les bits huit à quinze de l'expression, l'opérateur low extrait et retourne les bits de zéro à sept. Les opérateurs highword et lowword extraient le mot le plus significatif et le mot le moins significatif d'une expression (voir la figure 8.7).

Vous pouvez extraire les bits 16-23 et 24-31 à l'aide d'expressions de la forme low(highword(expr)) et high(highword(expr))[19], respectivement.

### 8.12.5 Précédence des opérateurs

Même si vous n'aurez besoin que rarement d'utiliser des expressions complexes employant plus de deux opérandes et un seul opérateur, le besoin se fait sentir à l'occasion. MASM suit une convention unique pour la précédence des opérateurs, basée sur les règles suivantes :

- Les opérateurs qui ont une plus haute priorité sont d'abord exécutés.
- Les opérateurs ayant une priorité identique ont une associativité gauche et s'évaluent de gauche à droite.
- Les parenthèses outrepassent la priorité normale.

Les parenthèses ne doivent entourer que les expressions. Certains opérateurs, comme sizeof ou lengthof requièrent un nom de type et non une expression. Et ils ne vous permettent pas de mettre des parenthèses

[19] Les parenthèses rendent cette expression plus lisible, mais elles ne sont pas requises.

autour du nom. Par conséquent, "(sizeof X)" est possible, mais "sizeof(X)" ne l'est pas. Gardez ceci à l'esprit quand vous utilisez des parenthèses, pour imposer un ordre de priorité dans une expression. Si MASM génère une erreur, il vous faudra sans doute réarranger vos parenthèses.

Comme on le fait dans les langages de haut niveau, c'est toujours une bonne idée d'utiliser les parenthèses pour établir de façon explicite l'ordre d'évaluation dans toute lexpression d'adresse complexe (et par complexe, on entend que l'expression a plus qu'un opérateur). Ceci généralement rend les expressions plus lisibles et aide à prévenir les bogues en rapport à la précédence des opérateurs.

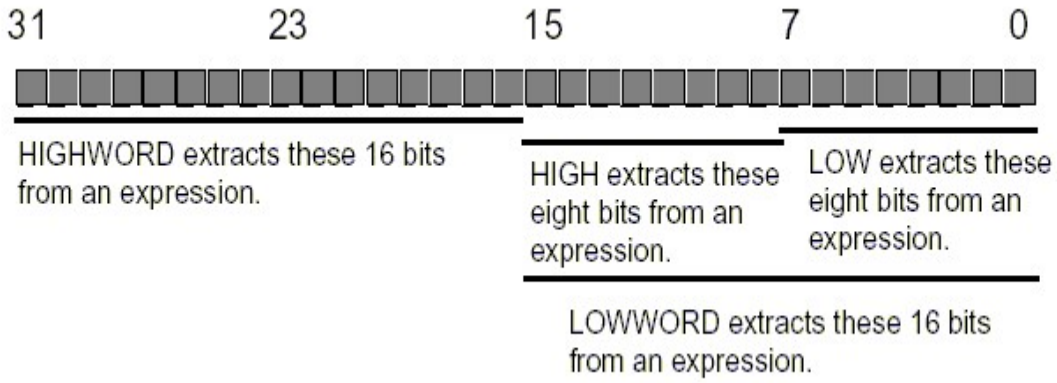


Figure 8.7 Les opérateurs HIGHWORD, LOWWORD, HIGH et LOW

Tableau 41 : Priorité des opérateurs

| Précédence | Opérateurs |
|---|---|
| (La plus élévée) | |
| 1 | length lengthof size sizeof ( ) [ ] < > |
| 2 | . (opérateur de champ d'une structure) |
| 3 | CS: DS: ES: FS: GS: SS: (préfixe de surcharge de segment) |
| 4 | ptr offset set type opattr this |
| 5 | high low highword lowword |
| 6 | + - (unaires) |
| 7 | * / mod shl shr |
| 8 | + - (binaires) |
| 9 | eq ne lt le gt ge |
| 10 | not |
| 11 | and |
| 12 | or xor |
| 13 | short .type |
| (la moins élevée) | |

## 8.13 Assemblage conditionnel

MASM fournit une interface très puissante d'assemblage conditionnel. Avec celui-ci, vous pouvez décider, à partir de certaines conditions, si et quand une certaine portion de code doit être assemblée. Il y a plusieurs directives d'assemblage conditionnel et cette section couvre la plupart d'elles.

Il est important que vous réalisez que ces directives évaluent leurs expressions *pendant l'assemblage* et jamais pendant l'exécution. La directive conditionnelle if n'est pas la même chose que le classique "if" du C ou du Pascal. Si vous connaissez le langage C, une directive comme #ifdef est à peu près équivalente aux directives d'assemblage conditionnel de MASM.

Les directives d'assemblage conditionnel de MASM sont importantes parce que elles permettent de générer différents codes objet pour différents environnements et différentes situations. Par exemple, supposez que vous voulez écrire un programme capable de s'exécuter sur différentes machines, mais vous voudriez optimiser le code surtout pour les processeurs 80386 ou ultérieurs. Évidemment, on ne peut pas exécuter un code 80386 sur un 8086, donc, comment résoudre le problème ?

Une solution possible serait de déterminer le type de processeur pendant l'exécution et d'exécuter différentes sections de code dans le programme selon la présence ou l'absence d'un processeur 386 ou ultérieur. Le problème avec cette approche est que votre programme aura alors besoin de deux séquences de code, une optimale pour le 80386 et l'autre compatible avec le 8086. Dans tout CPU donné, le système exécutera seulement une de ces séquences et l'autre occupera seulement de l'espace mémoire inutilisé, pouvant aussi causer des effets néfastes sur le cache du système.

Une seconde possibilité est d'écrire deux versions du code, une qui utilise seulement les instructions du 8086 et l'autre qui exploite toutes les nouvelles possibilités du 80386. Pendant l'installation l'utilisateur - ou le programme d'installation - sélectionnera la version 80386 si le système est basé sur un processeur moderne, sinon, le choix tombera sur la version 8086. Ce programme requerra plus d'espace disque, mais au moins il consommera moins de mémoire en s'exécutant. Le problème avec cette approche est que vous aurez à maintenir *deux versions séparées* du programme. Si vous corrigez une bogue sur la version 8086, vous aurez probablement à faire le même travail sur l'autre version. Maintenir plusieurs fichiers sources est une tâche difficile.

La troisième solution, la meilleure, est d'utiliser l'*assemblage conditionnel*. Par ce moyen, vous fusionnez les parties du code pour le 8086 et pour le 80386 dans le même fichier. Pendant l'assemblage, vous pourrez conditionnellement choisir si MASM assemble l'une ou l'autre version. De cette façon, vous aurez à travailler avec un seul fichier source et vous n'aurez pas à modifier des parties de code plusieurs fois. Il pourra tout au plus vous arriver de corriger la même bogue deux fois, mais au moins elles se trouvent dans deux séquences de code du même programme et vous aurez moins de chances d'en corriger une en oubliant l'autre.

Les directives d'assemblage conditionnel de MASM sont particulièrement utiles dans les *macros*. Elles vous aideront à produire un code plus efficace là-où une macro seule ne pourrait pas produire le même effet. Pour plus d'informations à propos des macros et leur usage combiné avec l'assemblage conditionnel, référez-vous au paragraphe 8.14.

Les macros et les directives d'assemblage conditionnel fournissent, en fait, un langage de programmation dans un langage de programmation. Les macros de l'assemblage conditionnel vous permettent d'écrire des programmes (dans le "langage des macros"), capables d'écrire des segments de code en assembleur à votre place. Ceci constitue aussi un moyen indépendant de générer des bogues dans vos applications. Vous risquez non seulement de produire des bogues dans vos programmes, mais également vous courez le même risque dans les macros (c'est-à-dire l'assemblage conditionnel) qui finira par produire des bogues dans votre code en assembleur. Gardez à l'esprit que si vous faites des choses trop sophistiquées avec ces outils vous pourrez obtenir des programmes trop difficiles à lire, comprendre et déboguer.

### 8.13.1 La directive IF

La directive IF comporte la syntaxe que voici :

```
        if              expression
<séquence d'instructions>
        else                            ;Ceci est optionnel !
<séquence d'autres instructions>
        endif
```

MASM évaluera *expression*. Si elle correspond à une valeur non nulle, alors la séquence d'instructions entre les directives if et else (ou *endif*, si *else* n'est pas présent) sera assemblée. Si l'expression est fausse (valeur

nulle) et que la section else est présente, alors MASM assemblera la séquence d'instructions entre les directives else et endif. Si la section else n'est pas présente, alors MASM n'assemblera rien à partir du code entre les directives if et endif.

Une chose importante à retenir est qu'*expression* n'est pas quelque chose que MASM peut évaluer pendant l'exécution d'un programme. Ce qui revient à dire qu'*expression* doit s'évaluer comme une constante. Des constantes manifestes, des équivalents comme les constantes symboliques (déclarées avec =), ou encore des valeurs que les opérateurs de type de MASM produisent, se rencontrent couramment dans les expressions des directives if. Par exemple, supposez que vous voulez écrire deux séquences de code pour deux processeurs différents, tel comme il a été décrit précédemment. Vous pourriez faire ce qui suit :

```
Processeur      =       80386   ;Constante symbolique égale à 8086 du code
                                ;uniquement 8086
                .
                .
                .
                if      Processeur eq 80386
                shl     ax, 4
                else            ;C'est le 8086
                mov     cl, 4
                shl     ax, cl
                endif
```

Il y a même d'autres moyens de produire la même chose. MASM fournit des variables prédéfinies indiquant si vous assemblez un code pour un processeur spécifique. On en parlera davantage plus tard.

### 8.13.2 Directive IFE

La directive ife est utilisée exactement comme la directive if, mais dans ce cas, le code sera assemblé seulement si l'expression évaluée est fausse (valant zéro),

### 8.13.3 IFDEF et IFNDEF

Ces deux directives requièrent un seul symbole comme opérande. ifdef assemblera son code associé seulement si le symbole est défini, alors que ifndef assemblera son code respectif seulement si le symbole (son opérande) n'est pas défini. On termine ces séquences conditionnelles avec else et endif.

Ces directives sont particulièrement populaires pour inclure ou ne pas inclure du code dans un programme afin de traiter certains cas spéciaux. Par exemple, on pourrait utiliser des instructions comme ce qui suit pour inclure dans le code des lignes de débogage :

```
ifdef   DEBUG

<instructions de débogage>

endif
```

Pour activer le code de débogage, définissez simplement le symbole DEBUG quelque part au début de votre programme (à savoir avant le premier ifdef qui mentionne DEBUG). Pour éliminer automatiquement le code de débogage, il vous suffira simplement d'effacer la définition de DEBUG. Vous pouvez définir ce symbole en utilisant une simple instruction comme :

```
DEBUG           =       0
```

Notez que la valeur que vous donnez à DEBUG n'est pas importante. Seul le fait d'avoir ou non défini le symbole est important.

### 8.13.4 IFB, IFNB

Ces directives sont utiles surtout dans un contexte de macros (voir le paragraphe 8.13.4), pour vérifier si une opérande est vide (ifb: b pour blank) ou non vide (ifnb). Considérez le code suivant :

```
Blank           textequ         <>
NotBlank        textequ         <not vide>

                ifb             Blank
                <ce code sera assemblé>
                endif

                ifb             NotBlank
                <ce code ne sera pas assemblé>
                endif
```

La directive ifnb fonctionne de manière opposée à ifb. Elle assemble ce que ifb n'assemble pas et vice-versa.

### 8.13.5 IFIDN, IFDIF, IFIDNI et IFDIFI

Ces directives d'assemblage conditionnel prennent deux opérandes et traitent le code associé si elles sont identiques (ifidn), différentes (ifdif), identiques sans tenir compte de la casse[20] (ifdni) ou différentes sans tenir compte de la casse (ifdifi). La syntaxe est :

```
                ifidn   op1, op2
        <instructions à assembler si <op1> = <op2>>
                endif

                ifdif   op1, op2
        <instructions à assembler si <op1> ≠ <op2>>
                endif

                ifidni  op1, op2
        <instructions à assembler si <op1> = <op2>>
                endif

                ifdifi  op1, op2
        <instructions à assembler si <op1> ≠ <op2>>
                endif
```

La différence entre IFxxx et IFxxxI est que la seconde version n'est pas sensible à la casse dans la comparaison des opérandes.

## 8.14 Macros

Une macro est comme une procédure qui insère, pendant l'assemblage, un bloc d'instructions dans divers endroits de votre programme. Il y a trois types généraux de macros supportées par MASM : les macros procédurales, les macros fonctionnelles et les macros en boucle. Avec l'assemblage conditionnel, ces outils permettent de réaliser les structures de contrôle similaires à celles des langages de haut niveau : les structures if, les boucles et les procédures. Mais contrairement aux instructions normales de l'assembleur, les macros et les directives conditionnelles ont lieu pendant l'assemblage. Elles n'existent plus au moment de l'exécution du programme. L'objectif de ces instructions spéciales et de déterminer quel type de code MASM devra assembler dans le fichier final ".exe". Alors que les directives conditionnelles incluent ou omettent certaines instructions, les macros, elles, vous permettent d'émettre des séquences répétitives d'instructions dans un fichier source, tout comme les boucles et les procédures des langages de haut niveau vous permettent d'exécuter des séquences répétitives d'instructions de haut niveau.

### 8.14.1 Macros procédurales

La séquence qui suit définit une macro :

```
nom_macro       macro   {paramètre1 {paramètre2 {,...}}}
                <instructions>
                endm
```

[20] La casse est la distinction entre majuscules et minuscules, n.d.t.

*nom_macro* doit être un symbole unique et valide dans le fichier source. Il faudra utiliser cet identificateur pour appeler une macro. Les noms de paramètres sont des arguments optionnels que l'on spécifie lors de l'appel de la macro ; les accolades dénotent la nature optionnelle des paramètres, elles ne font pas partie de la syntaxe. Ces noms de paramètres sont locaux à la macro et peuvent apparaître ailleurs dans le programme.

Voici un exemple de définition de macro :

```
COPY            macro   Dest, Source
                mov     ax, Source
                mov     Dest, ax
                endm
```

Cette macro copie le mot de l'adresse source dans le mot de l'adresse de destination. Les symboles Dest et Source sont locaux à la macro.

Notez que MASM n'assemble pas immédiatement les instructions entre les directives macro et endm quand MASM rencontre la macro. Au contraire, MASM stocke le texte correspondant à la macro dans une table spéciale (appelée *table des symboles*). MASM insère ces instructions dans le programme seulement lors de l'appel de la macro.

Pour invoquer (utiliser) une macro, spécifiez simplement son nom tout comme une instruction de MASM. Quand vous faites ceci, MASM insérera les instructions entre les directives macro et endm dans votre code à partir du point de l'appel de la macro. Si cette denière a des paramètres, MASM substituera les paramètres spécifiés à l'appel avec les paramètres faisant partie de la définition de la macro. À cet effet, MASM effectuera juste une substitution textuelle, en traitant les macros comme si elles étaient définies avec des constantes symboliques (equates).

Considérez le code suivant qui utilise la macro COPY définie plus tôt :

```
                call    SetUpX
                copy    Y, X
                add     Y, 5
```

Ce fragment de programme appelle SetUpX (qui vraisemblablement fait quelque chose avec la variable X), puis il invoque notre macro COPY, laquelle copie la valeur de X dans la variable Y. Finalement, il additionne 5 à la nouvelle valeur de Y.

Notez que cette dernière séquence d'instructions est *absolument* identique à :

```
                call    SetUpX
                mov     ax, X
                mov     Y, ax
                add     Y, 5
```

Dans certaines circonstances, utiliser des macros permet d'économiser une quantité considérable de frappes. Par exemple, supposez vouloir accéder à des éléments de divers tableaux bidimensionnels. Comme vous vous souvenez, la formule (orientée par ligne) pour déterminer l'offset d'un élément de tableau est :

adresse de l'élément = adresse de base + (premier index * taille ligne + deuxième index) * taille élément

Supposez que vous voulez écrire un code capable de réaliser quelque chose d'analogue au code C suivant :

```
int a[16][7], b[16][7], x[7][16];
int i,j;
        for(i = 0; i < 16; i++)
                for(j = 0; j < 7; j++)
                        x[j][i] = a[i][j] * b[15-i][j];
```

L'équivalent 80x86 de ces instructions serait fort complexe à cause du nombre d'accès aux tableaux. En voici le code complet :

```
                .386                    ;usage des instructions .386
                option  segment:use16   ;requis pour le mode réel
                .
                .
                .
a               sword   16 dup (7 dup (?))
b               sword   16 dup (7 dup (?))
```

```
x               sword   7 dup (16 dup (?))
                .
                .
                .
i               textequ <cx>            ;garder i dans le registre cx
j               textequ <dx>            ;garder j dans le registre dx

ForILp          mov     I, 0            ;initialiser l'index I à 0
                cmp     I, 16           ;I < 16 ?
                jnl     ForIDone        ;si I >= 16, sauter à l'étiquette

                mov     J, 0            ;Initialiser l'index J à 0
ForJLp          cmp     J, 7            ;J < 7 ?
                jnl     ForJDone        ;si J >= 7 sauter à l'étiquette

                imul    bx, I, 7        ;Déterminer l'index pour a[I][J]
                add     bx, J
                add     bx, bx          ;La taille de l'élément est de 2 octets
                mov     ax, a[bx]       ;Obtenir a[I][J]

                mov     bx, 15          ;Déterminer l'index pour b[15-I][J]
                sub     bx, I
                imul    bx, 7
                add     bx, J
                add     bx, bx          ;La taille de l'élément est de 2 octets
                imul    ax, b[bx]       ;Calculer a[I][J] * b[16-I][J]

                imul    bx, J, 16       ;Déterminer l'index pour X[J][I]
                add     bx, I
                add     bx, bx
                mov     X[bx], ax       ;Stocker le résultat

                inc     J               ;Itération suivante pour J
                jmp     ForJLp

ForJDone:       inc     I               ;Itération suivante pour I
                jmp     ForILp

ForIDone                                ;Terminaison de la boucle imbriquée
```

Trop de code pour cinq instructions C/C++, n'est-ce pas ? Si vous lisez ce code entre les lignes, vous remarquerez que beaucoup d'instructions calculent simplement les index des trois tableaux. De plus, les séquences de code calculant ces index sont très similaires. Si elles étaient strictement pareilles, il serait évident qu'on pourrait écrire une macro pour remplacer les trois calculs de ces index. Puisque ces calculs d'index *ne sont pas* identiques, on se demande alors s'il existe un moyen d'écrire une macro pour simplifier ce code. La réponse est oui. En utilisant les paramètres de macro, c'est très facile. Considérez le code suivant :

```
i       texteq          <cx>    ;Garder i dans le registre cx
j       texteq          <dx>    ;Garder j dans le registre dx

NDX2            macro   Index1, Index2, RowSize
                imul    bx, Index1, RowSize
                add     bx, Index2
                add     bx, bx
                endm

                mov     I, 0            ;Initialiser l'index I avec 0
ForILp:         cmp     I, 16           ;I < 16 ?
                jnl     ForIDone        ;Si oui, continuer, si non sauter

                mov     J, 0            ;Initialiser J à 0
ForJLp          cmp     J, 7            ;J < 7 ?
                jnl     ForJDone        ;Si oui,continuer, si non sauter
```

```
                NDX2    I, J, 7
                mov     ax, A[bx]       ;Obtenir a[i][j]

                mov     bx, 15          ;Calculer l'index pour b[15-I][J]
                sub     bx, I
                NDX2    bx, J, 7
                imul    ax, b[bx]       ;Calculer a[i][j]*b[15-i][j]

                NDX2    J, I, 16
                mov     X[bx], ax       ;Stocker le résultat

                inc     J               ;Itération suivante pour J
                jmp     ForJLp

ForJDone        inc     I               ;Itération suivante pour I
                jmp     ForILp
ForIDone                                ;Terminaison de la boucle imbriquée
```

Un problème avec la macro NDX2 est qu'il faut connaître la taille de la ligne d'un tableau (puisque c'est un paramètre de la macro. Dans un court exemple comme celui qu'on vient de voir, ce n'est pas un gros problème. Mais dans un gros programme, on peut facilement oublier les tailles et devoir les chercher une à une, ou pire encore, « s'en souvenir » incorrectement et générer des bogues insidieuses. Une question raisonnable qu'on se pose tout de suite est de savoir si MASM fournirait un moyen automatique pour déterminer les tailles des tableaux. La réponse est encore affirmative.

L'opérateur length de MASM est une rélique des vieilles versions antérieures à 6.0. Il devait retourner le nombre d'éléments d'un tableau. Cependant, il ne retourne en fait que la valeur de la première dimension. Par exemple *length a* retourne 16 si on se réfère à la définition du tableau de l'exemple précédent. MASM a corrigé l'erreur en introduisant l'opérateur lengthof qui retourne le nombre total réel des éléments d'un tableau. Par exemple, *lengthof a* retourne 112 (16 * 7), dans notre exemple. Certes, length a ne retourne pas la bonne valeur pour nos objectifs (car il retourne le nombre des colonnes au lieu du nombre total des éléments), cependant il peut être utilisé pour calculer le nombre des lignes avec l'expression lengthof a / length a. En tenant compte de ce fait, considérez les deux macros suivantes :

```
;LDAX - Cette macro charge ax avec le mot à l'adresse Array[Index1][Index2].
;Suppositions :                 Vous avez déclaré le tableau à l'aide d'une
;                               instruction comme
;                               Array word Colsize dup(RowSize dup(?))
;                               et le tableau est indexé en ordre orienté ligne.
; Si l'on spécifie le quatrième paramètre (optionnel), il s'agit d'une
;  instruction machine 80x86 à substituer à l'instruction mov qui charge AX
avec ; le tableau Array[bx]

LDAX            macro   Array, Index1, Index2, Instr
                imul    bx, Index1, (lengthof Array) / (length Array)
                add     bx, Index2
                add     bx, bx

;Voir si l'appelant a fourni le quatrième paramètre.

                ifb     <instr>
                mov     ax, Array[bx]   ;Sinon, émettre une instruction mov
                else
                instr   ax, Array[bx]   ;Si c'est le cas, émettre l'instruction
                endif
                endm

;STAX - Cette macro stocke la valeur de ax dans le mot à l'adresse
; Array[Index1][Index2]
;Suppositions :                 Le mêmes qu'en haut

STAX            macro   Array, Index1, Index2
                imul    bx, Index1, (lengthof Array) / (length Array)
```

```
                add     bx, Index2
                add     bx, bx
                mov     Array[bx], ax
                endm
```

Avec ces macros, le programme original devient :

```
i               textequ         <cx>    ; Garder i dans le registre cs
j               textequ         <dx>    ; Garder j dans le registre dx

                mov     I, 0            ; Initialiser l'index I avec 0
ForILp:         cmp     I, 16           ; I < 16 ?
                jnl     ForIDone        ; Si oui, continuer, si non sauter

                mov     J, 0            ; Initialiser l'index J avec 0
                cmp     J, 7            ; I < 7 ?
                jnl     ForJDone        ; Si oui, continuer, si non sauter

                ldax    A, I, J         ; Obtenir A[I][J]
                mov     bx, 16          ; Calculer 16-I
                sub     bx, I
                ldax    b, bx, J, imul  ; Effectuer a[i][j] * b[16-i][j]
                stax    x, J, I         ; Stocker dans X[J][I]

                inc J                   ; Itération suivante pour J
                jmp     ForJLp

                inc I                   ; Itération suivante pour I
                jmp     ForIJp
ForIDone                                ; Fin de la boucle imbriquée
```

Comme vous pouvez voir parfaitement, le code des boucles ci-dessus devient de plus en plus court grâce à ces macros. Sans doute, toute la séquence *réelle* est en réalité plus longue, car les macros représentent plus de lignes de code de ce qu'elles économisent dans le vrai programme. Mais, pour le programme en question, cette solution n'est qu'un pis-aller. Généralement vous aurez affaire à plus de trois accès de tableaux. Et encore, on peut toujours placer ces macros dans une bibliothèque et les inclure automatiquement toutes les fois qu'on a affaire à des tableaux de deux dimensions. Bien que techniquement votre programme contient plus d'instructions que sans macros, vous n'aurez cependant à écrire ce code qu'une fois. Et après, rien de plus simple que de les utiliser dans tous vos programmes.

On pourrait raccourcir encore ce code à l'aide de certaines macros additionnelles. Cependant, nous avons encore quelques matières à étudier avant de pouvoir le faire, donc continuez à lire.

### 8.14.2 Les Macros par rapport aux procédures 80x86

Les débutants confondent souvent les procédures avec les macros. Une procédure est une section de code qu'on peut appeler à partir de plusieurs points du programme. Une macro est une séquence d'instructions que MASM copie dans votre programme toutes les fois que vous l'utilisez. Considérez les deux fragments de code suivants :

```
Proc_1          proc    near
                mov     ax, 0
                mov     bx, ax
                mov     cx, 5
                ret
Proc_1          endp

Macro_1         macro
                mov     ax, 0
                mov     bx, ax
                mov     cx, 5
                endm
```

```
call    Proc_1
        .
        .
call    Proc_1
        .
        .
Macro_1
        .
        .
Macro_1
```

Bien que la macro et la procédure ci-dessus produisent le même résultat, elles le font de façons différentes. La définition de la procédure génère du code quand l'assembleur trouve la directive proc. Un appel à cette procédure requiert seulement trois octets. Pendant l'exécution, donc, l'assembleur :

- trouve l'instruction call.
- empile l'adresse de retour dans la pile.
- saute à Proc_1
- exécute le code relatif à Proc_1
- désempile l'adresse de retour de la pile
- retourne au code appelant

La macro, de son côté, n'émet pas de code lors du traitement des instructions entre les directives macro et endm. Cependant, chaque fois que MASM rencontre Macro_1 dans le champ mnémonique, il va assembler chaque instruction entre les directives macro et endm et émettre ce code dans le fichier de sortie. A l'exécution, le CPU exécute ces instructions sans la perte de performance due aux call/ret.

L'exécution d'un code généré par une macro est généralement plus rapide que l'exécution du même code implémenté dans une procédure. Nous avons donc là un autre exemple du dilemme vitesse/espace. Les macros s'exécutent plus rapidement en éliminant la séquence call/return, mais occupent plus d'espace parce que l'assembleur, au lieu de sauter plusieurs fois dans la section du code représentée par une procédure copie le code des macros à tout endroit où la macro est appelée. Si vous avez beaucoup d'appels de macros dans votre programme, le fichier de sortie sera beaucoup plus grand que celui d'un programme n'utilisant que des procédures.

Les appels de macros et de procédures sont considérablement différents. Pour appeler une macro, on en spécifie simplement le nom, comme s'il était une instruction ou une directive. Pour appeler une procédure, il vous faudra utiliser une instruction call. Dans beaucoup de contextes, ce n'est pas très heureux d'utiliser deux types d'appels différents pour des tâches similaires. Le véritable problème a lieu quand vous voulez passer d'une macro à une procédure et vice-versa. Il se peut que vous ayez utilisé une macro pour une opération particulière, mais maintenant vous l'avez utilisée tellement de fois que vous trouvez qu'il serait plus sensé d'utiliser une procédure à la place. Ou bien il peut arriver l'inverse, vous vous êtes servi d'une procédure, mais vous réalisez que générer du code en ligne serait beaucoup plus sensé pour améliorer la performance générale. Le problème dans les deux cas est qu'il vous faudra chercher chaque occurrence de l'appel de la macro ou de la procédure afin de pouvoir effectuer le changement. Modifier une procédure ou une macro est facile, mais localiser et changer tous les appels peut se révéler un drôle de travail. Heureusement, il y a une technique très simple que vous pouvez utiliser dans ce genre de situations, de façon que l'appel de procédure puisse partager la même syntaxe que l'invocation de la macro. L'astuce consiste à créer une macro ou un alias textuel (constante symbolique) pour chaque procédure écrite générant l'appel lui-même de la procédure. Par exemple, supposez que vous écrivez une procédure ClearArray qui met à zéro des tableaux. On pourrait procéder de la façon suivante :

```
ClearArray              textequ         <call $$ClearArray>
$$ClearArray            proc            near
                        .
                        .
                        .
$$ClearAray             endp
```

Et pour appeler cette procédure, on se servirait simplement d'une instruction comme celle qui suit :

```
                        .
                        .
                        .
```

```
                <Pousser les paramètres pour ClearArray>
                ClearArray
                .
                .
                .
```

Si vous avez besoin de transformer la procédure $$ClearAray en macro, tout ce que vous avez à faire, c'est de la nommer ClearArray et mettre le textequ en commentaire. De même, si vous avez une macro et vous voulez la convertir en procédure, appelez simplement la procédure $$nomproc et crééz un alias textuel vous permettant d'appeler cette procédure. Ce qui vous permet justement d'utiliser la même syntaxe d'appel aussi bien pour les procédures que pour les macros.

Ce livre ne se servira pas de cette technique, à l'exception des routines de la bibliothèque standard et non parce que ce n'est pas un bon moyen d'appeler les procédures, mais parce que cela peut causer des problèmes de confusion chez certains étudiants et, puisque le but de ce livre est didactique, on se servira exclusivement des appels explicites.

### 8.14.3 La directive LOCAL

Considérez la définition de macro suivante :

```
LJE             macro   Dest
                jne     SkipIt
                jmp     Dest
SkipIt:
                endm
```

Cette macro fait un « long saut si égal ». Cependant, il y a tout de même un problème. Puisque MASM ne fait que copier le contenu de la macro, le symbole SkipIt sera défini chaque fois que la macro LJE apparaîtra. Quand cela arrive, l'assembleur génèrera plusieurs erreurs de définition. Pour se débarrasser de ce problème, la directive *local* peut être utilisée pour définir un objet local. Considérez la définition suivante :

```
LJE             macro   Dest
                local   SkipIt
                jne     SkipIt
                jmp     Dest
SkipIt:
                endm
```

SkipIt est un symbole local dans la définition de cette macro. Par conséquent, l'assembleur générera une nouvelle copie de SkipIp chaque fois que vous utilisez la macro. Ce qui évitera de générer une erreur.

La directive local, si elle figure dans votre définition de macro, doit apparaître immédiatement après la directive *macro.* Pour définir plusieurs variables locales, on place simplement plusieurs définitions de variable en séparant chaque symbole par une virgule :

```
IFEQUAL         macro   a, b
                local   ElsePortion, Done
                mov     ax, a
                        cmp     ax, b
                jne     ElsePortion
                inc     bx
                jmp     Done
ElsePortion:    dec     bx
Done:
                endm
```

### 8.14.4 La directive EXITM

Cette directive provoque la terminaison immédiate de l'exécution de la macro. Ce qui arrive est exactement ce qui se produirait à l'occurrence de la directive endm. En trouvant la directive exitm, MASM ignore tout le code de la macro qui vient après. Mais alors, quelle serait l'utilité du texte ignoré ? La réponse se trouve dans

l'assemblage conditionnel. Un tel code peut être utilisé pour exécuter conditionnellement cette directive, permettant le reste de l'expansion de la macro seulement sous certaines circonstances. Considérez ce qui suit :

```
Bytes           macro   Count
                byte    Count
                if      Count eq 0
                exitm
                endif
                byte    Count dup (?)
                endm
```

Il va de soi que ce simple exemple aurait pu être codé sans directive exitm (les directives d'assemblage conditionnel suffiraient parfaitement), mais il sert justement à démontrer comment cette directive peut être utilisée dans une séquence d'assemblage conditionnel pour contrôler son influence.

### 8.14.5 Expansion des paramètres d'une macro et opérateurs de macro

Puisque MASM effectue un remplacement textuel pour les paramètres d'une macro quand vous l'appelez, il peut arriver qu'un appel de macro ne produise pas les résultats désirés. Par exemple, considérez la définition (certainement idiote) suivante :

```
Index           =       8

;Problème : cette macro essaie de charger ax avec l'élément d'un tableau word
;spécifié par le paramètre de la macro. Qui devrait bien évidemment être
;une constante déterminée lors de l'assemblage.

Probleme        macro   Parametre
                mov     ax, Array[Parametre*2]
                endm
                .
                .
                .
                Probleme 2
                .
                .
                .
                Probleme Index+2
```

À la première invocation de Probleme, MASM produit l'instruction :

```
                mov     ax, Array[2*2]
```

Ce qui semble acceptable. Ce code charge ax avec l'élément de Array. Et cependant considérez l'expression produite par le second appel de cette macro :

```
                mov     ax, Array[Index+2*2]
```

Puisque les expressions d'adresse de MASM supportent la précédence des opérateurs (voir le paragraphe 8.12.5), cette expansion de macro ne produit pas un résultat correct. Elle provoquera l'accès au sixième élément du tableau (à l'index 12) au lieu qu'au dixième élément (à l'index 20).

Le problème qu'on vient de voir se produit parce que MASM remplace simplement un paramètre formel par le texte du véritable paramètre et non par sa valeur. D'ailleurs, ce mécanisme de passage par nom devrait être familier aux programmeurs rompus au C ou C++ qui ont l'habitude d'utiliser l'instruction #define. Si vous pensez que les paramètres (passés par nom) marchent tout comme les paramètres passés par valeur dans Pascal ou C, vous vous exposez à un désastre éventuel.

Une solution possible qui fonctionne pour ce genre de macros est de placer des parenthèses autour des paramètres qui se trouvent dans des expressions au sein de la macro. Considérez le code suivant :

```
Probleme        macro   Parametre
                mov     ax, Array[(Parametre)*2]
                endm
                .
                .
                .
```

```
                Probleme        Index+2
```

Cet appel de macro devient :

```
                mov     ax, Array[(Index+2)*2]
```

Qui produit le résultat correct.

La substitution des paramètres textuels ne constitue que l'un des problèmes que vous rencontrerez à l'heure d'utiliser des macros. Un autre problème se produit parce que MASM admet deux types de valeurs pendant l'assemblage : numériques et textuelles. Malheureusement, MASM s'attend à des valeurs numériques dans certains contextes et à des valeurs textuelles dans certains autres. Et les deux types de valeurs ne sont pas toujours interchangeables. Par chance, MASM comprend un ensemble d'opérateurs vous permettant de convertir entre une type de valeur et l'autre (évidemment quand cela est possible). Pour comprendre les différences subtiles entre ces deux types de valeurs, jetez un égard aux deux instructions suivantes :

```
Numerique       =               10+2
Textuel         textequ         <10+2>
```

MASM évalue l'expression numérique 10+2 et associe la valeur 12 au symbole *Numerique*. Alors que pour l'autre symbole, MASM stocke simplement la chaîne 10+2 qui remplacera le symbole *Textuel* à tout endroit où vous utiliserez ce symbole dans une expression.

Dans beaucoup de contextes, vous pouvez utiliser l'un ou l'autre symbole. Par exemple, les deux instructions suivantes chargent ax avec 12 :

```
                mov     ax, Numerique           ;Idem que mov ax, 12
                mov     ax, Textuel             ;Idem que mov ax, 10+2
```

Cependant, considérez les deux expressions suivantes :

```
                mov     ax, Numerique*2         ;Idem que mov ax, 12*2
                mov     ax, Textuel*2           ;Idem que mov ax, 10+2*2
```

Comme vous pouvez voir, la substitution textuelle qui se produit avec les "equates" textuels peut mener aux mêmes problèmes qui se produisent avec la substitution textuelle des paramètres de macro.

MASM convertira automatiquement un objet textuel en valeur numérique, si la conversion est nécessaire. A part le problème de substitution textuelle décrit ci-dessus, vous pouvez utiliser une valeur numérique textuelle partout où MASM requiert une valeur numérique.

Dans le sens opposé, convertir des valeurs numériques en valeurs textuelles n'est pas automatique. Par conséquent, MASM fournit un opérateur permettant de convertir des données numériques en données textuelles. Il s'agit de l'opérateur %. Cet opérateur *d'expansion* impose une évaluation immédiate d'une expression pour ensuite convertir le résultat en une chaîne de chiffres. Considérez :

```
                Probleme 10+2           ;où le paramètre est "10+2"
                Probleme %10+2          ;cette fois il est "12"
```

Dans le deuxième exemple, l'opérateur d'expansion textuelle indique à MASM d'évaluer l'expression "10+2" et de convertir la valeur numérique résultante en une valeur textuelle consistant dans les chiffres qui représentent la valeur douze. Par conséquent, ces deux macros se développent respectivement en :

```
                mov     ax, Array[10+2*2]       ;Expansion de Probleme 10+2
                mov     ax, Array[12*2]         ;Expansion de Probleme %10+2
```

MASM fournit un second opérateur, l'opérateur de *substitution*, permettant de développer un nom de paramètre de macro là où MASM ne s'attend pas normalement à un symbole. L'opérateur de substitution est représenté par le caractère &. Si dans une macro vous entourez le nom de la macro par des &, MASM fera la substitution, *peu importe l'endroit où se trouve le symbole*. Ceci permet précisément de développer des paramètres de macro à l'intérieur d'autres identificateurs ou de chaînes littéraires. La macro suivante montre l'usage de cet opérateur :

```
DebugMsg        macro   Point, String
Msg&String&     byte    "At point &Point&: &string&"
                endm
                .
                .
                .
```

```
                DebugMsg        5, <Assertion fails>
```

L'appel de la macro produit l'instruction :

```
Msg5            byte    "At point 5: Assertion failed"
```

Notez comme l'opérateur de substitution a permis à cette macro de concaténer "Msg" et "5" pour produire une étiquette avec la directive byte. Notez aussi que l'opérateur d'expansion permet de développer des identificateurs de macro même s'ils apparaissent dans une chaîne littérale constante. Sans les ampersands ("&") dans la chaîne, MASM aurait émis :

```
Msg5            byte    "At point point: String"
```

Un autre opérateur important dans les macros est *l'opérateur de caractère littéral*, représenté par un point d'exclamation ("!"). Ce symbole indique à MASM de passer le caractère suivant sans aucune modification. On utilise normalement ce symbole si on a besoin d'inclure l'un des symboles suivants comme caractère à l'intérieur d'une macro :

```
                !       &       >       %
```

Par exemple, si vous vouliez en fait que la chaîne de la macro DebugMsg affiche le caractère & (ampersand), vous pourriez utiliser la définition :

```
DebugMsg        macro   Point, String
Msg&String&     byte    "At point !&Point!&: !&String!&"
                endm
```

Et la ligne "Debug5, <Assertion fails>" produirait :

```
Msg5            byte    "At point &Point&: &String&"
```

On utilise les symboles "<" et ">" pour délimiter du texte à l'intérieur de MASM. Les deux appels suivants de la macro PutData montrent comment on peut utiliser ces délimiteurs dans une macro :

```
PubData macro   TheName, TheData
PD_&TheName&    byte    TheData
                endm
                .
                .
                .
                PutData MyData, 5, 4, 3         ;Emet "PD_MyData byte 5"
                PutData MyData, <5, 4, 3>       ;Emet "PD_MyData byte 5, 4, 3"
```

Vous pouvez vous servir des délimiteurs de texte pour entourer des objets que vous désirez traiter en tant que paramètres individuels plutôt qu'en tant que liste de multiples paramètres. Dans la macro PutData de l'exemple ci-dessus, le premier appel passe quatre paramètres à PutData (PutData ignore les deux derniers). Dans le second appel, il y a deux paramètres, le second étant le texte "5,4,3".

Le dernier opérateur de MACRO digne d'intérêt est le ";;". Cet opérateur commence un *commentaire de macro*. Normalement, MASM copie tout le texte de la macro dans le corps du programme pendant l'assemblage, en incluant tous les commentaires. Cependant, si vous commencez un commentaire avec l'opérateur ;; au lieu qu'avec un seul point-virgule, MASM ne développera pas ce genre de commentaire. Ceci augmente un peu la vitesse de l'assembleur, et, plus important encore, il n'encombre pas le listing avec le même commentaire (pour en savoir plus sur la façon de contrôler les listings, consultez les paragraphes 8.19 et suivants).

Tableau 42 : Opérateurs des macros

| Opérateur | Description |
|---|---|
| & | Opérateur de substitution de texte |
| <> | Opérateur d'expression littérale |
| ! | Opérateur de caractère littéral |
| % | Opérateur d'expression |
| ;; | Commentaire de macro |

### 8.14.6 Une macro échantillon pour implémenter des boucles FOR

Vous souvenez-vous de la boucle for et des opérations de matrice utilisées dans l'exemple précédent ? À la fin de cette section il y avait un court commentaire expliquant qu'on pouvait améliorer encore plus ce code avec une utilisation plus intense des macros, mais l'exemple a été renvoyé a plus tard. Suite à la description des opérateurs de macro, nous sommes maintenant prêts à terminer cette discussion-là. Les macros qui implémentent la boucle for sont :

```
; D'abord trois macros permettant de construire des symboles obtenus par
; la concaténation d'autres symboles.
; Ceci est nécessaire parce que le code suivant a besoin de développer
; divers composants du genre "text equate", plusieurs fois avant d'arriver
; au bon symbole.
;
; MakeLbl - Émet une étiquette en concaténant les deux paramètres passés à
; la macro.

MakeLbl       macro   FirstHalf, SecondHalf
&FirstHalf&&SecondHalf&:
              endm

jgDone        macro   FirstHalf, SecondHalf
              jg      &FirstHalf&&SecondHalf&
              endm

jmpLoop       macro   FirstHalf, SecondHalf
              jmp     &FirstHalf&&SecondHalf&
              endm

; ForLp -     Cette macro apparaît au début de la boucle for. Pour l'appeler,
;             il faut se servir d'une instruction de la forme :
;
;             ForLp LoopCtrlVar, StartVal, StopVal
;
; Nota: "FOR" est un mot réservé de MASM et c'est pourquoi cette macro
; n'utilise pas ce nom

ForLp         macro   LCV, Start, Stop

; On a besoin de générer un symbole unique et global pour chaque boucle
; que l'on crée. Ce symbole doit être global parce que il faudra le
; référencer au bas de la boucle. Pour générer un symbole unique, cette
; macro concatène "FOR" avec le nom de la variable de contrôle de la boucle
; et une valeur numérique unique que cette macro incrémente chaque fois que
; l'utilisateur construit une boucle for avec la même variable de contrôle.

              ifndef  $$For&LCV&      ;;Symbole = $$FOR concaténé avec LCV
$$For&LCV&    = 0             ;;Si c'est la première boucle w/LCV, utilisez
              else            ;;zéro, sinon, incrémentez la valeur.
$$For&LCV&    =       $$For&LCV& + 1
              endif

; Émission des instructions pour initialiser la variable de contrôle :

              mov     ax, Start
              mov     LCV, ax

; Ceci provoque la sortie de l'étiquette au haut de la boucle for. Cette
; étiquette prend la forme
;             $$FOR LCV x
; où LCV est le nom de la variable de contrôle de la boucle et X est un
; nombre unique que cette macro incrémente pour chaque boucle for qui
; utilisera la même variable de contrôle.
```

```
                MakeLbl $$For&LCV&, %$$For&LCV&

; Bon, provoquer la sortie du code pour voir si la boucle est complète.
; La macro jgDone génère un jump (if greater) à l'étiquette que la
; macro Next émet après le bas de la boucle.

                mov     ax, LCV
                cmp     ax, Stop
                jgDone $$Next&LCV&, %$$For&LCV&
                endm

; La macro Next met fin à la boucle. Cette macro incrémente la variable
; de contrôle et ensuite retransfère le contrôle à l'étiquette du haut de la
; boucle for.

Next            macro   LCV
                inc     LCV
                jmpLoop $$For&LCV&, %$$For&LCV&
                MakeLbl $$Next&LCV&, %$$For&LCV&
                endm
```

Avec ces macros et les macros LDAX/STAX, le code de l'exemple de manipulation de tableaux qu'on a vu précédemment devient très simple :

```
                ForLp   J, 0, 15
                ForLp   J, 0, 6

                ldax    A, I, J                 ;Charger A[I][J]
                mov     bx, 15                  ;Calculer 16-I.
                sub     bx, I
                ldax    b, bx, J, imul          ;Multiplier dans B[15-I][J].
                stax    x, J, I                 ;Stocker dans X[J][I]
                Next J
                Next I
```

Bien que ce code ne soit pas aussi court que celui de l'exemple montré en C/C++, il s'en approche tout quand même !

Alors que le programme principal devient très simple, il reste une question à se poser sur les macros elles-mêmes. Les macros ForLp et Next sont *extrêmement* complexes ! S'il faut faire de tels efforts chaque fois qu'on veut créer une macro, les programmes assembleurs seront dix fois plus difficiles à écrire, si vous décidez d'utiliser des macros. Heureusement, vous n'aurez à écrire et à déboguer de telles macros qu'une fois. Après vous pourrez les utiliser aussi souvent que vous voudrez, dans divers programmes sans avoir à vous préoccuper de leur implémentation.

Étant donné la complexité des macros For et Next, c'est probablement une bonne idée de décrire *soigneusement* ce que chaque instruction fait. Cependant, avant des discuter des macros elles-mêmes, il faut parler de la façon dont on implémente des boucles for/next en assembleur. Ce livre explorera pleinement la boucle for un peu plus tard, mais on peut parfaitement examiner ici les principes de base. Considérez la boucle Pascal suivante :

```
        for variable := StartExpression to EndExpression do
                Some_Statement;
```

Pascal commence à déterminer la valeur de **StartExpression**. Il affecte ensuite cette valeur à **variable**. Puis il évalue **EndExpression** et enregistre cette valeur dans un emplacement temporaire. Ensuite, on entre dans le corps de la boucle. La première chose que la boucle fait est comparer la valeur de **variable** avec celle de **EndExpression**. Si la valeur de **variable** est supérieure à celle de **EndExpression**, Pascal transfère le contrôle à la première instruction après la boucle, sinon, il continue à exécuter **Some_Statement**. Après l'exécution de **Some_Statement**, Pascal incrémente **variable** et retourne à l'endroit où il compare compare derechef la valeur

de **variable** avec celle de la variable **EndExpression**. En convertissant ce code directement en assembleur, le code qui en résulte est le suivant :

```
; Nota: ce code présuppose que StartExpression et EndExpresison sont de simples
; variables. Si ce n'est pas le cas, il vous faudra calculer la valeur de
; ces expressions et l'affecter à ces variables.

                mov     ax, StartExpression
                mov     Variable, ax
ForLoop:        mov     ax, Variable
                cmp     ax, EndExpression
                jg ForDone

        <Code de Some_Statement>

                inc     Variable
                jmp     ForLoop
ForDone:
```

Pour implémenter ceci en tant que jeu de macros, on doit être capable de créer un court fragment de code qui *va écrire* les instructions ci-dessus pour nous. De premier abord, cela semble facile, pourquoi en effet ne pas utiliser le code suivant ?

```
ForLp           macro   Variable, Start, Stop
                mov     ax, Start
                mov     Variable, ax
ForLoop:        mov     ax, Variable
                cmp     ax, Stop
                jg      ForDone
                endm

Next            macro   Variable
                inc     Variable
                jmp     ForLoop
ForDone:
                endm
```

Ces deux macros vont créer un code correct - exactement une fois. Son second usage entraînerait un problème. Ceci est particulièrement évident si on utilise des boucles imbriquées :

```
        ForLp   I, 1, 10
        ForLp   J, 1, 10
                .
                .
                .
        Next    J
        Next    I
```

Ces macros émettent le code 80x86 suivant :

```
                mov     ax, 1           ;La macro ForLp I, 1, 10 émet ces
                mov     I, ax           ;instructions :
ForLoop:        mov     ax, I           ;       .
                cmp     ax, 10          ;       .
                jg      ForDone         ;       .
                mov     ax, 1           ;La macro ForLp J, 1, 10 émet :
                mov     J, ax           ;       .
ForLoop:        mov     ax, J           ;       .
                cmp     ax, 10          ;       .
                jg      ForDone         ;       .
                        .
                        .
                        .
                inc     J               ;Instructions émises par Next J
                jmp     ForLp           ;       .
ForDone:
                inc     I               ;Instructions émises par Next I
                jmp     ForLp           ;       .
```

```
ForDone:
```

Le problème, évident dans ce code est que chaque fois qu'on utilise la macro ForLp, vous émettez l'étiquette "ForLoop" dans le code. De même, chaque fois que vous utilisez la macro Next, l'étiquette "ForDone" sera émise. Par conséquent, si vous utilisez ces macros plus d'une fois (au cours de la même procédure), vous obtiendrez une erreur de duplication de symbole. Pour éviter ces erreurs, les macros doivent générer des étiquettes uniques chaque fois qu'elles sont utilisées. Malheureusement, la directive local ne fonctionne pas ici. Cette dernière utilise un symbole unique *à l'intérieur* d'un seul appel de macro. Si vous regardez soigneusement le code ci-dessus, vous verrez que la macro ForLp émet un symbole référencé par le code de la macro Next. De même, la macro Next émet une étiquette qui sera référencée par la macro ForLp. Par conséquent, les noms d'étiquettes doivent être globaux, puisque les deux macros y font mutuellement référence.

La solution qu'utilisent les variables macros ForLp et Next est de générer des étiquettes connues globalement de la forme "$$For" + "*nom de variable*"+ "*une valeur unique*" et "$$Next" + "*nom de variable*" + "*une valeur unique*". Pour les exemples qu'on a donnés, les véritables macros ForLp et Next, génèrent le code suivant :

```
                mov     ax, 1           ; La macro ForLp I, 1, 10
                mov     I, ax           ; émet ces instructions.
$$ForI0:        mov     ax, I           ;       .
                cmp     ax, 10          ;       .
                jg      $$NextI0        ;       .

                mov     ax, 1           ; La macro ForLp J, 1, 10
                mov     J, ax           ; émet ces instructions.
$$ForJ0:        mov     ax, J           ;       .
                cmp     ax, 10          ;       .
                jg      $$NextJ0        ;       .
                        .
                        .
                        .
                inc     J               ; Instructions émises par Next J
                jmp     $$ForJ0         ;       .
$$NextJ0:
                inc     I               ; Instructions émises par Next I
                jmp     $$ForI0         ;       .
$$NextI0:
```

La véritable question à se poser est donc : "Comment générer ces étiquettes ?".

Construire un symbole tel que "$$ForI" ou "$$ForJ" est très facile. Il suffit de créer un symbole en concaténant les chaînes "$$For" ou "$$Next" avec le nom de la variable de contrôle de la boucle. Le problème surgit quand on essaie d'y concaténer une valeur numérique. Les macros ForLp et Next réalisent ceci en créant des noms de variables de la forme "$$ForVariable_name" au moment de l'assemblage et en incrémentant cette variable pour chaque boucle associée au nom de la variable de contrôle. En appelant les macros MakeLbl, jgDone et jmpLoop, ForLp et Next produisent les étiquettes et les instructions appropriées.

Les macros ForLp et Next sont très complexes. Beaucoup plus complexes que celles qu'on trouve normalement dans un programme. Elles démontrent néanmoins la puissance que les macros de MASM permettent. Naturellement, il y a des moyens *beaucoup* plus efficaces pour créer ces symboles, notamment l'usage des *fonctions macro*. On en parlera immédiatement.

### 8.14.7 Fonctions macro

Une fonction macro (ou aussi *macro fonctionnelle*) est une macro dont le seul but est de retourner une valeur à utiliser comme opérande d'une autre instruction. Bien qu'il y a un parallélisme évident entre les procédures/fonctions des langages de haut niveau et les macros procédurales et fonctionnelles, l'analogie est loin d'être correcte. Les macros fonctionnelles ne vous permettent pas de créer des séquences de code émettant des instructions qui calculent des valeurs pendant l'exécution d'un programme. Tout ce qu'elles font réellement est de calculer des valeurs pendant l'assemblage que MASM pourra utiliser comme opérandes.

Un bon exemple de macro fonctionnelle est la fonction Date. Cette fonction compacte une valeur de jour sur cinq bits, de mois sur quatre bits et d'année sur sept bits, le tout dans un mot de 16 bits qui sera retourné comme résultat. En ayant besoin de créer un tableau initialisé de dates, on pourrait utiliser un code tel que :

```
DateArray       word    Date(2, 4, 84)
                word    Date(1, 1, 94)
                word    Date(7, 20, 60)
                word    Date(7, 19, 69)
                word    Date(6, 18, 74)
                .
                .
                .
```

La fonction Date compacte la date et la directive word émet dans le code objet la valeur compactée de 16 bits. On appelle les macros fonctionnelles par leur nom là où MASM attend une expression textuelle quelconque. Si la macro requiert des paramètres, il faudra les spécifier entre parenthèses, tout comme les paramètres de la macro Date.

Les macros fonctionnelles ressemblent exactement aux macros standard, à deux exceptions près : elles ne contiennent aucune instruction qui génère du code et elles retournent une valeur textuelle via une opérande placée dans la directive exitm. Notez qu'avec une fonction macro, vous *ne pouvez pas* retourner une valeur numérique. S'il faut retourner une valeur numérique, il faudra d'abord la convertir en valeur textuelle.

La macro fonctionnelle suivante implémente Date à l'aide du format de date de 16 bits donné au chapitre 1 (voir le paragraphe 1.10).

```
Date    macro   month, day, year
        local   Value
Value   =       (month shl 12) or (day shl 7) or year
        exitm   %Value
        endm
```

L'opérateur d'expression textuelle ("%") est nécessaire comme opérande de la directive exitm parce que les macros fonctionnelles retournent toujours des valeurs textuelles et non numériques. L'opérateur d'expansion convertit la valeur numérique en chaîne de chiffres, acceptable par exitm.

Un problème mineur dans ce code est que la fonction retourne n'importe quoi si la date n'est pas correcte. Une fonction mieux conçue devrait produire une erreur si la date entrée n'est pas correcte. Pour permettre cela, on peut utiliser la directive ".err" et l'assemblage conditionnel. La version suivante de la macro Date vérifie le mois, le jour et l'année avant de retourner une valeur.

```
Date    macro   month, day, year
        local   Value
        if      (month gt 12) or (month lt 1) or \
                (day gt 31) or (day lt 1) or \
                (year gt 99) (year lt 1)
        .err
        exitm   <0>     ;;Doit retourner quelque chose !
        endif
Value   =       (month shl 12) or (day shl 7) or year
        exitm   %Value
        endm
```

Avec cette version, toute tentative de spécifier une valeur farfelue provoque l'assemblage de la directive .err qui force une erreur au moment de l'assemblage.

### 8.14.8 Macros prédéfinies, macros fonctions et symboles

MASM fournit quatre macros prédéfinies et quatre macros fonctionnelles correspondantes. De plus, il fournit aussi un grand nombre de symboles prédéfinis auxquels on peut avoir accès durant l'assemblage. Bien que vous n'utiliserez que rarement ces macros, fonctions et variables en dehors de macros raisonnablement complexes, elles sont essentielles quand vous en avez besoin.

Tableau 43 : Macros prédéfinies de MASM

| Nom | Opérandes | Exemple | Description |
|---|---|---|---|
| substr | string, start, length<br>Retour : une valeur textuelle | NewStr substr OldStr, 1, 3 | Retourne la chaîne de caractères qui commence à start et termine à start+length. L'opérateur de longueur est optionnel. S'il n'est pas spécifié, MASM retourne tous les caractères de start jusq'à la fin de la chaîne. |
| instr | start, string, substr<br>Retour : une valeur numérique | Pos instr 2, Oldstr, <ax> | Recherche "substr" dans "string" à partir de la position "start". La valeur de départ est optionnelle. Si elle n'est pas spécifiée, MASM commence à chercher à partir de la position 1. Si MASM ne peut pas trouver la sous-chaîne, il retourne la valeur 0. |
| sizestr | string<br>Retour : une valeur numérique | StrSize sizestr OldStr | Retourne la taille de l'opérande chaîne. |
| catstr | string, string, ...<br>Retour : une valeur textuelle | NewStr catstr OldStr, <$$> | Crée une nouvelle chaîne en concaténant chacune des chaînes passées comme opérandes de la macro. |

Les macros substr et catstr retournent des valeurs textuelles. Sous certains aspects elles sont similaires à la directive texequ, car on les utilise pour affecter des données textuelles à un symbole pendant l'assemblage. instr et sizestr sont semblables à la directive "=", car elles retournent une valeur numérique.

La macro catstr peut éliminer le besoin de la macro MakeLbl de notre boucle ForLp. Comparez la version suivante de ForLp avec la version précédente (paragraphe 8.14.6).

```
ForLp           macro   LCV, Start, Stop
                local   ForLoop

                ifndef  $$For&LCV&
$$For&LCV&      =       0
                else
$$For&LCV&      =       $$For&LCV& + 1
                endif

                mov     ax, Start
                mov     LCV, ax

        ; A cause d'une bogue de MASM ceci ne fonctionnera pas, mais l'idée
        ; est bonne. Voici la solution

ForLoop         texteq @catstr($For&LCV&, %$$For&LCV&)
&ForLoop&:
                mov     ax, LCV
                cmp     ax, Stop
                jgDone  $$Next&LCV&, %$$For&LCV&
                endm
```

MASM fournit aussi des versions de macros fonctionnelles pour catstr, instr, sizestr et substr. Pour différencier entre les macros fonctionnelles et les macros prédéfinies correspondantes, MASM utilise les noms @catstr, @instr, @sizestr et @substr. Considérez les équivalences suivantes :

```
Symbol          catstr  String1, String2, ...
Symbol          texteq @catstr(String1, String2, ...)

Symbol          substr  SomeStr, 1, 5
Symbol          texteq @substr(SomeStr, 1, 5)

Symbol          instr   1, SomeStr, SearchStr
Symbol          =       @instr(1, SomeStr, SearchStr)

Symbol          sizestr SomeStr
Symbol          =       @sizestr(SomeStr)
```

Table 44 : Macros fonctionnelles prédéfinies de MASM

| Nom | Paramètres | Exemple |
|---|---|---|
| @substr | string, start, length<br>Retour : une valeur textuelle | ifidn @substr(parm, 1, 4), <bx> |
| @instr | start, string, substr<br>Retour : une valeur numérique | if @instr(parm, <bx>) |
| @sizestr | string<br>Retour : une valeur numérique | byte @sizestr(SomeStr) |
| @catstr | string, string, ...<br>Retour : une valeur textuelle | jg @catstr($$Next&LCV&, %$$For&LCV&) |

Le dernier exemple ci-dessus montre comment se débarrasser des macros jgDone et jmpLoop de notre boucle ForLp. Une version finale et améliorée des macros ForLp et Next (en éliminant les trois macros supplémentaires et en tenant compte de la bogue déjà mentionnée de MASM), ressemblerait à ce qui suit :

```
ForLp           macro   LCV, Start, Stop
                local   ForLoop

                ifndef  $$For&LCV&
$$For&LCV&      =       0
                else
$$For&LCV&      =       $$For&LCV& + 1
                endif

                mov     ax, Start
                mov     LCV, ax

ForLoop         textequ @catstr($For&LCV&, %$$For&LCV&)
&ForLoop&:
                mov     ax, LCV
                cmp     ax, Stop
                jg      @catstr($$Next&LCV&, %$$For&LCV&)
                endm

Next            macro   LCV
                local   NextLbl
                inc     LCV
                jmp     @catstr($$For&LCV&, %$$For&LCV&)
NextLbl         textequ @catstr($Next&LCV&, %$$For&LCV&)
&NextLbl&:
                endm
```

MASM fournit aussi un assez grand nombre de variables prédéfinies qui retournent des informations sur l'assemblage courant. La table suivante décrit toutes ces variables :

Tableau 45 : Variables prédéfinies de MASM

| Catégorie | Nom | Description | Résultat de retour |
|---|---|---|---|
| Date et heure | @Date | Retourne la date de l'assemblage. | Valeur textuelle |
| | @Time | Retourne une chaîne représentant l'heure de l'assemblage. | Valeur textuelle |
| Informations d'environnement | @CPU | Retourne une valeur word dont les bits indiquent la directive du processeur couramment active. Le fait de spécifier .8086, .186, .296, 386, 486 et .586 permet l'usage d'instructions additionnelles de MASM. Ces directives activent également les bits correspondants dans la variable @cpu, où chaque bit est un drapeau indiquant un processeur différent. Notez que MASM active tous les bits qui précèdent celui du processeur activé. Par exemple, pour la directive .386, MASM active | Bit 0 - instructions 8086<br>Bit 1 - instructions 80186<br>Bit 2 - instructions 80286<br>Bit 3 - instructions 80386<br>Bit 4 - instructions 80486<br>Bit 5 - instructions Pentium<br>Bit 6 - bit réservé pour 80686<br>Bit 7 - instructions mode protegé<br>Bit 8 - instructions 8087 |

| | | | |
|---|---|---|---|
| | | tous les bits de 0 à 3. | Bit 10 - instructions 80287<br>Bit 11 - instructions 80386<br>(le bit 11 est activé aussi pour les instructions 80486 et Pentium). |
| | @Environ | @Environ(nom) retourne le texte associé avec les noms des variables d'environnement du DOS. Le paramètre doit être une valeur textuelle valide qui évalue une variable d'environnement valide. | Valeur textuelle |
| | @Interface | Retourne une valeur numérique indiquant le type de langage couramment utilisé. Notez que cette information est similaire à celle donnée par l'attribut opattr.<br>Le bit le plus significatif détermine si vous êtes en train d'assembler le code sous DOS/Windows ou OS/2.<br>Cette directive est principalement utile pour ceux qui font usage des directives de segment simplifiées. Puisque ce livre ne traite pas ces directives, une discussion ultérieure sur cette variable n'est pas garantie. | Bits de 0 à 2 :<br>000 - Aucun type de langage<br>001 - C<br>010 - SYSCALL<br>011 - STDCALL<br>100 - Pascal<br>101 - FORTRAN<br>110 - BASIC<br>Bit 7<br>0 - MS-DOS ou Windows<br>1 - OS/2 |
| | @Version | Retourne une valeur numérique indiquant le nombre courant de version de MASM, multiplié par 100. Par exemple, la variable @version pour MASM 6.11 retourne 611. | Valeur numérique |
| | @FileCur | Retourne le nom du fichier source ou du fichier d'inclusion, en incluant toute information nécessaire sur les chemins d'accès. | Valeur textuelle |
| Informtations sur les fichiers | @File-Name | Retourne le nom du fichier source courant (et pas son chemin d'accès). Si on se trouve dans un fichier d'inclusion, elle retourne le nom du fichier source qui inclut le fichier courant. | Valeur textuelle |
| | @Line | Retourne le numéro de ligne courant dans le fichier source. | Valeur numérique. |

**Tableau 45 : Variables prédéfinies de MASM (suite)**

| Catégorie | Nom | Description | Résultat de retour |
|---|---|---|---|
| Informations de segment [a] | @code | Retourne le nom du segment de code courant | Valeur textuelle |
| | @data | Retourne le nom du segment de données courant | Valeur textuelle |
| | @FarData? | Retourne le nom du segment de données éloigné courant. | Valeur textuelle |
| | @Word-Size | Retourne 2 si c'est un segment de 16 bits, 4 si c'est un segment de 32 bits. | Valeur numérique |
| | @Code-Size | Retourne 0 pour un modèle Tiny, Small, Compact et Flat. Retourne 1 pour un modèle Medium, Large et Huge. | Valeur numérique |
| | @DataSize | Retourne 0 pour un modèle de mémoire Tiny, Small, Medim, et Flat. Retourne 1 pour un modèle de mémoire Compact et Large. Retourne 2 pour Huge. | Valeur numérique |
| | @Model | Retourne 1 pour un modèle Tiny, 2 pour Small, 3 pour Compact, 4 pour Medium, 5 pour Large, 6 pour Huge et 7 pour Flag. | Valeur textuelle |
| | @CurSeg | Retourne le nom de segment de code courant. | Valeur textuelle |
| | @stack | Retourne le nom du segment de pile courant | Valeur textuelle |

a. Ces fonctions sont prévues pour l'usage des directives de segment simplifiées. Ce chapitre ne s'occupe pas de ces directives, donc ces fonctions ne seront probablement objet d'aucune discussion.

Malgré le manque d'espace pour traiter dans le détail chacune des applications qu'on peut faire de ces variables, des exemples peuvent néanmoins montrer quelques possibilités. D'autres usages de ces variables seront illustrés au cours de ce livre. Cependant, les applications les plus impressionnantes sont celles que vous découvrirez *vous-même*.

La variable @CPU est très utile si on veut assembler différentes séquences de code pour différents processeurs. La section de ce chapitre traitant l'assemblage conditionnel a décrit comment on peut définir un symbole pour déterminer si votre code sera assemblé sur un 80386 ou ultérieur ou bien sur un processeur plus ancien. La variable @CPU vous fournit un symbole qui vous dira *exactement* quelles instructions sont possibles à tout point de votre programme. Voici un exemple se servant de cette variable :

```
if      @CPU and 100b   ;On a besoin que ce soit un 80286 ou ultérieur
shl     ax, 4           ;pour cette instruction
else                    ;Sinon,
mov     cl, 4           ;Voici l'équivalent pour un processeur 8086
shl     ax, cl
endif
```

On peut se servir de la directive @Line pour placer certains messages spéciaux de diagnostic. Le code suivant affiche un message d'erreur, en incluant le numéro de ligne de l'instruction incorrecte, si une erreur pendant l'exécution est détectée :

```
mov     ax, ErrorFlag
cmp     ax, 0
je      NoError
mov     ax, @Line       ;Charge AX avec le n° de ligne courante
call    PrintError      ;Afficher le message d'erreur et le n° de ligne
jmp     Quit            ;Arrêt du programme.
```

### 8.14.9 Macros vs constantes symboliques

Les macros, les macros fonctionnelles et les constantes symboliques effectuent toutes des substitutions textuelles dans un programme. Même si elles ont des fonctions similaires qui peuvent être confondues parfois, elles peuvent servier à des tâches différentes.

Les constantes symboliques effectuent une substitution individuelle dans une ligne. Elles ne permettent aucun paramètre. De cette façon, vous pouvez remplacer du texte *n'importe où* dans une ligne. Vous pouvez développer une constante symbolique dans le contexte d'une étiquette, d'un mot-clé, d'une opérande et même d'un commentaire. De plus, vous pouvez remplacer plusieurs champs, même une ligne entière avec un seul symbole.

Les macros fonctionnelles sont permises seulement comme opérandes. Cependant, vous pouvez leur passer des paramètres, ceci les rendant considérablement plus générales qu'une simple constante symbolique.

Les macros procédurales vous permettent d'émettre des séquences d'instructions (avec des constantes symboliques on peut émettre au plus une instruction).

### 8.14.10 Macros : bonnes et mauvaises nouvelles

Les macros offrent des commodités considérables. Elles permettent d'insérer diverses instructions dans vos fichiers source avec une simple commande. Ceci peut économiser beaucoup de frappes, surtout quand on a affaire à des opérations répétitives, par exemple d'énormes tables, dont chaque ligne contient des calculs bizarres mais répétés. Parfois, les macros sont utiles pour rendre vos programmes plus lisibles. Peu de gens contesteraient que la macro ForLp I,1,10 est plus lisible que le code correspondant. Malheureusement, avec les macros, il est également facile de s'emballer et d'obtenir un code inefficace, difficile à lire et à modifier.

Beaucoup de programmeurs soi-disant avancés se plaisent à inventer leurs propres instructions par des définitions de macros pour chaque fonction possible et imaginable sous le soleil. La macro COPY, mentionnée plus tôt, en est un bon exemple. Les processeurs 80x86 ne supportent pas des mov entre opérandes de mémoire. On crée donc une macro qui fait le travail pour nous et bientôt un programme ne ressemble en rien à un programme écrit en assembleur, car un grand nombre d'instructions se réduit à des appels de macros. Bien

sûr, cela peut être une grande idée pour le programmeur qui a créé ces macros et les connaît intimement. Mais pour tout autre programmeur, c'est du pur charabia. Maintenir un programme écrit par quelqu'un d'autre contenant de "nouvelles" instructions implémentées par des macros est une tâche horrible. Par conséquent, n'utilisez que rarement les macros comme moyen pour créer de nouvelles instructions.

Un autre problème avec les macros est qu'elles ont la tendance à cacher les effets de bord. Considérez encore la macro COPY. Si vous trouvez une instruction comme COPY VAR1,VAR2, vous pouvez simplement penser qu'il s'agit d'une innocente ligne qui se limite à copier VAR2 dans VAR1. Faux ! Cette macro détruit le contenu courant de ax en laissant une copie de la valeur de VAR2 dans ce registre. Mais dans l'appel de la macro, ceci n'est pas clair du tout. Considérez la séquence suivante :

```
mov     ax, 5
copy    Var2, Var1
mov     Var1, ax
```

Ce code copie la valeur de Var1 dans Var2 et puis (censément) affecte 5 à Var1. Malheureusement, la macro COPY a écrasé la valeur originale de ax (en y laissant la valeur de Var1), donc cette séquence d'instructions ne modifie pas du tout Var1 !

Un autre problème avec les macros est l'efficacité. Considérez les appels suivants de la macro COPY :

```
copy    Var3, Var1
copy    Var2, Var1
copy    Var0, Var1
```

Ces trois appels génèrent le code :

```
mov     ax, Var1
mov     Var3, ax
mov     ax, Var1
mov     Var2, ax
mov     ax, Var1
mov     Var0, ax
```

Certainement, les deux dernières instructions mov ax, Var1 sont superflues, car le registre ax contient déjà une copie de Var1, il n'y a donc pas besoin de charger chaque fois le registre ax avec Var1. Bien que cette inefficacité est parfaitement évidente dans le code généré, ce n'est pas du tout le cas pour sa macro correspondante.

Un autre problème avec les macros est la complexité. Afin de créer du code efficace, on finit par créer des macros extrêmement complexes avec l'assemblage conditionnel (spécialement ifb, ifidn, etc.), les boucles répétitives et d'autres directives. Malheureusement, ces macros sont des programmes à part entière. On peut avoir des bogues en écrivant des macros tout comme on peut en avoir en écrivant des programmes. Plus les macros seront complexes, plus ce sera probable qu'elles contiendront des bogues qui se répercuteront insidieusement dans votre programme.

Abuser de l'usage des macros, surtout les macros complexes, ne fait que produire un code difficile à lire et encore plus difficile à modifier. Malgré l'enthousiasme de ceux qui aiment les macros, l'usage immodéré de ces dernières provoque plus de bogues qu'il ne peut en prévenir. Si vous avez besoin d'utiliser des macros, allez-y avec beaucoup de précautions.

Les macros ont cependant un très bon côté. Si vous standardisez un jeu de macros et si vous avez soin de bien documenter tous les programmes qui l'utilisent, vous programmes seront plus faciles à écrire tout en restant lisibles. Surtout pour ce qui concerne les macros ayant des noms qui les identifient facilement. La bibliothèque standard UCR utilise les macros dans beaucoup de cas. Vous en lirez plus dans le prochain chapitre.

## 8.15 Opérations répétitives

Un autre format de macro (au moins en se tenant à la définition de Microsoft) est la macro *repeat*. Une telle macro n'est rien de plus qu'une boucle qui répète ses instructions un nombre spécifié de fois. Il y a trois types de macros répétitives que MASM fournit : repeat/rept, for/irp et forc/irpc. La macro repeat/rept utilise la syntaxe suivante :

```
repeat  expression
```

```
        <instructions>
                endm
```

*Expression* doit être une expression numérique qui peut être évaluée comme constante non signée. La directive *repeat* duplique toutes les instructions entre repeat et endm le nombre spécifié de fois. Le code suivant génère une table de 26 octets contenant les 26 lettres majuscules :

```
CodeASCII       =       'A'
                repeat 26
                byte    CodeASCII
CodeASCII       =       CodeASCII+1
                endm
```

Le symbole CodeASCII se voit affecter le code ASCII de la lettre "A". La boucle se répète 26 fois en émettant un octet avec la valeur courante de CodeASCII et en incrémentant ce symbole, de sorte qu'à chaque répétition il contient le code ASCII du prochain caractère du jeu de caractères. Ce qui génère en fait :

```
                byte    'A'
                byte    'B'
                 .
                 .
                 .
                byte    'Y'
                byte    'Z'
CodeASCII       =       27
```

Notez que la boucle repeat s'exécute pendant l'assemblage et non pendant l'exécution. repeat n'est pas un mécanisme pour créer des boucles à l'intérieur de votre programme, mais pour insérer des sections de code répétitives dans votre code source. Si votre but est de créer une boucle, utilisez l'instruction loop. Bien que les deux séquences de code suivantes produisent le même résultat, elles ne sont *pas* la même chose :

```
; Séquence de code produisant une boucle pendant l'exécution :

                mov     cx, 10
AddLp:          add     ax, [bx]
                add     bx, 2
                loop    AddLp

; Séquence de code produisant une boucle pendant l'assemblage :
                repeat  10
                add     ax, [bx]
                add     bx, 2
                endm
```

La première séquence émet dans le code objet quatre instructions machine. Pendant l'exécution, le CPU exécute les instructions entre AddLp et loop dix fois sous le contrôle de l'instruction loop. La seconde séquence émet 20 instructions dans le fichier de code objet. Pendant l'exécution, le CPU exécute simplement ces 20 instructions séquentiellement, sans transferts de contrôle. La seconde version sera plus rapide, car le processeur n'aura pas à exécuter une instruction loop à toutes les trois instructions. Mais la deuxième version est aussi beaucoup plus longue, car elle duplique le code du corps de la boucle dix fois dans le fichier de code objet.

Contrairement aux macros standard, vous n'avez pas à définir et à évoquer les macros repeat séparément. MASM émet le code entre repeat et endm à l'occurrence de la directive repeat. Il n'y a pas d'étape d'appel séparée. Si vous voulez créer une macro repeat qui pourra être appelée partout dans votre programme, considérez ce qui suit :

```
REPTMacro       macro   Count
                repeat  Count
        <instructions>
                endm
                endm
```

En plaçant une macro repeat à l'intérieur d'une macro standard, vous pouvez l'appeler partout dans votre programme en appelant simplement la macro REPTMacro. Notez que vous aurez besoin de deux directives endm, une pour mettre fin à la macro repeat, l'autre pour mettre fin à la macro standard.

La macro *rept*, également prédéfinie, est synonyme de *repeat*. *repeat* est juste la nouvelle version, MASM supporte les deux pour des raisons de compatibilité avec des anciens programmes. Mais vous devriez toujours utiliser *repeat*.

## 8.16 Les macros itératives FOR et FORC

Une autre version de la macro repeat est la macro *for.* Cette macro a la syntaxe suivante :

```
for     parameter,<item1 {,item2 {,item3 {,...}}}>
<instructions>
endm
```

Les "<>" autour des opérandes sont requis, alors que les accolades entourent des éléments optionnels et ne doivent pas figurer dans l'instruction.

La directive *for* développe les instructions entre les mots-clés for et endm. De plus, pour chaque itération, *parameter* prend la valeur de chacun des éléments qui se suivent. Considérez la boucle suivante :

```
for     value,<0,1,2,3,4,5>
byte    value
endm
```

Ici, la boucle émet six octets avec les valeurs successives 0, 1, 2, 3, 4 et 5, et c'est absolument identique à la séquence d'instructions :

```
byte 0
byte 1
byte 2
byte 3
byte 4
byte 5
```

Gardez à l'esprit que la boucle *for*, tout comme la boucle *repeat*, s'exécute pendant l'assemblage et non pendant l'exécution. La seconde opérande de la boucle for n'a pas besoin d'être une constante textuelle littérale. Vous pouvez lui suppléer un paramètre de macro, un résultat de macro fonctionnelle ou une affectation textuelle. Mais ce paramètre *doit* se développer sous forme d'une constante textuelle avec des délimiteurs de texte autour de lui.

La macro *irp* est un vieil et obsolète synonyme de *for*. MASM l'accepte uniquement pour des raisons de compatibilité avec d'anciens programmes. Mais pour les programmes actuels, il n'y a aucune raison de l'utiliser.

La troisième possibilité d'itération dans les macros est fournie par la macro *forc*. Sa différence avec la macro for est qu'elle répète une boucle le nombre de fois spécifié par la longueur d'une chaîne de caractères, au lieu que par le nombre d'opérandes présentes. Sa syntaxe est :

```
forc parametre,<string>
<instructions>
endm
```

Les instructions de la boucle se répètent pour chaque caractère dans l'opérande chaîne (string). Ici aussi les caractères "<>" doivent entourer le deuxième paramètre. Considérez la boucle suivante :

```
forc value,<012345>
byte value
endm
```

Cette boucle produit le même code que la boucle qu'on a vue précédemment.

La macro *irpc* est à forc ce que *irp* est à for. Elle est un vieil équivalent qui n'existe que pour des raisons de compatibilité. Vous n'avez pas de raisons pour l'utiliser.

## 8.17 La macro WHILE

Cette macro permet de développer un nombre indéfini de fois une séquence de code dans un fichier source. Pendant l'assemblage, l'expression est évaluée avant l'exécution du corps de la boucle (et donc l'émission du code). La syntaxe pour cette macro est :

```
while expression
<instructions>
endm
```

Cette macro évalue l'expression lors de l'assemblage. Si la valeur de l'expression est 0, alors tout le code à l'intérieur de la macro sera ignoré. Si au contraire l'expression a une valeur différente de zéro (vraie), alors MASM assemble les instructions jusqu'à la directive endm, puis il réévalue l'expression pour déterminer si les instructions du corps de la macro doivent être encore émises.

Normalement, la directive while répète les instructions entre *while* et *endm* tant que l'expression qu'elle évalue est vraie. Cependant, vous pouvez aussi utiliser la directive *exitm* pour terminer prématurément l'exécution du corps de la macro. Gardez à l'esprit que vous devez fournir en tout cas des conditions de fin de boucle, sinon MASM produira une boucle infinie et émettra continuellement son code dans le fichier objet, jusqu'à remplir complètement votre disque dur (ou simplement il produira une boucle infinie s'il n'y a pas d'instructions à émettre).

## 8.18 Paramètres de macro

Les macros standard de MASM sont très flexibles. Si le nombre de paramètres lors de l'appel ne correspond pas au nombre de paramètres formels (c'est-à-dire, ceux apparaissant comme opérandes dans la définition de macro), MASM ne se plaindra pas nécessairement. S'il y a plus de paramètres que les paramètres formels, MASM ignorera simplement ces paramètres additionnels et se limitera à générer un avertissement. Si au contraire il y a plus de paramètres formels que de paramètres passés à l'appel, MASM les remplacera par une chaîne vide "<>". En utilisant les directives d'assemblage conditionnel ifb et ifnb, vous pouvez tester cette dernière condition. Mais là où cette technique de substitution est flexible, elle laisse aussi ouverte la possibilité d'erreurs. Si vous demandez que les programmeurs passent exactement trois paramètres et qu'ils en passent moins, MASM ne générera pas d'erreur. Et si vous oubliez de tester la présence de chaque paramètre en utilisant ifb, vous pourriez générer du mauvais code. Pour surmonter cette limitation, MASM donne la possibilité de spécifier que certains paramètres de macros sont requis. Vous pouvez aussi attribuer une valeur par défaut à un paramètre si la programmation n'en fournit pas. Finalement, MASM permet aussi un nombre variable d'arguments[21] de macro.

Si vous voulez spécifier qu'un paramètre de macro est obligatoire, placez simplement ":req" (required) après le paramètre de macro dans la définition de votre macro. Pendant l'assemblage, MASM générera une erreur si ce paramètre particulier est omis.

```
Needs2Parms     macro   parm1:req, parm2:req
                .
                .
                .
                endm
                .
                .
                .
                Needs2Parms ax          ;Génère une erreur.
                Needs2Parms             ;Erreur.
                Needs2Parms ax, bx      ;Bien.
```

Une autre possibilité est d'attribuer une valeur par défaut pour une macro en cas d'omission. Pour ce faire, utilisez simplement l'opérateur ":=<texte>", immédiatement après le nom de paramètre dans la liste formelle des paramètres (c'est-à-dire dans la définition de la macro). Par exemple, la fonction int 10h du BIOS fournit plusieurs services de vidéo. L'un des plus communément utilisés est la fonction ah=0eh qui produit l'affichage à l'écran du caractère qui se trouve dans al. La macro qui suit permet à l'appelant de spécifier quelle fonction utiliser et fournit la fonction 0eh par défaut si le paramètre n'est pas spécifié :

```
Video           macro   service := <0eh>
                mov     ah, service
```

[21] "Arguments" et "Paramètres" sont des synonymes dans le jargon informatique, n.d.t.

```
                int     10h
endm
```

La dernière facilité que MASM fournit avec les paramètres est la possibilité de spécifier un nombre variable d'arguments. Pour ce faire, on place simplement l'opérateur ":vararg" après le dernier paramètre formel de la définition de la macro. MASM associe les *n* premiers paramètres réels avec les paramètres formels correspondants qui apparaissent avant le dernier paramètre formel et il crée ensuite une affectation textuelle de tous les paramètres restants au paramètre formel suffixé avec l'opérateur ":vararg". Vous pouvez vous servir de la macro for pour extraire chaque paramètre de la liste des paramètres variables. Par exemple, la macro suivante vous permet de déclarer un nombre arbitraire de tableaux à deux dimensions, tous de la même taille. Les premiers deux paramètres spécifient le nombre de lignes et des colonnes et tous les arguments optionnels restants permettent de spécifier les noms des tableaux :

```
MkArrays        macro   NumRows:req, NumCols:req, Names:vararg
                for     AryName, Names
&AryName&       word    NumRows dup (NumCols dup (?))
                endm
                endm
                .
                .
                .
MkArrays 8, 12, A, B, X, Y
```

## 8.19 Contrôle du listing

MASM offre plusieurs directives d'assemblage utiles pour contrôler la sortie de l'assembleur. Ces directives comprennent *echo*, *%out*, *title*, *subttl*, *page*, *.list*, *.nolist* et *.xlist*. Il y en a d'autres mais celles-ci sont les plus importantes.

### 8.19.1 Les directives ECHO et %OUT

Ces directives affichent simplement, pendant l'assemblage, tout ce qui apparaît dans leur opérande. Nous avons vu des exemples des directives echo et %out dans les sections sur l'assemblage conditionnel et sur les macros. Notez que %out est une ancienne version de echo, fournie par souci de compatibilité avec des anciennes versions. Vous devriez utiliser echo dans tous vos programmes.

### 8.19.2 La directive TITLE

Cette directive assigne un titre à votre fichier source. Seulement une directive *title* peut figurer dans votre programme. La syntaxe est :

```
                title   texte
```

MASM affichera le texte spécifié en haut de chaque page du listing.

### 8.19.3 La directive SUBTTL

La directive subttl (subtitle) est semblable à la directive title, sauf que plusieurs sous-titres peuvent apparaître dans votre fichier source. Les sous-titres apparaissent immédiatement au-dessous du titre dans l'en-tête de chaque page du listing. La syntaxe est :

```
                subttl  texte
```

Le texte spécifié sera le nouveau sous-titre. Notez que MASM n'affichera pas le nouveau sous-titre tant que la page courante n'est pas terminée. Si vous voulez placer un sous-titre dans la même page du code immédiatement après la directive, utilisez la directive *page* (décrite ci-dessous) pour forcer un saut de page.

### 8.19.4 La directive PAGE

Cette directive a deux fonctions : peut forcer un saut de page dans le listing assemblé et peut fixer la largeur et la hauteur du périphérique de sortie. Pour forcer un saut de page, la version à utiliser est celle-ci :

```
page
```

Si vous placez un signe "+" comme opérande, MASM effectue le saut de page, incrémente le numéro de section et réinitialise le numéro de page à 1. MASM affiche le nombre de pages en se servant du format :

```
section-page
```

Si vous voulez profiter de la possibilité de numéroter les pages par sections, vous aurez à insérer un saut de page avec une opérande + devant de chaque nouvelle section.

La seconde version de la commande page vous permet de spécifier la largeur et la longueur du listing imprimé. En voici la syntaxe :

```
page    longueur, largeur
```

où *longueur* est le nombre de lignes pour chaque page (qui normalement correspond à 50, mais le meilleur choix pour la plupart des imprimantes est entre 56 et 60), et *largeur* est le nombre de caractères pour chaque ligne. La largeur par défaut est de 80 caractères. Si votre imprimante peut imprimer 132 colonnes, changez cette valeur à 132, ainsi votre fichier source sera plus facile à lire. Notez que certaines imprimantes, même si leur carreau couvre au plus une largeur de 8 et ½ pouces, peuvent toutefois imprimer 132 colonnes en format condensé. Normalement, certains caractères de contrôle doivent être envoyés à l'imprimante pour la mettre en mode condensé. Vous pouvez insérer ce caractère de contrôle dans un commentaire au début de votre fichier source.

### 8.19.5 Les directives .LIST, .NOLIST et XLIST

Ces directives peuvent être utilisées pour n'afficher que certaines parties du code source. .list active la visibilité du code en question et .nolist la désactive, alors que .xlist n'est qu'une version obsolète de .nolist présente seulement pour des questions de compatibilité.

L'usage de ces trois directives vous permet d'afficher seulement les portions de code qui vous intéressent. Aucune d'elles n'accepte d'opérandes. Leur syntaxe est simplement :

```
.list
.nolist
.xlist
```

### 8.19.6 Autres directives de listing

MASM fournit plusieurs autres directives de contrôle de listing que ce chapitre ne couvrira pas. Celles-ci vous permettent de contrôler la sortie des macros, des segments d'assemblage conditionnel et ainsi de suite. Regardez les annexes pour avoir plus de détails.

## 8.20 Gérer de larges programmes

Beaucoup de programmes d'assembleur ne sont pas simplement des programmes d'un seul listing. En général, on appelle toujours diverses routines de bibliothèque standard ou d'autres routines qui ne sont pas définies dans votre programme principal. Par exemple, vous aurez probablement remarqué que l'architecture 80x86 ne fournit pas d'instructions comme "read", "write" ou "printf" pour effectuer des opérations d'entrées/sorties. En effet, les seules instructions qui s'en rapprochent sont in et out, qui ne sont autre chose que des versions spéciales de l'instruction mov ; et, bien-sûr, les directives echo et %out qui font la même chose, mais pendant l'assemblage, et non à l'exécution, comme vous le souhaiteriez. Dans les deux cas, pas question des entrées/sorties de haut niveau. Pour obtenir de tels résultats, il faut écrire des procédures qui effectuent des opérations telles que "read" et "write". Malheureusement, écrire de telles routines est une tâche souvent complexe, et bien évidemment, les débutants ne sont pas prêts à de tels exploits. C'est là que la bibliothèque

standard UCR vient en aide. On y trouve, par exemple, une procédure permettant d'effectuer des opérations d'entrées/sorties de façon transparente, tout comme "printf".

La bibliothèque standard UCR contient de milliers de lignes de code source. Imaginez comme ce serait difficile de programmer s'il fallait écrire ces longs codes dans vos simples programmes. Heureusement, on n'a pas à le faire.

Pour de petits programmes, travailler avec un seul fichier source, c'est correct. Pour des programmes plus grands, ceci devient vite ingérable (pour vous en convaincre, considérez l'exemple qu'on vient de faire, où il fallait inclure le contenu des bibliothèques dans vos programmes). De plus, une fois que vous avez débogué et testé une grosse portion de code, continuer à assembler ce même code chaque fois qu'on doit modifier d'autres portions du programme, devient une tâche ennuyeuse qui prend du temps. La bibliothèque standard, par exemple, emploie plusieurs minutes à s'assembler, même sur des machines rapides. Imaginez de devoir attendre de cinq à dix minutes sur un Pentium pour assembler un programme après avoir changé une ligne !

La solution, tout comme avec les langages de haut niveau, est la *compilation séparée* (ou l'assemblage séparé, dans le contexte de MASM). D'abord, on organise "son gros fichier source" en morceaux maniables. Ensuite, on assemble les fichiers séparément dans des modules de code objet. Finalment on relie (link) ensemble ces modules objet pour former un programme complet. Si on a besoin de faire un petit changement dans l'un des modules, on ne fait que réassembler ce module et ensuite le lier sans avoir besoin de réassembler le programme en entier.

La bibliothèque standard fonctionne précisément de cette façon. Elle est déjà assemblée et prête à l'usage. On ne fait rien d'autre qu'appeler ses routines et les relier avec le code objet de notre programme. Pour cela, on utilise un éditeur de liens (linker)[22], qui est un programme de linkage. Ceci permet d'économiser beaucoup de temps dans le développement des programmes qui utilisent le code de la bibliothèque standard. Vous pouvez bien évidemment créer vos propres modules de code objet et le relier ensuite avec votre code source principal. Vous pourriez même ajouter de nouvelles routines dans la bibliothèque standard pour les utiliser ensuite dans les programmes que vous écrivez.

« Programmer au long » (*programming in the large*), est un terme que les ingénieurs logiciels ont forgé pour décrire les processus, les méthodologies et les outils pour venir au bout du développement de grands projets logiciels. Alors que chacun a sa propre idée de ce que le terme "grand" implique, la compilation séparée et certaines conventions qui s'y rattachent, sont parmi les techniques principales de ce processus. Les sections qui suivent décrivent les outils que MASM fournit pour réaliser la compilation séparée et comment les employer au mieux dans vos programmes.

### 8.20.1 La directive INCLUDE

Quand cette directive fait son apparition dans un fichier source, elle fait passer l'entrée du programme du fichier courant au fichier spécifié par son paramètre. Ce qui vous permet de définir des fichiers textes spécifiques ne contenant que du code général, comme des constantes symboliques, des macros, des définitions de procédures et ainsi de suite, qui pourront être incluses dans le programme courant ou dans divers programmes séparés. La syntaxe de la directive include est :

```
include         nom_de_fichier
```

Où *nom_de_fichier* doit être un nom de fichier DOS correct. MASM fusionne le fichier spéficié à l'endroit où la directive include apparaît. Notez que vous pouvez imbriquer des includes à l'intérieur d'autres fichiers que vous incluez. Autrement dit, un fichier qui est inclus dans un autre fichier pendant l'assemblage peut inclure lui-même un autre fichier.

Utiliser la directive include en soi ne produit pas une compilation séparée. Vous pouvez par conséquent l'utiliser pour diviser un grand fichier en modules séparés et logiquement reliés, puis les joindre ensemble lors de l'assemblage du fichier principal. L'exemple suivant comporte l'inclusion du fichier PRINTF.ASM et PUTC.ASM pendant l'assemblage :

```
include         printf.asm
include         putc.asm
```

[22] La traduction française du terme "linker" est assez courante, n.d.t.

```
        <Ici le code de votre programme>

        end
```

Maintenant, votre programme tirera profit de la modularité obtenue par cette approche. Hélas ! Vous n'en gagnerez pas en temps de développement. La directive include insère le fichier spécifié dans votre fichier source pendant l'assemblage, exactement comme si vous l'aviez tapé manuellement dans votre fichier source. MASM devra encore assembler ce code supplémentaire, ce qui prend du temps. Si vous incluiez *tous* les fichiers de la bibliothèque standard, la durée de l'assemblage serait *interminable.*

En général, il *ne* faut *pas* utiliser la directive include de la manière qu'on a montré ci-dessus[23], mais il faut au contraire l'utiliser pour inclure un jeu commun de constantes (affectations textuelles ou *equates*), macros, déclarations de procédures externes et ainsi de suite. Normalement, un fichier d'inclusion ne contient *aucun* code machine (à part les macros). Le raison pour laquelle il faudrait se servir des fichiers d'inclusion de cette façon vous sera plus claire après avoir vu comment les déclarations externes et publiques fonctionnent.

### 8.20.2 Les directives PUBLIC, EXTERN et EXTRN

Techniquement, la directive include vous fournit tous les moyens de créer des programmes modulaires. Vous pouvez mettre en place une bibliothèque de modules, chacun contenant certaines routines spécifiques, et inclure ensuite tous les modules nécessaires à vos programmes. Mais MASM, à l'égal que éditeur de liens associé, fournit un meilleur moyen : les symboles *extern* et *public*.

Un des majeurs problèmes avec le mécanisme d'inclusion est qu'une fois que vous avez débogué une routine, le fait de l'inclure dans l'assemblage consomme beaucoup de temps, car MASM doit assembler de nouveau tout code libre de bogues chaque fois que vous assemblez le rpogramme principal. Une solution supérieure serait de préassembler les modules débogués et de lier ensuite les modules code objet au lieu de réassembler tout le programme chaque fois que vous modifiez un seul module. C'est précisément de ceci que les directives *public* et *extern* se chargent. *extrn* est une ancienne version de extern, maintenue seulement pour des questions de compatibilité. Comme toujours, vous devriez toujours utiliser extern.

Pour pouvoir vous servir des directives public et extern, il faut créer au moins deux codes sources, l'un contenant l'ensemble de variables et des procédures utilisées par l'autre. Le deuxième fichier utilise ces variables et ces procédures sans connaître comment celles-ci ont été implémentées. Comme exemple, considérez les deux modules suivants :

```
;Module #1:
                public  Var1, Var2, Proc1
DSEG            segment para public 'data'
                Var1    word    ?
                Var2    word    ?
                DSEG    ends

CSEG            segment para public 'code'
                assume  cs:cseg, ds:dseg
Proc1           proc    near
                mov     ax, Var1
                add     ax, Var2
                mov     Var1, ax
                ret
Proc1           endp
CSEG            ends
end

;Module #2:
                extern  Var1:word, Var2:word, Proc1:near
CSEG            segment para public 'code'
                .
                .
                .
                mov     Var1, 2
```

[23] Certes, il n'y a rien d'erroné dans cela, simplement on ne tire aucun avantage de cette technique.

```
                mov     Var2, 3
                call    Proc1
                .
                .
                .
CSEG            ends

end
```

Module #2 fait référence à Var, Var2 et Proc1, pourtant ces symboles sont externes à module #2. Par conséquent, vous devrez les déclarer comme tels, à l'aide de la directive *extern*. Cette directive a la syntaxe suivante :

```
extern  nom:type {, nom:type...}
```

Où *nom* est le nom du symbole externe et *type* est le type de ce symbole. Ce dernier peut être near, far, proc, byte, word, dword, qword, tbyte, abs (absolut, c'est-à-dire constant) ou bien un type défini par l'utilisateur.

Ni MASM, ni l'éditeur de liens ne font de vérification pour voir si les types importés par extern correspondent aux types définis par le module utilisant la directive name. Par conséquent, il faudra prêter beaucoup d'attention quand vous définissez des symboles externes. La directive *public* vous permet d'exporter la valeur d'un symbole dans un module externe. Une telle déclaration a la syntaxe suivante :

```
public  nom {, nom}
```

Chaque symbole figurant dans le champ de l'opérande est disponible comme symbole externe pouvant être utilisé par un autre module. De même, tous les symboles externes à l'intérieur d'un module doivent apparaître dans une directive public d'un autre module.

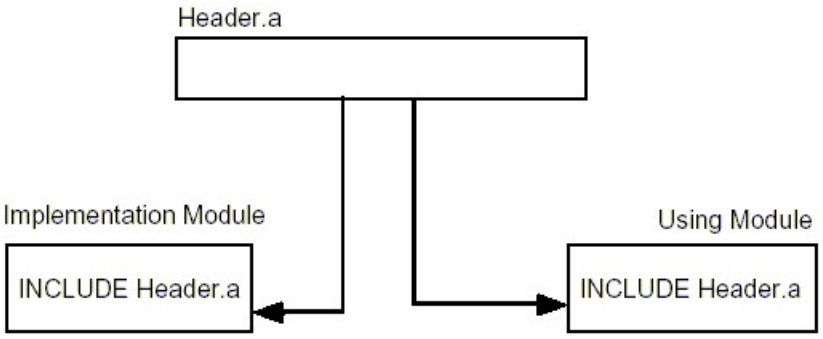


Figure 8.8 Utiliser un seul fichier d'inclusion pour l'implémentation et l'utilisation des modules

Une fois les modules sources créés, il vous faudra assembler en premier le fichier contenant les déclarations publiques. Avec MASM 6.x, vous pouvez utiliser une commande comme :

ML /c pubs.asm

L'option "/c" indique à MASM d'effectuer un assemblage de type "compile only". C'est-à-dire, il n'essaiera pas de relier le code après un assemblage bien mené. Ceci produit un module "pubs.obj".

Ensuite, il vous faudra assembler le fichier contenant les définitions externes et le lier en utilisant la commande :

ML exts.asm pubs.obj

En supposant qu'il n'y a pas d'erreurs, ceci produit le fichier "exts.exe" qui est la version liée et exécutable du programme.

Notez que la directive extern définit un symbole dans votre fichier source. Toute tentative de rédéfinir ce symbole ailleurs dans votre programme produira une erreur de duplication de symbole.

C'est le genre de problèmes que Microsoft a réglé en introduisant la directive externdef.

### 8.20.3 La directive EXTERNDEF

Cette directive est une combinaison des directives public et extern. Elle a la même syntaxe que la directive extern, à savoir on place la paire nom:type dans la liste des champs d'opérande. Si MASM ne trouve pas une autre définition du symbole dans le fichier source courant, alors externdef se comportera exactement comme extern. Alors que si ce symbole apparaît dans le fichier source courant, elle se comportera comme public. Avec externdef, il n'y a réellement pas besoin d'utiliser public ou extern, sauf dans le cas où vous le jugeriez strictement nécessaire.

L'avantage majeur d'externdef est qu'il permet de minimiser la duplication des tâches dans un fichier source. Supposez, par exemple, que vous voulez créer un module avec un groupe de routines de support pour d'autres programmes. En plus de partager des routines et des variables, supposez que vous voulez partager aussi des macros et des constantes. Le mécanisme de l'inclusion fournit un moyen parfait de le faire. Tout ce que vous aurez à faire sera de créer un fichier include contenant les constantes, les macros et les définitions externdef et de l'inclure ensuite dans tous les modules qui utiliseront vos routines.

Notez que dans ce cas, les directives extern et public ne vont pas fonctionner, car le module d'implémentation veut la directive public et le module d'usage veut la directive extern. Et vous ne voulez certes pas créer deux fichiers d'en-tête différents. Maintenir deux fichiers d'en-tête contenant des directives presque identiques n'est pas une bonne idée. La directive externdef offre une solution.

À l'intérieur de vos fichiers d'en-tête, vous créerez des définitions de segment qui correspondent à celles des modules. Assurez-vous de placer la directive externdef à l'intérieur des mêmes segments où le symbole est effectivement défini. Ceci associe une valeur de segment avec le symbole, permettant à MASM d'y apporter les optimisations éventuelles et d'autres calculs basés sur l'adresse complète du segment de façon appropriée.

```
; Du fichier "HEADER.A" :

cseg            segment         para public 'code'
                externdef       Routine1:near, Routine2:far
cseg            ends

dseg            segment         para public 'data'
                externdef       i:word, b:byte, flag:byte
dseg            ends
```

Ce livre adopte une convention de la bibliothèque standard consistant en utiliser le suffixe ".a" pour les fichiers d'en-tête. D'autres suffixes utilisés couramment comprennent ".inc" et ".def".

## 8.21 Fichiers make

Bien que la compilation séparée réduit la durée de l'assemblage et facilite la réutilisation du code et de la modularité, ceci n'est pas sans inconvénient. Supposez qu'un programme consiste en deux modules : pgma.asm et pgmb.asm. Supposez aussi que vous avez déjà assemblé les modules, de sorte que les fichiers pgma.obj et pgmb.obj existent. Finalement, vous avez modifié pgma.asm et vous assemblez pgma.asm, *mais vous avez oublié d'assembler le ficher pgmb.asm*. Il en résulte que ce dernier fichier *n'est plus actualisé*, car son fichier objet ne reflète pas les changements faits. Si vous liez ces deux modules, le fichier .exe contiendra seulement l'actualisation de pgma.asm et non celle du code objet associé à pgmb.asm. À mesure que les projets deviennent grands, le nombre des modules et des programmeurs qui y travaillent croît aussi. Et alors, il devient très difficile de garder la trace des fichiers qui sont à jour.

La complexité a souvent comme conséquence le fait que quelqu'un réassemble tous les modules, ne sachant plus lesquels sont mis à jour et lesquels ne le sont pas. En faisant cela, bien entendu, on perd beaucoup des bénéfices qu'on gagne avec la compilation séparée. Heureusement, il y a un outil permettant de gérer de gros projets sans confusion : nmake. Cet utilitaire, avec une petite contribution de votre part, est en mesure de se rappeler quels fichiers doivent être réassemblés et quels ont déjà leur fichier objet mis à jour. Avec un ficher *make* soigneusement défini, vous n'aurez à assembler que les modules qui en ont vraiment besoin.

Un fichier *make* est un fichier texte qui liste les dépendances lors de l'assemblage entre les divers fichiers. Par exemple, un fichier.exe *dépend* du code source dont l'assemblage produit la version exécutable. Si vous modifiez

ensuite le fichier source, vous aurez besoin d'assembler de nouveau le code source pour produire un nouveau fichier .exe[24].

Typiquement, les dépendances incluent ce qui suit :

- Un fichier exécutable (.exe) généralement ne dépend que du jeu des fichiers objet (.obj) que l'éditeur de liens combine pour réaliser l'exécutable.
- Un fichier objet donné dépend des fichiers source en code assembleur qui ont été assemblés pour le produire. Ceci comprend les fichiers source .asm et tous les fichiers inclus lors de l'assemblage (généralement, les fichiers .a).
- Les fichiers source et d'inclusion ne dépendent généralement de rien.

Un fichier *make* consiste généralement d'une instruction de dépendance suivie d'un ensemble de commandes qui gèrent cette dépendance. Une instruction de dépendance prend la forme suivante :

```
        fichiers dépendants : liste des fichiers
```

Par exemple :

```
pgm.exe: pgma.obj pgmb.obj
```

Cette instruction indique que le fichier "pgm.exe" est dépendant des fichiers pgma.obj et pgmb.obj. Tout changement apporté à un de ces fichiers requerra la génération d'un nouveau fichier pgm.exe.

Le programme nmake.exe utilise un "cachet" date/heure pour déterminer si un fichier dépendant est à actualiser ou non par rapport aux fichiers dont il dépend. Chaque fois que vous apportez un changement au fichier, MS-DOS et Windows mettront à jour *la date et heure de la modification* associée à ce fichier. nmake compare la date/heure de modification du fichier dépendant avec celle du fichier duquel il dépend. Si la date du fichier dépendant est antérieure à un ou plusieurs des fichiers dont il dépend, ou encore si ces derniers ne sont pas présents, alors nmake.exe suppose que certaines opérations sont nécessaires pour que le fichier dépendant soit mis à jour.

Quand une mise à jour est nécessaire, nmake.exe exécute le jeu de commandes DOS qui suit l'instruction de dépendance. Logiquement, ces commandes sont susceptibles de faire ce qu'il faut pour que le fichier soit à jour.

Une instruction de dépendance *doit* commencer à la première colonne de votre ligne. Et toute commande devant s'exécuter pour résoudre la dépendance doit commencer à la ligne suivant l'instruction de dépendance et, de plus, doit être indentée. L'instruction pgm.exe ci-dessus, ressemblerait probablement à ce qui suit :

```
pgm.exe: pgma.obj pgmb.obj
        ml /Fepgm.exe pgma.obj pgmb.obj
```

(L'option "/Fepgm.exe" indique à MASM de nommer "pgm.exe" le fichier exécutable).

Si pour résoudre les dépendances, vous avez besoin d'exécuter plus d'une commande, vous pouvez placer diverses commandes dans l'ordre approprié, après l'instruction de dépendance, naturellement en les indentant toutes d'une tabulation. nmake.exe ignore toute ligne vide dans un fichier make. Par conséquent, vous pouvez ajouter autant de lignes vides qu'il vous plaît, dans le but de rendre le fichier plus facile à lire et à comprendre.

Un fichier make peut avoir plus d'une instruction de dépendance. Dans l'exemple qu'on a vu, pgm.exe dépend de pgma.obj et pgmb.obj. Évidemment, ces fichiers dépendent à leur tour des fichiers source qui les ont générés. Par conséquent, avant de tenter de résoudre les dépendances du fichier pgm.exe, le programme nmake.exe vérifiera d'abord le reste du fichier make, afin de voir si pgma.obj et pbmb.obj dépendent de quelque chose. Si c'est le cas, alors nmake.exe résoudra avant tout ces dépendances. Considérez le fichier make qui suit :

```
pgm.exe: pgma.obj pgmb.obj
        ml /Fepgm.exe pgma.obj pgmb.obj
pgma.obj: pgma.asm
        ml /c pgma.asm
pgmb.obj: pgmb.asm
        ml /c pgmb.asm
```

[24] Bien évidemment, si vous ne changez dans votre fichier source que les commentaires ou d'autres instructions sans effet pour l'exécution, une récompilation ou un réassemblage seront tout à fait superflus. Mais, d'autre part, pour ne pas avoir des doutes, on suppose toujours que *tout* changement du fichier source implique et réquiert un second assemblage.

Le programme nmake.exe commence par évaluer la première dépendance trouvée dans le fichier. Cependant, les fichiers objet dépendent eux-mêmes d'autres fichiers. Par conséquent, nmake.exe s'assurera d'abord que pgma.obj et pgmb.obj soint à jour, avant de tenter d'exécuter MASM pour lier ces fichiers. Donc, si vous n'avez modifié que pgmb.asm, nmake.exe suivra les étapes suivantes (en supposant que pgma.obj existe et qu'il est à jour).

1. nmake.exe évalue la première instruction de dépendance. Il remarque l'existence de lignes de dépendance concernant les fichiers pgma.obj et pgmb.obj (les fichiers dont dépend pgm.exe). Donc, il évalue ces instructions en premier.
2. nmake.exe évalue la ligne de dépendance de pgma.obj et il remarque que pgma.obj est plus récent que le fichier pgma.asm. Donc, *il n'exécutera pas* la commande suivant cette instruction de dépendance.
3. nmake.exe évalue la ligne de dépendance de pgmb.obj et il remarque le fichier objet est plus vieux que pgmb.asm (car on vient justement de modifier le fichier source). Il exécutera donc la commande se trouvant sur la ligne suivante. Ceci générera un nouveau fichier objet, pgmb.obj, maintenant à jour.
4. Après avoir évalué les deux fichiers objet, nmake.exe dirige désormais son attention vers la première ligne de dépendance, à savoir, celle concernant le fichier exécutable. Puisque, nmake.exe vient juste de créer un nouveau fichier pgmb.obj, la date de modification de ce dernier sera plus récente que celle du fichier exécutable. Par conséquent, nmake.exe exécutera la commande ml qui lie pgma.obj et pgmb.obj ensemble pour produire le nouveau fichier pgm.exe.

Notez qu'un fichier make bien écrit indiquera à nmake de n'assembler que les fichiers absolument nécessaires à produire un fichier exécutable consistant. Dans l'exemple qu'on vient de voir, nmake.exe ne se soucie pas de mettre à jour le fichier pgma.obj, car il l'est déjà.

Il y a une dernière chose à signaler au sujet des dépendances. Souvent, les fichiers objet sont dépendants non seulement des fichiers source qui les ont produits, mais aussi de tout autre fichier que le fichier source a inclus. Dans le dernier exemple, apparemment, il n'y avait pas de fichiers d'inclusion. Souvent, ce n'est cependant pas le cas. Un fichier make plus typique, ressemblerait à ce qui suit :

```
pgm.exe: pgma.obj pgmb.obj
        ml /Fepgm.exe pgma.obj pgmb.obj
pgma.obj: pgma.asm pgm.a
        ml /c pgma.asm
pgmb.obj: pgmb.asm pgm.a
        ml /c pgmb.asm
```

Notez que tout changement que vous faites au fichier pgm.a, imposera un nouvel assemblage *des deux* fichiers pgma.asm et pgmb.asm, car les deux dépendent du fichier d'inclusion pgm.a. Laisser les fichiers d'inclusion en dehors d'une liste de dépendance constitue une erreur commune des programmeurs qui peut produire des fichiers .exe inconsistants.

Notez que normalement, vous n'avez pas à spécifier dans la liste des dépendances les fichiers d'inclusion de la bibliothèque standard de l'UCR ni les fichiers avec l'extension .lib de celle-ci. Certainement, vos exécutables qui en résultent dépendront aussi de ces codes, mais la bibliothèque standard ne change que rarement, donc vous pouvez parfaitement l'omettre la plupart du temps. Et à chaque nouvelle version de la bibliothèque, il ne vous restera qu'à effacer vos exécutables et à les refaire.

nmake.exe suppose par défaut qu'il évaluera un fichier make nommé "makefile". Quand vous exécutez nmake.exe, il commence à chercher "makefile" dans le répertoire courant. S'il ne trouve pas ce fichier, il émet une erreur et il arrête[25]. Par conséquent, c'est une bonne idée de placer tous les fichiers de chacun de vos projets dans leur propre répertoire et de donner à chaque projet son makefile. Puis, pour créer un exécutable, vous n'aurez qu'à aller dans le répertoire approprié et qu'à exécuter le programme nmake.exe.

[25]Il y a cependant une option en ligne de commande vous permettant de spécifier un autre nom de fichier. Pour plus de détails, référez-vous la documentation de MASM à propos de nmake.exe.

Bien que cette section décrit le programme nmake d'une manière assez détaillée pour travailler avec la plupart des projets, gardez à l'esprit que nmake.exe fournit également beaucoup de fonctionnalités que ce chapitre ne mentionne pas. Pour en savoir plus, consultez la documentation de MASM.

## 8.22 Exemple de programme

Voici un programme unique montrant la plupart des concepts de ce chapitre. Ce programme est composé de différents fichiers, dont un makefile, que vous pouvez assembler et exécuter à l'aide du programme nmake.exe. Le programme calcule le résultat "croisé" de diverses fonctions. La table de multiplication que vous avez apprise à l'école est un bon exemple de ce genre de calcul, comme le sont les tables de vérité que vous avez vues au chapitre 2. Ce programme génère des tables d'addition, soustraction, division et, optionnellement, de reste (modulo). En plus de mettre en pratique beaucoup des concepts de ce chapitre, il montre comment manipuler des tableaux alloués dynamiquement. Il demande à l'utilisateur de fournir la taille de la matrice et calcule ensuite un jeu approprié de lignes/colonnes pour ce tableau.

### 8.22.1 EX8.MAK

Ce programme contient divers modules. Le makefile suivant assemble tous les fichiers nécessaires pour assurer un fichier .exe consistant.

```
ex8.exe:ex8.obj geti.obj getarray.obj xproduct.obj matrix.a
        ml ex8.obj geti.obj getarray.obj xproduct.obj

ex8.obj: ex8.asm matrix.a
        ml /c ex8.asm

geti.obj: geti.asm matrix.a
        ml /c geti.asm

getarray.obj: getarray.asm matrix.a
        ml /c getarray.asm

xproduct.obj: xproduct.asm matrix.a
        ml /c xproduct.asm
```

### 8.22.2 Matrix.A

MATRIX.A est le fichier d'en-tête contenant les définitions que le programme principal utilisera. Il contient également les instructions externdef pour toutes les routines qui sont définies en dehors.

```
; MATRIX.A
;
; Ce fichier d'inclusion fournit les définitions externes et les définitions
; des types de données pour le programme de matrices d'exemple du chapitre 8.
;
; Certaines définitions de types utiles :

Integer         typedef word
Char            typedef byte

; Quelques constantes communes :

Bell            equ     07      ;Code ASCII pour le caractère cloche.

; Un “Dope Vector” est une structure contenant des informations à propos
; des tableaux qu'un programme alloue dynamiquement durant l'exécution.
; Celui-ci traite des tableaux à deux dimensions et utilise les champs
; suivants :
;
```

```
; TTL - Pointe sur une chaîne terminée par zéro et contenant une description
; des données du tableau.
;
; Func - Pointeur sur la fonction à calculer pour cette matrice.
;
; Data - Pointeur sur l'adresse de base du tableau.
;
; Dim1 - C'est un mot contenant le nombre de lignes dans le tableau.
;
; Dim2 - C'est un autre mot contenant le nombre d'éléments par ligne (autrement
; dit, le nombre de colonnes).
;
; ESize - Contient le nombre d'octets par élément du tableau.

DopeVec         struct
TTL             dword   ?
Func            dword   ?
Data            dword   ?
Dim1            word    ?
Dim2            word    ?
ESize           word    ?
DopeVec         ends

; Certaines égalités textuelles (text equates) que la matrice utilisera
; couramment :

Base            textequ <es:[di]>

byp             textequ <byte ptr>
wp              textequ <word ptr>
dp              textequ <dword ptr>

; Déclarations de procédure

InpSeg          segment para public 'input'

                externdef geti:far
                externdef getarray:far

InpSeg          ends


cseg            segment         para public 'code'
                externdef       CrossProduct:near

cseg            ends


; Déclaration de variables

dseg            segment         para public 'data'

                externdef       InputLine:byte

dseg ends


; Enlevez le commentaire des constantes symboliques suivantes si vous voulez
; activer les instructions de débogage ou si vous voulez inclure la
; fonction MODULO.

;debug          equ     0
;DoMOD          equ     0
```

### 8.22.3 EX8.ASM

Voici le programme principal. Il appelle les routines appropriées pour obtenir les entrées et donner les résultats.

```
; Programme d'exemple pour le chapitre Huit, montrant l'utilisation
; de plusieurs caractéristiques de MASM qui ont été discutées dans ce
; chapitre, en incluant les divers types d'étiquettes, les constantes,
; l'organisation des segments, les procédures, les constantes textuelles
; (text equates), les expressions d'adresse, les opérateurs de type et de
; coercition de type, les préfixes de segment, la directive assume,
; l'assemblage conditionnel, les macros, les directives de listing,
; l'assemblage séparé et, bien entendu, la bibliothèque standard UCR.
;
; Ce programme inclut les fichiers d'en-tête pour l'utilisation de la
; bibliothèque standard. Notez que le fichier "stdlib.a" définit deux
; segments. Dans ce programme, MASM charge ces segments en mémoire
; avant "dseg".
;
; La directive ".nolist" indique à MASM qu'il ne faut pas développer les macros
; de la bibliothèque standard pendant l'édition du listing de code. Le
; faire augmenterait la taille du listing de dizaines de pages supplémentaires
; qui ne feraient que rendre confus et obscur le code réel de ce programme.
;
; La directive ".list" active de nouveau la fonction de listing après que
; MASM aura dépassé les fichiers de la bibliothèque standard. Notez
; que ces deux directives ne sont actives que quand vous produisez un listing
; en utilisant le paramètre de ligne de commande "/F1".

                .nolist
                include stdlib.a
                includelib stdlib.lib
                .list

; La prochaine instruction inclut le fichier d'en-tête spécial pour ce
; programme spécifique. Il contient des définitions externes et diverses
; définitions de types de données.

                include matrix.a

; Les deux instructions qui suivent nous permettent d'utiliser dans ce
; programme les instructions du processeur 80386. La directive ".386"
; active ce jeu d'instructions, alors que la directive "option" indique
; à MASM d'utiliser des segments de 16 bits par défaut (car en mode
; "jeu d'instructions 80386" le défaut serait des segments de 32 bits).
; Mais les programmes qui s'exécutant sous DOS en mode réel doivent être
; écrits avec des segments de 16 bits.

                .386
                option  segment:use16
dseg            segment para public 'data'

Rows            integer ?       ;Nombre de lignes dans la matrice
Columns         integer ?       ;Nombre de colonnes dans la matrice

; "InputLine" est un tampon d'entrée utilisé par ce code pour lire une
; chaîne textuelle fournie par l'utilisateur. En particulier, la
; procédure GetWholeNumber passe l'adresse de InputLine à la routine GETS,
; laquelle lit une ligne de texte depuis l'entrée standard et place chaque
; caractère dans ce tableau (InputLine). GETS lit un maximum de 127 caractères,
; plus Entrée. Puis il fait terminer cette chaîne avec zéro (en remplaçant le
```

```
; code ASCII de la touche Entrée avec zéro). Par conséquent ce tampon[26] (buffer)
; doit faire 128 octets de long pour éviter la possibilité de dépassement de
; capacité du tampon.
;
; Notez que le module GetArray utilise aussi ce tampon.

InputLine       char    128 dup (0)

; Les deux pointeurs suivants, pointent sur un tableau d'entiers.
; Ce programme alloue dynamiquement de l'espace pour les données du tableau
; courant, une fois que l'utilisateur a indiqué sa taille au programme.
; Les variables Rows et Columns déterminent les tailles respectives
; des lignes et des colonnes. Après avoir alloué de l'espace à l'aide
; de MALLOC, ce programme stocke les pointeurs sur ces tableaux dans les
; deux variables pointeur suivantes :

RowArray        dword   ?       ;Pointeur sur les lignes
ColArray        dword   ?       ;Pointeur sur les colonnes

; ResultArrays est un tableau du genre "dope vector"(*) servant à stocker les
; résultats des opérateurs de matrice :
;
; [0]- table d'addition
; [1]- table de soustraction
; [2]- table de multiplication
; [3]- table de division
;
; [4]- table de modulo (reste de division) -- si le symbole “DoMOD” est défini.
;
; La constante symbolique qui suit la déclaration de ResultArrays calcule le
; nombre d'éléments dans le tableau. “$” est l'offset dans dseg immédiatement
; après le dernier octet de ResultArrays. En soustrayant cette valeur de
; ResultArrays, on obtient le nombre d'octets qu'il contient. En divisant
; ceci par la taille d'un "dope vector" produit le nombre d'éléments du
; tableau. Il s'agit d'un excellent exemple qui montre comment on peut utiliser
; des expressions d'adresse en assembleur.
;
; Le code IFDEF DoMOD montre combien c'est facile d'étendre la matrice.
; Définir le symbole “DoMOD” a pour effet d'inclure une autre entrée
; pour ce tableau. Le reste du programme s'y adapte automatiquement.
;
; Vous pouvez facilement ajouter de nouveaux éléments à ce tableau de
; "vecteurs dope". Vous aurez simplement à fournir un titre et une fonction
; pour donner ces nouvelles capacités.
; Bref, ce programme est capable de s'adapter automatiquement à tout nouveau
; ajout qu'on peut faire au tableau de "dope vectors".
;
; (*) Un “Dope Vector”, mot intraduisible, représente une structure de données
; décrivant un tableau alloué dynamiquement. Un vecteur dope typique contient
; la valeur maximale pour chaque dimension, un pointeur sur le tableau en
; mémoire, et d'autres informations possibles. Ce programme possède également
; un pointeur sur titre de un tableau et un autre pointeur sur une fonction
; arithmétique dans le vecteur dope.

ResultArrays    DopeVec {AddTbl,Addition}, {SubTbl,Subtraction}
                DopeVec {MulTbl,Multiplication}, {DivTbl,Division}

                ifdef   DoMOD
                DopeVec {ModTbl,Modulo}
                endif
```

[26] Dans le jargon informatique, les mots "tampon" et "tableau" sont presque des synonymes, n.d.t.

```
; On peut ici ajouter toute nouvelle fonction, avant la constante symbolique
; suivante :

 RASize         =       ($-ResultArrays) / (sizeof DopeVec)

; Titres pour chacune des quatre (ou cinq) matrices.

AddTbl          char    "Addition Table",0
SubTbl          char    "Subtraction Table",0
MulTbl          char    "Multiplication Table",0
DivTbl          char    "Division Table",0

                ifdef   DoMOD
ModTbl          char "Modulo (Remainder) Table",0
                endif

; Ce serait ici un bon endroit pour placer un titre pour tout nouveau tableau
; que vous créez.

dseg            ends

; Mettre PrintMat à l'intérieur de son propre segment montre que vous
; pouvez avoir plusieurs segments de code dans un même programme. Il n'y a
; pas de raison de ne pas placer "PrintMat" dans CSEG, sinon que pour montrer
; un appel éloigné dans un segment différent.

PrintSeg        segment para public 'PrintSeg'

; PrintMat :    Affiche une matrice montrant une table de calcul.
;
;               Au début de l'exécution :
;
;                       DS doit pointer sur DSEG.
;                       DS:SI pointe sur la position de ResultArrays qu'il faut
;                       afficher.
;
; La sortie a l'allure suivante :
;
; Titre de la matrice
;
;               <- valeur des colonnes de la matrice ->
;
;       ^       *-------------------------*
;       |       |                         |
;       V       |                         |
;       a       | Valeurs croisées        |
;       l       | de la matrice           |
;       e       |                         |
;       u       |                         |
;       r       |                         |
;       s       |                         |
;               |                         |
;       l       |                         |
;       i       |                         |
;       g       |                         |
;       n       |                         |
;       e       |                         |
;       s       |                         |
;       |       |                         |
;       v       *-------------------------*

PrintMat        proc    far
                assume  ds:dseg
```

```
; Notez l'utilisation de l'assemblage conditionnel pour insérer des
; instructions de débogage additionnelles si un symbole spécial "debug"
; est défini pendant l'assemblage. Dans le cas contraire, les instructions
; qui suivent seront ignorées :

                ifdef   debug
                print
                char    "In PrintMat",cr,lf,0
                endif

; D'abord, afficher le titre de cette table. Le champ TTL dans le "dope vector"
; contient un pointeur sur une chaîne terminée par zéro. Charger ensuite ce
; pointeur dans es:di et appeler PUTS pour afficher cette chaîne.

                putcr
                les     di, [si].DopeVec.TTL
                puts

; Maintenant, afficher les valeurs des colonnes. Notez l'utilisation de
; PUTISIZE, qui fait prendre à chaque valeur exactement six positions
; d'affichage. La boucle suivante répète une fois pour chaque élément du
; tableau des colonnes (le nombre d'éléments dans ce tableau est donné par
; le champ Dim2 du "dope vector").

                print                           ;Sauter des espaces pour aller
                char    cr,lf,lf,"      ",0      ;après les valeurs des lignes.
                mov     dx, [si].DopeVec.Dim2   ;Répéter la boucle Dim2 n fois.
                les     di, ColArray            ;Adresse de base du tableau.
ColValLp:       mov     ax, es:[di]             ;Obtenir l'élément courant.
                mov     cx, 6                   ;Afficher la valeur en utilisant
                putisize                        ;un minimum de six positions.
                add     di, 2                   ;Prochain élément.
                dec     dx                      ;Répéter.
                jne     ColValLp
                putcr                           ;Fin de l'affichage de colonne
                putcr                           ;Insérer une ligne vide.

; Afficher chaque ligne de la matrice. Notez qu'on aura à afficher la valeur
; de RowArray avant chaque ligne de la matrice.
;
; RowLp est la boucle externe qui se répète pour chaque ligne.

                mov     Rows,   0               ;Répéter de 0 à Dim1-1 lignes.
RowLp:          les     di, RowArray            ;Imprimer la valeur courante de
                mov     bx, Rows                ;RowArray à gauche de la
                add     bx, bx                  ;matrice.
                mov     ax, es:[di][bx]         ;ES:DI = la base, BX = l'index
                mov     cx, 5                   ;Afficher utilisant 5 positions.
                putisize
                print
                char    ": ",0

; ColLp est la boucle intérieure qui se répète pour chaque élément
; de chaque ligne.

                mov     Columns, 0      ;Répéter pour 0..Dim2-1 colonnes.
ColLp:          mov     bx, Rows        ;Calculer l'index du tableau
                imul    bx, [si].DopeVec.Dim2 ; index := (Rows*Dim2 +
                add     bx, Columns     ; columns) * 2
                add     bx, bx

; Notez que nous avons seulement un pointeur sur l'adresse de base du tableau,
```

```
; donc, on doit charger celui-ci et l'indexer pour accéder à l'élément désiré.
; Ce code charge le pointeur avec l'adresse de base du tableau dans la paire
; de registres es:di.

                les     di, [si].DopeVec.Data   ;Adresse de base du tableau.
                mov     ax, es:[di][bx]         ;Obtenir l'élément du tableau

; Les fonctions calculant les valeurs pour le tableau stockent la valeur
; 8000h dans l'élément du tableau s'il se produit une erreur.
; Bien sûr, on peut produire 8000h comme résultat courant, mais cela vaut
; toujours la peine de réserver une valeur spécifique comme code d'erreur. Le
; code suivant vérifie si une erreur s'est produite pendant le calcul. Si
; c'est le cas, le code affiche " ****", sinon, il affiche simplement
; le résultat courant.

                cmp     ax, 8000h       ;Vérification d'erreur
                jne     GoodOutput
                print
                char    " ****",0       ;Afficher en cas d'erreur.
                jmp     DoNext
GoodOutput:     mov     cx, 6           ;Utiliser six positions d'affichage.
                putisize                ;Afficher une valeur correcte.
DoNext:         mov     ax, Columns     ;Prochain élément du tableau
                inc     ax
                mov     Columns, ax
                cmp     ax, [si].DopeVec.Dim2 ;Voir si la fin de cette colonne
                jb ColLp                      ; a été atteinte.
                putcr                   ;Terminer chaque colonne avec CR/LF
                mov     ax, Rows        ;Prochaine ligne.
                inc     ax
                mov     Rows, ax
                cmp     ax, [si].DopeVec.Dim1 ;Toutes les lignes terminées ?
                jb RowLp                      ;Répéter sinon.
                ret
                PrintMat endp
                PrintSeg ends

cseg            segment para public 'code'
                assume  cs:cseg, ds:dseg

; GetWholeNum : Cette routine lit un nombre entier plus grand que zéro fourni
; par l'utilisateur. Si l'utilisateur fournit un nombre incorrect, il devra
; redonner une valeur valide.

GetWholeNum     proc    near
                lesi    InputLine       ;es:di pointe sur le tableau InputLine.
                gets
                call    Geti            ;Obtenir un index de ligne.
                jc      BadInt          ;Drapeau de retenue activé si erreur
                cmp     ax, 0           ;Il faut au moins une ligne ou une
                                        ;colonne !
                jle     BadInt
                ret
BadInt:         print
                char    Bell
                char "Illegal integer value, please re-enter",cr,lf,0
                jmp     GetWholeNum
GetWholeNum     endp

; Voici plusieurs routines à appeler pour effecteur le calcul proprement dit.
; Au début, AX contient la première opérande et dx la seconde.
; Ces routines retournent leur résultat dans AX, ou bien AX=8000h si une erreur
; se produit.
```

```
;
; Noter que la fonction CrossProduct appelle ces routines indirectement.

addition        proc    far
                add     ax, dx
                jno     AddDone         ;Vérifie un dépassement de capacité
                                        ;arithmétique signé.
                mov     ax, 8000h       ;Retourne 8000h en cas d'erreur.
AddDone:        ret
addition        endp

subtraction     proc    far
                sub     ax, dx
                jno     SubDone
                mov     ax, 8000h       ;Retourne 8000h si un dépassement de
                                        ;capacité se produit.
SubDone:        ret
subtraction     endp

multiplication proc     far
                imul    ax, dx
                jno     MulDone
                mov     ax, 8000h       ;Erreur si dépassement de capacité.
MulDone:        ret
multiplication  endp

division        proc    far
                push    cx              ;Préserve les valeurs des registres
                                        ; qu'on écrase.
                mov     cx, dx
                cwd
                test    cx, cx          ;Vérification de la division par zéro.
                je      BadDivide
                idiv    cx
                mov     dx, cx          ;Restaurer l'ancienne valeur du
                                        ;registre.
                pop     cx
                ret
BadDivide:      mov     ax, 8000h
                mov     dx, cx
                pop     cx
                ret
division        endp

; La fonction suivante calcule le reste d'une division si le symbole “DoMOD”
; est défini.

                ifdef   DoMOD

modulo          proc    far
                push    cx
                mov     cx, dx
                cwd
                test    cx, cx          ;Vérification de la division par zéro.
                je      BadDivide
                idiv    cx
                mov     ax, dx          ;Il faut placer le reste dans AX.
                mov     dx, cx          ;Restauration de la valeur originale
                                        ; des registres.
                pop     cx
                ret
BadMod:         mov     ax, 8000h
                mov     dx, cx
```

```
                pop     cx
                ret
modulo          endp
                endif

; Si on veut ajouter de nouvelles fonctionnalités au programme, on peut élargir
; le tableau "dope vector" ResultArrays. Ici, c'est un bon endroit pour définir
; la ou les fonctions pour ces nouvelles capacités.

; Le programme principal qui saisit des données de l'utilisateur, appelle les
; routines appropriées et puis affiche les résultats.

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit

; Invitation à fournir le nombre de lignes et de colonnes :

GetRows:        print
                byte    "Enter the number of rows for the matrix:",0
                call    GetWholeNum
                mov     Rows, ax

; OK, lire chacune des valeurs de ligne et de colonnes fournies par
; l'utilisateur :

                print
                char    "Enter values for the row (vertical) array",cr,lf,0

; Malloc alloue le nombre d'octets spécifiés dans le registre CX.
; AX contient le nombre d'éléments de tableau qu'on veut. On doit
; multiplier cette valeur par deux, car on veut un tableau de mots.
; Après l'appel à malloc, es:di pointe sur le tableau alloué dans le "tas"
; et l'enregistre dans la variable "RowArray".
;
; Notez l'utilisation du symbole "wp". Ceci est un equate textuel pour
; l'expression "word ptr" apparaissant dans le fichier d'inclusion "matrix.a".
; Notez aussi l'utilisation de l'expression d'adresse "RowArray+2" pour accéder
; à la portion segment du pointeur de 32 bits.

                mov     cx, ax
                shl     cx, 1
                malloc
                mov     wp RowArray, di
                mov     wp RowArray+2, es

; OK, appeler "GetArray" pour y lire "ax" valeurs d'entrée fournies par
; l'utilisateur. GetArray s'attend le nombre de valeurs à lire dans AX et
; au pointeur sur l'adresse de base du tableau dans es:di.

                print
                char    "Enter row data:",0
                mov     ax, Rows                ;Nombre de valeurs à lire.
                call    GetArray                ;ES:DI pointe encore sur ce
                                                ; tableau.

; OK, l'opération se répète pour lire les valeurs pour les colonnes.

GetCols:        print
                byte    "Enter the number of columns for the matrix:",0
                call    GetWholeNum     ;Obtenir cette valeur.
```

```
                mov     Columns, ax     ;Enregistrer cette valeur.

; Bien, lire chacune des valeurs pour les colonnes fournies par l'utilisateur :

                print
                char    "Enter values for the column (horz.) array",cr,lf,0

; malloc alloue le nombre d'octets spécifié dans le registre CX.
; AX contient le nombre des éléments du tableau qu'on veut. Il faut
; multiplier cette valeur par deux, car il s'agit d'un tableau de mots.
; Après malloc, es:di pointe sur le tableau alloué dans le tas. Enregistrement
; ce pointeur dans la variable "RowArray".

                mov     cx, ax          ;Convertir n colonnes en n octets
                shl     cx, 1           ; en multipliant par deux.
                malloc                  ;Obtention de la mémoire.
                mov     wp ColArray, di   ;Enregistrement du pointeur sur le
                mov     wp ColArray+2, es ;vecteur des colonnes.

; Bien, appeler "GetArray" pour lire "ax" valeurs d'entrées fournies par
; l'utilisateur. GetArray s'attend le nombre de valeurs à lire dans AX et
; un pointeur sur l'adresse de base du tableau dans es:di.

                print
                char    "Enter Column data:",0
                mov     ax, Columns             ;Nombre de valeurs à lire.
                call    GetArray                ;ES:DI pointe sur le tableau
                                                ; des colonnes.

; OK, initialiser les matrices qui garderont les résultats.
; Générer RASize copies du code suivant.
; La macro "repeat" répète les instructions entre les directives "repeat" et
; "endm" RASize fois. Notez l'utilisation du symbole Item pour générer
; automatiquement des index différents à chaque répétition du code suivant.
; La ligne "Item = Item+1" assure que l'élément prendra les valeurs
; 0, 1, 2, ..., RASize pour chaque opération dans la boucle.
;
; N'oubliez pas que la macro "repeat..endm" copie les instructions
; plusieurs fois dans le fichier source et elle n'exécute pas
; un "repeat..until" pendant l'exécution du programme. C'est à dire,
; la macro suivante équivaut à faire "RASize" copies du code, en
; substituant des valeurs différentes pour Item à chaque copie.
;
; Ce qui est bon à propos de ces lignes est qu'elles génèrent automatiquement
; la juste quantité de code d'initialisation, quel que soit le nombre
; d'éléments placé dans le tableau ResultArrays.

Item            =       0
                repeat  RASize
                mov     cx, Columns     ;Calculer la taille, en octets,
                imul    cx, Rows        ; de la matrice et allouer
                add     cx, cx          ; suffisamment d'espace pour
                malloc                  ; le tableau.
                mov     wp ResultArrays[Item * (sizeof DopeVec)].Data, di
                mov     wp ResultArrays[Item * (sizeof DopeVec)].Data+2, es
                mov     ax, Rows
                mov     ResultArrays[Item * (sizeof DopeVec)].Dim1, ax
                mov     ax, Columns
                mov     ResultArrays[Item * (sizeof DopeVec)].Dim2, ax
                mov     ResultArrays[Item * (sizeof DopeVec)].ESize, 2
Item            =       Item+1
                endm
```

```
; Bon, on a obtenu les valeurs d'entrée fournies par l'utilisateur, maintenant
; complétons les tables d'addition, de soustraction, de multiplication et
; de division. Encore une fois, une macro réduit le besoin de devoir taper
; du code répétitif et permet ici de traiter automatiquement divers éléments
; présents dans le tableau ResultArrays.

element         =       0
                repeat  RASize
                lfs     bp, RowArray ;Pointeur sur une ligne.
                lgs     bx, ColArray ;Pointeur sur une colonne.
                lea     cx, ResultArrays[element * (sizeof DopeVec)]
                call    CrossProduct
element         =       element+1
                endm

; Afficher ici les tableaux. Encore une fois, il est bon de rappeler que
; l'usage de la macro repeat..endm est très utile pour réduire la rédaction
; et automatiser le traitement de plusieurs éléments présents dans le tableau
; ResultArray.
.

Item            =       0
                repeat  RASize
                mov     si, offset ResultArrays[item * (sizeof DopeVec)]
                call    PrintMat
Item            =       1Item+1
                endm

; Techniquement, on n'aurait pas besoin de libérer l'espace alloué par
; malloc pour chacun des tableaux, car le programme est en train de
; terminer. Cependant, c'est une bonne idée de s'habituer à toujours
; libérer tout ce que vous allouez. Par exemple, si on ajoute du code
; par la suite dans le programme, on disposerait de cette mémoire
; libérée pour le nouveau code.

                les     di, ColArray
                free
                les     di, RowArray
                free

Item            =       0
                repeat  RASize
                les     di, ResultArrays[Item * (sizeof DopeVec)].Data
                free
Item            =       Item+1
                endm

Quit:           ExitPgm         ;La macro du DOS pour sortir du programme.
Main            endp

cseg            ends

sseg            segment         para stack 'stack'
stk             byte            1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment         para public 'zzzzzz'
LastBytes       byte            16 dup (?)
zzzzzzseg       ends
                end Main
```

### 8.22.4 GETI.ASM

GETI.ASM contient une routine (geti) permettant de lire une valeur entière fournie par l'utilisateur.

```
; GETI.ASM
;
; Ce module contient la routine réalisant l'invite de saisie des entiers pour
; le programme d'exemple du Chapitre Huit.

                .nolist
                include stdlib.a
                .list

                include matrix.a

InpSeg          segment         para public 'input'

; Geti : au début, es:di pointe sur la chaîne de caractères.
;       Cette routine saute tous les espaces principaux et les virgules et
;       puis elle teste le premier caractère (qui n'est pas une espace ni une
;       virgule), pour voir s'il s'agit d'un chiffre.
;       Si ce n'est pas le cas, alors elle active en retour le drapeau de
;       retenue pour indiquer qu'une erreur s'est produite. Si au contraire,
;       le premier caractère est un chiffre, alors elle appelle la routine
;       de la bibliothèque standard "atoi2" pour convertir la valeur en un
;       entier. Elle s'assure ensuite que le nombre termine par une espace,
;       une virgule ou un octet zéro de terminaison de chaîne.
;
; Valeurs de retour:
;       Le drapeau de retenue à zéro et la valeur dans AX si pas d'erreur.
;       Le drapeau de retenue à un, dans le cas contraire.
;
;       Cette routine fait pointer ES:DI sur le caractère courant quand on
;       convertit la chaîne en un entier. Si la conversion réussit, alors
;       ES:DI pointera sur une espace, une virgule ou un octet zéro de
;       terminaison.

geti            proc    far

                ifdef   debug
                print
                char    "Inside GETI",cr,lf,0
                endif

; Sauter tout espace ou virgule.
; Noter l'utilisation du symbole "byp" pour abréger la clause "byte ptr".
; BYP est une constante symbolique apparaissant dans le fichier macros.a.
; Une coercition de type "byte ptr" est requise ici, car MASM ne peut pas
; déduire la taille de l'opérande de mémoire (byte, word, dword, etc.),
; à partir des opérandes. Autrement dit, "es:[di]" et ' ' pourraient avoir
; n'importe laquelle de ces tailles.
;
; Noter aussi une petite astuce ici : en décrémentant di avant l'entrée
; dans la boucle et ensuite en l'incrémentant immédiatement après, on peut
; produire l'incrémentation de di avant de tester le caractère dans le corps
; de la boucle. Ceci rend la boucle un peu plus efficace et beaucoup plus
; élégante.

                dec     di
SkipSpcs:       inc     di
                cmp     byp es:[di], ' '
                je      SkipSpcs
                cmp     byp es:[di], ','
```

```
                je      SkipSpcs

; Vérifier si le premier caractère qui n'est pas un espace ni une virgule est
; un chiffre décimal :

                mov     al, es:[di]
                cmp     al, '-'         ;Le signe moins est également correct.
                jne     TryDigit
                mov     al, es:[di+1]   ;Obtenir le prochain caractère si  "-"

TryDigit:       isdigit
                jne BadGeti             ;Sauter si pas un chiffre.

; Bien, convertir le caractère qui suit en un entier :

ConvertNum:     atoi2                   ;Laisse l'entier dans AX
                jc      BadGeti         ;Se plaindre si conversion pas réussie.

; S'assurer que ce nombre se termine par un caractère raisonnable (espace,
; virgule, ou zéro de terminaison de chaîne) :

                cmp     byp es:[di], ' '
                je      GoodGeti
                cmp     byp es:[di], ','
                je      GoodGeti
                cmp     byp es:[di], 0
                je      GoodGeti

                ifdef   debug
                print
                char    "GETI: Failed because number did not end with "
                char    "a space, comma, or zero byte",cr,lf,0
                endif

BadGeti:        stc                     ;Retourner une erreur.
                ret

GoodGeti:       clc                     ;Retourner un entier dans AX.
                ret
geti            endp

InpSeg          ends
                end
```

### 8.22.5 GetArray.ASM

Ce module contient la routine d'entrée GetArray. Elle saisit les données obtenues en entrée pour le tableau produisant une table de calcul. Notez que GetArray lit les données pour un tableau unidimensionnel (ou pour une ligne d'un tableau multidimensionnel). Le programme contient la logique du calcul et lit deux sortes de vecteur : l'un pour le tableau des colonnes et l'autre pour le tableau des lignes. Notez : cette routine utilise des sous-routines de la bibliothèque standard qui apparaîtront dans le prochain chapitre.

```
; GETARRAY.ASM
;
; Ce module contient la routine d'entrée GetArray. Cette routine lit un
; ensemble de valeurs pour d'une colonne d'un tableau.

                .386
                option          segment:use16

                .nolist
                include stdlib.a
```

```
                .list

                include matrix.a

; Certaines variables locales pour ce module :

localdseg       segment         para public 'LclData'

                NumElements     word    ?
                ArrayPtr        dword   ?

Localdseg       ends

InpSeg          segment         para public 'input'
                assume ds:Localdseg

; GetArray :    Lit un ensemble de valeurs et les garde dans un tableau.
;
;               Au début :
;
;               es:di pointe sur l'adresse de base du tableau.
;               ax contient le nombre d'éléments du tableau.
;
;               Cette routine lit le nombre d'éléments de
;               tableau spécifiés en entrée et les stocke ensuite
;               dans le tableau. S'il se produit une erreur d'entrée quelconque,
;               alors cette routine répète l'invite pour l'utilisateur.

GetArray        proc    far
                pusha           ;Préserve tous les registres
                push    ds      ;que le code suivant modifiera.
                push    es
                push    fs

                ifdef   debug
                print
                char    "Inside GetArray, # of input values =",0
                puti
                putcr
                endif

                mov     cx, Localdseg           ;Faire pointer cs sur notre
                mov     ds, cx                  ; segment de données local.
                mov     wp ArrayPtr, di         ;Enregistrer en cas d'erreur
                mov     wp ArrayPtr+2, es       ; pendant la saisie.
                mov     NumElements, ax

; La boucle suivante lit une ligne de texte entrée par l'utilisateur, contenant
; un certain nombre de valeurs d'entrée. Cette boucle continue à se répéter
; si un caractère invalide est saisi.
;
; Note: LESI est une macro du fichier d'inclusion stdlib.a. Il charge ES:DI
; avec l'adresse de son opérande (contrairement à les di, InputLine qui
; chargent ES:DI avec la valeur dword qui se trouve à l'adresse InputLine)

RetryLp:        lesi    InputLine               ;Saisie de l'utilisateur.
                gets
                mov     cx, NumElements         ;Nombre de valeurs à lire.
                lfs     si, ArrayPtr            ;Stocker ici les entrées.

; Cette boucle interne lit "ax" entiers de la ligne d'entrée. S'il se produit
; une erreur, elle transfère le contrôle à RetryLp ci-dessus.
```

```
ReadEachItem:   call    geti                    ;Lire la prochaine valeur
                                                ;disponible.
                jc      BadGA
                mov     fs:[si], ax             ;Enregistrer dans le tableau.
                add     si, 2                   ;Prochain élément.
                loop    ReadEachItem            ;Répéter pour chaque élément.
                pop     fs                      ;Rétablir l'ancienne valeur des
                pop     es                      ; registres dans la pile avant
                pop     ds                      ; le retour.
                popa
                ret

; Si une erreur se produit, répéter la saisie de l'utilisateur pour toute la
; ligne :

BadGA:          print
                char    "Illegal integer value(s).",cr,lf
                char    "Re-enter data:",0
                jmp     RetryLp
                getArray endp
InpSeg          ends
                end
```

### 8.22.6 XProduct.ASM

Ce fichier contient le code effectuant les calculs.

```
; XProduct.ASM :
;
; Ce fichier contient le code effectuant les calculs.

                .386
                option  segment:use16

                .nolist
                include         stdlib.a
                includelib      stdlib.lib
                .list

                include         matrix.a

; Variables locales pour ce module

dseg            segment         para public 'data'
                DV              dword   ?
                RowNdx          integer ?
                ColNdx          integer ?
                RowCntr         integer ?
                ColCntr         integer ?
dseg            ends

cseg            segment         para public 'code'
                assume          ds:dseg

; CrossProduct : cette procédure calcule le produit de deux vecteurs.
;
;       Au début :
;
;       FS:BP - Pointe sur la matrice des lignes.
;       GS:BX - Pointe sur la matrice des colonnes.
;       DS:CX - Pointe sur le dope vecteur pour la destination.
;
```

```
;       Dans ce code, on présume que ds pointe sur le segment dseg.
;       Cette routine ne préserve que les registres de segment.

RowMat          textequ <fs:[bp]>
ColMat          textequ <gs:[bx]>
DVP             textequ <ds:[bx].DopeVec>

CrossProduct    proc    near

ifdef           debug
                print
                char    "Entering CrossProduct routine",cr,lf,0
                endif

                xchg    bx, cx          ;Obtenir le pointeur sur le dope vector.
                mov     ax, DVP.Dim1    ;Placer les valeurs Dim1 et Dim2 où
                mov     RowCntr, ax     ; elles sont faciles à accéder.
                mov     ax, DVP.Dim2
                mov     ColCntr, ax
                xchg    bx, cx

; Bon, ici on effectue les opérations en tant que telles. Ceci est défini comme
; suit :
;
;       for RowNdx := 0 to NumRows-1 do
;               for ColNdx := 0 to NumCols-1 do
;                       Result[RowNdx, ColNdx] = Row[RowNdx] op Col[ColNdx];

                mov     RowNdx, -1      ;En fait, commencer à zéro.
OutsideLp:      add     RowNdx, 1
                mov     ax, RowNdx
                cmp     ax, RowCntr
                jge     Done

                mov     ColNdx, -1      ;En fait, commencer à zéro.
InsideLp:       add     ColNdx, 1
                mov     ax, ColNdx
                cmp     ax, ColCntr
                jge     OutSideLp
                mov     di, RowNdx
                add     di, di
                mov     ax, RowMat[di]
                mov     di, ColNdx
                add     di, di
                mov     dx, ColMat[di]
                push    bx              ;Enregistrer le pointeur sur la matrice
                                        ; des colonnes.
                mov     bx, cx          ;Placer le pointeur sur le dope vector
                                        ; là-où on peut l'utiliser.

                call    DVP.Func        ;Calculer le résultat.

                mov     di, RowNdx      ;L'index du tableeau est
                imul    di, DVP.Dim2    ; (RowNdx*Dim2 + ColNdx) * ElementSize
                add     di, ColNdx
                imul    di, DVP.ESize

                les     bx, DVP.Data    ;Obtenir l'adresse de base du tableau.
                mov     es:[bx][di], ax ;Enregistrer le résultat.

                pop     bx              ;Rétablir le pointeur sur le tableau
                                        ; des colonnes.
                jmp     InsideLp
```

```
Done:           ret
CrossProduct    endp
cseg            ends
                end
```

## 8.23 Exercices de laboratoire

Dans cet ensemble d'exercices, vous assemblerez plusieurs petits programmes, vous produirez des listings d'assembleur et vous observerez le code objet que l'assembleur produira pour certaines séquences simples d'instructions. Vous étudierez aussi un fichier make afin d'observer comment il traite les dépendances.

### 8.23.1 Procédures *near* et *far*

Le court programme qui suit montre comment MASM génère automatiquement des appels near et far et les instructions ret, selon le champ de l'opérande de la directive proc (ce programme se trouve aussi dans le répertoire du chapitre 8 des exemples des programmes).

Assemblez ce programme en vous servant de l'option /Fl pour produire un listing en assembleur. Analysez les opcodes pour les instructions d'appel near et far, ainsi que pour l'instruction ret (voyez l'annexe D pour cela). Comparez ces valeurs avec les opcodes que ce programme émet. **Pour votre rapport de laboratoire :** décrivez comment MASM détermine quelles instructions doivent être near ou far. Incluez le listing assemblé dans votre rapport et identifiez quelles instructions sont des appels near ou far. Identifiez également les instructions de retour.

```
; EX8_1.asm (Exercice de laboratoire 8.1)

cseg            segment para public 'code'
                assume  cs:cseg, ds:dseg

Procedure1      proc    near

; MASM émettra un appel *far* à la procédure2 parce qu'il s'agit d'une
; procédure de type far.

                call    Procedure2

; Puisque cette instruction de retour se trouve à l'intérieur d'une procédure
; near, MASM émettra un retour de type near.

                ret
Procedure1      endp

Procedure2      proc    far

; MASM émettra un appel *near* à la procedure1, car il s'agit d'une
;procédure de type near.

                call    Procedure1

; Puisque cette instruction de retour se trouve à l'intérieur d'une procédure
; far, MASM émettra un retour de type far.

                ret
Procedure2      endp

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
```

```
; MASM émettra les instructions d'appel appropriées pour les procédures
; suivantes.

                call    Procedure1
                call    Procedure2

Quit:           mov     ah, 4ch
                int     21h
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             byte    1024 dup ("stack ")
sseg            ends
                end     Main
```

### 8.23.2 Exercices d'alignement des données

Dans cet exercice, vous assemblerez deux programmes différents à l'aide de l'option de ligne de commande "/Fl" de MASM, de sorte à pouvoir observer les adresses que MASM attribue aux variables. Le premier programme (Ex8_2.asm) utilise la directive even pour aligner les objets sur une limite de taille word. Le second programme (EX8_2b.asm) se sert de la directive align pour aligner des objets sur des limites de différentes tailles. **Pour votre rapport de laboratoire :** incluez le listing des programmes dans votre rapport en décrivant qu'est-ce que les directives even et align font et commentez de quelle manière ceci produit des programmes qui s'exécutent plus rapidement.

```
; EX8_2a.asm
;
; Cet exemple montre la directive EVEN.

dseg            segment

; Forcer un compteur d'emplacement impair à l'intérieur de ce segment :

                i       byte    0

; Ce mot commence à une adresse impaire, ce qui est mauvais !

                j       word    0

; Forcer le prochain mot à s'aligner sur une adresse paire, afin de pouvoir
; l'y accéder plus rapidement.

                even
                k       word    0

; Notez que la directive even n'a pas d'effet si le mot commençait déjà
; à une adresse paire.

                even
                l       word    0
dseg            ends

cseg            segment
                assume  ds:dseg
procedure       proc
                mov     ax, [bx]
                mov     i, al
                mov     bx, ax
```

```
; L'instruction suivante devrait commencer à  une adresse impaire, mais
; la directive EVEN insère un NOP, donc l'instruction commencera à une
; adresse paire.

                even
                mov     bx, cx

; Puisqu'on est déjà à une adresse paire, la directive EVEN suivante n'aura
; pas d'effect.

                even
                mov     dx, ax
                ret
procedure       endp
cseg            ends
                end



; EX8_2b.asm
;
; Cet exemple montre la directive align.

dseg            segment

; Forcer un compteur d'emplacement impair dans ce segment :

i               byte    0

; Ce mot tombe sur une adresse impaire, ce qui est mauvais !

j               word    0

; Forcer le prochain mot à s'aligner sur une adresse paire, afin de pouvoir
; l'y accéder plus rapidement.

                align   2
k               word    0

; Forcer encore une adresse impaire :

k_odd           byte    0

; Aligner la prochaine instruction sur un double-mot, à savoir sur une
; adresse qui est multiple de 4.

                align   4
l               dword   0

; Aligner la prochaine instruction sur une donnée de type quad word :

                align   8
RealVar         real8   3.14159

; Aligner ce qui suit sur un paragraphe :

                align   16
Table           dword   1,2,3,4,5
dseg            ends
                end
```

### 8.23.3 Exercice sur les constantes symboliques (*equates*)

Dans cet exercice, vous découvrirez une différence majeure entre une constante symbolique numérique et une constante symbolique textuelle (ex8_3.asm). MASM évalue l'opérande de la constante numérique lorsqu'il trouve cette dernière, alors que dans le cas de la constante textuelle, MASM ne l'évalue que lors de son développement (c'est à dire quand vous utilisez l'equate dans le programme). **Pour votre rapport de laboratoire :** assemblez le programme suivant à l'aide de l'option "Fl" et regardez le code objet émis pour les deux constantes symboliques. Expliquez pourquoi les opérandes des instructions sont différentes, même quand elles sont virtuellement identiques.

```
; Ex8_3.asm
;
; Comparaison entre les constantes symboliques numériques et les constantes
; symboliques textuelles, ainsi que la différence qu'elles produisent
; pendant l'assemblage.
;
cseg            segment

equ1            equ     $+2     ;Evalue "$" à cette instruction.
equ2            equ     <$+2>   ;Evalue "$" à son utilisation.

MyProc          proc
                mov     ax, 0
                lea     bx, equ1
                lea     bx, equ2
                lea     bx, equ1
                lea     bx, equ2
MyProc          endp

cseg            ends
                end
```

### 8.23.4 Exercice avec IFDEF

Dans cet exercice, vous assemblerez un programme utilisant l'assemblage conditionnel et vous observerez ses résultats. Le programme Ex8_4.asm utilise la directive ifdef pour tester la présence des symboles DEBUG1 et DEBUG2. DEBUG1 apparaît dans le programme alors que DEBUG2 n'y figure pas. **Pour votre rapport de laboratoire :** assemblez ce code en utilisant le paramètre de ligne de commande "/Fl". Incluez le listing dans votre rapport de laboratoire et expliquez les actions de la directive ifdef.

```
; Ex8_4.asm
;
; Démonstration de la directive IFDEF pour gérer le débogage. Ce code suppose
; qu'il y a deux niveaux de débogage, contrôlés par les deux symboles DEBUG1 et
; DEBUG2. Ici, DEBUG1 est défini alors que DEBUG2 ne l'est pas.

                .xlist
                include stdlib.a
                .list
                .nolistmacro
                .listif

DEBUG1          =       0

cseg            segment
DummyProc       proc
                ifdef   DEBUG2
                print
                byte    "In DummyProc"
                byte    cr,lf,0
                endif
```

```
                ret
DummyProc       endp

Main            proc
                ifdef   DEBUG1
                print
                byte    "Calling DummyProc"
                byte    cr,lf,0
                endif
                call    DummyProc
                ifdef   DEBUG1
                print
                byte    "Return from DummyProc"
                byte    cr,lf,0
                endif
                ret
Main            endp
cseg            ends
                end
```

### 8.23.5 Exercice sur un fichier m*ake file*

Dans cet exercice, vous ferez une expérience avec un exemple de *make file* et vous verrez comment mmake.exe choisit les fichiers qu'il faut réassembler. Vous utiliserez les programmes Ex8_5a.asm, Ex8_5b.asm, Ex8_5.a et Ex8_5.mak que vous pouvez trouver dans le répertoire du chapitre 8. Copiez ces fichiers dans un répertoire de votre disque dur. Ces fichiers contiennent un programme qui lit une chaîne textuelle de l'utilisateur et affiche toutes les voyelles de cette chaîne. Vous apporterez des changements mineurs dans les fichiers se terminant par .asm et .a et exécuterez le fichier make pour observer les résultats.

La première chose qu'il faudra faire, sera assembler le programme et créer une version mise à jour des fichiers .exe et .obj. Vous pourrez le faire à l'aide de la commande DOS suivante :

```
                nmake   Ex8_5.mak
```

En supposant que les fichiers .obj et .exe n'étaient pas déjà présents dans le répertoire courant, la commande nmake qu'on vient de voir assemble et fait l'édition des liens des fichiers Ex8_5a.asm et Ex8_5b.asm, en produisant ainsi l'exécutable Ex8.exe.

En utilisant l'éditeur, apportez un changement mineur (par exemple, insérer un espace dans une ligne commentaire), au ficher Ex8a.asm. Exécutez la commande nmake ci-dessus et rapportez dans votre rapport de laboratoire ce que le fichier make fait.

Ensuite modifiez le fichier Ex8_5b.asm, exécutez l'instruction ci-dessus et faites le même rapport en expliquant les résultats.

Finalement, changez encore légèrement le fichier Ex8_5.a. Exécutez la commande nmake et décrivez encore les résultats dans votre rapport de laboratoire.

**Pour votre rapport :** expliquez comment les changements de chacun des fichiers décrits ci-dessus affectent l'opération make en indiquant pourquoi nmake fait ce qu'il fait. **Pour aller plus loin :** essayez d'effacer (un ficher à la fois), les fichiers Ex8_5a.obj, Ex8_5b.obj et Ex8_5.exe et d'exécuter ensuite la commande nmake. Expliquez pourquoi nmake fait ce qu'il fait quand vous effacez individuellement chacun de ces fichiers.

Voici le make file Ex8_5.mak :

```
ex8_5.exe: ex8_5a.obj ex8_5b.obj
        ml /Feex8_5.exe ex8_5a.obj ex8_5b.obj

ex8_5a.obj: ex8_5a.asm ex8_5.a
        ml /c ex8_5a.asm

ex8_5b.obj: ex8_5b.asm ex8_5.a
        ml /c ex8_5b.asm
```

Le fichier d'en-tête Ex8_5a :

```
; Fichier d'en-tête pour le projet Ex8_5.
; Ce fichier inclut la directive EXTERNDEF qui rend le nom PrintVowels public
; et externe. Il comprend aussi la macro PrtVowels qui nous permet d'appeler la
; routine PrintVowels à la manière des routines de la bibliothèque standard
; UCR.

                externdef       PrintVowels:near

PrtVowels       macro
                call PrintVowels
                endm
```

Le fichier source Ex8_5a.asm :

```
; Ex8_5a.asm
;
; Randall Hyde
; 2/7/96
;
; Ce programme lit une chaîne de symboles de l'utilisateur et en affiche les
; voyelles. Il montre l'utilisation des fichiers make.

                .xlist
                include         stdlib.a
                includelib      stdlib.lib
                .list

; Le fichier d'inclusion suivant importe les définitions externes des routines
; du module Lab6x10b. Particulièrement, il donne à ce module accès à la routine
; "PrtVowels" située dans le fichier Ex8_5b.asm[27].

                include         Ex8_5.a

cseg            segment para public 'code'

Main            proc

                meminit

; Lit une chaîne de l'utilisateur, affiche les voyelles trouvées dans la chaîne
; et puis libère la mémoire allouée par la routine GETSM :

                print
                byte    "I will find all your vowels"
                byte    cr,lf
                byte    "Enter a line of text: ",0

                getsm
                print
                byte    "Vowels on input line: ",0
                PrtVowels
                putcr
                free

Quit:           ExitPgm
Main            endp

cseg            ends

sseg            segment para stack 'stack'
```

[27] Dans l'original, il était écri Lab8_5b.asm, mais ce fichier n'existe pas, n.d.t.

```
stk             byte    1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte    16 dup (?)
zzzzzzseg       ends
                end     Main
```

## 8.24 Projets de programmation

1) Écrivez un programme qui accepte en entrée matrices d'entiers 4x4 et calcule leur produit matriciel. L'algorithme de la multiplication des matrices (c'est-à-dire C:= A * B) est le suivant :

```
for i := 0 to 3 do
        for j := 0 to 3 do begin
                c[i,j] := 0;
                for k := 0 to 3 do
                        c[i,j] := c[i,j] + a[i,k] * b[k,j];
        end;
```

Vous pouvez vous servir des macros ForLp et Next du Chapitre Six.

2) Modifiez le programme d'exemple du paragraphe 8.23 afin d'utiliser les boucles FORLP et NEXT vues dans ce chapitre. Remplacez toutes les occurrences des boucles avec les macros correspondantes.

3) Écrivez un programme invitant l'utilisateur à entrer trois valeurs entières m, p et n. Le programme alloue ensuite de l'espace pour trois tableaux A[0…m-1, 0…p-1], B[0…p-1, 0…n-1] et C[0…m-1, 0…, n-1]. Ensuite, saisir des valeurs pour ces tableaux. Et, finalement, le programme doit calculer le produit matriciel des tableaux A et B, selon l'algorithme suivant :

```
for i := 0 to m-1 do
        for j := 0 to n-1 do begin
                c[i,j] := 0;
                for k := 0 to p-1 do
                        c[i,j] := c[i,j] + a[i,k] * b[k,j];
        end;
```

À la fin, le programme doit afficher les tableaux A, B et C. Utilisez les macros ForLp et Next. Il vous faudra encore jeter un coup d'œil sur le programme du paragraphe 8.23, afin d'observer la manière d'allouer dynamiquement des tableaux et d'y accéder, car leurs dimensions ne sont pas connues avant l'exécution.

4) Les macros ForLp et Next données dans ce chapitre n'incrémentent que leur variable de contrôle de boucle d'une unité après chaque itération. Écrivez une nouvelle macro, ForTo, vous permettant de spécifier une constante d'*incrémentation*, de sorte à faire incrémenter la variable de contrôle selon la valeur de cette constante. Écrivez ensuite un programme se servant de cette macro. Astuce : il vous faudra créer une étiquette globale pour passer l'information d'incrémentation à la macro Next, où vous effectuerez l'opération d'incrémentation à l'intérieur de la macro ForLp.

5) Écrivez une troisième version des macros ForLp et Next vous permettant de spécifier des incrémentations *négatives* (quelque chose de semblable à l'instruction for…downto de Pascal). Appelez cette macro ForDT (for…downto).

## 8.25 Résumé

Ce chapitre a présenté plusieurs directives et pseudo-opcodes de l'assembleur supportés par MASM et a fourni une description complète de tout ce que MASM peut offrir. Donc suffisamment d'information pour vous permettre de démarrer.

Le format des instructions d'assembleur est généralement assez libre et on trouve normalement dans un fichier source une instruction par ligne. Bien que MASM accepte n'importe quelle disposition des instructions (par

exemple, vous pourriez écrire trois instructions par ligne), il faut tout de même structurer soigneusement vos fichiers sources, afin de les rendre plus faciles à lire.

• Voir "Les Instructions de l'assembleur" (8.1)

MASM garde trace de l'offset d'une instruction ou d'une variable dans un segment, à l'aide du *compteur d'emplacement*, qui est incrémenté d'une unité chaque fois qu'un octet est ajouté dans le fichier de sortie.

• Voir "Le compteur d'emplacement" (8.2)

Comme les langages de haut niveau, MASM vous permet d'utiliser des noms symboliques pour les variables et les étiquettes d'instructions. Avoir affaire à des symboles est plus pratique qu'utiliser des offsets numériques dans un programme en assembleur. Les symboles de MASM ressemblent beaucoup à leurs équivalents LHN avec quelques extensions.

• Voir "symboles" (8.3)

MASM fournit divers types de constantes littérales, en incluant les constantes entières binaires, décimales et hexadécimales, ainsi que les constantes chaîne et texte.

• Voir "Constantes littérales" (8.4)
• Voir "Constantes entières" (8.4.1)
• Voir "Constantes chaînes" (8.4.2)
• Voir "Constantes textuelles" (8.4.4)

Pour vous aider à manipuler des segments dans vos programmes, MASM fournit les directives segment/ends, avec lesquelles on peut contrôler l'ordre et l'alignement du chargement des modules en mémoire.

• Voir "Segments" (8.8)
• Voir "Les noms des segments" (8.8.1)
• Voir "Ordre de chargement des segments" (8.8.2)
• Voir "Opérandes de segment" (8.8.3)
• Voir "Le type CLASS" (8.8.4)
• Voir "Définitions typiques des segments" (8.8.7)
• Voir "Pourquoi contrôler l'ordre de chargement" (8.8.8)

MASM fournit les directives proc/endp pour déclarer des procédures dans vos programmes. Bien qu'elles ne soient pas strictement nécessaire, ces directives rendent vos programmes beaucoup plus faciles à lire et à maintenir. Elles vous permettent aussi d'utiliser des noms d'instructions locaux dans vos procédures.

• Voir "Procédures" (8.7)

Les constantes symboliques vous permettent de définir des constantes de toutes sortes dans vos programmes. MASM met en place trois directives pour définir de telles constantes : la directive "=", la directive "equ" et la directive "textequ". Tout comme dans des langages de haut niveau, une utilisation judicieuse peut vous aider à rendre vos programmes plus lisibles.

• Voir "Déclarer des constantes manifestes à l'aide des constantes symboliques" (8.5)

Comme vous avez vu au chapitre 4, MASM vous donne la possibilité de déclarer des variables dans le segment de données en utilisant les directives byte, word, dword, etc. Entre autres, MASM est un assembleur fortement typé et attache aussi bien un type qu'un emplacement à un nom de variable (beaucoup d'assembleurs n'attribuent qu'un emplacement). Ceci aide MASM à localiser des bogues obscures dans vos programmes.

• Voir "Variables" (8.10)
• Voir "Types d'identificateurs" (8.11)
• Voir "Comment attribuer un type particulier à un identificateur" (8.11.1)
• Voir "Valeurs des identificateurs" (8.11.2)
• Voir "Conflits de type" (8.11.3)

MASM supporte des *expressions d'adresse* vous permettant d'utiliser des opérateurs arithmétiques pour mettre en place des adresses constantes pendant l'assemblage. Il vous permet aussi de modifier le type d'une valeur d'adresse (*overriding*) et d'extraire diverses sources d'informations d'un symbole. Ce qui est très utile pour écrire des programmes faciles à maintenir.

- Voir "Expressions d'adresse" (8.12)
- Voir "Types de symboles et modes d'adressage" (8.12.1)
- Voir "Opérateurs logiques et arithmétiques" (8.12.2)
- Voir "Coercition" (8.11.3)
- Voir "Opérateurs de type" (8.12.4)
- Voir "Précédence des opérateurs" (8.12.5)

MASM fournit différents outils pour indiquer à l'assembleur quel segment associer à un registre de segment. Il fournit également la possibilité de modifier un choix par défaut. Tout ceci permet de gérer divers segments à la fois dans un même programme avec un minimum de tracas.

- Voir "Préfixes de segment" (8.8.9)
- Voir "Contrôler les segments à l'aide de la directive ASSUME" (8.8.10)

MASM supporte également l' "assemblage conditionnel" vous permettant de choisir quels segments de code doivent être réellement assemblés. Ce qui est utile pour insérer du code de débogage dans vos programmes (supprimable avec une seule instruction) et pour écrire des programmes pouvant s'exécuter sous différents environnements (en insérant ou en supprimant des sections de code distinctes).

- Voir "Assemblage conditionnel" (8.13)
- Voir "Directive IF" (8.13.1)
- Voir "Directive IFE" (8.13.2)
- Voir "IFDEF et IFNDEF" (8.13.3)
- Voir "IFB, IFNB" (8.13.4)
- Voir "IFIDN, IFDIF, IFIDNI et IFDIFI" (8.13.5)

MASM, comme son nom l'indique, fournit également la capacité puissante d'écrire des macros. Il s'agit de sections de code que vous pouvez produire en plaçant simplement le nom de la macro dans votre code. Les macros, si convenablement utilisées, peuvent aider à écrire des programmes plus courts, plus faciles à lire et plus robustes. Hélas, maladroitement utilisées, les macros produisent exactement l'inverse.

- Voir "Macros" (8.14)
- Voir "Macros procédurales " (8.14.1)
- Voir "La directive LOCAL " (8.14.3)
- Voir "La directive EXITM " (8.14.4)
- Voir "Macros : le pour et le contre " (8.14.10)
- Voir "Opérations répétitives" (8.15)

MASM fournit également diverses directives pour produire des "listings assemblés" ou des impressions de vos programmes, avec beaucoup d'informations (utiles !) générées par l'assembleur. Ces directives vous permettent d'activer/désactiver l'opération de listing, d'afficher des informations à l'écran pendant l'assemblage et de placer des titres sur la sortie.

- Voir "Contrôle du listing" (8.19)
- Voir "Les directives ECHO et %OUT" (8.19.1)
- Voir "La directive TITLE" (8.19.2)
- Voir "La directive SUBTTL" (8.19.3)
- Voir "La directive PAGE " (8.19.4)
- Voir "Les directives .LIST, .NOLIST et XLIST " (8.19.7)
- Voir "Autres directives de listing " (8.19.6)

Pour gérer de gros projets, il faut la "compilation séparée" (ou l' "assemblage séparé" dans le cas de l'assembleur). MASM fournit diverses directives vous permettant de fusionner des fichiers source durant l'assemblage, d'assembler séparément des modules et communiquer des noms des variables et des procédures entre les modules.

- Voir "Gérer de longs programmes" (8.20)
- Voir "La directive INCLUDE" (8.20.1)
- Voir "Les directives PUBLIC, EXTERN et EXTRN" (8.20.2)
- Voir "La directive EXTERNDEF" (8.20.3)

## 8.26 Questions

1. Quelle est la différence entre les séquences d'instructions suivantes ?

```
MOV     AX, VAR+1
```

et

```
MOV     AX, VAR
INC     AX
```

2. Quel est le format de ligne d'une instruction en assembleur ?

3. Quelle est la fonction de la directive ASSUME ?

4. Qu'est-ce que c'est le compteur d'emplacement ?

5. Lesquels des symboles suivants sont valides ?

| | |
|---|---|
| a) CeciEstUnSymbole | b) Ceci_Est_Un_Symbole |
| c) Ceci.Est.Un.Symbole | d) .Ceci_Est_Il_Un_Symbole? |
| e) _____ | f) @_$?_Pour_Vous |
| g) 1MoyenDYAller | h) %BonjourATous |
| i) F000h | j) ?A_0$1 |
| k) $1234 | l) Bonjour Mecs |

6. Comment spécifie-t-on l'ordre de chargement des segments ?

7. Quel est le type des symboles déclarés dans les instructions qui suivent ?

```
a) symbol1      equ     0
b) symbol2:
c) symbol3      proc
d) symbol4      db      ?
e) symbol5      dw      ?
f) symbol6      proc    far
g) symbol7      equ     this word
h) symbol8      equ     byte ptr symbol7
i) symbol9      dd      ?
j) symbol10     macro
k) symbol11     segment para public 'data'
l) symbol12     equ     this near
m) symbol13     equ     'ABCD'
n) symbol14     equ     <MOV AX, 0>
```

8. Lesquels des symboles de la question 7 ne se voient pas affecter un compteur d'emplacement ?

9. Expliquez la fonction des opérateurs suivants :

| | | | | |
|---|---|---|---|---|
| a) PTR | b) SHORT | c) THIS | d) HIGH | e) LOW |
| f) SEG | g) OFFSET | | | |

10. Quelle est la différence entre les valeurs chargées dans le registre BX (s'il y en a) dans la séquence de code suivante ?

```
mov     bx, offset Table
lea     bx, Table
```

11. Quelle est la différence entre la macro REPEAT et l'opérateur DUP ?

12. Dans quel ordre les segment suivants seront-ils chargés en mémoire ?

```
CSEG    segment para public 'CODE'
        …
CSEG    ends
DSEG    segment para public 'DATA'
        …
DSEG    ends
ESEG    segment para public 'CODE'
```

```
        …
ESEG    ends
```

13. Laquelle des expressions d'adresse suivantes ne produit pas le même résultat que les autres ?

a) Var1[3][5] b) 15[Var1]c) Var1[8]
d) Var1+ 2[6] e) Var1*3*5 f) Var1+3+5