

Une chaîne est une collection d'objets stockés dans des emplacements de mémoire contigus. Les chaînes sont habituellement des tableaux de bytes, de mots, ou (sur 80386 et processeurs postérieurs) de doubles mots. La famille du microprocesseur 80x86 supporte plusieurs instructions spécifiquement conçues pour manipuler des chaînes. Ce chapitre explore certaines des utilisations de ces instructions de chaîne.

Les 8088, les 8086, les 80186, et les 80286 peuvent traiter deux types de chaînes : chaînes de bytes et chaînes de mots. Les 80386 et les processeurs postérieurs manipulent également les chaînes de double mots. Ils peuvent déplacer des chaînes, comparer des chaînes, rechercher une valeur spécifique dans une chaîne, initialiser une chaîne à une valeur fixe, et font d'autres opérations primitives sur des chaînes. Les instructions de chaîne du 80x86 sont également utiles pour manipuler des tableaux, des tables, et des enregistrements. Vous pouvez facilement assigner ou comparer de telles structures de données en utilisant les instructions de chaîne. Employer des instructions de chaîne peut accélérer considérablement votre code de manipulation de tableau.

---

### 15.0 Vue d'Ensemble du Chapitre

Ce chapitre passe en revue le mécanisme des instructions de chaînes du 80x86. Ensuite, il montre comment traiter des chaînes de caractères en utilisant ces instructions. En conclusion, il conclut en discutant les instructions de chaînes disponibles dans la Bibliothèque Standard de l'UCR. Les sections ci-dessous qui ont le préfixe "●" sont essentielles. Les sections avec un "□" discutent de sujets avancés que vous pouvez reporter pendant un moment.

- Les instructions de chaîne du 80x86
- Chaînes de caractères
- Fonctions de chaîne de caractères
- Fonctions de chaîne de la Bibliothèque Standard de l'UCR.
- Utilisation des instructions de chaîne avec d'autres types de données.

---

### 15.1 Les Instructions de Chaînes du 80x86

Tous les membres de la famille 80x86 supportent cinq instructions de chaînes différentes : **movs**, **cmps**, **scas**, **lods**, et **stos**<sup>1</sup>. Ce sont les opérations de chaînes primitives puisque vous pouvez établir la plupart des autres opérations de chaîne à partir de ces cinq instructions. Comment employer ces cinq instructions est le sujet des prochaines sections.

---

#### 15.1.1 Comment Fonctionnent les Instructions de Chaînes

Les instructions de chaîne opèrent sur des blocs (tableaux linéaires contigus) de mémoire. Par exemple, l'instruction **movs** déplace une séquence de bytes d'un emplacement de mémoire à un autre. L'instruction **cmps** compare deux blocs de mémoire. L'instruction **scas** balaye un bloc de mémoire à la recherche d'une valeur particulière. Ces instructions de chaîne exigent souvent trois opérandes, une adresse de bloc de destination, une adresse de bloc de source, et (en option) un compte d'éléments. Par exemple, en utilisant l'instruction **movs** pour copier une chaîne, vous avez besoin d'une adresse de source, d'une adresse de destination, et d'un compte (le nombre d'éléments de chaîne à déplacer).

A la différence d'autres instructions qui opèrent sur la mémoire, les instructions de chaîne sont des instructions de byte unique qui n'ont pas d'opérande explicite. Les opérandes pour les instructions de chaîne incluent

- le registre **si** (index de source)
- le registre **di** (index de destination)
- le registre **cx** (compteur)
- le registre **ax**, et
- le drapeau de direction dans le registre **FLAGS**.

<sup>1</sup>Le 80186 et les processeurs postérieurs supportent deux instructions de chaînes supplémentaires, **INS** et **OUTS** qui entrent des chaînes de données depuis un port d'entrée ou sortent des chaînes de données sur un port de sortie. Nous ne prendrons pas ces instructions en compte dans ce chapitre.

Par exemple, une variante de l'instruction **movs** (déplacer une chaîne) copie une chaîne de l'adresse de source indiquée par **ds:si** à l'adresse de destination indiquée par **es:di**, de longueur **cx**. De même, l'instruction **cmps** compare la chaîne pointée par **ds:si**, de longueur **cx**, à la chaîne pointée par **es:di**.

Toutes les instructions n'ont pas des opérandes de source et de destination (seulement **movs** et **cmps** les supportent). Par exemple, l'instruction **scas** (balayer une chaîne) compare la valeur dans l'accumulateur aux valeurs dans la mémoire. Malgré leurs différences, les instructions de chaînes du 80x86 ont toutes une chose en commun – leur utilisation exige que vous manipuliez deux segments, le segment de données et le segment supplémentaire.

### 15.1.2 Les Préfixes REP/REPE/REPZ et REPNE/REPZ

Les instructions de chaînes, par elles-mêmes, n'opèrent pas sur des chaînes des données. L'instruction **movs**, par exemple, déplacera un byte, un mot ou un double mot unique. Exécutée toute seule, l'instruction **movs** ignore la valeur dans le registre **cx**. Les préfixes de répétition indiquent au 80x86 faire une opération de chaîne multi-byte. La syntaxe pour le préfixe de répétition est :

Champ:

Étiquette      repeat      mnémonique   opérande      ; commentaire

Pour **MOVS**:

rep              movs              {opérandes}

Pour **CMPS**:

repe              cmps              {opérandes}  
repz              cmps              {opérandes}  
repne              cmps              {opérandes}  
repnz              cmps              {opérandes}

Pour **SCAS**:

repe              scas              {opérandes}  
repz              scas              {opérandes}  
repne              scas              {opérandes}  
repnz              scas              {opérandes}

Pour **STOS**:

rep              stos              {opérandes}

Normalement, vous n'employez pas les préfixes de répétition avec l'instruction **lods**.

Comme vous pouvez le voir, la présence des préfixes de répétition introduit un nouveau champ dans la ligne source - le champ du préfixe de répétition. Ce champ apparaît seulement sur des lignes source contenant des instructions de chaînes. Dans votre fichier source :

- le champ étiquette devrait toujours commencer à la colonne une
- le champ répétition devrait commencer au premier arrêt de tabulatrice, et
- le champ mnémonique devrait commencer au deuxième arrêt de tabulatrice.

En spécifiant le préfixe de répétition avant une instruction de chaîne, l'instruction de chaîne se répète **cx** fois<sup>2</sup>. Sans préfixe de répétition, l'instruction ne fonctionne que sur un byte, un mot, ou un double mot unique.

Vous pouvez employer des préfixes de répétition pour traiter des chaînes entières avec une instruction unique. Vous pouvez employer les instructions de chaîne, sans préfixe de répétition, comme opérations de chaînes primitives pour synthétiser des opérations de chaînes plus puissantes.

Le champ opérande est facultatif. Si présent, MASM l'utilise simplement pour déterminer la taille de la chaîne sur laquelle opérer. Si le champ opérande est le nom d'une variable byte, l'instruction de chaîne opère sur des bytes. Si

<sup>2</sup>Excepté l'instruction **cmps** qui se répète au plus le nombre de fois ont indiqué dans le registre **cx**.

l'opérande est une adresse de mot, l'instruction opère sur des mots. De même pour des double mots. Si le champ opérande n'est pas présent, vous devez ajouter un "B", "W", ou "D" à l'instruction de chaîne pour spécifier la taille, par exemple, **movsb**, **movsw** ou **movsd**.

---

### 15.1.3 Le Drapeau de Direction

En plus des registre **si**, **di**, **cx** et **ax**, un autre registre commande les instructions de chaîne du 80x86 - le registre flags. Spécifiquement, le *drapeau de direction* dans le registre flags contrôle comment la CPU traite les chaînes.

Si le drapeau de direction est à zéro, la CPU incrémente **si** et **di** après avoir agi sur chaque élément de la chaîne. Par exemple, si le drapeau de direction est à zéro, alors l'exécution de **movs** déplacera le byte, le mot, ou le double mot à **ds:si** vers **es:di** et incrémentera **si** et **di** de un, deux, ou quatre. En spécifiant le préfixe de **rep** avant cette instruction, le CPU incrémente **si** et **di** pour chaque élément dans la chaîne. A terme, les registres **si** et **di** pointeront sur le premier item au delà de la chaîne.

Si le drapeau de direction est à un, alors le 80x86 décrémentera **si** et **di** après traitement de chaque élément de la chaîne. Après une opération de chaîne répétée, les registres **si** et **di** pointeront sur le premier byte ou mot avant les chaînes si le drapeau de direction était à un.

Le drapeau de direction peut être mis à un ou à zéro en utilisant les instructions **cld** (clear direction flag) et **std** (set direction flag). En utilisant ces instructions à l'intérieur d'une procédure maintenez à l'esprit qu'elles modifient l'état de la machine. Par conséquent, vous pouvez avoir besoin de sauver le drapeau de direction pendant l'exécution de cette procédure. L'exemple suivant montre les genres de problèmes que vous pourriez rencontrer :

StringStuff:

```
                cld
                <faire quelques opérations>
                call Str2
                <faire quelques opérations de chaînes nécessitant D=0>
                .
                .
                .
Str2            proc near
                std
                <faire quelques opérations de chaînes>
                ret
Str2            endp
```

Ce code ne fonctionnera pas correctement. Le code appelant suppose que le drapeau de direction est à zéro après que **Str2** retourne. Cependant, ce n'est pas vrai. Par conséquent, les opérations de chaîne exécutées après l'appel à **Str2** ne fonctionneront pas correctement.

Il y a deux façons de résoudre ce problème. La première, et probablement la plus évidente, est de toujours insérer des instructions **cld** ou **std** juste avant exécuter une instruction de chaîne. L'autre alternative est de sauver et reconstituer le drapeau de direction en utilisant les instructions **pushf** et **popf**. En utilisant ces deux techniques, le code ci-dessus ressemblerait à ceci :

Toujours définir **cld** ou **std** avant une instruction de chaîne :

StringStuff:

```
                cld
                <faire quelques opérations>
                call Str2
                cld
                <faire quelques opérations de chaînes nécessitant D=0>
                .
                .
                .
```

```

Str2          proc near
               std
               <faire quelques opérations de chaînes>
               ret
Str2          endp
    Sauver et restaurer le registre flags:
StringStuff:
               cld
               <faire quelques opérations>
               call Str2
               <faire quelques opérations de chaînes nécessitant D=0>
               .
               .
               .
Str2          proc near
               pushf
               std
               <faire quelques opérations de chaînes>
               popf
               ret
Str2          endp

```

Si vous employez les instructions **pushf** et **popf** pour sauver et restaurer le registre flags, maintenez à l'esprit que vous sauvez et restaurez de tous les drapeaux. Par conséquent, de telles routines ne peuvent renvoyer aucune information dans les drapeaux. Par exemple, vous ne pourrez pas renvoyer une condition d'erreur dans le drapeau de retenue si vous employez **pushf** et **popf**.

---

### 15.1.4 L'Instruction MOVS

L'instruction **movs** prend quatre formes de base. **Movs** déplace des bytes, mots ou doubles mots, **movsb** déplace des chaînes de bytes, **movsw** déplace des chaînes de mots et **movsd** déplace des chaînes de double mots (sur les processeurs 80386 et postérieurs). Ces quatre instructions emploient la syntaxe suivante :

```

{REP}         MOVSB
{REP}         MOVSW
{REP}         MOVSD      ;Disponible seulement sur 80386 et plus
{REP}         MOVS Dest, Source

```

L'instruction **movsb** (move string, bytes) prend le byte à l'adresse **ds:si**, le stocke à l'adresse **es:di**, et puis incrémente ou décrémente les registres **si** et **di** de un. Si le préfixe **rep** est présent, la CPU vérifie **cx** pour voir s'il contient zéro. Si ce n'est pas le cas, alors elle déplace le byte de **ds:si** à **es:di** et décrémente le registre **cx**. Ce processus se répète jusqu'à ce que **cx** arrive à zéro.

L'instruction **movsw** (move string, words) prend le mot à l'adresse **ds:si**, le stocke à l'adresse **es:di**, et puis incrémente ou décrémente les registres **si** et **di** de deux. S'il y a un préfixe **rep**, la CPU répète cette procédure autant de fois que spécifié dans **cx**.

L'instruction **movsd** fonctionne d'une manière semblable sur les double mots. Incrémentant ou décrémentant **si** et **di** par quatre pour chaque transfert de données.

MASM calcule automatiquement la taille de l'instruction **movs** en regardant la taille des opérandes indiqués. Si vous avez défini les deux opérandes avec une directive **byte** (ou comparable), alors MASM émettra une instruction **movsb**. Si vous avez déclaré les deux étiquettes par l'intermédiaire de **word** (ou comparable), MASM produira d'une instruction **movsw**. Si vous avez déclaré les deux étiquettes avec **dword**, MASM émet une instruction **movsd**. L'assembleur vérifiera également que les segments des deux opérandes correspondent aux assertions courantes

(via la directive **assume**) en ce qui concerne les registres **es** et **ds**. Vous devriez toujours employer les formes **movsb**, **movsw** et **movsd** et oublier la forme **movs**.

Bien que, en théorie, la forme **movs** semble être une manière élégante de gérer l'instruction de déplacement de chaîne, dans la pratique il crée plus d'ennuis que ça n'en vaut la peine. En outre, cette forme de l'instruction de déplacement de chaîne implique que **movs** ait des opérandes explicites, alors que, en fait, les registres **si** et **di** indiquent implicitement les opérandes. Pour cette raison, nous emploierons toujours les instructions **movsb**, **movsw** ou **movsd**. Utilisée avec le préfixe **rep**, l'instruction **movsb** déplacera le nombre de bytes indiqués dans le registre **cx**. Le segment de code suivant copie 384 bytes de **String1** à **String2** :

```
        cld
        lea  si, String1
        lea  di, String2
        mov  cx, 384
rep     movsb
        .
        .
        .
String1  byte 384 dup (?)
String2  byte 384 dup (?)
```

Ce code, bien sûr, suppose que **String1** et **String2** sont dans le même segment et que les registres **ds** et **es** pointent sur ce segment. Si vous substituez **movsw** à **movsb**, alors le code ci-dessus déplacera 384 mots (768 bytes) au lieu de 384 bytes :

```
        cld
        lea  si, String1
        lea  di, String2
        mov  cx, 384
rep     movsw
        .
        .
        .
String1  word 384 dup (?)
String2  word 384 dup (?)
```

Rappelez-vous, le registre **cx** contient le compte d'éléments, pas le compte d'octets. En utilisant l'instruction **movsw**, la CPU déplace le nombre de mots indiqués dans le registre **cx**.

Si vous avez mis à un le drapeau de direction avant d'exécuter une instruction **movsb/movsw/movsd**, la CPU décrémente les registres **si** et **di** après avoir déplacé chaque élément de la chaîne. Ceci signifie que les registres **si** et **di** doivent pointer sur la fin de leurs chaînes respectives avant qu'on émette une instruction **movsb**, **movsw** ou **movsd**. Par exemple,

```
        std
        lea  si, String1+383
        lea  di, String2+383
        mov  cx, 384
rep     movsb
        .
        .
        .
String1  byte 384 dup (?)
String2  byte 384 dup (?)
```

Bien qu'il y ait des moments où le traitement d'une chaîne de la fin au début est utile (voiez la description de **cmps** dans la prochaine section), généralement vous traiterez des chaînes dans la direction avant puisqu'elle est plus naturelle. Il y a une classe d'opérations de chaîne où pouvoir traiter des chaînes dans les deux directions est

absolument obligatoire : traitement des chaînes quand les blocs de source et de destination se chevauchent. Considérez ce qui se produit dans le code suivant :

```
        cld
        lea  si, String1
        lea  di, String2
        mov  cx, 384
rep     movsb
        .
        .
        .
String1  byte ?
String2  byte 384 dup (?)
```

### Figure 15.1 Réécriture des Données Pendant une Opération de Déplacement en Bloc

Cette séquence d'instructions traite **String1** et **String2** comme une paire de chaînes de 384 bytes. Cependant, les 383 derniers bytes du tableau **String1** recouvrent les 383 premiers bytes du tableau **String2**. Suivons l'opération de ce code byte par byte.

Quand la CPU exécute l'instruction **movsb**, elle copie le byte à **ds:si** (**String1**) dans le byte pointé par **es:di** (**String2**). Puis elle incrémente **si** et **di**, décrémente **cx** de un et répète ce processus. Maintenant le registre **si** pointe sur **String1+1** (qui est l'adresse de **String2**) et le registre **di** pointe sur **String2+1**. L'instruction **movsb** copie le byte pointé par **si** dans le byte pointé par **di**. Cependant, c'est le byte copié à l'origine depuis l'emplacement **String1**. Ainsi l'instruction **movsb** copie la valeur à l'origine dans l'emplacement **String1** aux emplacements **String2** et **String2+1**. De nouveau, la CPU incrémente **si** et **di**, décrémente **cx** de un et répète cette opération. Maintenant l'instruction **movsb** copie le byte de l'emplacement **String1+2** (**String2+1**) à l'emplacement **String2+2**. Mais de nouveau, c'est la valeur qui était à l'origine à l'emplacement **String1**. Chaque répétition de la boucle copie l'élément suivant dans **String1** à l'emplacement disponible suivant dans la tableau **String2**. Graphiquement, cela ressemble quelque peu à la Figure 15.1.

### Figure 15.2 Manière Correcte de Déplacer des Données Avec une Opération de Déplacement en Bloc

Le résultat final est que X est recopié dans toute la chaîne. L'instruction de déplacement copie l'opérande source dans l'emplacement de mémoire qui deviendra l'opérande source pour l'opération de déplacement qui suit immédiatement, ce qui en provoque la copie.

Si vous voulez vraiment déplacer un tableau dans un autre quand ils se chevauchent, vous devriez déplacer chaque élément de la chaîne source vers la chaîne de destination en commençant à la fin des deux chaînes comme représenté sur la Figure 15.2.

La mise à un du drapeau de direction et faire pointer **si** et **di** à la fin des chaînes vous permettront de déplacer (correctement) une chaîne à l'autre quand les deux chaînes se chevauchent et que la chaîne source commence à une adresse inférieure de celle de la chaîne de destination. Si les deux chaînes se chevauchent et que la chaîne source commence à une adresse plus élevée que la chaîne de destination, alors mettez à zéro le drapeau de direction et faites pointer **si** et **di** au début des deux chaînes.

Si les deux chaînes ne se chevauchent pas, alors vous pouvez employer l'une ou l'autre technique pour déplacer les chaînes dans la mémoire. Généralement, fonctionner avec le drapeau de direction à zéro est le plus facile, et le plus naturel.

Vous ne devriez pas employer l'instruction **movs** pour remplir tableau avec une valeur de byte, mot ou double mot unique. Une autre instruction de chaîne, **stos**, est bien mieux appropriée à cette usage. Cependant, pour des tableaux dont les éléments sont plus grands que quatre bytes, vous pouvez employer l'instruction **movs** pour initialiser le tableau entier à la valeur du premier élément. Voyez les questions pour plus d'information.

---

### 15.1.5 L'Instruction CMPS

L'instruction **cmps** compare deux chaînes. La CPU compare la chaîne référencée par **es:di** à la chaîne pointée par **ds:si**. **Cx** contient la longueur des deux chaînes (lorsqu'on utilise le préfixe **rep**). Comme l'instruction **movs**, l'assembleur MASM permet plusieurs formes différentes de cette instruction :

```
{ REPE }      CMPSB
{ REPE }      CMPSW
{ REPE }      CMPSD          ; Disponible seulement sur 80386 et plus
{ REPE }      CMPS dest, source
{ REPNE }     CMPSB
{ REPNE }     CMPSW
{ REPNE }     CMPSD          ; Disponible seulement sur 80386 et plus
{ REPNE }     CMPS dest, source
```

Comme l'instruction **movs**, les opérandes présents dans la zone d'opérande de l'instruction **cmps** déterminent la taille des opérandes. Vous indiquez les adresses d'opérande réelles dans les registres **si** et **di**.

Sans préfixe de répétition, l'instruction **cmps** soustrait la valeur à l'emplacement **es:di** de la valeur à **ds:si** et met à jour les drapeaux. À part mettre à jour les drapeaux, la CPU n'utilise pas la différence produite par cette soustraction. Après avoir comparé les deux emplacements, **cmps** incrémente ou décrémente les registres **si** et **di** de un, deux, ou quatre (pour **cmpsb/cmpsw/cmpsd**, respectivement). **cmps** incrémente les registres **si** et **di** si le drapeau de direction est à zéro et sinon les décrémente.

Naturellement, vous ne capturez pas les potentialités de l'instruction **cmps** en l'utilisant pour comparer des bytes ou des mots uniques dans la mémoire. Cette instruction est remarquable pour comparer des chaînes entières. Avec **cmps**, vous pouvez comparer les éléments consécutifs dans une chaîne jusqu'à ce que vous trouviez une correspondance ou jusqu'à ce que les éléments consécutifs ne correspondent pas.

Pour comparer deux chaînes pour voir si elles sont égales ou non égales, vous devez comparer les éléments correspondants dans une chaîne jusqu'à ce qu'ils ne correspondent pas. Considérez les chaînes suivantes :

"String1"

"String1"

La seule manière de déterminer que ces deux chaînes sont égales est de comparer chaque caractère dans la première chaîne au caractère correspondant dans la seconde. Après tout, la deuxième chaîne pourrait avoir été "String2" qui n'est certainement pas égal à "String1". Naturellement, une fois que vous rencontrez un caractère dans la chaîne de destination qui n'égale pas le caractère correspondant dans la chaîne source, la comparaison peut s'arrêter. Vous n'avez pas besoin de comparer d'autre caractère dans les deux chaînes.

Le préfixe **repe** permet cette opération. Il comparera les éléments successifs dans une chaîne aussi longtemps qu'ils sont égaux et que **cx** est plus grand que zéro. Nous pourrions comparer les deux chaînes ci-dessus en employant le code suivant en assembleur 80x86 :

```
; On suppose que les deux chaînes sont dans le même segment et que ES
; et DS pointent les deux sur ce segment.
```

```
cld
lea si, AdrsChaîne1
lea di, AdrsChaîne2
mov cx, 7
repe cmpsb
```

Après l'exécution de l'instruction **cmpsb**, vous pouvez tester les drapeaux en utilisant les instructions de branchement conditionnel standards. Ceci vous permet de vérifier l'égalité, l'inégalité, moins que, plus grand que, etc...

Des chaînes de caractères sont habituellement comparées en utilisant *l'ordre lexicographique*. Dans l'ordre lexicographique, l'élément le moins signifiant d'une chaîne porte le poids le plus grand. C'est tout le contraire des comparaisons standards des nombres entiers où la partie la plus significative du nombre porte le poids le plus grand. En outre, la longueur d'une chaîne affecte la comparaison seulement si les deux chaînes sont identiques jusqu'à la longueur de la chaîne plus courte. Par exemple, "zèbre" est plus petit que "zèbres", parce qu'il est le plus court des deux chaînes, cependant, "zèbre" est plus grand que "AAAAAAAAAAH!" bien qu'il soit plus court. Les comparaisons lexicographiques comparent les éléments correspondants jusqu'à ce qu'elles rencontrent un caractère qui ne correspond pas, ou jusqu'à ce qu'elles rencontrent la fin de la chaîne plus courte. Si une paire de caractères correspondants ne correspondent pas, alors cet algorithme compare les deux chaînes basées sur ce caractère unique. Si les deux chaînes correspondent jusqu'à la longueur de la chaîne plus courte, nous devons comparer leur longueur. Les deux chaînes sont égales si et seulement si leurs longueurs sont égales et chaque paire correspondante de caractères dans les deux chaînes est identique. L'ordre lexicographique est l'ordre l'alphabétique standard avec lequel vous avez grandi.

Pour des chaînes de caractères, employez l'instruction **cmps** de la façon suivante :

- Le drapeau de direction doit être à zéro avant de comparer les chaînes.
- Employez l'instruction **cmpsb** pour comparer les chaînes sur la base de byte par byte. Même si les chaînes contiennent un nombre pair de caractères, vous ne pouvez pas employer l'instruction **cmpsw**. Elle ne compare pas des chaînes dans l'ordre lexicographique.
- Le registre **cx** doit être chargé avec la longueur de la chaîne la plus petite.
- Employez le préfixe **repe**.
- Les registres **ds:si** et **es:di** doivent pointer sur le tout premier caractère dans les deux chaînes que vous voulez comparer.

Après l'exécution de l'instruction **cmps**, si les deux chaînes étaient égales, leurs longueurs doivent être comparées afin de finir la comparaison. Le code suivant compare une paire de chaînes de caractères :

```

        lea    si, source
        lea    di, dest
        mov    cx, lengthSource
        mov    ax, lengthDest
        cmp    cx, ax
        ja     NoSwap
        xchg   ax, cx
NoSwap:    repe cmpsb
        jne    NotEqual
        mov    ax, lengthSource
        cmp    ax, lengthDest
NotEqual:

```

Si vous employez des bytes pour contenir les longueurs de chaîne, vous devrez modifier ce code de manière appropriée.

Vous pouvez également employer l'instruction **cmps** pour comparer des valeurs de nombre entier multi-mots (c'est-à-dire, valeurs de nombre entier avec précision étendue). En raison de la lourdeur de l'installation exigée pour une comparaison de chaîne, ce n'est pas pratique pour des valeurs de nombre entier de longueur inférieure à trois ou quatre mots, mais pour de grandes valeurs entières, c'est une excellente manière de comparer de telles valeurs. Au contraire des chaînes de caractères, nous ne pouvons pas comparer des chaînes de nombre entier en utilisant un ordre lexicographique. En comparant des chaînes, nous comparons les caractères du byte le moins significatif au byte le plus significatif. En comparant des nombres entiers, nous devons comparer les valeurs du byte le plus significatif (ou mot/double mot) vers le byte, mot ou double mot le moins significatif. Ainsi, pour comparer deux valeurs de nombre entier de huit mots (128-bits), employez le code suivant sur les 80286 :

```

std
lea    si, SourceInteger+14

```



```

        lea    di, DestInteger+14
        mov    cx, 8
repe    cmpsw

```

Ce code compare les nombres entiers de leur mot le plus significatif vers le mot le moins significatif. L'instruction **cmpsw** finit quand les deux valeurs sont inégales ou lorsque **cx** est décrémenté à zéro (impliquant que les deux valeurs sont égales). De nouveau, les drapeaux fournissent le résultat de la comparaison.

Le préfixe **repne** demandera à l'instruction **cmps** de comparer les éléments successifs de la chaîne aussi longtemps qu'ils ne correspondent pas. Les drapeaux du 80x86 sont de peu d'utilité après l'exécution de cette instruction. Soit le registre **cx** est à zéro (dans ce cas les deux chaînes sont totalement différentes), soit il contient le nombre d'éléments comparés dans les deux chaînes jusqu'à une correspondance. Alors que cette forme de l'instruction **cmps** n'est pas particulièrement utile pour comparer des chaînes, elle est utile pour localiser la première paire d'éléments correspondants dans une paire de tableaux de bytes ou de mots. En général, cependant, vous emploierez rarement le préfixe **repne** avec **cmps**.

Une dernière chose à garder à l'esprit en employant l'instruction **cmps** - la valeur dans le registre **cx** détermine le nombre d'éléments à traiter, pas le nombre de bytes. Par conséquent, en utilisant **cmpsw**, **cx** indique le nombre de mots à comparer. Ceci, naturellement, est deux fois le nombre de bytes à comparer.

---

### 15.1.6 L'Instruction SCAS

L'instruction **cmps** compare deux chaînes entre elles. Vous ne pouvez pas l'employer pour rechercher un élément particulier dans une chaîne. Par exemple, vous ne pourriez pas employer l'instruction **cmps** pour balayer rapidement une autre chaîne à la recherche d'un zéro. Vous pouvez employer l'instruction **scas** (scan string) pour cette tâche.

A la différence des instructions **movs** et **cmps**, l'instruction **scas** exige seulement une chaîne de destination (**es:di**) au lieu d'une chaîne source et une de destination. L'opérande source est la valeur dans le registre **al** (**scasb**), **ax** (**scasw**) ou **eax** (**scasd**).

L'instruction **scas**, par elle-même, compare la valeur dans l'accumulateur (**al**, **ax** ou **eax**) avec la valeur dirigée par **es:di** et puis incrémente (ou décrémente) **di** de un, deux ou quatre. La CPU modifie les drapeaux selon le résultat de la comparaison. Bien que ceci puisse être utile occasionnellement, **scas** est beaucoup plus utile en utilisant les préfixes **repe** et **repne**.

Quand le préfixe **repe** (répétition tant qu'égal) est présent, **scas** balaye la chaîne à la recherche d'un élément qui ne correspond pas à la valeur dans l'accumulateur. En utilisant le préfixe **repne** (répétition tant que non égal), **scas** balaye la chaîne à la recherche du premier élément de chaîne qui est égal à la valeur dans l'accumulateur.

Vous vous demandez probablement "pourquoi ces préfixes font-ils exactement le contraire de ce qu'ils doivent faire?" Les paragraphes ci-dessus n'ont pas décrit l'opération de l'instruction **scas** tout à fait correctement. En utilisant le préfixe **repe** avec **scas**, le 80x86 balaye la chaîne tant que la valeur dans l'accumulateur est égale à l'opérande de chaîne. Cela équivaut à rechercher dans la chaîne le premier élément qui ne correspond pas la valeur dans l'accumulateur. L'instruction **scas** avec **repne** balaye la chaîne tant que l'accumulateur n'est pas égal à l'opérande de chaîne. Naturellement, cette forme recherche la première valeur dans la chaîne qui correspond à la valeur dans le registre accumulateur. L'instruction **scas** prend les formes suivantes :

```

{ REPE }    SCASB
{ REPE }    SCASW
{ REPE }    SCASD      ; Disponible seulement sur 80386 et plus
{ REPE }    SCAS dest
{ REPNE }   SCASB
{ REPNE }   SCASW
{ REPNE }   SCASD      ; Disponible seulement sur 80386 et plus
{ REPNE }   SCAS dest

```

Comme les instructions **cmps** et **movs**, la valeur dans le registre **cx** indique le nombre d'éléments à traiter, pas des bytes, en utilisant un préfixe de répétition.

---

### 15.1.7 L'Instruction STOS

L'instruction **stos** stocke la valeur dans l'accumulateur à l'emplacement indiqué par **es:di**. Après le stockage de la valeur, l'unité centrale de traitement d'incrmente ou de décrémente **di** selon l'état du drapeau de direction. Bien que l'instruction **stos** ait beaucoup d'utilisations, son utilisation primaire est d'initialiser des tableaux et des chaînes avec une valeur constante. Par exemple, si vous avez un tableau de 256 bytes que vous voulez remplir avec des zéros, employez le code suivant :

```
; Le registre ES est supposé pointer sur le segment contenant
; DestChaîne
    cld
    lea di, DestChaîne
    mov cx, 128          ;256 bytes soit 128 words.
    xor ax, ax           ;AX := 0
rep  stosw
```

Ce code écrit 128 mots au lieu de 256 bytes parce qu'une opération **stosw** unique est plus rapide que deux opérations **stosb**. Sur des 80386 ou plus ce code pourrait avoir écrit 64 double mots pour accomplir la même chose encore plus rapidement. L'instruction **stos** prend quatre formes. Elles sont

```
{REP}    STOSB
{REP}    STOSW
{REP}    STOSD
{REP}    STOS      dest
```

L'instruction **stosb** stocke la valeur dans le registre **al** dans le(s) emplacement(s) de mémoire indiqué(s), l'instruction **stosw** stocke le registre **ax** dans le(s) emplacement(s) de mémoire indiqué(s) et l'instruction **stosd** stocke **eax** dans le(s) emplacement(s) indiqué(s). L'instruction **stos** est une instruction **stosb**, **stosw** ou **stosd** dépendant de la taille de l'opérande indiqué.

Gardez à l'esprit que l'instruction **stos** est utile seulement pour initialiser un tableau de bytes, words ou dwords à une valeur constante. Si vous devez initialiser un tableau avec des valeurs différentes, vous ne pouvez pas employer l'instruction **stos**. Vous pouvez employer des **movs** dans une telle situation, voyez les exercices pour les détails supplémentaires.

---

### 15.1.8 L'Instruction LODS

L'instruction **lods** est unique parmi les instructions de chaîne. Vous n'emploierez jamais un préfixe de répétition avec cette instruction. L'instruction **lods** copie le byte ou le mot pointé par **ds:si** dans le registre **al**, **ax**, ou **eax**, après quoi elle incrémente ou décrémente le registre **si** de un, deux ou quatre. La répétition de cette instruction par l'intermédiaire du préfixe de répétition ne servirait à rien puisque le registre accumulateur serait écrasé chaque fois que l'instruction **lods** se répète. À l'in de l'opération de répétition, l'accumulateur contiendra la dernière valeur indiquée de la mémoire.

Au lieu de cela, employez l'instruction **lods** pour chercher des bytes (**lods b**), des mots (**lods w**) ou des double mots (**lods d**) en mémoire pour un traitement ultérieur. En employant l'instruction **stos**, vous pouvez synthétiser des opérations de chaînes puissantes. Comme l'instruction **stos**, l'instruction **lods** prend quatre formes :

```
{REP}    LODSB
{REP}    LODSW
{REP}    LODSD          ;Disponible slmt sur 80386 et plus
{REP}    LODS      dest
```

Comme mentionné précédemment, vous employerez rarement, si jamais, les préfixes **rep** avec ces instructions<sup>3</sup>. Le 80x86 incrémente ou décrémente **si** de un, deux ou quatre selon le drapeau de direction et selon que vous employez l'instruction **lods b**, **lods w** ou **lods d**.

<sup>3</sup>Ils apparaissent ici uniquement parce qu'ils sont permis. Ils ne sont pas utiles, mais ils sont permis.

---

### 15.1.9 Construction de Fonctions de Chaînes Complexes avec LODS et STOS

Le 80x86 supporte seulement cinq instructions de chaînes différentes : **movs**, **cmps**, **scas**, **lods** et **stos**<sup>4</sup>. Ce ne sont certainement pas les seules opérations de chaînes que vous voudrez employer. Cependant, vous pouvez employer les instructions **lods** et **stos** pour produire facilement n'importe quelle opération particulière de chaîne que vous désirez. Par exemple, supposez que vous vouliez une opération de chaîne qui convertit tous les caractères majuscules dans une chaîne en minuscules. Vous pourriez employer le code suivant :

```
; ES et DS sont supposés avoir été initialisés pour pointer sur le
; même segment, celui contenant la chaîne à convertir.
```

```
        lea    si, Chaîne2Convert
        mov    di, si
        mov    cx, LengthOfChaîne
Convert2Lower: lodsb                ;Obtient caractère suivant.
               cmp    al, 'A'        ;Majuscule?
               jnb    NotUpper
               cmp    al, 'Z'
               ja     NotUpper
               or     al, 20h         ;Convertit en minuscule.
NotUpper: stosb                    ;Stocke dans destination.
               loop   Convert2Lower
```

A supposer que vous soyez prêt à dépenser 256 bytes pour un tableau, cette opération de conversion peut être quelque peu accélérée en utilisant l'instruction **xlat**:

```
; ES et DS sont supposés avoir été initialisés pour pointer sur le
; même segment, celui contenant la chaîne à convertir.
```

```
        cld
        lea    si, Chaîne2Convert
        mov    di, si
        mov    cx, LengthOfChaîne
        lea    bx, ConversionTable
Convert2Lower: lodsb                ;Obtient caractère suivant.
               xlat                 ;Convertit si besoin.
               stosb                ;Stocke dans destination.
               loop   Convert2Lower
```

La table de conversion, naturellement, contiendrait l'index dans la table à chaque emplacement excepté aux offsets 41h..5Ah. A ces emplacements la table de conversion contiendrait les valeurs 61h..7Ah (c.-à-d., aux index 'A'..'Z' la table contiendrait les codes pour 'a'..'z').

Puisque les instructions **lods** et **stos** utilisent l'accumulateur comme intermédiaire, vous pouvez employer n'importe quelle opération de l'accumulateur pour manipuler rapidement des éléments de chaîne.

---

### 15.1.10 Les Préfixes et les Instructions de Chaînes

Les instructions de chaîne accepteront des préfixes de segment, des préfixes de blocage (lock) et des préfixes de répétition. En fait, vous pouvez indiquer les trois types d'instruction de préfixes si vous le désirez. Cependant, en raison d'un bogue dans les premières puces 80x86 (pré-80386), vous ne devriez jamais employer plus d'un seul préfixe (répétition, blocage ou surcharge de segment) pour une instruction de chaîne à moins que votre code fonctionne seulement sur les processeurs postérieurs ; ce qui est probable de nos jours. Si vous devez absolument employer deux préfixes ou plus et avoir besoin de fonctionner sur un processeur plus ancien, assurez-vous de bloquer les interruptions en exécutant l'instruction de chaîne.

<sup>4</sup>Sans compter INS et OUTS que nous ignorons ici..

---

## 15.2 Chaînes de Caractères

Puisque vous rencontrerez des chaînes de caractères plus souvent que d'autres types de chaînes, elles méritent une attention particulière. Les sections suivantes décrivent des chaînes de caractères et de divers types d'opérations de chaîne.

---

### 15.2.1 Types de Chaînes

Au niveau le plus basique, l'instruction de chaîne du 80x86 fonctionne seulement sur des tableaux de caractères. Cependant, puisque la plupart des types de données de chaîne contiennent un tableau de caractères comme composant, les instructions de la chaîne du 80x86 sont maniables pour manipuler cette partie de la chaîne.

Probablement, la plus grande différence entre une chaîne de caractères et un tableaux de caractères est l'attribut de longueur. Un tableau de caractères contient un nombre fixe de caractères. Jamais plus, jamais moins. Une chaîne de caractères, cependant, a une longueur dynamique en contexte d'exécution, c.-à-d., le nombre de caractères contenus dans la chaîne à un certain point dans le programme. Les chaînes de caractères, à la différence des tableaux de caractères, ont la capacité de changer leur taille pendant l'exécution (dans certaines limites, naturellement).

Pour compliquer des choses encore plus, il y a deux types génériques de chaînes : chaînes allouées statiquement et chaînes allouées dynamiquement. Des chaînes allouées statiquement se voient donner une longueur fixe et maximum au moment de la création du programme. La longueur de la chaîne peut changer lors de l'exécution, mais seulement entre zéro et cette longueur maximum. La plupart des systèmes allouent et libèrent les chaînes dynamiquement allouées dans un "bassin" de mémoire lors de l'utilisation de chaînes. De telles chaînes peuvent être n'importe quelle longueur (jusqu'à une certaine valeur maximum raisonnable). L'accès à de telles chaînes est moins efficace qu'accéder aux chaînes allouées statiquement. En outre, la collecte des résidus<sup>5</sup> peut prendre du temps supplémentaire. Néanmoins, les chaînes allouées dynamiquement sont beaucoup plus efficaces en espace que les chaînes allouées statiquement et, parfois, l'accès à des chaînes allouées dynamiquement est plus rapide aussi. La plupart des exemples de ce chapitre emploieront les chaînes allouées statiquement.

Une chaîne avec une longueur dynamique a besoin d'une certaine manière de garder une trace cette longueur. Alors qu'il y a plusieurs manières possibles de représenter des longueurs de chaîne, les deux les plus populaires sont les chaînes à préfixe de longueur et les chaînes terminées par zéro. Une chaîne à préfixe de longueur se compose d'un byte ou d'un mot unique qui contient la longueur de cette chaîne. Juste après cette valeur de longueur, se trouvent les caractères qui composent la chaîne. En supposant qu'on utilise le byte comme longueur de préfixe, vous pourriez définir la chaîne "BONJOUR" comme suit :

```
HelloStr    byte 7, "BONJOUR"
```

Les chaînes à préfixe de longueur s'appellent souvent les chaînes Pascal puisque c'est le type de variable de chaîne supporté par la plupart des versions du Pascal<sup>6</sup>.

Une autre manière populaire d'indiquer des longueurs de chaîne est d'employer les chaînes terminées par zéro. Une chaîne terminée par zéro se compose d'une chaîne des caractères terminés avec un byte zéro. Ces types de chaînes s'appellent souvent les chaînes C puisqu'elles sont le type employé par le langage de programmation C/C++<sup>+</sup>. La Bibliothèque Standard de l'UCR, puisqu'elle imite la bibliothèque standard du C, emploie également les chaînes terminées par zéro.

Les chaînes de Pascal sont bien meilleures que les chaînes de C/C++ pour plusieurs raisons. D'abord, le calcul de la longueur d'une chaîne de Pascal est trivial. Vous devez seulement chercher le premier byte (ou mot) de la chaîne et vous avez la longueur de la chaîne. Le calcul de la longueur d'une chaîne C/C++ est considérablement moins efficace. Vous devez balayer la chaîne entière (par exemple, en utilisant l'instruction **scasb**) à la recherche d'un byte zéro. Si la chaîne C/C++ est longue, ceci peut prendre un bon moment. En outre, les chaînes de C/C++ ne peuvent pas contenir le caractère NULL. D'autre part, les chaînes de C/C++ peuvent être de n'importe quelle longueur, tout en coûtant un seul byte supplémentaire. Les chaînes Pascal, au contraire, ne peuvent faire plus de

<sup>5</sup>La récupération des stockages inutilisés.

<sup>6</sup>Au moins ces versions de Pascal qui supportent les chaînes.

255 caractères en utilisant seulement un byte unique de longueur. Pour des chaînes plus longues que 255 bytes, vous aurez besoin de deux bytes pour contenir la longueur d'une chaîne Pascal. Puisque la plupart des chaînes font moins de 256 caractères de longueur, ce n'est pas réellement un inconvénient.

Un avantage des chaînes terminées par zéro est qu'il est facile de les employer dans un programme en assembleur. C'est particulièrement vrai des chaînes qui sont assez longues pour avoir besoin de lignes multiples dans le code source dans vos programmes en assembleur. Compter chaque caractère dans une chaîne est si pénible que ce n'est même pas la peine d'y penser. Cependant, vous pouvez écrire une macro qui construira facilement des chaînes Pascal à votre place :

```
PString      macro      String
              local      StringLength, StringStart
              byte      StringLength
StringStart  byte      String
StringLength =      $-StringStart
              endm
              .
              .
              .
PString      "Cette chaîne a un préfixe de longueur"
```

Tant que la chaîne tient entièrement sur une ligne de source, vous pouvez employer cette macro pour produire des chaînes du modèle Pascal.

Des fonctions de chaînes courantes comme la concaténation, la longueur, l'extraction de sous-chaîne, l'indexation, etc. sont beaucoup plus faciles à écrire en utilisant les chaînes à préfixe de longueur. Ainsi nous emploierons des chaînes Pascal sauf indication contraire. En outre, la bibliothèque standard de l'UCR fournit un grand nombre de fonctions de chaîne C/C++, aussi il n'est aucunement besoin de répéter ces fonctions ici.

---

### 15.2.2 Assignment de Chaîne

Vous pouvez facilement assigner une chaîne à une autre en employant l'instruction **movsb**. Par exemple, si vous voulez assigner la chaîne à préfixe de longueur Chaîne1 à Chaîne2, employez ce qui suit :

; ES et DS sont supposés être initialisés

```
        lea      si, String1
        lea      di, String2
        mov      ch, 0           ;Etend len à 16 bits.
        mov      cl, String1     ;Obtient longueur de chaîne.
        inc      cx              ;Inclut le byte de longueur.
rep     movsb
```

Ce code incrémente **cx** de un avant l'exécution de **movsb** parce que le byte de longueur contient la longueur de la chaîne moins le byte de longueur lui-même.

Généralement, des variables de chaîne peuvent être initialisées à des constantes en employant la macro **PString** décrite plus tôt. Cependant, si vous devez placer une variable de chaîne à une certaine valeur constante, vous pouvez écrire une routine **StrAssign** qui assigne la chaîne qui suit immédiatement le **call**. La procédure suivante fait exactement cela :

```
        include      stdlib.a
        includelib   stdlib.lib

cseg    segment      para public 'code'
        assume       cs:cseg, ds:dseg, es:dseg, ss:sseg

; Procédure d'assignation de chaîne
```

```

MainPgm    proc        far
            mov     ax, seg dseg
            mov     ds, ax
            mov     es, ax

            lea     di, ToString
            call    StrAssign
            byte    "This is an example of how the "
            byte    "StrAssign routine is used",0
            nop
            ExitPgm
MainPgm    endp

StrAssign  proc near
            push bp
            mov     bp, sp
            pushf
            push ds
            push si
            push di
            push cx
            push ax
            push di                ;Sauve encore pour utiliser plus tard.
            push es
            cld

; Obtient l'adresse de la chaîne source

            mov     ax, cs
            mov     es, ax
            mov     di, 2[bp]        ;Obtient adresse de retour.
            mov     cx, 0ffffh       ;Balaie sans arrêt.
            mov     al, 0             ;Cherche un zéro.
            repne scasb               ;Calcule la longueur de la chaîne.
            neg     cx                 ;Convertit la longueur en nb positif.
            dec     cx                 ;Car on a commencé avec -1, pas 0.
            dec     cx                 ;Saute le byte zéro de terminaison.

; Maintenant copie les chaînes

            pop     es                 ;Obtient segment de destination.
            pop     di                 ;Obtient adresse de destination.
            mov     al, cl             ;Stocke byte de longueur.
            stosb

; Maintenant copie la chaîne source.

            mov     ax, cs
            mov     ds, ax
            mov     si, 2[bp]
            rep     movsb

; Met à jour l'adresse de retour et quitte:

```

```

        inc    si                ;Saute le byte zéro.
        mov    2[bp], si
        pop    ax
        pop    cx
        pop    di
        pop    si
        pop    ds
        popf
        pop    bp
        ret
StrAssign endp

cseg      ends

dseg      segment      para public 'data'
ToString byte          255 dup (0)
dseg      ends

sseg      segment      para stack 'stack'
          word          256 dup (?)
sseg      ends
          end            MainPgm

```

Ce code emploie l'instruction **scas** pour déterminer la longueur de la chaîne juste après l'instruction **call**. Une fois que le code a déterminé la longueur, il stocke cette longueur dans le premier byte de la chaîne de destination et puis copie le texte suivant l'appel dans la variable de chaîne. Après avoir copié la chaîne, ce code ajuste l'adresse de retour de sorte qu'elle pointe juste après le byte de terminaison zéro. Alors la procédure renvoie le contrôle à l'appelant.

Naturellement, cette procédure d'assignation de chaîne n'est pas très efficace, mais elle est très facile à employer. Définir **es:di** est tout ce que vous devez faire pour employer cette procédure. Si vous avez besoin d'assignation de chaîne rapide, employez uniquement l'instruction **movs** comme suit :

; ES et DS sont supposés être déjà initialisés

```

        lea    si, SourceString
        lea    di, DestString
        mov    cx, LengthSource
rep     movsb
        .
        .
        .
SourceString byte LengthSource-1
          byte "This is an example of how the "
          byte "StrAssign routine is used"
LengthSource =    $-SourceString

DestString  byte 256 dup (?)

```

Employer des instructions en-ligne exige considérablement plus de préparation (et de dactylographie !), mais elles sont beaucoup plus rapides que la procédure **StrAssign**. Si vous n'aimez pas taper du code, vous pouvez toujours écrire une macro pour faire l'assignation de chaîne à votre place.

---

### 15.2.3 Comparaison de Chaînes

La comparaison de deux chaînes de caractères a été déjà largement débattu dans la section sur l'instruction **cmps**. A part fournir quelques exemples concrets, il n'y a aucune raison de traiter ce sujet plus en détail.

Note : tous les exemples suivants supposent que **es** et **ds** pointent sur les segments appropriés contenant les chaînes de destination et de source.

Comparer Str1 à Str2

```
        lea    si, Str1
        lea    di, Str2
; Obtenir la longueur minimum des deux chaînes.

        mov    al, Str1
        mov    cl, al
        cmp    al, Str2
        jnb    CmpStrs
        mov    cl, Str2

; Comparer les deux chaînes.

CmpStrs:  mov    ch, 0
          cld
          repe cmpsb
          jne    StrsNotEqual

; Si CMPS trouve qu'elles sont égales, comparer leurs longueurs
; juste pour être sûrs.

          cmp    al, Str2
StrsNotEqual:
```

À l'étiquette **StrsNotEqual**, les drapeaux contiendront toutes les informations intéressantes sur l'ordre de ces deux chaînes. Vous pouvez employer les instructions de branchement conditionnel pour examiner le résultat de cette comparaison.

---

### 15.3 Fonctions de Chaînes de Caractères

La plupart des langages de niveau élevé, comme le Pascal, le BASIC, le "C", et le PL/I, fournissent plusieurs fonctions et procédures de chaîne (soit intégrées au langage, soit comme éléments d'une bibliothèque standard). A part les cinq opérations de chaîne fournies ci-dessus, le 80x86 ne supporte aucune fonction de chaîne. Par conséquent, si vous avez besoin d'une fonction de chaîne particulière, vous devrez l'écrire vous-même. Les sections suivantes décrivent plusieurs des fonctions de chaîne les plus populaires et comment les mettre en application en assembleur.

---

#### 15.3.1 Substr

La fonction **Substr** (sous-chaîne) copie une partie d'une chaîne vers une autre. Dans un langage de niveau élevé, cette fonction prend habituellement la forme :

```
DestStr := Substr(SrcStr, Index, Length) ;
```

où :

- **DestStr** est le nom de la variable de chaîne où vous voulez stocker la sous-chaîne
- **SrcStr** est le nom de la chaîne source (où la sous-chaîne doit être prise)
- **Index** est la position du caractère de début dans la chaîne (1..length(SrcStr)), et
- **Length** est la longueur de la sous-chaîne que vous voulez copier dans **DestStr**.

Les exemples suivants montrent comment **Substr** fonctionne.



```
SrcStr := 'This is an example of a string';
DestStr := Substr(SrcStr,11,7);
write(DestStr);
```

Ceci imprime "exemple". La valeur d'index est onze, aussi, la fonction **Substr** commencera à copier des données en commençant au onzième caractère dans la chaîne. Le onzième caractère est le "e" ans "exemple". La longueur de la chaîne est sept.

Cette invocation copie les sept caractères d' "exemple" dans **DestStr**.

```
SrcStr := 'This is an example of a string';
DestStr := Substr(SrcStr,1,10);
write(DestStr);
```

Ceci imprime "This is an". Puisque l'index est un, cette occurrence de la fonction **Substr** commence à copier 10 caractères en commençant par le premier caractère de la chaîne.

```
SrcStr := 'This is an example of a string';
DestStr := Substr(SrcStr,20,11);
write(DestStr);
```

Ceci imprime "of a string". Cet appel à **Substr** extrait les onze derniers caractères de la chaîne.

Que se produit-il si les valeurs d'index et de longueur sont hors des limites ? Par exemple, que se produit-il si l'index est zéro ou est plus grand que la longueur de la chaîne ? Que se produit-il si l'index est correct, mais la somme de l'index et de la longueur est plus grande que la longueur de la chaîne source ? Vous pouvez gérer ces situations anormales de trois manières : (1)ignorer la possibilité d'erreur ; (2)arrêter le programme avec une erreur d'exécution ; (3)traiter un nombre raisonnable de caractères en réponse à la requête.

.La première solution part du principe que l'appelant ne fait jamais d'erreur en calculant les valeurs pour les paramètres de la fonction **Substr**. Elle suppose aveuglément que les valeurs passées à la fonction **Substr** sont correctes et traite la chaîne en se basant sur cette hypothèse. Ceci peut produire quelques effets bizarres. Considérez les exemples suivants, qui emploient des chaînes à en-tête de longueur :

```
SourceStr := '1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ';
DestStr := Substr(SourceStr,0,5);
Write('DestStr');
```

imprime "\$1234". La raison, naturellement, est que SourceStr est une chaîne à en-tête de longueur. Par conséquent la longueur, 36, apparaît à l'offset zéro dans la chaîne. Si **Substr** emploie l'index illégal zéro alors la longueur de la chaîne sera retournée comme premier caractère. Dans ce cas particulier, la longueur de la chaîne, 36, se trouve correspondre au code ASCII du caractère "\$".

La situation est bien pire si la valeur indiquée pour l'index est négative ou est plus grande que la longueur de la chaîne. En ce cas, la fonction **Substr** renverrait une sous-chaîne contenant des caractères apparaissant avant ou après la chaîne source. Ce n'est pas un résultat raisonnable.

Malgré les problèmes rencontrés en ignorant la possibilité d'erreur dans la fonction **Substr**, il y a un grand avantage à traiter des sous-chaînes de cette manière : le code de Substr résultant est plus efficace s'il ne doit exécuter aucune vérification des données à l'exécution. Si vous savez que les valeurs d'index et de longueur sont toujours dans une marge acceptable, alors il n'y a aucun besoin de faire de vérification dans la fonction **Substr**. Si vous pouvez garantir qu'une erreur ne se produira pas, vos programmes fonctionneront (quelque peu) plus rapidement en éliminant le contrôle à l'exécution.

Puisque la plupart des programmes sont rarement sans erreur, vous faites un grand pari si vous supposez que tous les appels à la routine **Substr** passent des valeurs raisonnables. Par conséquent, une un contrôle à l'exécution est souvent nécessaire pour déceler des erreurs dans votre programme. Une erreur se produit dans les conditions suivantes :

- Le paramètre d'index (**Index**) est plus petit que un

- **Index** est plus grand que la longueur de la chaîne
- Le paramètre de longueur de **Substr (Length)** est plus grand que la longueur de la chaîne
- La somme de **Index** et de **Length** est plus grande que la longueur de la chaîne.

Une alternative à ignorer une de ces erreurs est d'arrêter le programme avec un message d'erreur. C'est probablement très bien pendant la phase d'élaboration du programme, mais une fois que votre programme est aux mains des utilisateurs, cela pourrait être un vrai désastre. Vos clients ne seraient pas très heureux s'ils passaient toute la journée à saisir des données dans un programme et qu'il s'arrête, entraînant la perte des données qu'ils ont saisies. Une alternative à arrêter le programme quand une erreur se produit est de faire retourner la fonction **Substr** avec une condition d'erreur. Laissez alors le soin au code appelant de déterminer si une erreur s'est produite. Cette technique marche bien avec la troisième alternative de gestion des erreurs : traiter la sous-chaîne du mieux que vous pouvez.

La troisième alternative, gérer l'erreur du mieux que vous pouvez, est probablement la meilleure alternative. Gérez les conditions d'erreur de la façon suivante :

- Le paramètre d'index (**Index**) est plus petit que un. Il y a deux manières de gérer cette condition d'erreur. L'une est de mettre automatiquement le paramètre **Index** à un et renvoyer la sous-chaîne commençant par le premier caractère de la chaîne source. L'autre alternative est de renvoyer la *chaîne vide*, une chaîne de longueur zéro, comme sous-chaîne. Des variations sur ce thème sont également possibles. Vous pourriez renvoyer la sous-chaîne commençant par le premier caractère si l'index est zéro et une chaîne vide si l'index est négatif. Une autre alternative est d'employer des nombres non signés. Alors vous n'avez à vous préoccuper que du cas où **Index** est zéro. Un nombre négatif, si le code appelant en produisait accidentellement un, ressemblerait à un grand nombre positif.
- **Index** est plus grand que la longueur de la chaîne. Si c'est le cas, alors la fonction Substr devrait renvoyer une chaîne vide. Intuitivement, c'est la réponse appropriée dans cette situation
- Le paramètre de longueur de **Substr (Length)** est plus grand que la longueur de la chaîne. - ou -
- La somme de **Index** et de **Length** est plus grande que la longueur de la chaîne. Les points trois et quatre sont le même problème, la longueur de la sous-chaîne désirée se prolonge au delà de la fin de la chaîne source. Dans une telle éventualité, Substr devrait renvoyer la sous-chaîne se composant de ces caractères commençant à l'index jusqu'à la fin de la chaîne source.

Le code suivant pour la fonction **Substr** attend quatre paramètres : les adresses des chaînes source et destination, l'index de début, et la longueur de la sous-chaîne désirée. **Substr** attend les paramètres dans les registres suivants :

<b>ds:si-</b>	L'adresse de la chaîne source
<b>es:di-</b>	L'adresse de la chaîne destination
<b>ch-</b>	L'index de début.
<b>cl-</b>	La longueur de la sous-chaîne.

**Substr** renvoie les valeurs suivantes :

- La sous-chaîne, à l'emplacement **es:di**.
- **Substr** met à zéro le drapeau de retenue s'il n'y avait aucune erreur. **Substr** met à un le drapeau de retenue s'il y avait une erreur.
- **Substr** préserve tous les registres.

Si une erreur se produit, alors le code appelant doit examiner les valeurs dans **si**, **di** et **cx** pour déterminer la cause exacte de l'erreur (si c'est nécessaire). En cas d'une erreur, la fonction **Substr** renvoie les sous-chaînes suivantes :

- Si le paramètre **Index** (**ch**) est zéro, Substr utilise un à la place
- Les paramètres **Index** et **Length** sont les deux des valeurs octet non signées, donc ils ne sont jamais négatifs
- Si le paramètre **Index** est plus grand que la longueur de la chaîne de source, **Substr** renvoie-t-il une chaîne vide

- Si la somme des paramètres **Index** et **Length** est plus grande que la longueur de la chaîne de source, **Substr** renvoie seulement les caractères depuis **Index** jusqu'à la fin de la chaîne source. Le code suivant réalise la fonction sous-chaîne.

```
; Fonction sous-chaîne.
;
; forme HLL:
;
;procedure substring(var Src:string;
;                    Index, Length:integer;
;                    var Dest: string);
;
; Src- Adresse d'une chaîne source.
; Index- Index dans la chaîne source.
; Length- Longueur de la sous-chaîne à extraire.
; Dest- Adresse d'une chaîne destination.
;
; Copie la chaîne source de l'adresse [Src+index] de longueur
; Length dans la chaîne destination.
;
; Si une erreur se produit, le drapeau carry est renvoyé à un,
; sinon à zéro.
;
; Les paramètres sont passés comme suit:
;
; DS:SI- adresse chaîne source.
; ES:DI- adresse chaîne destination.
; CH- Index dans la chaîne source.
; CL- Longueur de la chaîne source.
;
; Note: les chaînes pointée par les registres SI et DI sont
; des chaînes à préfixe de longueur. C'est à dire, le premier byte de
; chaque chaîne contient la longueur de cette chaîne.

Substring      proc near
                push ax
                push cx
                push di
                push si
                cld                                ;On suppose qu'il n'y a pas d'erreur.
                pushf                               ;Sauve le statut du flag direction.

; Vérifie la validité des paramètres.

                cmp     ch, [si]    ;L'index est-il au-delà de la fin de
ja             ReturnEmpty        ; la chaîne source?
mov     al, ch                    ;Voir si la somme d'index et
dec     al                        ; length est au-delà de la fin de
add     al, cl                    ; la chaîne.
jc      TooLong                   ;Erreur if > 255.
cmp     al, [si]                  ;Au-delà de la fin de source?
jbe     OkaySoFar
```

```
; Si la sous-chaîne ne tient pas complètement dans la chaîne source,
; la tronquer:
```

```
TooLong:      popf
              stc                      ;Retourne un flag d'erreur.
              pushf
              mov  al, [si]             ;Obtient la longueur maximum.
              sub  al, ch               ;Soustrait la valeur d'index.
              inc  al                   ;Ajuste de manière appropriée.
              mov  cl, al               ;Sauve comme nouvelle longueur.

OkaySoFar:   mov  es:[di], cl ;Sauve longueur chaîne destination.
              inc  di
              mov  al, ch               ;Obtient l'index dans la source.
              mov  ch, 0                ;Etend à zéro la valeur longueur CX.
              mov  ah, 0                ;Etend à zéro l'index AX.
              add  si, ax               ;Calcule adresse de la sous-chaîne.
              cld
              rep  movsb                ;Copie la sous-chaîne.

              popf

SubStrDone:   pop  si
              pop  di
              pop  cx
              pop  ax
              ret

; Retourne une chaîne vide ici:

ReturnEmpty:  mov  byte ptr es:[di], 0
              popf
              stc
              jmp  SubStrDone

SubString     endp
```

---

### 15.3.2 Index

La fonction de chaîne **Index** recherche la première occurrence d'une chaîne dans une autre et renvoie l'offset de cette occurrence. Considérez la forme HLL suivante:

```
SourceStr := 'Hello world';
TestStr   := 'world';
I := INDEX(SourceStr, TestStr);
```

La fonction **Index** balaye la chaîne source à la recherche de la première occurrence de la chaîne test. Si elle la trouve, elle renvoie l'index où la chaîne test commence dans la chaîne source. Dans l'exemple ci-dessus, la fonction **Index** renverrait sept puisque la sous-chaîne "world" commence à la septième position de caractères dans la chaîne source.

La seule erreur possible se produit si **Index** ne peut pas trouver la chaîne test dans la chaîne source. Dans une telle situation, la plupart des implémentations retournent zéro. Notre version fera de même. La fonction **Index** qui suit fonctionne de la façon suivante :

- 1) Elle compare la longueur de la chaîne test avec la longueur de la chaîne source. Si la chaîne test est plus longue, l'index renvoie immédiatement zéro puisqu'il n'y a aucune manière que la chaîne test sera trouvée dans la chaîne source dans cette situation.

- 2) la fonction **index** fonctionne comme suit :

```
i := 1;
while (i < (length(source)-length(test)) and
      test <> substr(source, i, length(test)) do
    i := i+1;
```

Quand cette boucle se termine, si  $(i < \text{length}(\text{source}) - \text{length}(\text{test}))$  alors elle contient l'index où test commence dans la source. Autrement test n'est pas une sous-chaîne de source. En utilisant l'exemple précédent, cette boucle compare test à la source de la façon suivante :

i=1		
test:	world	No match
source:	Hello world	
i=2		
test:	world	No match
source:	Hello world	
i=3		
test:	world	No match
source:	Hello world	
i=4		
test:	world	No match
source:	Hello world	
i=5		
test:	world	No match
source:	Hello world	
i=6		
test:	world	No match
source:	Hello world	
i=7		
test:	world	Match
source:	Hello world	

Il y a (algorithmiquement) de meilleures manières de faire cette comparaison<sup>7</sup>, cependant, l'algorithme ci-dessus se prête bien à l'utilisation des instructions de chaîne du 80x86 et est très facile à comprendre. Le code d'**Index** suit :

```
; INDEX- calcule l'offset d'une chaîne au sein d'une autre.
;
; En entrée:
;
; ES:DI- Pointe sur la chaîne test qu'INDEX va rechercher
; dans la chaîne source.
; DS:SI- Pointe sur la chaîne source qui est supposée
```

<sup>7</sup>Le lecteur intéressé pourra chercher l'algorithme Knuth-Morris-Pratt dans "Data Structure Techniques" par Thomas A. Standish. L'algorithme Boyer-Moore est une autre routine de recherche dans une chaîne très rapide, bien que quelque peu plus complexe.

```
; contenir la chaîne qu'INDEX recherche.
```

```
;
```

```
; En sortie:
```

```
;
```

```
; AX- Contient l'offset dans la chaîne source où la chaîne
```

```
; test a été trouvée.
```

```
INDEX      proc near
            push si
            push di
            push bx
            push cx
            pushf                ;Sauve le flag de direction.
            cld

            mov  al, es:[di]      ;Obtient longueur de chaîne test.
            cmp  al, [si]        ;Voit si elle est plus longue que
            ja   NotThere        ; la longueur de chaîne source.
```

```
; Calcule l'index du dernier caractère avec lequel nous avons besoin
; de comparer la chaîne test dans la chaîne source.
```

```
            mov  al, es:[di]      ;Longueur de la chaîne test.
            mov  cl, al           ;Sauve pour plus tard.
            mov  ch, 0
            sub  al, [si]        ;Longueur de la chaîne source.
            mov  bl, al           ;nb de fois à répéter la boucle.
            inc  di               ;Saute l'octet de longueur.
            xor  ax, ax           ;Initialise index à zéro.
CmpLoop:    inc  ax               ;Augmente index de un.
            inc  si               ;Passe au car suivant dans source.
            push si               ;Sauve les pointeurs de chaîne et la
            push di               ; longueur de la chaîne test.
            push cx
            rep  cmpsb            ;Compare les chaînes.
            pop  cx               ;Restaure les pointeurs de chaîne
            pop  di               ; et longueur.
            pop  si
            je   Foundindex      ;Si nous avons trouvé la sous-chaîne.
            dec  bl
            jnz  CmpLoop         ;Essaie le suivant dans chaîne source.
```

```
; Si nous tombons ici, la chaîne test n'apparaît pas au sein de la
; chaîne source.
```

```
NotThere:   xor  ax, ax          ;Renvoie INDEX = 0
```

```
; Si la sous-chaîne a été trouvée dans la boucle ci-dessus, enlève les
; résidus laissés sur la pile
```

```
FoundIndex: popf
            pop  cx
            pop  bx
            pop  di
```

```

                                pop    si
                                ret
INDEX                          endp

```

---

### 15.3.3 Repeat

La fonction de chaîne **Repeat** attend trois paramètres - l'adresse d'une chaîne, une longueur et un caractère. Elle construit une chaîne de la longueur indiquée contenant la copie fois "longueur" du caractère indiqué. Par exemple, Repeat(STR, 5, '\*') stocke la chaîne "\*\*\*\*\*" dans la variable de chaîne **STR**. C'est une fonction de chaîne très facile à écrire, grâce à l'instruction **stosb** :

```

; REPEAT- Construit une chaîne de longueur CX où chaque élément
; est initialisé au caractère passé dans AL.
;
; En entrée:
;
; ES:DI- Pointe sur la chaîne à construire.
; CX- Contient la longueur de la chaîne.
; AL- Contient le caractère avec lequel chaque element de
; la chaîne sera initialisé.

REPEAT    proc near
            push di
            push ax
            push cx
            pushf                ;Sauve le flag de direction.
            cld
            mov  es:[di], cl      ;Sauve la longueur de la chaîne.
            mov  ch, 0            ;Par précaution.
            inc  di               ;Débute la chaîne à l'octet suivant.
rep        stosb
            popf
            pop  cx
            pop  ax
            pop  di
            ret
REPEAT    endp

```

---

### 15.3.4 Insert

La fonction de chaîne **Insert** insère une chaîne dans une autre. Elle attend trois paramètres, une chaîne source, une chaîne destination et un index. **Insert** insère la chaîne source dans la chaîne destination en commençant à l'offset indiqué par le paramètre index. Les HLLs appellent habituellement la procédure **Insert** comme suit :

```

source := ' there';
dest   := 'Hello world';
INSERT(source, dest, 6);

```

L'appel à **Insert** ci-dessus changerait la source pour qu'elle contienne la chaîne "Hello there world". Elle fait ceci en insérant la chaîne " there" avant le sixième caractère de "Hello world".

La procédure d'insertion utilise l'algorithme suivant :

**Insert(Src, DEST, index) ;**

- 1) Déplacer les caractères de l'emplacement **dest+index** jusqu'à la fin de la chaîne de destination de **length (Src)** bytes vers le haut dans la mémoire.

- 2) Copier les caractères de la chaîne **Src** à l'emplacement **dest+index**.
- 3) Ajuster la longueur de la chaîne de destination de sorte que ce soit la somme des longueurs de destination et de source.

Le code suivant met en application cet algorithme :

```
; INSERT- Insère une chaîne dans une autre.
;
; En entrée:
;
; DS:SI Pointe sur la chaîne source à insérer
;
; ES:DI Pointe sur la chaîne destination dans laquelle la chaîne
; source sera insérée.
;
; DX Contient l'offset dans la chaîne de destination où la chaîne
; source sera insérée.
;
;
; Tous les registres sont préservés.
;
; Condition d'erreur -;
; Si la longueur de la chaîne nouvellement créée est plus grande que
; 255, l'opération d'insertion ne sera pas effectuée et le flag carry
; sera renvoyé à un.
;
; Si l'index est plus grand que la longueur de la chaîne de
; destination, alors la chaîne source sera ajoutée à la fin de la
; chaîne de destination.
```

```
INSERT    proc near
           push si
           push di
           push dx
           push cx
           push bx
           push ax
           cld                                ;On suppose qu'il n'y a pas d'erreur.
           pushf
           mov  dh, 0                          ;Par précaution.
```

```
; D'abord, voyons si la nouvelle chaîne sera trop longue.
```

```
mov  ch, 0
mov  ah, ch
mov  bh, ch
mov  al, es:[di]    ;AX = longueur de chaîne dest.
mov  cl, [si]       ;CX = longueur de chaîne source.
mov  bl, al         ;BX = longueur de nouvelle chaîne.
add  bl, cl
jc   TooLong        ;Arrête si trop longue.
mov  es:[di], bl     ;Met à jour la longueur.
```

```
; Voit si la valeur d'index value est trop grande:
```



```

        cmp    dl, al
        jbe    IndexIsOK
        mov    dl, al
IndexIsOK:

; Maintenant, fait de la place pour la chaîne que nous allons
; insérer.

        push  si                ;Sauve pour plus tard.
        push  cx
        mov   si, di            ;Fait pointer SI sur la fin de la
        add   si, ax            ; chaîne de destination en cours.
        add   di, bx            ;Pointe DI sur la fin de la nouvelle.
        std
rep      movsb                  ;Fait place pour la nouvelle chaîne.

; Maintenant, copie la chaîne source dans l'espace dégagé.

        pop   cx
        pop   si
        add   si, cx            ;Pointe à la fin de la source source.
rep      movsb
        jmp   INSERTDone

TooLong:  popf
        stc
        pushf
INSERTDone: popf
        pop   ax
        pop   bx
        pop   cx
        pop   dx
        pop   di
        pop   si
        ret
INSERT   endp

```

---

### 15.3.5 Delete

La fonction de chaîne **Delete** enlève des caractères d'une chaîne. Elle attend trois paramètres - l'adresse d'une chaîne, un index dans cette chaîne et le nombre des caractères à enlever de cette chaîne. Un appel de HLL à **Delete** prend habituellement la forme :

```
Delete(Str, index, length);
```

Par exemple,

```
Str := 'Hello there world';
Delete(str, 7, 6);
```

Cet appel à **Delete** donnera "Hello world" comme contenu à **Str**. L'algorithme pour l'opération **Delete** est le suivant :

- 1) Soustraire la valeur du paramètre longueur de la longueur de la chaîne de destination et mettre à jour la longueur de la chaîne de destination avec cette nouvelle valeur.
- 2) Copier tous caractères suivant la sous-chaîne supprimée par-dessus la sous-chaîne supprimée.

Il y a deux erreurs qui peuvent se produire en utilisant la procédure **Delete**. La valeur d'index pourrait être zéro ou plus grande que la taille de la chaîne indiquée. Dans ce cas, la procédure **Delete** ne devrait faire rien à la chaîne. Si la somme de l'index et du paramètres de longueur est plus grande que la longueur de la chaîne, alors la procédure d'effacement devrait supprimer tous les caractères à la fin de la chaîne. Le code suivant met en application la procédure **Delete**:

```
; DELETE - enlève une sous-chaîne d'une chaîne.
;
; En entrée:
;
; DS:SI Pointe sur la chaîne source.
; DX Index dans la chaîne du début de la sous-chaîne
; à supprimer.
; CX Longueur de la sous-chaîne à supprimer.
;
; Conditions d'erreur -:
; Si DX est plus grand que la longueur de la chaîne, alors
; l'opération est annulée.
;
; Si DX+CX est plus grand que la longueur de la chaîne, DELETE ne
; supprime que les caractères depuis DX jusqu'à la fin de la chaîne.
```

```
DELETE    proc near
           push es
           push si
           push di
           push ax
           push cx
           push dx
           pushf                                ;Sauve le flag de direction.
           mov  ax, ds                          ;Chaînes source et destination
           mov  es, ax                          ; sont les mêmes.
           mov  ah, 0
           mov  dh, ah                          ;Par précaution.
           mov  ch, ah

; Voit si une condition d'erreur existe.

           mov  al, [si]                        ;Obtient la longueur de la chaîne
           cmp  dl, al                          ;L'index est-il trop grand?
           ja   TooBig
           mov  al, dl                          ;Voyons si INDEX+LENGTH
           add  al, cl                          ;est trop grand
           jc   Truncate
           cmp  al, [si]
           jbe  LengthIsOK

; Si la sous-chaîne est trop grande, la tronquer pour qu'elle rentre.

Truncate:  mov  cl, [si]                        ;Calcule la longueur maximum
           sub  cl, dl
           inc  cl

; Calcule la longueur de la nouvelle chaîne.
```

```

LengthIsOK: mov al, [si]
             sub  al, cl
             mov  [si], al

```

; OK, maintenant, supprimer la sous-chaîne spécifiée.

```

             add  si, dx           ;Calcule l'adresse de la sous-chaîne
             mov  di, si           ; à supprimer, et l'adresse du
             add  di, cx           ; premier caractère la suivant.
             cld
rep movsb    ;Supprime la chaîne.
TooBig:      popf
             pop  dx
             pop  cx
             pop  ax
             pop  di
             pop  si
             pop  es
             ret
DELETE      endp

```

---

### 15.3.6 Concaténation

L'opération de concaténation prend deux chaînes et ajoute l'une à la fin de l'autre. Par exemple, **Concat**('Hello ', 'world') produit la chaîne "Hello, world". Certains langages de niveau élevé traitent la concaténation comme appel à une fonction, d'autres comme appel à une procédure. Puisqu'en assembleur tout est un appel à une procédure de toute façon, nous adopterons la syntaxe procédurale. Notre procédure **Concat** prendra la forme suivante :

```
Concat(source1, source2, dest);
```

Cette procédure copiera **source1** dans **dest**, puis concatènera **source2** à la fin de **dest**. **Concat** suit :

```

; Concat- Copie la chaîne pointée par SI dans la chaîne
; pointée par DI et ensuite concatène la chaîne
; pointée par BX dans la chaîne destination.
;
; En entrée-;
; DS:SI- Pointe sur la première chaîne source
; DS:BX- Pointe sur la seconde chaîne source
; ES:DI- Pointe sur la chaîne destination.
;
; Conditions d'erreur -
;
; La somme des longueurs des deux chaînes est plus grande que 255.
; Dans ce cas, la seconde chaîne sera tronquée afin que
; la chaîne entière fasse moins que 256 caractères de longueur.

```

```

CONCAT      proc near
             push si
             push di
             push cx
             push ax
             pushf

```

; Copie la première chaîne dans la chaîne destination:

```
                mov     al, [si]
                mov     cl, al
                mov     ch, 0
                mov     ah, ch
                add     al, [bx]           ;Calcule la somme des longueurs
                adc     ah, 0             ; des chaînes.
                cmp     ax, 256
                jb      SetNewLength
                mov     ah, [si]         ;Sauve longueur de la chaîne.
                mov     al, 255         ;Fixe longueur de chaîne à 255.
SetNewLength:   mov     es:[di], al      ;Sauve nouvelle longueur chaîne.
                inc     di              ;Saute les bytes de longueur.
                inc     si
rep             movsb                    ;Copie source1 dans chaîne dest.
```

; Si la somme des deux chaînes est trop grande, la seconde chaîne  
; doit être tronquée.

```
                mov     cl, [bx]         ;Obtient longueur seconde chaîne.
                cmp     ax, 256
                jb      LengthsAreOK
                mov     cl, ah           ;Calcule longueur tronquée.
                neg     cl               ;CL := 256-Length(Str1).

LengthsSontOK:  lea     si, 1[bx]        ;Pointe sur seconde chaîne et
                                           ; saute longueur de chaîne.
                cld
                rep     movsb           ;Exécute la concaténation.

                popf
                pop     ax
                pop     cx
                pop     di
                pop     si
                ret
CONCAT         endp
```

---

## 15.4 Fonctions de Chaîne dans la Bibliothèque Standard de l'UCR

La bibliothèque standard de l'UCR pour les programmeurs en assembleur 80x86 fournit un ensemble très riche de fonctions de chaîne que vous pouvez utiliser. Ces routines, pour la plupart, sont tout à fait semblables aux fonctions de chaîne fournies dans la Bibliothèque Standard du C. En tant que telles, ces fonctions supportent les chaînes terminées par zéro plutôt que les chaînes à en tête de longueur supportées par les fonctions dans les sections précédentes.

Puisqu'il y a un si grand nombre de routines de chaîne différentes dans la StdLib de l'UCR et que les sources pour toutes ces routines sont dans le domaine public (et sont présentes sur le CD-ROM d'accompagnement de ce texte), les sections suivantes ne discuteront pas l'implémentation de chaque routine. Au lieu de cela, les sections suivantes se concentreront sur la façon dont employer ces routines de bibliothèque.

La bibliothèque de l'UCR fournit souvent plusieurs variantes de la même routine. Généralement un suffixe "l", "m" ou "ml" apparaît à la fin du nom de ces variantes de routines. Le suffixe "l" représente la "constante littérale". Les

routines avec le suffixe "l" (ou "ml") exigent deux opérandes chaîne. Le premier est généralement pointé par **es:di** et le deuxième suit immédiatement l'appel dans le flux de code.

La plupart des routines de chaîne de StdLib opèrent sur la chaîne spécifiée (ou une des chaînes si la fonction a deux opérandes). Le suffixe "m" (ou "ml") demande à la fonction de chaîne d'allouer un stockage sur le tas (heap) en employant **malloc**, d'où le suffixe "m", pour la nouvelle chaîne et d'y stocker le résultat modifié au lieu de changer la (ou les) chaîne(s) source. Ces routines renvoient toujours un pointeur sur la chaîne nouvellement créée dans les registres **es:di**. En cas d'une erreur d'allocation de mémoire (mémoire insuffisante), ces routines avec le suffixe "m" ou "ml" renvoient le drapeau de retenue à un. Ils le renvoient à zéro si l'opération était réussie.

---

#### 15.4.1 StrBDel, StrBDelm

Ces deux routines suppriment les espaces initiaux d'une chaîne. **StrBDel** enlève tous les espaces initiaux de la chaîne pointée par **es:di**. Il modifie en fait la chaîne source. **StrBDelm** fait une copie de la chaîne sur le tas en enlevant les espaces initiaux. S'il n'y a aucun espace initial, alors les routines **StrBDel** renvoient la chaîne originale sans modification. Notez que ces routines affectent seulement les espaces initiaux (ceux qui apparaissent au début de la chaîne). Ils n'enlèvent pas les espaces finaux ni les espaces au milieu de la chaîne. Voir **Strtrim** si vous voulez enlever les espaces finaux. Exemples :

```
MyString    byte    "      Hello there, this is my string",0
MyStrPtr    dword    MyString
.
.
.
        les        di, MyStrPtr
        strbdelm                    ;Crée une nouvelle chaîne sans espaces
                                   ; initiaux,
        jc      error                ; pointeur sur la chaîne dans ES:DI au
                                   ; retour.
        puts                     ;Imprime la chaîne pointée par ES:DI.
        free                      ;Libère stockage alloué par strbdelm.
        .
        .
        .
```

; Notez que "MyString" contaient toujours les espaces initiaux.  
; L'appel suivant à printf imprimera la chaîne avec ces  
; espaces initiaux. "strbdelm" ci-dessus n'a pas changé MyString.

```
        printf
byte      "MyString = '%s'\n",0
dword    MyString
.
.
.
        les        di, MyStrPtr
        strbdel
```

; Maintenant, nous avons vraiment enlevé les espaces initiaux de  
; "MyString"

```
        printf
byte      "MyString = '%s'\n",0
dword    MyString
```

.  
.  
.

Sortie de ce fragment de code :

```
Hello there, this is my string
MyString = '      Hello there, this is my string'
MyString = 'Hello there, this is my string'
```

---

#### 15.4.2 StrCat, StrCatl, StrCatm, StrCatml

Les routines **strcat**(xx) exécutent une concaténation de chaîne. En entrée, **es:di** pointe sur la première chaîne, et pour **strcat/strcatm dx:si** pointe sur la deuxième chaîne. Pour **strcatl** et **strcatlm**, la deuxième chaîne suit l'appel dans le flux de code. Ces routines créent une nouvelle chaîne en apposant la deuxième chaîne à la fin de la première. Dans le cas de **strcat** et **strcatl**, la deuxième chaîne est directement apposée à la fin de la première chaîne (**es:di**) dans la mémoire. Vous devez vous assurer qu'il y a suffisamment de mémoire à la fin de la première chaîne pour contenir les caractères ajoutés. **Strcatm** et **strcatml** créent une nouvelle chaîne sur le tas (en employant **malloc**) contenant le résultat concaténé. Exemples :

```
String1    byte "Hello ",0
           byte 16 dup (0)           ;Place pour la concatenation.
```

```
String2    byte "world",0
```

```
; La macro suivante charge ES:DI avec l'adresse de l'opérande
; spécifié.
```

```
lesi      macro      operand
           mov        di, seg operand
           mov        es, di
           mov        di, offset operand
           endm
```

```
; La macro suivante charge DX:SI avec l'adresse de l'opérande
; spécifié.
```

```
ldxi      macro      operand
           mov        dx, seg operand
           mov        si, offset operand
           endm
.
.
.
lesi      String1
ldxi      String2
strcatm                                ;Crée "Hello world"
jc        error                       ;Si mémoire insuffisante.
print
byte      "strcatm: ",0
puts                                ;Imprime "Hello world"
putcrl
free                                ;Libère stockage de la chaîne.
.
```

```

.
.
lesi      String1      ;Crée la chaîne
strcatml  ; "Hello there"
jc        error        ;Si mémoire insuffisante.
byte      "there",0
print
byte      "strcatml: ",0
puts                      ;Imprime "Hello there"
putc
free
.
.
.
lesi      String1
ldxi      String2
strcat                      ;Crée "Hello world"
printf
byte      "strcat: %s\n",0
.
.
.

```

; Note: puisque strcat ci-dessus a réellement modifié String1,  
; l'appel suivant à strcatl ajoute "there" à la fin  
; de la chaîne "Hello world".

```

lesi      String1
strcatl
byte      "there",0
printf
byte      "strcatl: %s\n",0
.
.
.

```

Le code ci-dessu produit tla sortie suivante :

```

strcatm: Hello world
strcatml: Hello there
strcat: Hello world
strcatl: Hello world there

```

---

### 15.4.3 StrChr

Strchr recherche la première occurrence d'un caractère unique dans une chaîne. A l'exécution, elle est très similaire à l'instruction **scasb**. Cependant, vous ne devez pas indiquer une longueur explicite pour utiliser cette fonction alors que vous devez le faire pour **scasb**.

En entrée, **es:di** pointe sur la chaîne à travers laquelle vous voulez chercher, **al** contient la valeur à rechercher. Au retour, le drapeau de retenue affiche le succès ( C=1 signifie que le caractère n'était *pas* présent dans la chaîne, C=0 signifie que le caractère était présent). Si le caractère a été trouvé dans la chaîne, **cx** contient l'index dans la chaîne où **strchr** a localisé le caractère. Notez que le premier caractère de la chaîne est à l'index zéro. Ainsi **strchr**

renverra zéro si al correspond au premier caractère de la chaîne. Si le drapeau de retenue est à un, alors la valeur dans **cx** n'a aucune signification. Exemple :

```
; Notez que la chaîne suivante a un point à l'emplacement
; "HasPeriod+24".
```

```
HasPeriod byte "This string has a period.",0
.
.
.
lesi HasPeriod          ;Voir strcat pour définition lesi
mov  al, "."            ;Recherche un point.
strchr
jnc  GotPeriod
print
byte "No period in string",cr,lf,0
jmp Done
```

```
; Si nous avons trouvé le point, afficher l'offset dans la chaîne:
```

```
GotPeriod: print
byte "Found period at offset ",0
mov  ax, cx
puti
putcrl
```

```
Done:
```

Ce fragment de code produit la sortie :

```
Found period at offset 24
```

---

#### 15.4.4 Strcmp, Strcmpl, Stricmp, Stricmpl

Ces routines comparent des chaînes en utilisant un ordre lexicographique. En entrée à **strcmp** ou **stricmp**, **es:di** pointe sur la première chaîne et **dx:si** pointe sur la deuxième chaîne. **Strcmp** compare la première chaîne à la seconde et renvoie le résultat de la comparaison dans le registre flags. **Strcmpl** fonctionne de façon similaire, sauf que la deuxième chaîne suit l'appel dans le flux de code. Les routines **stricmp** et **stricmpl** diffèrent de leurs contreparties du fait elles ignorent la casse lors de la comparaison. Là où **strcmp** renverrait "non égal" en comparant "Strcmp" à "strcmp", les routines **stricmp** (et **stricmpl**) renvoient "égal" puisque la seule différence est une majuscule au lieu d'une minuscule. Le "i" dans **stricmp** et **stricmpl** signifie "ignore la casse". Exemples :

```
String1 byte "Hello world", 0
String2 byte "hello world", 0
String3 byte "Hello there", 0
.
.
.
lesi String1          ;Voir strcat pour lesi definition
ldxi String2          ;Voir strcat pour ldxi definition
strcmp
jae  IsGtrEq1
printf
byte "%s is less than %s\n",0
dword String1, String2
jmp Try1
```



```

IsGtrEq1: printf
          byte      "%s is greater or equal to %s\n",0
          dword     String1, String2

Try1:     lesi       String2
          strcmpl
          byte      "hi world!",0
          jne       NotEq1
          printf
          byte      "Hmmm..., %s is equal to 'hi world!'\n",0
          dword     String2
          jmp       Tryi

NotEq1:   printf
          byte      "%s is not equal to 'hi world!'\n",0
          dword     String2

Tryi:     lesi       String1
          ldxi       String2
          stricmp
          jne       BadCmp
          printf
          byte      "Ignoring case, %s equals %s\n",0
          dword     String1, String2
          jmp       Tryil

BadCmp:   printf
          byte      "Wow, stricmp doesn't work! %s <> %s\n",0
          dword     String1, String2

Tryil:    lesi       String2
          strcmpl
          byte      "hELLO THERE",0
          jne       BadCmp2
          print
          byte      "Stricmpl worked",cr,lf,0
          jmp       Done

BadCmp2:  print
          byte      "Stricmp did not work",cr,lf,0

Done:

```

---

#### 15.4.5 Strcpy, Strcpyl, Strdup, Strdupl

Les routines **strcpy** et **strdup** copient une chaîne dans une autre. Il n'y a aucune routine **strcpym** ou **strcpyml**. **Strdup** et **strdupl** correspondent à ces opérations. La bibliothèque standard de l'UCR emploie les noms **strdup** et **strdupl** plutôt que **strcpym** et **strcpyml** de manière à employer les mêmes noms que la bibliothèque standard du C.

**Strcpy** copie la chaîne pointée par **es:di** dans les emplacements de mémoire commençant à l'adresse dans **dx:si**. Il n'y a aucune vérification des erreurs ; vous devez vous assurer qu'il y a suffisamment d'espace libre à

l'emplacement **dx:si** avant d'appeler **strcpy**. **Strcpy** retourne avec **es:di** pointant sur la chaîne destination (c'est-à-dire, la valeur originale de **dx:si**). **Strcpyl** fonctionne de façon similaire, sauf que la chaîne source suit l'appel.

**Strdup** reproduit la chaîne sur laquelle pointe **es:di** et renvoie un pointeur sur la nouvelle chaîne dans le tas. **Strdupl** fonctionne de façon similaire, sauf que la chaîne suit l'appel. Comme d'habitude, le drapeau de retenue est à un s'il y a une erreur d'attribution de mémoire en utilisant **strdup** ou **strdupl**. Exemples :

```
String1    byte        "Copy this string ",0
String2    byte        32 dup (0)
String3    byte        32 dup (0)
StrVar1    dword       0
StrVar2    dword       0
.
.
.
lesi       String1      ;Voir strcat pour lesi definition
ldxi       String2      ;Voir strcat pour ldxi definition
strcpy

ldxi       String3
strcpyl
byte       "This string, too!",0

lesi       String1
strdup
jc         error        ;Si mém insuffisante.
mov        word ptr StrVar1, di    ;Sauve ptr sur la
mov        word ptr StrVar1+2, es  ; chaîne.

strdupl
jc         error
byte       "Also, this string ",0
mov        word ptr StrVar2, di
mov        word ptr StrVar2+2, es

printf
byte       "strcpy: %s\n"
byte       "strcpyl: %s\n"
byte       "strdup: %^s\n"
byte       "strdupl: %^s\n",0
dword     String2, String3, StrVar1, StrVar2
```

---

#### 15.4.6 Strdel, Strdelm

**Strdel** et **strdelm** suppriment des caractères d'une chaîne. **Strdel** supprime les caractères indiqués dans la chaîne, **strdelm** crée une nouvelle copie de la chaîne source sans les caractères indiqués. En entrée, **es:di** pointe sur la chaîne à traiter, **cx** contient l'index dans la chaîne où la suppression doit commencer, et **ax** contient le nombre de caractères à supprimer dans la chaîne. Au retour, **es:di** pointe sur la nouvelle chaîne (qui est sur le tas si vous appelez **strdelm**). Pour **strdelm** seulement, si le drapeau de retenue est à un au retour, il y avait une erreur d'attribution de mémoire. Comme avec toutes les routines de chaîne de la StdLib de l'UCR, les valeurs d'index pour la chaîne sont basées sur zéro. C'est-à-dire, zéro est l'index du premier caractère dans la chaîne source. Exemple :

```
String1    byte        "Hello there, how are you?",0
.
```

```

.
.
lesi      String1      ;Voir strcat pour lesi definition
mov       cx, 5         ;Part à la position 5 (" there")
mov       ax, 6         ;Supprime six caractères.
strdelm   ;Crée une nouvelle chaîne.
jc        error        ;Si mémoire insuffisante.
print
byte      "New chaîne:",0
puts
putcr

lesi      String1
mov       ax, 11
mov       cx, 13
strdel
printf
byte      "Modified string: %s\n",0
dword    String1

```

Ce code imprime ce qui suit :

New string: Hello, how are you?

Modified string: Hello there

---

#### 15.4.7 Strins, Strinsl, Strins, Strinsm, Strinsml

Les fonctions **strins**(xx) insèrent une chaîne dans une autre. Pour chacune des quatre routines **es:di** pointe sur la chaîne source dans laquelle vous voulez insérer une autre chaîne. **Cx** contient le point d'insertion (0..longueur de la chaîne source). Pour **strins** et **strinsm**, **dx:si** pointe sur la chaîne que vous souhaitez insérer. Pour **strinsl** et **strinsml**, la chaîne à insérer apparaît comme constante littérale dans le flux de code. **Strins** et **strinsl** insèrent la deuxième chaîne directement dans la chaîne pointée par **es:di**. **Strinsm** et **strinsml** font une copie de la chaîne source et insèrent la deuxième chaîne dans cette copie. Elles renvoient un pointeur sur la nouvelle chaîne dans **es:di**. S'il y a une erreur d'attribution de mémoire, alors **strinsm/strinsml** met à un le drapeau de retenue au retour. Pour **strins** et **strinsl**, la première chaîne doit avoir un stockage alloué suffisant pour contenir la nouvelle chaîne. Exemples :

```

InsertInMe   byte      "Insert >< Here",0
              byte      16 dup (0)
InsertStr    byte      "insert this",0
StrPtr1      dword     0
StrPtr2      dword     0
.
.
.
lesi         InsertInMe ;Voir strcat pour lesi definition
ldxi         InsertStr  ;Voir strcat pour ldxi definition
mov          cx, 8      ;Insère avant "<"
strinsm
mov          word ptr StrPtr1, di
mov          word ptr StrPtr1+2, es

lesi         InsertInMe
mov          cx, 8
strinsml

```

```

byte    "insert that",0
mov     word ptr StrPtr2, di
mov     word ptr StrPtr2+2, es

lesi    InsertInMe
mov     cx, 8
strinsl
byte    " ",0          ;Deux espaces

lesi    InsertInMe
ldxi    InsertStr
mov     cx, 9          ;Avant le 1er espace ci-dessus
strins

printf
byte    "First string: %s\n"
byte    "Second string: %s\n"
byte    "Third string: %s\n",0
dword   StrPtr1, StrPtr2, InsertInMe

```

Notez que les opérations **strins** et **strinsl** ci-dessus inserent toutes deux des chaînes dans la même chaîne destination. La sortie du code ci-dessus est.

```

First string: Insert >insert this< here
Second string: Insert >insert that< here
Third string: Insert > insert this < here

```

---

#### 15.4.8 Strlen

**Strlen** calcule la longueur de la chaîne pointée par **es:di**. Elle renvoie le nombre de caractères jusqu'au byte de terminaison zéro, mais sans celui-ci. Elle renvoie cette longueur dans le registre **cx**. Exemple :

```

GetLen   byte "This string is 33 characters long",0
.
.
.
lesi GetLen          ;Voir strcat pour lesi definition
strlen
print
byte "The string is ",0
mov ax, cx           ;Puti demande la longueur dans AX!
puti
print
byte " characters long",cr,lf,0

```

---

#### 15.4.9 Strlwr, Strlwrn, Strupr, Struprn

**Strlwr** et **Strlwrn** convertissent tous les caractères majuscules dans une chaîne en minuscules. **Strupr** et **Struprn** convertissent toutes les minuscules dans une chaîne en majuscules. Ces routines n'affectent aucun autre caractère présent dans la chaîne. Pour chacune des quatre routines, **es:di** pointe sur la chaîne source à convertir. **Strlwr** et **strupr** modifient les caractères directement dans cette chaîne. **Strlwrn** et **struprn** font une copie de la chaîne sur le tas et puis convertissent les caractères dans la nouvelle chaîne. Ils renvoient également un pointeur sur cette nouvelle chaîne dans **es:di**. Comme d'habitude pour des routines de la StdLib de l'UCR, **strlwrn** et **struprn** renvoient le drapeau de retenue à un s'il y a une erreur d'attribution de mémoire. Exemples :

```

String1    byte    "This string has lower case.",0
String2    byte    "THIS STRING has Upper Case.",0
StrPtr1    dword   0
StrPtr2    dword   0
.
.
.
lesi       String1    ;Voir strcat pour lesi definition
struprm    ;Convertit minuscules en majuscules.
jc         error
mov        word ptr StrPtr1, di
mov        word ptr StrPtr1+2, es
lesi       String2
strlwr     ;Convertit majuscules en minuscules.
jc         error
mov        word ptr StrPtr2, di
mov        word ptr StrPtr2+2, es

lesi       String1
strlwr     ;Convertit en minuscules, sur place.

lesi       String2
strupr    ;Convertit en majuscules, sur place.

printf
byte       "strupr: %^s\n"
byte       "strlwr: %^s\n"
byte       "strlwr: %s\n"
byte       "strupr: %s\n",0
dword     StrPtr1, StrPtr2, String1, String2

```

Le fragment de code ci-dessus imprime ce qui suit :

```

strupr: THIS STRING HAS LOWER CASE
strlwr: this string has upper case
strlwr: this string has lower case
strupr: THIS STRING HAS UPPER CASE

```

---

#### 15.4.10 Strrev, Strrevm

Ces deux routines inversent les caractères dans une chaîne. Par exemple, si vous passez à **strrev** la chaîne "ABCDEF", elle convertira cette chaîne en "FEDCBA". Comme vous vous en doutez maintenant, la routine **strrev** inverse la chaîne dont vous passez l'adresse dans **es:di** ; **strrevm** fait d'abord une copie de la chaîne sur le tas et inverse ces caractères laissant la chaîne originale inchangée. Naturellement **strrevm** renverra le drapeau de retenue à un s'il y avait une erreur d'attribution de mémoire. Exemple :

```

Palindrome    byte    "radar",0
NotPaldrn     byte    "x + y - z",0
StrPtr1       dword   0
.
.
.
lesi          Palindrome ;Voir strcat pour lesi definition
strrevm
jc            error

```

```

mov      word ptr StrPtr1, di
mov      word ptr StrPtr1+2, es

lesi     NotPaldrm
strrev

printf
byte "First string: %s\n"
byte "Second string: %s\n", 0
dword StrPtr1, NotPaldrm

```

Le code ci-dessus produit la sortie suivante :

```

First string: radar
Second string: z - y + x

```

---

#### 15.4.11 Strset, Strsetm

**Strset** et **strsetm** recopient un caractère unique dans une chaîne. Leur comportement, cependant, n'est pas tout à fait identique. En particulier, alors que **strsetm** est tout à fait semblable à la fonction *repeat* (voir "Repeat" à la section 15.3.3), **strset** ne l'est pas. Les deux routines attendent une valeur unique de caractère dans le registre **al**. Elles recopieront ce caractère dans toute la chaîne indiquée. **Strsetm** exige également un compte dans le registre **cx**. Elle crée une chaîne sur le tas se composant de **cx** caractères et renvoie un pointeur sur cette chaîne dans **es:di** (à supposer qu'il n'y a aucune erreur d'attribution de mémoire). **Strset**, d'autre part, s'attend à ce que vous lui passiez l'adresse d'une chaîne existante dans **es:di**. Elle remplacera chaque caractère dans cette chaîne avec le caractère dans **al**. Notez que vous n'indiquez pas une longueur en utilisant la fonction **strset**, **strset** emploie la longueur de la chaîne existante. Exemple :

```

String1    byte    "Hello there", 0
.
.
.
lesi       String1          ;Voir strcat pour lesi definition
mov        al, '*'
strset

mov        cx, 8
mov        al, '#'
strsetm

print
byte       "String2: ", 0
puts
printf
byte       "\nString1: %s\n", 0
dword     String1

```

Le code ci-dessus produit la sortie :

```

String2: #####
String1: *****

```

---

#### 15.4.12 Strspan, Strspanl, Strcspan, Strcspanl

Ces quatre routines recherchent dans une chaîne un caractère qui est soit dans un jeu de caractères spécifié (**strspan**, **strspanl**) soit qui n'est pas un membre d'un jeu de caractères (**strcspan**, **strcspanl**). Ces routines apparaissent dans la bibliothèque standard de l'UCR seulement en raison de leur présence dans la Bibliothèque

Standard du C. Vous devriez rarement utiliser ces routines. La bibliothèque standard de l'UCR inclut d'autres routines pour les manipulations des jeux de caractères et les opérations de correspondance de caractères. Néanmoins, ces routines sont parfois utiles occasionnellement et valent la peine d'être mentionnées ici.

Ces routines s'attendent à ce que vous leur passiez les adresses de deux chaînes : une chaîne source et une chaîne jeu de caractères. Elles attendent l'adresse de la chaîne source dans **es:di**. **Strspan** et **strcspan** demandent l'adresse de la chaîne jeu de caractères dans **dx:si** ; la chaîne jeu de caractères suit l'appel avec le **strspanl** et le **strcspanl**. Au retour, **cx** contient un index dans la chaîne, défini comme suit :

**strspan, strspanl** : Index du premier caractère dans la source trouvé dans le jeu de caractère,  
**strcspan, strcspanl** : Index de premier caractère dans la source non trouvé dans le jeu de caractères.

Si tous les caractères sont dans l'ensemble (ou ne sont pas dans l'ensemble) alors **cx** contient l'index dans la chaîne du byte de terminaison zéro.

Exemple :

Source	byte	"ABCDEFGH 0123456",0	
Set1	byte	"ABCDEFGH IJKLMNOPQRSTUVWXYZ",0	
Set2	byte	"0123456789",0	
Index1	word	?	
Index2	word	?	
Index3	word	?	
Index4	word	?	
.			
.			
.			
lesl	Source		;Voir strcat pour lesl definition
ldxi	Set1		;Voir strcat pour ldxi definition
strspan			;Cherche le 1er car ALPHA.
mov	Index1, cx		;Index du 1er car alphabétique.
lesl	Source		
lesl	Set2		
strspan			;Cherche le 1 <sup>er</sup> car numérique.
mov	Index2, cx		
lesl	Source		
strcspanl			
byte	"ABCDEFGH IJKLMNOPQRSTUVWXYZ",0		
mov	Index3, cx		
lesl	Set2		
strcspanl			
byte	"0123456789",0		
mov	Index4, cx		
printf			
byte	"First alpha char in Source is at offset %d\n"		
byte	"First numeric char is at offset %d\n"		
byte	"First non-alpha in Source is at offset %d\n"		
byte	"First non-numeric in Set2 is at offset %d\n",0		
dword	Index1, Index2, Index3, Index4		

Ce code sort ce qui suit :

First alpha char in Source is at offset 0

```

First numeric char is at offset 8
First non-alpha in Source is at offset 7
First non-numeric in Set2 is at offset 10

```

---

#### 15.4.13                      **Strstr, Strstrl**

**Strstr** recherche la première occurrence d'une chaîne dans une autre. **es:di** contient l'adresse de la chaîne dans laquelle vous voulez rechercher une deuxième chaîne. **dx:si** contient l'adresse de la deuxième chaîne pour la routine **strstr** ; pour **strstrl** la seconde chaîne à rechercher suit immédiatement l'appel dans le flux de code.

Au retour de **strstr** ou **strstrl**, le drapeau de retenue sera à un si la deuxième chaîne n'est pas présente dans la chaîne source. Si le drapeau de retenue est à zéro, alors la deuxième chaîne est présente dans la chaîne source et **cx** contiendra l'index (basé sur zéro) où la deuxième chaîne a été trouvée. Exemple :

```

SourceStr  byte      "Search for 'this' in this string",0
SearchStr  byte      "this",0
.
.
.
lesi       Source      ;Voir strcat pour lesi definition
ldxi       Set1        ;Voir strcat pour ldxi definition
strstr
jc         NotPresent
print
byte       "Found string at offset ",0
mov        ax, cx      ;Demande offset dans AX pour puti
puti
putc

lesi       SourceStr
strstrl
byte       "for",0
jc         NotPresent
print
byte       "Found 'for' at offset ",0
mov        ax, cx
puti
putc

NotPresent:

```

Le code ci-dessus imprime ce qui suit :

```

Found string at offset 12
Found 'for' at offset 7

```

---

#### 15.4.14                      **Strtrim, Strtrimm**

Ces deux routines sont tout à fait semblables à **strbdel** et **strbdelm**. Au lieu d'enlever les espaces initiaux, cependant, elles enlèvent tous les espaces en fin de chaîne. **Strtrim** enlève tous les espaces terminaux directement sur la chaîne indiquée dans la mémoire. **Strtrimm** copie d'abord la chaîne source et puis enlève tous les espaces terminaux de la copie. Les deux routines s'attendent à ce que vous passiez l'adresse de la chaîne source dans **es:di**. **Strtrimm** renvoie un pointeur sur la nouvelle chaîne (s'il a pu l'allouer) dans **es:di**. Elle renvoie également la retenue à un ou zéro pour dénoter une éventuelle erreur. Exemple :

```

String1    byte      "Spaces at the end      ",0
String2    byte      "      Spaces on both sides      ",0

```



```

StrPtr1    dword    0
StrPtr2    dword    0
.
.
.

```

```

; TrimSpcs enlève les espaces des deux extrémités d'une chaîne.
; Notez qu'il est un peu plus efficace de réliser d'abord
; strbdel, puis strtrim. Cette routine crée la nouvelle
; chaîne sur le tas et renvoie un pointeur sur cette chaîne
; dans ES:DI.

```

```

TrimSpcs    proc
             strbdelm
             jc      BadAlloc          ;Retourne simplement si erreur.
             strtrim
             clc
BadAlloc:    ret
TrimSpcs    endp
.
.
.
             lesi      String1          ;Voir strcat pour ldx definition
             strtrimm
             jc      error
             mov       word ptr StrPtr1, di
             mov       word ptr StrPtr1+2, es
             lesi      String2
             call      TrimSpcs
             jc      error
             mov       word ptr StrPtr2, di
             mov       word ptr StrPtr2+2, es
             printf
             byte      "First string: '%s'\n"
             byte      "Second string: '%s'\n", 0
             dword     StrPtr1, StrPtr2

```

Ce fragment de code imprime ce qui suit :

```

First string: 'Spaces at the end'
Second string: 'Spaces on both sides'

```

---

#### 15.4.15 Autres Routines de Chaîne de la Bibliothèque Standard de l'UCR

En plus des routines "**strxxx**" énumérées dans cette section, il y a beaucoup de routines supplémentaires de chaîne disponibles dans la Bibliothèque Standard de l'UCR. Des routines pour convertir des types numériques (nombre entier, hexa, réel, etc...) en chaîne ou vice versa, recherche de correspondance et routines de jeu de caractères, et beaucoup d'autres utilitaires de conversion et de chaîne. Les routines décrites en ce chapitre sont celles dont les définitions apparaissent dans le dossier d'en-tête "strings.a" et sont spécifiquement orientées pour les traitements de chaîne génériques. Pour plus de détails sur les autres routines de chaîne, consultez la section référence de la bibliothèque standard de l'UCR dans les annexes.

---

#### 15.5 Routines de Jeux de Caractères de la Bibliothèque Standard de l'UCR

La bibliothèque standard de l'UCR fournit une collection étendue de routines de jeux de caractère. Ces routines vous permettent de créer des jeux, réinitialiser des jeux (les convertir à l'ensemble vide), ajouter ou enlever un ou

plusieurs éléments, tester l'appartenance au jeu de caractères, copier des jeux, en calculer l'union, l'intersection, ou la différence et extraire des éléments d'un jeu. Bien que prévu pour manipuler des jeux de caractères, vous pouvez employer les routines de jeu de caractères de la StdLib pour manipuler tout ensemble de 256 éléments ou moins.

La première chose inhabituelle à noter au sujet des jeux de caractères de la StdLib est leur format de stockage. Un tableau de 256 bits consomme normalement 32 bytes consécutifs. Pour des raisons de performance, le format de jeux de la bibliothèque standard de l'UCR compacte huit jeux séparés dans 272 bytes (256 bytes pour les huit jeux plus 16 bytes d'en-tête). Pour déclarer des variables jeu de caractères dans votre segment de données il est préférable que vous utilisiez la macro **set**. Cette macro prend la forme :

```
set    SetName1, SetName2, ..., SetName8
```

**SetName1..SetName8** représentent les noms de huit variables jeu de caractère maximum. Vous pouvez avoir moins de huit noms dans la zone d'opérande, mais cela gaspillera un quelques bits dans le tableau de jeux.

La routine **CreateSets** fournit un autre mécanisme pour créer des variables jeu de caractères. À la différence de la macro **set**, que vous utilisez pour créer des variables jeu de caractères dans votre segment de données, la routine **CreateSets** alloue le stockage pour huit jeux maximum dynamiquement à l'exécution. Elle renvoie un pointeur sur la première variable jeu de caractères dans **es:di**. Les sept jeux restants suivent aux emplacements **es:di+1**, **es:di+2...**, **es:di+7**. Un programme classique qui alloue des variables jeu de caractères dynamiquement pourrait employer le code suivant :

```
Set0      dword    ?
Set1      dword    ?
Set2      dword    ?
Set3      dword    ?
Set4      dword    ?
Set5      dword    ?
Set6      dword    ?
Set7      dword    ?
.
.
.
CreateSets
mov  word ptr Set0+2, es
mov  word ptr Set1+2, es
mov  word ptr Set2+2, es
mov  word ptr Set3+2, es
mov  word ptr Set4+2, es
mov  word ptr Set5+2, es
mov  word ptr Set6+2, es
mov  word ptr Set7+2, es

mov  word ptr Set0, di
inc  di
mov  word ptr Set1, di
inc  di
mov  word ptr Set2, di
inc  di
mov  word ptr Set3, di
inc  di
mov  word ptr Set4, di
inc  di
mov  word ptr Set5, di
inc  di
mov  word ptr Set6, di
inc  di
```

```

mov    word ptr Set7, di
inc    di

```

Ce segment de code crée huit jeux différents sur le tas, tous vides, et stocke des pointeurs sur eux dans les variables pointeur appropriées.

Le fichier SHELL.ASM fournit une ligne de code en commentaire dans le segment de données qui inclut le fichier STDSETS.A. Ce fichier inclut les définitions des bits pour huit jeux de caractères communément utilisés. Ce sont **alpha** (lettres majuscules et minuscules), **lower** (lettres minuscules), **upper** (lettres majuscules), **digits** ("0".."9"), **xdigits** ("0".."9", "A".."F", et "a".."f"), **alphanum** (lettres majuscules et minuscules plus les chiffres), **whitespace** (espace, tabulation, retour de chariot et retour à la ligne), et **delimiters** (whitespace plus virgule, point-virgule, moins que, plus grand que, et barre verticale). Si vous voulez employer ces jeux de caractères standard dans votre programme, vous devrez enlever le point-virgule au début de l'instruction include dans le fichier SHELL.ASM.

La bibliothèque standard de l'UCR fournit 16 routines de jeu de caractères : **CreateSets**, **EmptySet**, **RangeSet**, **AddStr**, **AddStrL**, **RmvStr**, **RmvStrL**, **AddChar**, **RmvChar**, **Member**, **CopySet**, **SetUnion**, **SetIntersect**, **SetDifference**, **NextItem** et **RmvItem**. Toutes ces routines à part **CreateSets** exigent un pointeur sur une variable de jeu de caractères dans les registres **es:di**. Des routines spécifiques peuvent exiger d'autres paramètres supplémentaires.

La routine **EmptySet** met à zéro tous les bits dans un jeu de caractères produisant l'ensemble vide. Cette routine exige l'adresse de la variable jeu de caractères dans **es:di**. L'exemple suivant initialise le jeu pointé par **Set1** :

```

les    di, Set1
EmptySet

```

**RangeSet** remplit une variable jeu de caractères pointée par **es:di** à partir d'une plage de valeurs. Le registre **al** contient la limite inférieure de la plage des éléments, **ah** contient la limite supérieure. Notez qu'**al** doit être inférieur ou égal à **ah**. L'exemple suivant construit l'ensemble avec tous les caractères de contrôle (codes ASCII un à 31, le caractère nul [code ASCII zéro] n'est pas permis dans les jeux) :

```

les    di, CtrlCharSet      ;Ptr sur jeu de car de ctrl.
mov    al, 1
mov    ah, 31
RangeSet

```

**AddStr** et **AddStrL** ajoutent tous les caractères d'une chaîne terminée par zéro dans un jeu de caractères. Pour **AddStr**, la paire de registres **dx:si** pointe sur la chaîne terminée par zéro. Pour **AddStrL**, la chaîne terminée par zéro suit l'appel à **AddStrL** dans le flux de code. Ces routines placent chaque caractère de la chaîne indiquée dans le jeu. Les exemples suivants ajoutent les chiffres et quelques caractères spéciaux dans le jeu **FPDigits**:

```

Digits      byte    "0123456789", 0
             set      FPDigitsSet
FPDigits     dword    FPDigitsSet
             .
             .
             .
             ldxi     Digits          ;Met DX:SI à adr Digits.
             les      di, FPDigits
             AddStr
             .
             .
             .
             les      di, FPDigits
             AddStrL
             byte     "Ee.+-", 0

```

**RmvStr** et **RmvStrl** enlèvent des caractères d'un jeu. Vous fournissez les caractères dans une chaîne terminée par zéro. Pour **RmvStr**, **dx:si** pointe sur la chaîne des caractères à enlever de la chaîne. Pour **RmvStrl**, la chaîne terminée par zéro suit l'appel. L'exemple suivant emploie **RmvStrl** pour enlever les symboles spéciaux de **FPDigits** ci-dessus :

```
les      di, FPDigits
RmvStrl
byte     "Ee.+-",0
```

Les routines **AddChar** et **RmvChar** vous permettent ajouter ou enlever des caractères individuels. Comme d'habitude, **es:di** pointe sur le jeu ; le registre **al** contient le caractère que vous souhaitez ajouter au jeu ou enlever du jeu. L'exemple suivant ajoute une espace à l'ensemble **FPDigits** et enlève le caractère "," (si présent) :

```
les  di, FPDigits
mov  al, ' '
AddChar
.
.
.
les  di, FPDigits
mov  al, ','
RmvChar
```

La fonction **Member** vérifie si un caractère est dans un jeu. En entrée, **es:di** doit pointer sur le jeu et **al** doit contenir le caractère à vérifier. En sortie, le drapeau zéro est à un si le caractère est membre du jeu, le drapeau zéro sera à zéro si le caractère n'est pas dans l'ensemble. L'exemple suivant lit des caractères au clavier jusqu'à ce que l'utilisateur appuie sur une touche qui n'est pas un caractère de **whitespace** :

```
SkipWS:  get                ;Lit car utilisateur dans AL.
         lesi WhiteSpace    ;Adresse du jeu WS dans es:di.
         member
         je    SkipWS
```

Les routines **CopySet**, **SetUnion**, **SetIntersect** et **SetDifference** opèrent toutes sur deux jeux de caractères. Le registre **es:di** pointe sur le jeu de caractères destination, la paire de registre **dx:si** pointe sur un jeu de caractères source. **CopySet** copie les bits du jeu source dans le jeu destination, remplaçant les bits originaux dans le jeu destination. **SetUnion** calcule l'union des deux jeux et stocke le résultat dans le jeu destination. **SetIntersect** calcule l'intersection des jeux et stocke le résultat dans le jeu destination. Enfin, la routine **SetDifference** calcule **DestSet** : = **DestSet** - **SrcSet**.

Les routines **NextItem** et **RmvItem** vous permettent d'extraire des éléments à partir d'un jeu. **NextItem** renvoie en **al** le code ASCII du premier caractère qu'il trouve dans un jeu. **RmvItem** fait la même chose sauf qu'il enlève en plus le caractère du jeu. Ces routines renvoient zéro en **al** si le jeu est vide (les jeux de StdLib ne peuvent pas contenir le caractère NULL). Vous pouvez employer la routine **RmvItem** pour construire un itérateur rudimentaire pour un jeu de caractères.

Les routines du jeu de caractères de la bibliothèque standard de l'UCR sont très puissantes. Avec elles, vous pouvez facilement manipuler des données de chaîne de caractères, particulièrement pour rechercher des agencements particuliers dans une chaîne. Nous étudierons ces routines de nouveau quand nous étudierons la recherche de modèle plus tard dans ce texte (voir "Recherche de modèle" au Chapitre 16).

---

## 15.6 Utilisation des Instructions de Chaîne avec d'autres Types de Données

Les instructions de chaîne fonctionnent avec d'autres types de données que les chaînes de caractères. Vous pouvez employer des instructions de chaîne pour copier des tableaux entiers d'une variable à une autre, initialiser de grandes structures de données à une valeur unique ou comparer des structures de données entières pour en vérifier l'égalité ou l'inégalité. Chaque fois que vous manipulez des structures de données contenant plusieurs bytes, vous avez l'occasion d'employer les instructions de chaîne.

---

### 15.6.1 Chaînes d'Entiers en Multi-précision

L'instruction **cmps** est utile pour comparer de (très) grandes valeurs de nombre entier. Contrairement aux chaînes de caractères, nous ne pouvons pas comparer des nombres entiers avec **cmps** du byte de L.O. jusqu'au byte de H.O.. Au lieu de cela, nous devons les comparer en descendant du byte de H.O. jusqu'au byte de L.O.. Le code suivant compare deux nombres entiers de 12 bytes :

```
        lea    di, integer1+10
        lea    si, integer2+10
        mov    cx, 6
        std
repe    cmpsw
```

Après l'exécution de l'instruction **cmpsw**, les drapeaux contiendront le résultat de la comparaison.

Vous pouvez facilement assigner une longue chaîne de nombre entier à une autre en employant l'instruction **movs**. Rien de tordu ici, vous n'avez qu'à charger les registres **si**, **di**, et **cx** et l'exécuter. Vous devrez faire les autres opérations, y compris des opérations arithmétiques et logiques, en employant les méthodes de précision étendue décrites dans le chapitre sur les opérations arithmétiques.

---

### 15.6.2 Gérer des Tableaux et des Enregistrements Entiers

Les seules opérations qui s'appliquent, en général, à toutes les structures tableau et enregistrement sont l'assignation et la comparaison (pour égalité/inégalité seulement). Vous pouvez employer les instructions **movs** et **cmps** pour ces opérations.

Des opérations telles que l'addition scalaire, la transposition, etc., peuvent être facilement synthétisées en utilisant les instructions **lods** et **stos**. Le code suivant montre comment vous pouvez facilement ajouter la valeur 20 à chaque élément du tableau de nombres entiers A :

```
        lea    si, A
        mov    di, si
        mov    cx, SizeOfA
        cld
AddLoop: lodsw
        add    ax, 20
        stosw
        loop  AddLoop
```

Vous pouvez produire d'autres opérations de manière similaire.

---

## 15.7 Programmes Exemples

Dans cette section il y a trois programmes exemples. Le premier recherche dans un fichier une chaîne particulière et affiche le numéro de ligne de toutes les lignes contenant cette chaîne. Ce programme démontre l'utilisation de la fonction **strsr** (entre autres). Le deuxième programme est un programme de démonstration qui emploie plusieurs des fonctions de chaîne disponibles dans la dotation concernant les chaînes de la bibliothèque standard de l'UCR. Le troisième programme démontre comment employer l'instruction 80x86 **cmps** pour comparer des données dans deux fichiers. Ces programmes (find.asm, strdemo.asm, et fcmp.asm) sont disponibles sur le CD-ROM d'accompagnement.

---

### 15.7.1 Find.asm

```
; Find.asm
;
; Ce programme ouvre un fichier spécifié sur la ligne de commande et
; recherche une chaîne (spécifiée sur la ligne de commande).
;
```

```
; Utilisation du programme:
;
; find "string" filename
```

```
.xlist
include    stdlib.a
includelib stdlib.lib
.list
```

```
wp      textequ    <word ptr>
dseg    segment    para public 'data'
StrPtr  dword      ?
FileName dword      ?
LineCnt dword      ?
```

```
FVar    filevar    {}
```

```
InputLine byte      1024 dup (?)
dseg     ends
```

```
cseg     segment    para public 'code'
          assume     cs:cseg, ds:dseg
```

```
; ReadLn- Cette procédure lit une ligne de texte dans le fichier
; entrée et la garde dans le tableau "InputLine".
```

```
ReadLn    proc
           push  es
           push  ax
           push  di
           push  bx

           lesi  FVar                ;Lit dans notre fichier.
           mov   bx, 0                ;Index dans InputLine.
ReadLp:    fgetc                     ;Obtient car suivant du fichier.
           jc    EndRead              ;Quitte si EOF

           cmp   al, cr                ;Ignore retours chariot.
           je    ReadLp
           cmp   al, lf                ;Fin de ligne si retour de ligne.
           je    EndRead

           mov   InputLine[bx], al
           inc   bx
           jmp   ReadLp
```

```
; Si nous arrivons en fin de ligne ou à la fin du fichier,
; termine la chaîne par zéro.
```

```
EndRead:   mov   InputLine[bx], 0
           pop   bx
           pop   di
           pop   ax
           pop   es
```

```

        ret
ReadLn   endp

; Le programme principal suivant extrait la chaîne à rechercher et le
; nom du fichier de la ligne de commande, ouvre le fichier, et
; ensuite recherche la chaîne dans ce fichier.

Main     proc
        mov     ax, dseg
        mov     ds, ax
        mov     es, ax
        meminit

        argc
        cmp     cx, 2
        je      GoodArgs
        print
        byte    "Usage: find 'chaîne' filename", cr, lf, 0
        jmp     Quit

GoodArgs: mov     ax, 1                ;Obtient la chaîne à rechercher
        argv                    ; dans la ligne de commande.
        mov     wp StrPtr, di
        mov     wp StrPtr+2, es
        mov     ax, 2                ;Obtient le nom de fichier
        argv                    ; dans la ligne de commande.
        mov     wp Filename, di
        mov     wp Filename+2, es

; Ouvre le fichier entrée en lecture

        mov     ax, 0                ;Ouvre en lecture.
        mov     si, wp FileName
        mov     dx, wp FileName+2
        lesi     Fvar
        fopen
        jc      BadOpen

; OK, commence la recherche de la chaîne dans le fichier.

        mov     wp LineCnt, 0
        mov     wp LineCnt+2, 0
SearchLp: call    ReadLn
        jc      AtEOF

; Augmente le numéro de ligne de un. Notez que c'est du code 8086
; aussi nous devons utiliser la précision arithmétique étendue pour
; faire une addition 32-bits. LineCnt est une variable 32-bit car
; des fichiers ont plus de 65,536 lignes.

        add     wp LineCnt, 1
        adc     wp LineCnt+2, 0

; Recherche la chaîne spécifiée par l'utilisateur dans la ligne

```

```

; en cours.

        lesi InputLine
        mov  dx, wp StrPtr+2
        mov  si, wp StrPtr
        strstr
        jc   SearchLp          ;Saute si pas trouvé.

; Imprime un message approprié si nous avons trouvé la chaîne.

        printf
        byte "Found '%^s' at line %ld\n",0
        dword StrPtr, LineCnt
        jmp  SearchLp

; Ferme le fichier quand nous avons fini.

AtEOF:   lesi FVar
        fclose
        jmp  Quit

BadOpen:  printf
        byte "Error attempting to open %^s\n",cr,lf,0
        dword FileName

Quit:     ExitPgm              ;Macro DOS pour quitter le programme.
Main     endp

cseg      ends

sseg      segment      para stack 'stack'
stk       db           1024 dup ("stack ")
sseg      ends

zzzzzzseg segment      para public 'zzzzzz'
LastBytes db           16 dup (?)
zzzzzzseg ends
end        Main

```

---

## 15.7.2 StrDemo.asm

Ce court programme de démo montre juste manière d'employer plusieurs des routines de chaîne trouvées dans le package de chaînes standard de bibliothèque de l'UCR.

```

; StrDemo.asm- Démonstration de quelques unes des diverses routines
; de chaîne de l'UCR Standard Library

```

```

        include      stdlib.a
        includelib   stdlib.lib

dseg      segment      para public 'data'

MemAvail  word         ?
String    byte         256 dup (0)

```



```

dseg      ends

cseg      segment      para public 'code'
          assume       cs:cseg, ds:dseg

Main      proc
          mov  ax, seg dseg          ;Installe registres de segment
          mov  ds, ax
          mov  es, ax

          MemInit
          mov  MemAvail, cx
          printf
          byte "There are %x paragraphs of memory available."
          byte cr,lf,lf,0
          dword MemAvail

; Démonstration de StrTrim:

          print
          byte "Testing strtrim on 'Hello there      '",cr,lf,0
          strdupl
HelloThere1 byte "Hello there      ",0
          strtrim
          mov  al, ""
          putc
          puts
          putc
          putcr
          free

;Démonstration de StrTrimm:

          print
          byte "Testing strtrimm on 'Hello there      '",cr,lf,0
          lesi HelloThere1
          strtrimm
          mov  al, ""
          putc
          puts
          putc
          putcr
          free

; Démonstration de StrBdel

          print
          byte "Testing strbdel on '      Hello there      '",cr,lf,0
          strdupl
HelloThere3 byte "      Hello there      ",0
          strbdel
          mov  al, ""
          putc
          puts

```

```

        putc
        putcr
        free

; Démonstration de StrBdelm

        print
        byte "Testing strbdelm on ' Hello there '",cr,lf,0
        lesi HelloThere3
        strbdelm
        mov  al, ""
        putc
        puts
        putc
        putcr
        free

; Démontre StrCpyl:

        ldxi string
        strcpyl
        byte "Copy this string to the 'String' variable",0

        printf
        byte "STRING = '%s'",cr,lf,0
        dword string

; Démontre StrCatl:

        lesi String
        strcatl
        byte ". Put at end of 'String'",0

        printf
        byte "CHAÎNE = ",'"%s"',cr,lf,0
        dword String

; Démontre StrChr:

        lesi String
        mov  al, ""
        strchr

        print
        byte "StrChr: First occurrence of ", '"', '" '
        byte '" found at position ',0
        mov  ax, cx
        puti
        putcr

; Démontre StrStrl:

        lesi String
        strstrl

```

```

        byte "String",0

        print
        byte 'StrStr: First occurrence of "String" found at '
        byte 'position ',0

        mov  ax, cx
        puti
        putcr

; Démo de StrSet

        lesi String
        mov  al, '*'
        strset

        printf
        byte "Strset: '%s'",cr,lf,0
        dword String
; Démo de strlen

        lesi String
        strlen

        print
        byte "String length = ",0
        puti
        putcr

Quit:    mov  ah, 4ch
        int  21h
Main     endp

cseg     ends

sseg     segment      para stack 'stack'
stk       db          256 dup ("stack ")
sseg     ends

zzzzzzseg segment      para public 'zzzzzz'
LastBytes db          16 dup (?)
zzzzzzseg ends
end       Main

```

---

### 15.7.3 Fcmp.asm

C'est un programme de comparaison de fichier. Il démontre l'utilisation de l'instruction cmps du 80x86 (et aussi des E/S en bloc sous DOS).

```

; FCMP.ASM- Un programme de comparaison de fichiers qui démontre
;           l'utilisation des instructions de chaîne du 80x86.

```

```

.xlist
include  stdlib.a
includelib stdlib.lib

```

```

        .list

dseg      segment      para public 'data'

Name1     dword        ?           ;Ptr sur nom fichier #1
Name2     dword        ?           ;Ptr sur nom fichier #2
Handle1   word         ?           ;Handle fichier pour fichier #1
Handle2   word         ?           ;Handle fichier pour fichier #2
LineCnt   word         0           ;# de lignes dans le fichier.

Buffer1   byte         256 dup (0) ;Block de données de fichier 1
Buffer2   byte         256 dup (0) ;Block de données de fichier 2

dseg      ends

wp        equ          <word ptr>

cseg      segment      para public 'code'
          assume       cs:cseg, ds:dseg

; Error- Imprime un message DOS d'erreur selon le type d'erreur.

Error     proc          near
          cmp          ax, 2
          jne          NotFNF
          print
          byte         "File not found",0
          jmp          ErrorDone

NotFNF:   cmp          ax, 4
          jne          NotTMF
          print
          byte         "Too many open files",0
          jmp          ErrorDone

NotTMF:   cmp ax, 5
          jne          NotAD
          print
          byte         "Access denied",0
          jmp          ErrorDone

NotAD:    cmp          ax, 12
          jne          NotIA
          print
          byte         "Invalid access",0
          jmp          ErrorDone

NotIA:
ErrorDone: putcr
          ret
Error     endp

; OK, voici le programme principal. Il ouvre deux fichiers, les
; compare et signale s'il sont differents.

```

```

Main      proc
          mov     ax, seg dseg      ;Initialise registres de segment
          mov     ds, ax
          mov     es, ax
          meminit

; Routine de comparaison de fichiers. D'abord, on ouvre les deux
; fichiers source.

          argc
          cmp     cx, 2             ;Avons-nous deux noms de fichier?
          je      GotTwoNames
          print
          byte    "Usage: fcmp file1 file2",cr,lf,0
          jmp     Quit

GotTwoNames: mov     ax, 1           ;Obtient le premier nom
          argv
          mov     wp Name1, di
          mov     wp Name1+2, es

; Ouvre les fichiers avec un appel au DOS.

          mov     ax, 3d00h         ;Ouvre en lecture
          lds     dx, Name1
          int     21h
          jnc     GoodOpen1
          printf
          byte    "Error opening %^s:",0
          dword   Name1
          call    Error
          jmp     Quit

GoodOpen1: mov     dx, dseg
          mov     ds, dx
          mov     Handle1, ax

          mov     ax, 2             ;Obtient le nom du second fichier
          argv
          mov     wp Name2, di
          mov     wp Name2+2, es
          mov     ax, 3d00h         ;Ouvre en lecture
          lds     dx, Name2
          int     21h
          jnc     GoodOpen2
          printf
          byte    "Error opening %^s:",0
          dword   Name2
          call    Error
          jmp     Quit

GoodOpen2: mov     dx, dseg
          mov     ds, dx

```

```

        mov         Handle2, ax

; Lit les données des fichiers en utilisant les E/S par bloc
; et les compare.

        mov         LineCnt, 1
CmpLoop: mov         bx, Handle1             ;Lit 256 bytes du
        mov         cx, 256                 ; premier fichier dans
        lea         dx, Buffer1             ; Buffer1.
        mov         ah, 3fh
        int         21h
        jc          FileError
        cmp         ax, 256                 ;Quitte si EOF.
        jne         EndOfFile
        mov         bx, Handle2             ;Lit 256 bytes du
        mov         cx, 256                 ; second fichier dans
        lea         dx, Buffer2             ; Buffer2
        mov         ah, 3fh
        int         21h
        jc          FileError
        cmp         ax, 256                 ;Si on n'a pas lu 256 bytes,
        jne         BadLen                 ; les fichiers sont differents.

; OK, nous venons de lire 256 bytes de chaque fichier, comparer les
; buffers pour voir si les données sont les mêmes dans les deux
; fichiers.

        mov         ax, dseg
        mov         ds, ax
        mov         es, ax
        mov         cx, 256
        lea         di, Buffer1
        lea         si, Buffer2
        cld
repe     cmpsb
        jne         BadCmp
        jmp         CmpLoop

FileError: print
        byte        "Error reading files: ",0
        call        Error
        jmp         Quit

BadLen:   print
        byte        "File lengths were different",cr,lf,0

BadCmp:   print
        byte        7,"Files were not equal",cr,lf,0
        mov         ax, 4c01h ;Exit with error.
        int         21h

; Si nous avons atteint la fin du premier fichier, comparer les bytes
; éventuels restant dans ce premier fichier avec les bytes restants
; dans le second fichier.

```

```

EndOfFile: push      ax                      ;Sauve longueur finale.
            mov      bx, Handle2
            mov      cx, 256
            lea      dx, Buffer2
            mov      ah, 3fh
            int      21h
            jc       BadCmp

            pop      bx                      ;Récupère la longueur de file1.
            cmp      ax, bx                  ;Voit si file2 correspond.
            jne      BadLen

            mov      cx, ax                  ;Compare les bytes restants
            mov      ax, dseg                ; ici.
            mov      ds, ax
            mov      es, ax
            lea      di, Buffer2
            lea      si, Buffer1
            repe     cmpsb
            jne      BadCmp

Quit:       mov      ax, 4c00h              ;Met le code de Sortie à OK.
            int      21h

Main        endp
cseg        ends

```

; Alloue un volume raisonnable d'espace pour la pile (2k).

```

sseg        segment      para stack 'stack'
stk         byte         256 dup ("stack ")
sseg        ends

zzzzzzseg   segment      para public 'zzzzzz'
LastBytes   byte         16 dup (?)
zzzzzzseg   ends
            end           Main

```

---

## 15.8 Exercices de Laboratoire

Ces exercices utilisent les fichiers Ex15\_1.asm, Ex15\_2.asm, Ex15\_3.asm et Ex15\_4.asm qui se trouvent sur le CD-ROM d'accompagnement. Dans cet ensemble d'exercices de laboratoire vous mesurerez l'exécution des instructions **movs** du 80x86 et (si tout va bien) les différences mineures d'exécution entre les opérations sur les chaînes à en tête de longueur et les opérations sur celles terminées par zéro.

---

### 15.8.1 Exercice N°1 sur les Performances de MOVS

Les instructions **movsb**, **movsw** et **movsd** fonctionnent à différentes vitesses, même lorsqu'elles déplacent le même nombre de bytes. En général, l'instruction **movsw** est deux fois plus rapide que **movsb** pour déplacer le même nombre de bytes. De même, **movsd** s'exécute environ deux fois plus rapidement que **movsw** (et environ quatre fois plus rapidement que **movsb**) pour déplacer le même nombre de bytes. Ex15\_1.asm est un court programme qui démontre ce fait. Ce programme se compose de trois sections qui copient 2048 bytes d'un tampon à un autre 100.000 fois. Les trois sections répètent cette opération en utilisant les instructions **movsb**, **movsw** et **movsd**. Exécutez ce programme et chronométrez chaque phase. **Pour votre rapport de laboratoire** : présentez les

chronométrages sur votre machine. Assurez-vous de préciser le type de processeur et la fréquence d'horloge dans votre rapport de laboratoire. Expliquez pourquoi les chronométrages sont différents entre les trois phases de ce programme. Expliquez la difficulté qu'il y a à employer l'instruction **movsd** (au lieu de **movsw** ou **movsb**) dans des programmes sur des processeurs 80386 ou postérieurs. Pourquoi n'est-ce pas un remplacement général pour **movsb**, par exemple ? Comment pouvez-vous résoudre ce problème ?

```
; EX15_1.asm
;
; Ce programme démontre l'utilisation correcte des instructions de
; chaîne du 80x86.
```

```

        .386
        option      segment:use16
        include     stdlib.a
        includelib  stdlib.lib

dseg     segment      para public 'data'

Buffer1  byte         2048 dup (0)
Buffer2  byte         2048 dup (0)

dseg     ends

cseg     segment      para public 'code'
        assume       cs:cseg, ds:dseg

Main     proc
        mov          ax, dseg
        mov          ds, ax
        mov          es, ax
        meminit

; Démo des instructions movsb, movsw et movsd

        print
        byte "Le code suivant déplace un bloc de 2048 bytes "
        byte "100,000 fois en mémoire.",cr,lf
        byte "La première phase utilise l'instruction movsb; "
        byte "la seconde phase ",cr,lf
        byte "utilise l'instruction movsw; "
        byte "la troisième phase utilise",cr,lf
        byte "l'instruction movsd.",cr,lf,lf,lf
        byte "Appuyez sur une touche pour la phase une :",0

        getc
        putcr

        mov     edx, 100000

movsbLp:  lea     si, Buffer1
        lea     di, Buffer2
        cld
        mov     cx, 2048
        rep     movsb
```



```

        dec    edx
        jnz    movsbLp

        print
        byte  cr,lf
        byte  "Phase une terminée ",cr,lf,lf
        byte  "Appuyez sur une touche pour la phase deux :",0

       getc
        putcr

        mov    edx, 100000

movswLp:  lea    si, Buffer1
        lea    di, Buffer2
        cld
        mov    cx, 1024
        rep    movsw
        dec    edx
        jnz    movswLp

        print
        byte  cr,lf
        byte  "Phase deux terminée",cr,lf,lf
        byte  "Appuyez sur une touche pour la phase trois :",0

       getc
        putcr

        mov    edx, 100000

movsdLp:  lea    si, Buffer1
        lea    di, Buffer2
        cld
        mov    cx, 512
        rep    movsd
        dec    edx
        jnz    movsdLp

Quit:    ExitPgm                ;Macro DOS pour quitter le programme.
Main     endp

cseg     ends

sseg     segment    para stack 'stack'
stk      db         1024 dup ("stack ")
sseg     ends

zzzzzzseg segment para public 'zzzzzz'
LastBytes db         16 dup (?)
zzzzzzseg ends
end      Main

```

---

## 15.8.2 Exercice N°2 sur les Performances de MOVS

Dans cet exercice vous chronométrerez de nouveau l'ordinateur qui déplace des blocs de 2.048 bytes. Comme Ex15\_1.asm dans l'exercice précédent, Ex15\_2.asm contient trois phases ; la première phase déplace des données en utilisant l'instruction **movsb** ; la deuxième phase déplace les données en employant les instructions **lods** et **stos** ; la troisième phase emploie une boucle avec des instructions **mov** uniques. Exécutez ce programme et chronométrez les trois phases. **Pour votre rapport de laboratoire** : incluez les chronométrages et une description de votre machine (unité centrale de traitement, fréquence d'horloge, etc.). Discutez les chronométrages et expliquez les résultats (consultez l'annexe D si besoin est).

```
; EX15_2.asm
;
; Ce programme compare les performances de l'instruction MOVS avec une
; opération manuelle de mouvement en bloc. Il compare aussi MOVS
; avec une boucle LODS/STOS

        .386
        option      segment:use16

        include     stdlib.a
        includelib  stdlib.lib

dseg     segment      para public 'data'

Buffer1  byte         2048 dup (0)
Buffer2  byte         2048 dup (0)

dseg     ends

cseg     segment      para public 'code'
        assume       cs:cseg, ds:dseg

Main     proc
        mov          ax, dseg
        mov          ds, ax
        mov          es, ax
        meminit

; Version MOVSB :

        print
        byte "Le code suivant déplace un bloc de 2,048 bytes "
        byte "en mémoire 100,000 fois.",cr,lf
        byte "La première phase le fait avec l'instruction "
        byte "movsb; la seconde",cr,lf
        byte "phase le fait avec les instructions lods/stos ; "
        byte "la troisième phase le fait",cr,lf
        byte "avec une boucle contenant des "
        byte "instructions. MOV ",cr,lf,lf,lf
        byte "Appuyez sur une touche pour la phase une:",0

        getc
        putcr
```

```

        mov     edx, 100000

movsbLp:  lea     si, Buffer1
        lea     di, Buffer2
        cld
        mov     cx, 2048
        rep     movsb
        dec     edx
        jnz     movsbLp
        print
        byte    cr,lf
        byte    "Phase une terminée",cr,lf,lf
        byte    "Appuyez sur une touche pour la phase deux :",0

        getc
        putcr

        mov     edx, 100000

LodsStosLp: lea si, Buffer1
        lea     di, Buffer2
        cld
        mov     cx, 2048
lodsstoslp2: lodsb
        stosb
        loop    LodsStosLp2
        dec     edx
        jnz     LodsStosLp

        print
        byte    cr,lf
        byte    "Phase deux terminée",cr,lf,lf
        byte    "Appuyez sur une touche pour la phase trois :",0

        getc
        putcr

        mov     edx, 100000

MovLp:    lea     si, Buffer1
        lea     di, Buffer2
        cld
        mov     cx, 2048
MovLp2:   mov     al, ds:[si]
        mov     es:[di], al
        inc     si
        inc     di
        loop    MovLp2
        dec     edx
        jnz     MovLp

Quit:     ExitPgm           ;Macro DOS pour quitter le programme.
Main      endp

```

```

cseg      ends

sseg      segment    para stack 'stack'
stk       db         1024 dup ("stack ")
sseg      ends

zzzzzzseg segment    para public 'zzzzzz'
LastBytes db         16 dup (?)
zzzzzzseg ends
end        Main

```

---

### 15.8.3 Exercice sur les Performances de la Mémoire

Dans les deux exercices précédents, les programmes ont accédé à un maximum de 4K de données. Puisque la plupart des caches modernes sur la puce de la CPU font au moins cette taille, la majeure partie de l'activité a eu lieu directement dans la CPU (qui est très rapide). L'exercice suivant est une légère modification qui déplace les données du tableau de façon à détruire les performances du cache. Exécutez ce programme et chronométrez les résultats. **Pour votre rapport de laboratoire** : basé sur ce que vous avez appris sur le mécanisme du cache du 80x86 au chapitre trois, expliquez les différences de performances.

```

; EX15_3.asm
;
; Ce programme compare les performances de l'instruction MOVS avec
; une opération de mouvement manuelle en bloc. Il compare aussi
; MOVS avec une boucle LODS/STOS. Cette version le fait de manière
; à effacer le cache embarqué dans la CPU

```

```

        .386
        option      segment:usel6
        include     stdlib.a
        includelib  stdlib.lib

dseg     segment    para public 'data'

Buffer1  byte      16384 dup (0)
Buffer2  byte      16384 dup (0)

dseg     ends

cseg     segment    para public 'code'
        assume     cs:cseg, ds:dseg

Main     proc
        mov  ax, dseg
        mov  ds, ax
        mov  es, ax
        meminit

; version MOVSB :

        print
        byte "Le code suivant déplace un bloc de 16,384 bytes "
        byte "en mémoire 12,500 fois.",cr,lf
        byte "La première phase le fait avec l'instruction "

```

```

byte " movsb; la seconde",cr,lf
byte "phase le fait avec les instructions lods/stos; "
byte "la troisième phase le fait",cr,lf
byte "avec une boucle comportant des instructions MOV."
byte cr,lf,lf,lf
byte "Appuyez sur une touche pour la phase une :",0

getc
putcr

mov    edx, 12500

movsbLp: lea    si, Buffer1
         lea    di, Buffer2
         cld
         mov    cx, 16384
rep     movsb
         dec    edx
         jnz    movsbLp

print
byte cr,lf
byte "Phase une terminée",cr,lf,lf
byte "Appuyez sur une touche pour la phase deux :",0

getc
putcr

mov    edx, 12500

LodsStosLp: lea si, Buffer1
           lea di, Buffer2
           cld
           mov cx, 16384
lodsstoslp2: lodsb
            stosb
            loop LodsStosLp2
            dec  edx
            jnz  LodsStosLp

print
byte cr,lf
byte "Phase deux terminée",cr,lf,lf
byte "Appuyez sur une touche pour la phase trois :",0

getc
putcr

mov    edx, 12500

MovLp:  lea    si, Buffer1
         lea    di, Buffer2
         cld
         mov    cx, 16384

```

```

MovLp2:    mov     al, ds:[si]
            mov     es:[di], al
            inc     si
            inc     di
            loop    MovLp2
            dec     edx
            jnz     MovLp

Quit:      ExitPgm                ;Macro DOS pour quitter le programme.
Main      endp
cseg      ends

sseg      segment    para stack 'stack'
stk       db         1024 dup ("stack ")
sseg      ends

zzzzzzseg segment    para public 'zzzzzz'
LastBytes db         16 dup (?)
zzzzzzseg ends
            end      Main

```

---

#### 15.8.4 Les Performances des Chaînes à En-tête de Longueur et celles Terminées par Zéro

Le programme suivant (Ex15\_4.asm sur le CD-ROM d'accompagnement) exécute deux millions d'opérations de chaîne. Pendant la première phase de l'exécution, ce code exécute une séquence d'opérations de chaîne à en-tête de longueur 1.000.000 fois. Lors de la deuxième phase, il exécute un ensemble d'opérations identiques sur des chaînes terminées par zéro. Mesurez le temps d'exécution de chaque phase. **Pour votre rapport de laboratoire :** rapportez les différences de temps d'exécution et commentez l'efficacité respective des chaînes à en-tête de longueur par rapport aux chaînes terminées par zéro. Notez que les performances relatives de ces séquences dépendront du processeur que vous employez. A partir de ce que vous avez appris au chapitre trois et des chronométrages de cycle de l'annexe D, expliquez quelques raisons possibles des différences d'exécution relatives entre ces séquences avec différents processeurs.

```

; EX15_4.asm
;
; Ce programme compare les performances des chaînes à en-tête de
; longueur par rapport aux chaînes terminées par zéro terminated avec
; quelques exemples.
;
; Note: ces routines supposent toutes que les chaînes sont dans
; le segment de données et que ds and es pointent déjà tous les
; deux dans le segment de données.

        .386
        option      segment:use16

        include     stdlib.a
        includelib  stdlib.lib

dseg     segment     para public 'data'

LStr1    byte        17,"This is a string."
LResult  byte        256 dup (?)

```

```

ZStr1      byte      "This is a string",0
ZResult    byte      256 dup (?)

dseg       ends

cseg       segment   para public 'code'
            assume    cs:cseg, ds:dseg

; LStrCpy: Copie une chaîne à en-tête de longueur pointée par SI
;          dans la chaîne à en-tête de longueur pointée par DI.

LStrCpy    proc
            push si
            push di
            push cx
            cld
            mov  cl, [si]                ;Obtient longueur de la chaîne.
            mov  ch, 0
            inc  cx                      ;Inclut le byte de longueur.
rep        movsb

            pop  cx
            pop  di
            pop  si
            ret
LStrCpy    endp

; LStrCat- Concatène la chaîne pointée par SI à la fin de la chaîne
;          pointée par DI en utilisant des chaînes à en-tête de
;          longueur.

LStrCat    proc
            push si
            push di
            push cx

            cld

; Calcule la longueur finale de la chaîne concaténée

            mov  cl, [di]                ;Obtient longueur d'origine.
            mov  ch, [si]                ;Obtient 2ème longueur.
            add  [di], ch                ;Calcule nouvelle longueur.

; Déplace SI au premier byte au delà de la fin de la première chaîne.

            mov  ch, 0                    ;Etend par zéro longueur d'orig.
            add  di, cx                    ;Saute à la fin de chaîne.
            inc  di                        ;Saute le byte de longueur.

; Concatène la seconde chaîne (SI) à la fin de la première chaîne (DI)

rep        movsb                        ;Copie 2ème à la fin d'origine.

```

```

        pop    cx
        pop    di
        pop    si
        ret
LStrCat    endp

; LStrCmp- Comparaison de chaînes utilisant deux chaînes à en-tête de
;          longueur.
;          SI pointe sur la première chaîne, DI pointe sur la
;          chaîne à laquelle on la compare.

LStrCmp    proc
        push    si
        push    di
        push    cx

        cld

; En comparant les chaînes, il nous faut comparer les chaînes
; jusqu'à la longueur de la chaîne la plus courte. Le code suivant
; calcule la longueur minimum des deux chaînes.

        mov     cl, [si]           ;Obtient le minimum des 2 longueurs
        mov     ch, [di]
        cmp     cl, ch
        jnb     HasMin
        mov     cl, ch
HasMin:    mov     ch, 0

        repe    cmpsb              ;Compare les deux chaînes.
        je      CmpLen
        pop     cx
        pop     di
        pop     si
        ret

; Si les chaînes sont égales jusqu'à la fin de la chaîne la plus
; courte, il nous faut comparer leurs longueurs

CmpLen:    pop     cx
        pop     di
        pop     si

        mov     cl, [si]
        cmp     cl, [di]
        ret
LStrCmp    endp

; ZStrCpy- Copie la chaîne terminée par zéro pointée par SI
;          dans la chaîne terminée par zéro pointée par DI.

ZStrCpy    proc
        push    si

```



```

        push di
        push ax

ZSCLp:   mov  al, [si]
        inc  si
        mov  [di], al
        inc  di
        cmp  al, 0
        jne  ZSCLp

        pop  ax
        pop  di
        pop  si
        ret
ZStrCpy  endp

; ZStrCat- Concatène la chaîne pointée par SI à la fin de la
;          chaîne pointée par DI en utilisant des chaînes
;          terminées par zéro.

ZStrCat  proc
        push si
        push di
        push cx
        push ax

        cld

; Trouve la fin de la chaîne destination :

        mov  cx, 0FFFFh
        mov  al, 0                      ;Recherche le byte zéro.
        repne scasb

; Copie la chaîne source à la fin de la chaîne destination :

ZcatLp:   mov  al, [si]
        inc  si
        mov  [di], al
        inc  di
        cmp  al, 0
        jne  ZcatLp

        pop  ax
        pop  cx
        pop  di
        pop  si
        ret
ZStrCat  endp

; ZStrCmp- Compare deux chaînes terminées par zéro.
;          C'est vraiment plus facile que la comparaison
;          à en-tête de longueur.

```

```

ZStrCmp    proc
            push cx
            push si
            push di

; Compare les deux chaînes jusqu'à ce qu'elles ne soient plus égales
; ou jusqu'à ce que nous rencontrions le byte zéro. Elles sont égales
; si nous rencontrons un byte zéro après avoir comparé les deux
; caractères des chaînes.

ZCmpLp:     mov  al, [si]
            inc  si
            cmp  al, [di]
            jne  ZCmpDone
            inc  di
            cmp  al, 0
            jne  ZCmpLp

ZCmpDone:   pop  di
            pop  si
            pop  cx
            ret

ZStrCmp    endp

Main       proc
            mov  ax, dseg
            mov  ds, ax
            mov  es, ax
            meminit

            print
            byte "Le code suivant fait 1,000,000 opérations de"
            byte " chaîne en utilisant",cr,lf
            byte "des chaînes à en-tête de longueur. Mesurez le "
            byte "temps que ce code",cr,lf
            byte "met à s'exécuter.",cr,lf,lf
            byte "Appuyez sur une touche pour commencer:",0

            getc
            putcr

            mov  edx, 1000000
LStrCpyLp:  lea  si, LStr1
            lea  di, LResult
            call LStrCpy
            call LStrCat
            call LStrCat
            call LStrCat
            call LStrCpy
            call LStrCmp
            call LStrCat
            call LStrCmp
            dec  edx
            jne  LStrCpyLp

```

```

    print
    byte "Le code suivant fait 1,000,000 opérations de"
    byte " chaîne en utilisant",cr,lf
    byte "des chaînes terminatées par zéro. Mesurez le "
    byte "temps que ce code",cr,lf
    byte "met à s'exécuter.",cr,lf,lf
    byte "Appuyez sur une touche pour commencer:",0

    getc
    putcr

    mov     edx, 1000000
ZStrCpyLp: lea     si, ZStr1
            lea     di, ZResult
            call    ZStrCpy
            call    ZStrCat
            call    ZStrCat
            call    ZStrCat
            call    ZStrCpy
            call    ZStrCmp
            call    ZStrCat
            call    ZStrCmp
            dec     edx
            jne     ZStrCpyLp

Quit:      ExitPgm                ;Macro DOS pour quitter le programme.
Main      endp

cseg       ends

sseg       segment      para stack 'stack'
stk        db           1024 dup ("stack ")
sseg       ends

zzzzzzseg  segment      para public 'zzzzzz'
LastBytes  db           16 dup (?)
zzzzzzseg  ends
end        Main

```

---

## 15.9 Projets de Programmation

- 1) Ecrivez une fonction **Substr** qui extrait une sous-chaîne d'une chaîne terminée par zéro. Passez un pointeur à la chaîne dans **ds:si**, un pointeur à la chaîne destination dans **es:di**, la position de départ dans la chaîne dans **ax**, et la longueur de la sous-chaîne dans **cx**. Suivez toutes les règles données dans la section 15.3.1 au sujet des conditions en mode dégradé.
- 2) Ecrivez un *itérateur* de mots (voir "Les Itérateurs" à la section 12.5) auquel vous passez une chaîne (par référence, sur la pile). Chaque itération each de la boucle foreach correspondante devrait extraire un mot de cette chaîne, allouer avec malloc un stockage suffisant pour cette chaîne sur le tas, copier ce mot (sous-chaîne) à l'emplacement alloué, et renvoyer un pointeur sur le mot. Écrivez un programme principal qui appelle l'itérateur avec différentes chaînes pour le tester.
- 3) Modifiez le programme *find.asm* (voir "Find.asm" à la section 15.7.1) de sorte qu'il recherche la chaîne désirée dans plusieurs fichiers en utilisant des noms de fichier ambigus (c.-à-d., avec des caractères joker).

Voir "Find First File" à la section 13.3.8.8 pour des détails au sujet du traitement des noms de fichier qui contiennent des caractères joker. Vous devrez écrire une boucle qui traite tous les noms de fichier correspondants et exécute le code noyau de `find.asm` sur chaque nom de fichier qui correspond au nom de fichier ambigu fourni par l'utilisateur.

- 4) Ecrivez une routine **strncpy** qui se comporte comme **strcpy** à la différence qu'elle copie un maximum de `n` caractères (byte zéro de terminaison compris). Passez l'adresse de la chaîne source dans **es:di**, l'adresse de la chaîne destination dans **dx:si**, et la longueur maximum dans **cx**.
- 5) L'instruction **movsb** peut ne pas fonctionner correctement si les blocs source et destination se recouvrent (voir "L'Instruction MOVS" à la section 15.1.4). Écrivez une procédure **bcopy** à laquelle vous passez l'adresse d'un bloc source, l'adresse d'un bloc destination et une longueur, qui copiera correctement les données même si les blocs source et destination se recouvrent. Faites-le par vérification du recouvrement des blocs et ajustement des pointeurs source, destination et du drapeau de direction si besoin est.
- 6) Comme vous l'avez découvert dans les expériences de laboratoire, l'instruction **movsd** peut déplacer un bloc de données beaucoup plus rapidement que **movsb** ou **movsw**. Malheureusement, elle peut seulement déplacer un bloc qui contient un nombre de bytes multiple de quatre. Écrivez une routine **fastcopy** qui emploie l'instruction **movsd** pour copier tout sauf les derniers un à trois bytes d'un bloc source dans un bloc destination et puis copiez manuellement les bytes restants d'un bloc à l'autre. Écrivez un programme principal avec plusieurs cas limite pour vérifier l'exécution correcte. Comparez l'exécution de votre procédure **fastcopy** avec l'utilisation de l'instruction **movsb**.

---

## 15.10 Résumé

Le 80x86 fournit un puissant jeu d'instructions de chaîne. Cependant, ces instructions sont très primitives, utiles principalement pour manipuler des blocs de bytes. Elles ne correspondent pas aux instructions de chaîne qu'on s'attend à trouver dans un langage de niveau élevé. Vous pouvez, cependant, employer les instructions de chaîne 80x86 pour synthétiser les fonctions normalement associées aux HLLs. Ce chapitre explique comment construire la plupart des fonctions de chaîne les plus populaires. Naturellement, il est idiot de réinventer constamment la roue, aussi ce chapitre décrit également plusieurs des fonctions de chaîne disponibles dans la Bibliothèque Standard de l'UCR.

Les instructions de chaîne 80x86 fournissent la base de plusieurs des opérations de chaîne apparaissant en ce chapitre. Par conséquent, ce chapitre commence par passer en revue et une discuter dans le détail les instructions de chaîne 80x86 : les préfixes de répétition et le drapeau de direction. Ce chapitre discute de l'exécution de chacune des instructions de chaîne et décrit comment vous pouvez employer chacune d'entre elles pour exécuter des tâches en rapport avec les chaînes. Pour voir comment les instructions de la chaîne 80x86 fonctionnent, voyez les sections suivantes :

- "Les Instructions de Chaîne du 80x86" à la section 15.1
- "Comment Fonctionnent les Instructions de Chaîne" à la section 15.1.1
- "Les préfixes REP/REPE/REPZ et REPNZ/REPNE" à la section 15.1.2
- "Le Drapeau de Direction" à la section 15.1.3
- "L'Instruction MOVS" à la section 15.1.4
- "L'Instruction CMPS" à la section 15.1.5
- "L'Instruction SCAS" à la section 15.1.6
- "L'Instruction STOS" à la section 15.1.7
- "L'Instruction LODS" à la section 15.1.8
- "Construction de Fonctions de Chaînes Complexes avec LODS et STOS" à la section 15.1.9
- "Les Préfixes et les Instructions de Chaîne" à la section 15.1.10

Bien qu'Intel les appelle "instructions de chaîne" elles ne travaillent pas réellement sur le type de données abstrait auquel nous associons normalement le terme chaîne de caractères. Les instructions de chaîne manipulent uniquement des tableaux d'octets, de mots, ou de doubles mots. Il faut un peu d'efforts pour amener ces instructions

à traiter de véritables chaînes de caractères. Malheureusement, il n'y a pas une définition unique de chaîne de caractères ce qui est, sans doute, la raison pour laquelle il n'y a pas vraiment lesd'instructions spécifiques pour les chaînes de caractères dans le jeu d'instruction 80x86. Deux des types de chaîne de caractères les plus populaires incluent les chaînes à en tête de longueur et les chaînes terminées par zéro qu'utilisent respectivement le Pascal et le C. Les détails sur les formats de chaîne apparaissent dans les sections suivantes :

- "Chaînes de Caractères" à la section 15.2
- "Types de Chaînes" à la section 15.2.1

Une fois que vous décidez d'un type de données spécifique pour vos chaînes de caractères, la prochaine étape est de mettre en application diverses fonctions pour traiter ces chaînes. Ce chapitre fournit des exemples de fonctions de chaîne différentes conçues spécifiquement pour les chaînes à en tête de longueur. Pour apprendre ces fonctions et voir le code qui les met en application, voyez les sections suivantes :

- "Assignment de Chaîne" à la section 15.2.2
- "Comparaison de Chaînes" à la section 15.2.3
- "Fonctions de Chaînes de Caractères" à la section 15.3
- "Substr" à la section 15.3.1
- "Index" à la section 15.3.2
- "Repeat" à la section 15.3.3
- "Insert" à la section 15.3.4
- "Delete" à la section 15.3.5
- "Concatenation" à la section 15.3.6

La bibliothèque standard de l'UCR fournit un très riche jeu de fonctions de chaîne spécifiquement conçues pour les chaînes terminées par zéro. Pour une description de ces routines, lisez les sections suivantes :

- " Fonctions de Chaîne dans la Bibliothèque Standard de l'UCR" à la section 15.4
- "StrBDel, StrBDelm" à la section 15.4.1
- "Strcat, Strcatl, Strcatm, Strcatml" à la section 15.4.2
- "Strchr" à la section 15.4.3
- "Strcmp, Strcmpl, Stricmp, Stricmpl" à la section 15.4.4
- "Strcpy, Strcpyl, Strdup, Strdupl" à la section 15.4.5
- "Strdel, Strdelm" à la section 15.4.6
- "Strins, Strinsl, Strinsm, Strinsml" à la section 15.4.7
- "Strlen" à la section 15.4.8
- "Strlwr, Strlwrn, Strupr, Struprm" à la section 15.4.9
- "Strrev, Strrevm" à la section 15.4.10
- "Strset, Strsetm" à la section 15.4.11
- "Strspan, Strspanl, Strcspan, Strcspanl" à la section 15.4.12
- "Strstr, Strstrl" à la section 15.4.13
- "Strtrim, Strtrimm" à la section 15.4.15
- "Autres Routines de Chaîne dans la Bibliothèque Standard de l'UCR" à la section 15.4.15

Ainsi que mentionné précédemment, les instructions de chaîne sont très utiles pour beaucoup d'opérations autres que la manipulation de chaîne de caractères. Ce chapitre se ferme avec quelques sections décrivant d'autres utilisations pour les instructions de chaîne. Voyez

- "Utilisation des Instructions de Chaîne sur d'Autres Types de Données" à la section 15.6
- "Chaînes d'Entiers en Multiprécision " à la section 15.6.1
- ? "Gérer des Tableaux et des Enregistrements Entiers " à la section 15.6.2

Les ensembles sont un autre type de données abstrait courant généralement trouvé dans les programmes aujourd'hui. Un ensemble est une structure de données qui représente l'appartenance (ou la non-appartenance) à un certain groupe d'objets. Si tous les objets sont du même type de base fondamental et qu'il y a un nombre limité

d'objets possibles dans l'ensemble, alors nous pouvons employer un *vecteur de bits* (tableau de booléens) pour représenter l'ensemble. L'exécution de vecteur de bits est très efficace pour de petits ensembles. La bibliothèque standard de l'UCR fournit plusieurs routines pour manipuler des jeux de caractères et d'autres ensembles avec un maximum de 256 membres. Pour plus de détails

- " Routines de Jeux de Caractères de la Bibliothèque Standard de l'UCR " à la section 856.

## 15.11 Questions

- 1) A quoi servent les préfixes de répétition ?
- 2) Quels préfixes de chaîne employe-t-on avec les instructions suivantes ?  
a) MOVS                      b) CMPS                      c) STOS                      d) SCAS
- 3) Pourquoi n'utilise-t-on pas normalement les préfixes de répétition avec l'instruction LODS ?
- 4) Qu'arrive-t-il aux registres SI, DI et CX quand l'instruction MOVSB est exécutée (sans préfixe de répétition) et que :  
a) le drapeau de direction est à un                      b) le drapeau de direction est à zéro.
- 5) Expliquez comment fonctionnent les instructions MOVSB et MOVSW. Décrivez comment elles affectent la mémoire et les registres avec et sans préfixe de répétition. Décrivez ce qui se produit quand le drapeau de direction est à un et à zéro.
- 6) Comment préservez-vous la valeur du drapeau de direction à travers un appel de procédure ?
- 7) Comment pouvez-vous vous assurer que le drapeau de direction contient toujours une valeur appropriée avant une instruction de chaîne sans le sauvegarder à l'intérieur d'une procédure ?
- 8) Quelle est la différence entre les instructions "MOVSB", "MOVSW", et "MOVS oprnd1, oprnd2" ?
- 9) Considérez la définition de tableau Pascal suivante :

```
a:array [ 0..31 ] of record
    a, b, c:char ;
    i, j, k:integer ;
end;
```

En supposant que A[0] a été initialisé à une certaine valeur, expliquez comment vous pouvez employer l'instruction MOVS pour initialiser les éléments restants de A à la même valeur qu'A[0].

- 10) Donnez un exemple d'une opération MOVS qui exige que le drapeau de direction soit :  
a) à zéro                      b) à un
- 11) Comment fonctionne l'instruction CMPS ? (que fait elle, comment affectet-elle les registres et les drapeaux, etc...)
- 12) Quel segment contient la chaîne source ? La chaîne destination ?
- 13) Pour quoi faire l'instruction SCAS est-elle employée ?
- 14) Comment initialiseriez-vous rapidement tous les éléments d'un tableau à zéro ?
- 15) Comment les instructions LODS et STOS sont-elles utilisées comme moyen d'implémenter des opérations de chaîne complexes ?
- 16) Comment utiliseriez-vous la fonction SUBSTR pour extraire une sous-chaîne de longueur 6 commençant à l'offset 3 dans la variable de StrVar, et stocker la sous-chaîne dans la variable NewStr ?
- 17) Quels types d'erreurs peut-on rencontrer quand la fonction Substr est exécutée ?
- 18) Donnez un exemple démontrant l'utilisation de chacune des fonctions de chaîne suivantes :  
a) INDEX                      b) REPEAT                      c) INSERT                      d) DELETE                      e) CONCAT
- 19) Ecrivez une courte boucle qui multiplie chaque élément d'un tableau unidimensionnel par 10. Employez les instructions de chaîne pour aller chercher et stocker chaque élément du tableau.
- 20) La bibliothèque standard de l'UCR ne fournit pas de routine STRCPYM. Quelle est la routine qui se charge de cette tâche ?

- 21) Supposez que vous écrivez un "jeu d'aventure" dans lequel le joueur tape des phrases et que vous vouliez récupérer les deux mots "GO" et "NORTH", s'ils sont présents, dans la ligne d'entrée. Quelle fonction de chaîne (non StdLib -UCR) apparaissant en ce chapitre emploieriez-vous pour rechercher ces mots ? Quelle routine standard de la bibliothèque de l'UCR emploieriez-vous ?
- 22) Expliquez comment effectuer une comparaison de nombre entier de précision étendue à l'aide de CMPS.