La Bibliothèque standard UCR Chapitre 7

La plupart des langages de programmation fournissent des fonctions intégrées pour automatiser certaines tâches courantes et réduire l'effort d'écrire des programmes. En général, ceux qui programment en assembleur, ne disposent pas de ces facilités. C'est pourquoi, écrire en assembleur peut parfois se révéler peu productif, car il faut réinventer la roue pour chaque programme écrit. La bibliothèque standard UCR vient en aide à cette situation en fournissant un certain nombre de routines utiles. Ce chapitre en décrit quelques unes. Après l'avoir lu, vous serez en mesure de découvrir les autres routines à l'aide de la documentation fournie avec la bibliothèque UCR.

7.0 Vue d'ensemble du chapitre

Ce chapitre offre une introduction de base aux fonctions disponibles dans la bibliothèque UCR. Cette brève introduction couvre les sujets suivants :

- Introduction à la bibliothèque standard
- Les routines de gestion de la mémoire
- Routines d'entrée
- Routines de sortie
- Conversions
- Constantes prédéfinies et macros

7.1 Une introduction à la bibliothèque standard UCR

La « Bibliothèque standard UCR pour les programmeurs des systèmes 80x86 » constitue un ensemble de sous-routines modèles semblables à la bibliothèque standard du langage C. Entre autres choses, la bibliothèque standard comprend des procédures pour faciliter les entrées, les sorties, les conversions, les divers contrôles et comparaisons, le traitement des chaînes, la gestion de la mémoire, les opérateurs du jeu de caractères, les opérations en virgule flottante, le traitement des listes, les E/S des ports série, la concurrence et les co-routines et la reconnaissance de modèles (pattern matching).

Ce chapitre n'a pas pour but de décrire toutes les routines de la bibliothèque. Avant tout, la bibliothèque est en constante évolution et toute référence complète deviendrait rapidement obsolète. Ensuite, certaines routines sont uniquement destinées à des programmeurs avancés et leur description irait au-delà des buts de cet ouvrage. Et, finalement, il y a de milliers de routines dans la bibliothèque. Tenter de les décrire toutes constituerait un ouvrage à part et vous distrairait de votre but primaire : apprendre l'assembleur.

Par conséquent, che chapitre couvrira seulement les routines nécessaires pour pouvoir travailler avec le moins d'effort possible. Notez que la documentation complète de la bibliothèque, ainsi que le code source et divers exemples, se trouvent dans le cédérom d'accompagnement de ce livre[1]. Un guide de référence figure également à la fin de cet ouvrage. Mais vous pouvez également trouver les dernières versions de la bibliothèque sur beaucoup de services en ligne, BBS, ainsi que divers autres sites free-ware. Elle est également disponible via FTP anonyme[2].

Quand on utilise la bibliothèque UCR il faut toujours utiliser le fichier SHELL.ASM fourni comme "modèle" pour tout nouveau programme. Ce fichier contient les segments nécessaires et fournit les bonnes directives *include* ; il initialise également les routines de bibliothèque nécessaires. En général, vous ne devriez pas créer un nouveau programme à partir de zéro sans d'abord être très familier avec le fonctionnement interne le la bibliothèque standard.

Notez aussi que beaucoup de routines de la bibliothèque utilisent des macros et sont appelées de façon différente par rapport à l'appel traditionnel avec l'instruction call. Par exemple, vous ne pouvez pas appeler

[1]Naturellement, il s'agit du répertoire "stdlib" qu'on a téléchargé à partir du site de ce livre, déjà à la lecture du chapitre 1, n.d.t.
[2]Voir la fin de ce chapitre pour en savoir plus, n.d.t.

directement la routine putc, mais évoquer la macro putc, laquelle inclut un appel à la procédure sl_putc ("sl" veut dire « standard library », bibliothèque standard).

Si vous décidez de ne pas utiliser le fichier shell.asm, votre programme devra alors inclure plusieurs instructions pour activer la bibliothèque et satisfaire certaines exigences. Si vous choisissez ce chemin, veuillez jeter un œil à la documentation qui accompagne la bibliothèque. Mais, tant que vous n'aurez pas acquis suffisamment d'expérience en assembleur, on vous conseille fortement de vous confier au fichier SHELL.ASM comme point de départ de vos programmes.

7.1.1 Routines de gestion de la mémoire : MEMINIT, MALLOC et FREE

La bibliothèque standard fournit plusieurs routines pouvant gérer la mémoire libre du tas (ou *heap*, en anglais). Ces routines donnent aux programmeurs la facilité d'allouer dynamiquement de la mémoire pendant l'exécution d'un programme et de restituer cette mémoire au système quand le programme n'en a plus besoin. Grâce à l'allocation dynamique de la mémoire, on peut faire un usage intelligent de cette ressource sur un PC.

La routine meminit initialise le gestionnaire de mémoire et vous devez impérativement l'appeler avant d'invoquer toute routine qui l'utilise. Comme beaucoup de routines de la bibliothèque standard utilisent le gestionnaire de mémoire, il est conseillé d'appeler cette procédure tôt dans le programme. Le fichier SHELL.ASM fait cet appel pour vous.

La routine malloc alloue de l'espace dans le heap (tas, en français) et retourne un pointeur dans le registre es:di sur le bloc disponible. Mais, avant d'appeler malloc, il faut charger la taille (en octets) du bloc dans le registre cx. En retour, malloc active le drapeau carry si une erreur s'est produite (par exemple, si la mémoire est insuffisante). Si le drapeau carry est à 0, alors es:di pointera sur le bloc de la mémoire allouée. Ce bloc aura la taille indiquée.

```
mov     cx, 1024              ;réserve 1024 octets dans le heap
malloc                        ;appelle MALLOC
jc      MallocError           ;en cas d'erreur
mov     word ptr PNTR, DI     ;sauvegarder le pointeur du bloc
mov     word ptr PNTR+2, ES
```

Quand vous appelez malloc, le gestionnaire de mémoire promet que le bloc qu'il vous donne est libre et à zéro et ne réallouera pas ce bloc à moins que vous ne le libériez explicitement. Pour retourner un bloc de mémoire au gestionnaire de mémoire de façon à pouvoir (éventuellement) le réutiliser, utilisez la routine *free*. Cette routine s'attend à ce que vous lui passiez le pointeur retourné par malloc.

```
les     di,  PNTR             ;récupérer le pointeur
free                          ;libérer le bloc
jc      BadFree               ;en cas d'erreur...
```

Comme c'est d'usage dans la plupart des routines, si free encourt des difficultés, elle activera le drapeau carry.

7.1.2 Les routines d'entrée standard : GETC, GETS et GETSM

Parmi les nombreuses routines que la bibliothèque standard met à disposition, il y en a trois que vous utiliserez tout le temps : getc (*get a character*), gets (*get a string*) et getsm (*get a malloc's string*). getc lit un seul caractère du clavier à chaque appel et place (retourne) ce caractère dans le registre al. Elle retourne EOF (*end of file*) dans le registre ah si elle ne trouve pas de caractère (quand ah=0 EOF ne se produit pas et quand ah=1 il se produit). getc ne modifie aucun autre registre. Comme d'habitude, le drapeau de retenue retourne l'état d'erreur. Vous n'avez pas besoin de passer à getc aucune valeur des registres. Cette routine ne reproduit pas les caractères d'entrées (*echo*) à l'écran. Vous devez explicitement afficher le caractère si vous voulez l'afficher.

Le programme suivant boucle jusqu'à ce que l'utilisateur presse la touche Entrée :

```
;A noter : cr est un symbole qui apparaît dans le fichier d'en-tête
;"consts.h". Il correspond à la valeur 13 (0xDh), qui est le code
;ASCII du retour chariot.
Wait4Enter:     getc
                cmp     al, cr
```

```
                        jne     Wait4Enter
```

La routine gets lit une ligne complète de texte depuis le clavier. Elle garde chaque caractère successif de la ligne d'entrée dans un tableau d'octets, duquel vous passez l'adresse de base dans la paire de registres es:di. Ce tableau doit avoir une taille minimale de 128 octets. gets lira chaque caractère et le placera dans le tableau, sauf le caractère de retour chariot. Cette routine fait terminer le tableau avec un caractère nul (lequel est compatible avec les routines de traitement de chaînes de la bibliothèque standard du langage C). gets affiche à l'écran chaque caractère entré et fournit aussi des fonctions simples d'édition de ligne, par exemple backspace. Comme d'habitude, elle active CF si une erreur se produit. L'exemple suivant lit une ligne de texte depuis le périphérique d'entrée standard (le clavier), ensuite elle compte le nombre de caractères tapés. Ce code est un peu tordu, notez qu'il initialise le compteur et le pointeur à -1 avant d'entrer dans la boucle et les incrémente tout de suite après. Ceci ajuste le pointeur à 0, de sorte qu'il pointe sur le premier élément de la chaîne. Cette simplification produit un code plus efficace en comparaison avec une solution plus simple à programmer.

```
DSEG            segment

MyArray         byte    128 dup (?)

DSEG            ends

CSEG            segment
                :
                :
;Note: LESI est une macro (qu'on trouve dans consts.a) qui charge
;ES:DI avec l'adresse de son opérande. C'est l'équivalent de :
;
;               mov di, seg operand
;               mov es, di
;               mov di, offset operand
;
;Vous utiliserez cette macro avant beaucoup d'appels à la bibliothèque

                lesi    MyArray         ;obtient l'adresse du tampon d'entrée
                gets                    ;lit une ligne de texte
                mov     ah, -1          ;enregistre le compteur dans ah
                lea     bx, -1[di]      ;pointe juste avant la chaîne
CountLoop:      inc     ah              ;incrémente le compteur
                inc     bx              ;pointe sur le premier caractère
                cmp     byte ptr es:[bx], 0
                jne     CountLoop

;Maintenant, ah contient le nombre de caractères de la chaîne
                :
                :
```

La routine getsm lit aussi une chaîne du clavier et retourne un pointeur sur cette chaîne (dans es:di). La différence entre gets et getsm est que vous n'avez pas à passer l'adresse d'un tampon d'entrée à es:di. getsm alloue automatiquement de l'espace sur le heap à l'aide d'un appel à malloc et retourne l'adresse du tampon directement dans es:di. Mais n'oubliez pas d'appeler *meminit* au début de votre programme avant d'utiliser cette routine. Le fichier SHELL.ASM le fait automatiquement pour vous. N'oubliez pas non plus d'appeler free pour libérer l'espace occupé une fois que vous avez terminé avec la ligne d'entrée.

```
                getsm                   ;retourne un pointeur sur es:di
                :
                :
                free                    ;libère l'espace dans le heap
```

7.1.3 Les Routines de sortie standard : PUTC, PUTCR, PUTS, PUTH, PUTI, PRINT et PRINTF

La bibliothèque standard fournit une vaste gamme de routines de sortie, beaucoup plus de ce que vous verrez ici. Les routines qui suivent sont représentatives des routines que vous trouverez dans la bibliothèque.

putc produit un caractère dans la sortie standard (l'écran). Elle affiche le caractère qui se trouve dans le registre al. Elle n'affecte aucun autre registre, sauf en cas d'erreur de sortie (CF signale erreur/pas d'erreur, comme d'habitude). Pour plus de détails, consultez la documentation de la bibliothèque standard.

putcr produit une nouvelle ligne (combinaison de retour chariot / saut de ligne[3]). Elle est complètement équivalente à :

```
mov     al, cr          ;cr et lf sont des constantes qui
putc                    ;apparaissent dans le fichier d'en-tête
mov     al, lf          ;consts.a
putc
```

La routine puts (*put a string*) affiche la chaîne terminée par zéro sur laquelle point la paire es:di[4]. Notez que puts *ne produit pas* automatiquement un retour à la ligne après que la chaîne est affichée. Pour obtenir cet effet, vous devez ajouter la combinaison cr/lf à la fin de la chaîne, ou appeler putcr après avoir appelé puts. puts n'affecte pas les registres (sauf en cas d'erreur), et, en particulier, ne change pas la valeur d'es:di. La séquence d'instructions suivante illustre ce fait :

```
getsm           ;lit une chaîne
puts            ;affiche cette chaîne
putcr           ;ajout d'une nouvelle ligne
free            ;libération de la mémoire allouée par getsm
```

Puisque ces routines préservent es:di (sauf, bien sûr, getsm). l'appel à free désalloue la mémoire occupée par getsm.

La routine puth affiche la valeur du registre al selon un format d'exactement deux chiffres hexadécimaux, en incluant un préfixe de zéro si la valeur est dans la plage 0..Fh. La boucle suivante lit une séquence de frappes du clavier et en affiche la valeur ASCII (en format hexadécimal) jusqu'à ce que l'utilisateur appuie sur Entrée.

```
KeyLoop:        getc
                cmp     al, cr
                je      Done
                puth
                putcr
                jmp     KeyLoop
Done:
```

La routine puti affiche la valeur de ax en tant que valeur signée de 16 bits. Le fragment de code suivant affiche à l'écran la somme de I et J :

```
mov     ax, I
add     ax, J
puti
putcr
```

putu est semblable à puti, mais elle affiche seulement des entiers non signés.

Des routines comme puti et putu affichent toujours un nombre en utilisant le nombre minimal de positions d'affichage possibles. Par exemple, puti utilise trois positions d'affichage pour la chaîne affichant la valeur 123. Parfois cependant, vous voulez plutôt forcer ces routines de sortie à afficher des valeurs utilisant un nombre déterminé de positions d'affichage et en remplissant les positions additionnelles avec des espaces. Les routines putisize et putusize permettent cette fonctionnalité. Elles attendent une valeur numérique dans le registre ax et un champ spécifiant la longueur dans cx. Elles afficheront la valeur dans un champ de largeur d'au moins cx positions. Si la valeur de cx est supérieure au nombre des positions d'affichage que la valeur requiert, ces routines justifieront à droite la valeur du nombre de positions spécifié dans cx. Si la valeur contenue dans cx est

[3]Line Feed (abrégé LF) en anglais, n.d.t.

[4]Une chaîne terminée par zéro est une séquence de caractères terminant avec un octet dont la valeur est zéro. C'est le caractère de formatage standard de la bibliothèque.

inférieure au nombre de positions d'affichage que la valeur requiert, alors putisize et putusize ignoreront la valeur spécifiée dans cx et utiliseront à la place le nombre exact de positions que la valeur demande.

```
;La boucle suivante affiche toutes les valeurs d'une matrice 3x3 en
;forme matricelle. La valeur initiale correspond à l'élément [0,0]
;de la matrice organisée en rangées
                mov     dx, 3           ;répéter pour chaque rangée
PrtMatrix:      mov     ax, [bx]        ;obtenir le premier élément de cette
                                        ;rangée
                mov     cx, 7           ;utiliser 7 positions d'affichage
                putisize                ;afficher cette valeur
                mov     ax, 2[bx]       ;atteindre le deuxième élément
                putisize                        ;CX vaut encore 7
                mov     ax, 4[bx]       ;atteindre le troisième élément
                putisize
                putcr                   ;sortir une nouvelle ligne
                add     bx, 6           ;se déplacer à la ligne suivante
                dec     dx              ;répéter pour chaque ligne
                jne     PrtMatrix
```

La routine print est l'une des routines les plus souvent appelées de la bibliothèque standard. Elle affiche la chaîne terminée par zéro qui la suit immédiatement :

```
                print
                byte    "Afficher cette chaîne à l'écran",cr,lf,0
```

Cet exemple affiche la chaîne "Afficher cette chaîne à l'écran", suivie par un saut de ligne. Notez que print affiche n'importe quel caractère qui suit immédiatement l'appel jusqu'au premier octet valant zéro qu'elle rencontre. Remarquez en particulier que vous pouvez afficher la séquence de nouvelle ligne, ainsi que tout autre caractère de contrôle, comme on le montre ci-dessus. Notez finalement que vous n'êtes pas limité à une seule ligne de texte. Par exemple :

```
                print
                byte    "Cet exemple de la routine PRINT ",cr,lf
                byte    "affiche plusieurs lignes de texte. ",cr,lf
                byte    "Notez aussi,",cr,lf,"que les lignes sources "
                byte    "n'ont pas à correspondre à la sortie."
                byte    cr,lf
                byte    0
```

Ce qui affiche :

```
Cet exemple de la routine PRINT
affiche plusieurs lignes de texte.
Notez aussi,
que les lignes sources n'ont pas à correspondre à la sortie.
```

Il est très important que vous *n'oubliiez pas* l'octet de fin de chaînes. La routine print recommence l'exécution du code machine 80x86 à la première instruction suivant le zéro de fin de chaîne. Si vous oubliez de placer ce caractère à la fin de vos chaînes, la routine print avalera toutes les instructions qui suivent votre chaîne affichant à l'écran (en les affichant à l'écran) jusqu'à la prochaine occurrence d'un octet dont la valeur est 0 (les octets à zéro son communs dans les programmes en assembleur). Ceci fera planter votre programme, pouvant ainsi constituer une énorme source d'erreurs. Ce genre d'erreurs est typique des programmeurs débutants. Gardez toujours cela à l'esprit.

printf, tout comme sa fonction cousine en langage C permet d'afficher des sorties formatées. Un appel typique de printf suit toujours la syntaxe suivante :

```
        printf
        byte    "chaîne formatée",0
        dword   operand1, operand2, ..., operandn
```

La chaîne de format est comparable à celle du langage C. Pour la plupart des caractères, printf affiche simplement les caractères de la chaîne de format jusqu'à la première occurrence de l'octet 0. Les deux exceptions sont constituées par les caractères préfixés par une barre oblique inverse ("\"), et les caractères préfixés par un signe de pourcentage ("%"). Comme il arrive dans le printf du C, celui de notre bibliothèque

standard utilise la barre oblique inverse comme caractère d'échappement et le signe de pourcentage comme début d'un format d'affichage.

printf utilise "\" pour afficher des caractères spéciaux de manière semblable, mais non identique, à son homologue en C. La routine printf de la bibliothèque standard admet les caractères spéciaux suivants :

- \n nouvelle ligne (retour de chariot / nouvelle ligne) (*new line=carriage return + line feed*)
- \b retour arrière (*backspace*)
- \t tabulation (*tab*)
- \l avance de ligne (sans retour chariot) (*line feed*)
- \f avance de page (*form feed*)
- \\ caractère barre oblique inverse (*backslash*)
- \% caractère pourcentage (*percent*)
- <0xhh le code ASCII hh, représenté par deux chiffres hexadécimaux

Les programmeurs du C devraient noter une paire de différences entre leurs séquences et celles de notre bibliothèque standard. D'abord l'usage de "\%" pour afficher un signe de pourcentage et non "%%". Le langage C ne permet pas l'utilisation de "\%" parce que le compilateur C traite le "\%" en temps de compilation (en laissant un seul "%" dans le code objet), alors que printf traite cette chaîne formatée en temps d'exécution. Elle verrait un seul "%" et le traiterait comme un signe précédant un caractère de format. De son côté, le printf de notre bibliothèque standard traite chaque caractère, le "\" et le "%" en temps d'exécution et peut donc les distinguer.

Les chaînes de la forme 0x*hh* doivent contenir exactement deux chiffres hexadécimaux. La routine courante printf n'est pas assez robuste pour traiter des cas comme 0x*h* contenant seulement un chiffre hexadécimal. Gardez ceci à l'esprit si vous voyez printf sauter des caractères après avoir affiché une valeur.

Il n'y a absolument aucune raison d'utiliser une séquence d'échappement avec les chiffres hexadécimaux, sauf dans le cas de "\0x00", car dans ce cas, printf pourrait confondre le zéro hexadécimal avec l'octet de fin de chaîne (c'est la seule façon d'afficher le caractère nul). La routine printf de la bibliothèque standard ne se soucie pas du nombre de caractères à traiter. En particulier, vous n'êtes pas limité à utiliser une seule chaîne après l'appel à printf. Le fragment de code suivant est parfaitement correct:

```
printf
byte     "Ceci est une chaîne",13,10
byte     "Ceci est une nouvelle ligne",13,10
byte     "Affichage d'un backspace à la fin de cette ligne: "
byte     8,13,10,0
```

Votre code est susceptible de s'exécuter un peu plus rapidement si vous évitez l'usage des séquences d'échappement. Sachez d'ailleurs que tout caractère d'échappement occupe au moins deux octets en mémoire. Vous pouvez les encoder presque tous en remplaçant simplement chaque séquence par son équivalent hexadécimal (c'est-à-dire, son code ASCII). N'oubliez pas que vous ne pouvez pas encoder un octet nul dans le flux du code. Un tel octet terminerait la chaîne de format. Si vous voulez spécifier la valeur hexadécimale 0, au lieu du caractère de fin de chaîne, utilisez "\0h00".

Les séquences de format commencent toujours par le symbole "%". Pour chaque séquence vous devez fournir un pointeur far sur la donnée associée qui suit immédiatement la chaîne, par exemple,

```
printf
byte     "%i %i",0
dword    i,j
```

Les séquences de format ont la syntaxe suivante : "%s \cn^f" où :

- "%" est toujours le caractère "%". Si vous voulez réellement afficher un signe de pourcentage, utilisez "\%".
- *s* est soit rien, soit un signe "-".
- "\c" est également optionnel et il représente tout caractère affichable.
- "n" représente une chaîne d'un à n chiffres décimaux.
- "^" est simplement le caractère de flèche en haut.
- "f" représente un des caractères de format : *i*, *d*, *x*, *h*, *u*, *c*, *s*, *ld*, *li*, *lx*, ou *lu*.

Les éléments "s", "\c", "n" et "^" sont optionnels, alors que "%" et "f" doivent être présents. De plus, l'ordre d'apparition de ces éléments dans la chaîne de format est très important. Par exemple, "\c" ne peut pas précéder "s". De même, le caractère "^", si présent, doit suivre tout sauf le(s) caractère(s) de l'élément "f".

Les caractères de format i, d, x, h, etc. contrôlent la sortie formatée des données. Les caractères i et d font exactement la même chose, à savoir, ils indiquent à printf d'afficher des valeurs entières décimales (et signées) de 16 bits. Les caractères x et h indiquent si la valeur à afficher doit être une valeur hexadécimale de 16 ou de 8 bits (respectivement). Si on spécifie u, printf affichera un entier décimal de 16 bits non signé. Alors que c affichera un caractère isolé. S indique à printf qu'on lui fournit l'adresse d'une chaîne à afficher terminée par zéro, printf affichera cette chaîne de caractères. Les combinaisons ld, li, lx et lu sont les versions longues (de 32 bits) des données représentées par les caractères d-i, x et u respectivement. L'adresse qui correspond au format pointe sur la valeur de 32 bits que printf formatera et affichera sur la sortie standard.

L'exemple suivant illustre tous ces concepts :

```
printf
byte    "I=%i, U=%u, HexC=%h,, HexI=%x, C=%c, "
dbyte   "S=%s",13,10
byte    "L=%ld",13,10,0
dword   i,u,c,i,c,s,l
```

Le nombre des adresses far (spécifié par les opérandes dans le pseudo-opcode "dd") doit correspondre au nombre d'éléments de format % dans la chaîne de format. printf compte le nombre des éléments formatés en sautant les adresses far qui suivent le format. Si le nombre d'éléments ne correspond pas, l'adresse de retour pour printf sera incorrecte et le programme plantera très probablement. De même, la chaîne de format doit toujours terminer avec un octet nul. Les adresses des éléments qui suivent la chaîne du format doivent pointer directement sur les adresses de mémoire où la donnée spécifiée se trouve.

Utilisée avec le format ci-dessus, printf affiche toujours les valeurs en utilisant un nombre minimum de positions d'affichage pour chaque opérande. Si vous voulez spécifier une largeur minimale de champ, vous pouvez le faire en utilisant l'option de format "n". Un élément ayant le format "%10d" affichera un entier décimal en se servant d'au moins 10 positions d'affichage. De même, "%16s" affichera une chaîne avec 16 positions minimum d'affichage. Si la valeur à afficher requiert plus de positions d'affichage que le nombre spécifié, printf en utilisera autant de supplémentaires que nécessaire. Si la valeur à afficher requiert moins de positions, printf affichera toujours le nombre spécifié en comblant le reste de l'espace avec des blancs. Dans le champ d'affichage, printf justifie toujours à droite (quel que soit le type de donnée). Si vous voulez afficher la valeur selon une justification à gauche, utilisez le caractère de format "-" comme préfixe à la largeur du champ, par exemple :

```
printf
byte    "%-17s",0
dword   string
```

Dans cet exemple, printf affiche la chaîne en utilisant un champ de longueur de 17 caractères et en justifiant la sortie à gauche.

Par défaut, printf laisse des espaces blancs si la valeur à afficher requiert moins de places que le format spéficié. L'élément de format "\c" permet de changer le caractère de remplissage. Par exemple, pour afficher une valeur, justifiée à droite et utilisant "*" comme caractères de remplissage, le format à utiliser serait "%*10d". Pour afficher la même chose justifiée à gauche, on utiliserait "%-*10d". Notez que le "-" doit précéder le "*". Celle-ci est une limitation de la version courante du logiciel. Les opérandes doivent apparaître dans cet ordre. Normalement, la ou les adresses qui suivent le format de printf doivent être des pointeurs far sur la donnée à afficher.

A l'occasion, spécialement lorsqu'on alloue de l'espace dans le tas (via malloc), vous pouvez ne pas savoir (au moment de l'assemblage) l'adresse de l'objet que vous voulez afficher. Vous pouvez n'avoir qu'un pointeur sur cette donnée. L'option de format "^" indique à printf que le pointeur far qui suit la chaîne de format est l'adresse d'un pointeur sur la donnée, au lieu de l'adresse de la donnée elle-même. Cette option vous permet donc d'accéder à la donnée indirectement.

Notez: contrairement à la version du langage C, le printf de notre bibliothèque standard ne supporte pas les sorties en virgule flottante. Donner à printf cette capacité augmenterait terriblement la taille de la routine. Puisque la plupart des gens n'ont pas besoin d'afficher des nombres en virgule flottante formatés, fournir cette capacité ne semble pas indispensable. Il y une routine séparée, printff, faisant précisément ceci.

La routine printf représente un élément complexe de la bibliothèque standard. Cependant, elle est très flexible et extrêmement utile. Vous devriez lui dédier un certain temps pour en maîtriser les fonctions principales, car vous l'utiliserez souvent dans vos programmes en assembleur.

Le package de sortie standard comprend diverses autres routines additionnelles. Si nous ne les traitons pas toutes, c'est simplement par manque d'espace, mais si vous voulez vous documenter, consultez alors la documentation de la bibliothèque standard.

7.1.4 Routines de sortie formatées: PUTSIZE, PUTUSIZE, PUTLSIZE ET PUTULSIZE

Les routines puti et putl affichent à l'écran une chaîne numérique en utilisant un nombre minimal de positions. Par exemple, puti utilise trois caractères de position pour afficher la valeur -12. A l'occasion, vous pouvez avoir besoin de spécifier une largeur de champ différente de façon à pouvoir afficher des colonnes de nombres ou effectuer toute autre tâche de formatage. Bien qu'on pourrait utiliser printf pour obtenir ce résultat, printf présente deux inconvénients : elle affiche uniquement une valeur qui se trouve en mémoire (et pas, par exemple, une valeur qui se trouve dans un registre) et la largeur du champ spécifiée doit être une constante[5]. Les routines putsize, putusize et putlsize permettent de dépasser ces limitations.

Tout comme les routines puti, putu et putl qui leur correspondent, ces routines affichent des entiers signés, des entiers non signés et des valeurs de 32 bits signées. Elles attendent une valeur à afficher dans le registre ax (putisize et putusize) ou la paire de registrers dx:ax (putlsize). Elles attendent aussi une valeur de largeur minimale dans le registre cx. Comme printf, si la valeur du registre cx est inférieure au nombre des positions d'affichage nécessaires, ces trois routines ignoreront la valeur du registre cx et affichent la valeur en utilisant juste le nombre nécessaire des positions requises pour afficher la valeur comme il faut.

7.1.5 Routines de taille de champ de sortie : ISIZE, USIZE et LSIZE

Parfois, on peut avoir besoin de connaître le nombre de positions d'affichage qu'une valeur nécessitera pour être affichée, avant d'être affichée. Par exemple, vous voulez calculer la largeur d'affichage maximale d'un ensemble de nombres à afficher en colonnes et vous voulez adapter la largeur d'affichage au nombre le plus large. Les routines isize, usize et lsize vous le permettent.

isize attend un entier signé dans le registre ax. Elle retourne la largeur minimale de champ requise pour cette valeur (en incluant la position pour le signe moins, si nécessaire), dans le registre ax. usize fait la même chose avec un entier non signé alors que lsize le fait avec des entiers signés de 32 bits en attendant ces valeurs dans la paire dx:ax (en incluant la position pour le signe moins, si nécessaire). Elle retourne la largeur dans le registre ax.

7.1.6 Routines de conversion : ATO*x* et *x*TOA

La bibliothèque standard fournit diverses routines pour convertir entre chaînes de caractères et valeurs numériques. Ces rotines comprennent atoi, atoh, atou, itoa, htoa, wtoa et utoa (et d'autres encore). Les routines ATOx convertissent une chaîne ASCII en valeur numérique selon le format approprié en laissant le résultat dans le registre ax (ou al). Les routines ITOx convertissent la valeur de al/ax en chaînes de chiffres et gardent cette chaîne dans le tampon dont l'adresse se trouve dans es:di[6]. Il y a plusieurs variantes de chaque routine traitant différents cas. Les paragraphes qui suivent décrivent chacune d'elles.

La routine atoi s'attend à que es:di pointe sur une chaîne contenant des chiffres entiers (et à l'occurrence aussi un signe moins, si le nombre est négatif). Elle convertit la chaîne en entier et retourne le résultat à ax. Après le retour, es:di pointe encore sur le début de la chaîne d'origine. Si es:di ne pointe pas sur une chaîne de chiffres ou bien si un dépassement de capacité se produit, atoi modifie le drapeau de retenue en le mettant à 1. Elle préserve la valeur de la paire es:di. Une variante de atoi, atoi2 convertit également une chaîne ASCII en nombre entier, mais *sans préserver* la valeur du registre di. Cette routine est particulièrement utile si vous avez besoin de convertir une séquence de nombres apparaissant dans la même chaîne. Chaque appel à atoi2 laisse le registre di pointer sur le premier caractère suivant la chaîne qui vient d'être traitée. Vous pouvez facilement sauter tout

[5]Sauf si vous ne décidez de recourir à un code automodificateur.

[6]Il y a aussi un jeu de routines xTOAM qui allouent automatiquement de l'espace du tas.

espace, virgule ou autres délimiteurs tant que vous n'avez pas atteint le prochain nombre dans la chaîne ; ensuite, vous pouvez appeler atoi2 pour convertir cette chaîne en nombre. Et vous pouvez répéter ce processus pour chaque nombre de la ligne.

atoh fonctionne comme atoi, mais elle attend des chaînes qui contiennent des chiffres hexadécimaux (sans signe moins). En retour, ax contiendra la valeur convertie de 16 bits et le drapeau de retenue indiquera l'état erreur/non-erreur. Comme atoi, atoh préserve la valeur de la paire es:di. Vous pouvez appeler atoh2 pour permettre à di de pointer sur le premier caractère après la chaîne hexadécimale qui vient d'être convertie.

atou convertit une chaîne ASCII de chiffres décimaux (dans la plage 0..65535) en valeur entière et retourne le résultat au registre ax. Sauf pour le fait qu'ici le signe moins n'est pas permis, atou fonctionne juste comme atoi ; il s'agit juste d'une conversion non signée. Il y a bien sût un atou2, analogue à atoi2.

Pusique dans la bibliothèque standard il n'y a pas de routines geti, geth ou getu disponibles, vous allez devoir les programmer vous mêmes. Le code suivant montre comment lire un entier du clavier :

```
print
byte    "Entrez une valeur entière: ",0
getsm
atoi    ;Conversion en entier dans ax
free    ;Retourner l'espace alloué par getsm
print
byte    "Vous avez entré ",0
puti    ;Affichage de la valeur retournée par ATOI
putcr
```

Les routines itoa, utoa, htoa et wtoa sont l'inverse logique des routines ATOx. Elles convertissent des valeurs numériques en leur équivalent de chaîne entière, non signée et hexadécimale. Il y a diverses variations de ces routines en dépendant si vous voulez allouer de l'espace automatiquement pour la chaîne ou si vous voulez préserver le registre dx.

itoa convertit l'entier signé de 16 bits dans ax en chaîne et garde les caractères de celle-ci à l'emplacement pointé par es:di. Quand on appelle itoa, il faut s'assurer que es:di pointe sur un tableau de caractères suffisamment large pour garder la chaîne résultante. itoa requiert un maximum de sept octets pour la conversion : cinq chiffres numériques, un signe et un octet zéro de terminaison. Elle préserve les valeurs dans es:di, donc lors d'un retour, es:di pointe sur le début de la chaîne produite par itoa.

Occasionnellement, vous pouvez ne pas vouloir préserver cette valeur. Par exemple, si vous voulez créer une seule chaîne contenant plusieurs valeurs converties, il serait bon si itoa laissait di pointer sur la fin de la chaîne, au lieu de son début. La routine itoa2 fait exactement ceci ; elle permet à di de pointer sur le zéro terminal à la fin de la chaîne. Considérez le segment de code suivant qui produit une chaîne contenant les représentations ASCII de trois variables entières, Int1, Int2 et Int3 :

```
; En supposant que es:di pointe déjà à l'adresse de départ pour stocker
; les valeurs entières converties

                mov ax, Int1
                itoa2           ;conversion de Int1 en chaîne

; D'accord, faire afficher un espace entre les nombres et déplacer di, de
; sorte qu'il pointe sur la prochaine position disponible dans la chaîne

                mov byte ptr es:[di], ' '
                inc di

; Convertir la seconde valeur

                mov ax, Int2
                itoa2
                mov byte ptr, es:[di], ' '
                inc di

; Convertir la troisième valeur

                mov ax, Int3
                itoa2
```

```
; À ce point, di pointe sur la fin de la chaîne contenant les valeurs
; converties. On espère que vous connaissiez encore où le commencement de la
chaîne se trouve, de façon à pouvoir la manipuler !
```

Une autre variante de la routine itoa, itoam, ne requiert pas l'initialisation de es:di. Elle appelle malloc pour allouer automatiquement de l'espace pour vous. Elle retourne dans la paire es:di un pointeur sur le tas où se trouve la chaîne convertie. Quand vous avez terminé avec cette chaîne, il vous faudra appeler free pour retourner l'espace au tas.

```
; Le fragment de code suivant convertit l'entier dans AX en chaîne
; et l'affiche ensuite. Sans doute, vous pourriez obtenir la même chose avec
; PUTI, mais ce code sert juste à montrer comment appeler itoam
                itoam           ;fait la conversion
                puts            ;affiche la chaîne
                free            ;libère l'espace sur le tas
```

Les routines utoa, utoa2 et utoam fonctionnent tout comme itoa, itoa2 et itoam, sauf qu'elles convertissent une valeur entière non signée (dans ax) en chaîne. Notez que utoa et utoa2 requièrent au plus six octets, car elles ne produisent jamais un caractère de signe moins.

wtoa, wtoa2 et wtoam convertissent en chaîne une valeur hexadécimale de 16 bits (dans ax) ayant exactement quatre caractères hexadécimaux, plus un octet de terminaison. Sinon, elles se comportent exactement comme itoa, itoa2 et itoam. Notez que ces routines affichent des zéros de préfixe, de façon que la valeur a toujours une longueur de quatre chiffres.

Les routines htoa, htoa2 et htoam sont similaires à wtoa, wtoa2 et wtoam, mais elles convertissent une valeur de huit bits dans al, en une chaîne de deux caractères hexadécimaux, plus un octet zéro de terminaison.

La bibliothèque standard fournit diverses autres routines de conversion. Voir la documentation qui l'accompagne pour avoir de plus amples détails.

7.1.7 Routines qui testent des caractères pour vérifier leur appartenance à un ensemble

La bibliothèque standard UCR fournit diverses routines permettant de tester un caractère dans al pour vérifier s'il fait partie d'un certain ensemble de caractères. Ces routines retournent toutes leur état dans le flag zéro. Si la condition est vraie, ce drapeau vaudra 1, si elle est fausse, il vaudra 0 (de façon à vous permettre de tester cet état par une instruction comme jne). Ces routines sont :

- IsAlNum vérifie si al contient un caractère alphanumérique.
- IsXDigit vérifie si al contient un caractère représentant un chiffre hexadécimal.
- IsDigit vérifie la présence d'un caractère représentant un chiffre décimal.
- IsAlpha vérifie la présence d'un caractère alphabétique.
- IsLower vérifie si al contient un caractère alphabétique minuscule.
- IsUpper vérifie si al contient un caractère alphabétique majuscule.

7.1.8 Routines de Conversion de Caractères : ToUpper et ToLower

Les routines ToUpper et ToLower vérifient le caractère dans le registre al et le convertissent à la casse correspondante.

Si al contient un caractère alphabétique minuscule,ToUpper le convertit en majuscule. Si al contient tout autre caractère, ToUpper le retournera inchangé.

Si al contient un caractère alphabétique majuscule, ToLower le convertira en minuscule. Si la valeur de al ne correspond pas à un caractère majuscule, alors la conversion sera ignorée et ce caractère sera retourné tel quel.

7.1.9 Génération de nombres aléatoires : ramdom et randomize

La routine Random de la bibliothèque standard, génère une séquence de nombres pseudo-aléatoires et retourne une valeur aléatoire dans le registre ax après chaque appel. Vous pouvez traiter cette valeur comme signée et comme non signée, car Random manipule les 16 bits du registre ax en totalité.

Vous pouvez utiliser les instructions div et idiv pour forcer la sortie de random à une plage spécifique. Il suffit de diviser la valeur retournée par un nombre n et le reste de cette division sera une valeur allant de 0 à n-1. Par exemple, pour obtenir un nombre aléatoire de la plage 1.10 on pourrait faire ceci :

```
        random          ;obtention d'un nombre aléatoire de la plage 0..65535
        sub     dx, dx  ;extension à zéro sur 16 bits
        mov     bx, 10  ;on veut une valeur de 1 à 10..
        div     bx      ;le résultat ira dans dx!
        inc     dx      ;conversion de 0..9 à 1..10
;Et maintenant, un nombre aléatoire de la plage 1..10 se trouve dans dx.
```

La routine random retourne toujours la même séquence de valeurs, une fois qu'un programme a été chargé du disque et exécuté. Random se sert d'une table interne de valeurs *sémille (*on dit *seed* en anglais*)* qu'elle stocke dans son code. Puisque ces valeurs sont fixes et elles se chargent toujours en mémoire avec le programme, l'algorithme que random utilise donnera toujours la même séquence de résultats à partir du moment où un programme est chargé pour être exécuté. Ceci ne sonne pas très "aléatoire", mais, en fait, c'est une caractéristique utile, car ce serait très difficile de tester un programme générant de *vraies* valeurs aléatoires. Au contraire en produisant toujours la même séquence de nombres, tout test qu'on peut appliquer à un programme peut être répété.

Maheureusement, il y a beaucoup d'exemples de programmes que vous voudriez écrire (par exemple, les jeux) où une même séquence de résultats aléatoires n'est pas acceptable. Pour ces applications, on peut appeler la routine *randomize*. Cette routine utilise la valeur courante de l'horloge système pour générer une séquence de départ quasi aléatoire. Donc, si vous avez besoin d'une nouvelle série de nombres aléatoires chaque fois que vous commencez l'exécution d'un programme, appelez randomize avant même d'appeler random. Notez qu'il n'y a qu'un maigre avantage d'appeler randomize plus d'une fois dans le même programme. Une fois que random établit un nombre de départ pour sa formule, des appels successifs à randomize n'amélioreront pas la qualité (c'est-à-dire le hasard de parution) des nombres générés.

7.1.10 Constantes, macros et instructions diverses

En incluant un fichier d'en-tête "stdlib.a", vous définissez également certaines macros (voir le chapitre 8 pour une exposition sur les macros), et les constantes d'usage courant. Elles comprennent ce qui suit :

```
;Codes ASCII courants
NULL            =       0
BELL            =       07              ;le caractère "cloche"
bs              =       08              ;le caractère retour (backspace)
tab             =       09              ;le caractère de tabulation
lf              =       0ah             ;le caractère de retour de ligne
cr              =       0dh             ;le caractère de retour chariot
```

En plus des constantes ci-dessus, "stdlib.a" définit aussi certaines macros utiles, par exemple ExitPgm, lesi et idxi. Ces macros contiennent les instructions suivantes :

```
;ExitPgm - Retour du contrôle à MS-DOS
ExitPgm macro
                mov     ah, 4ch          ;opcode de terminaison d'un programme DOS
                int     21h              ;appel au DOS
                endm

;LESI ADRS-
;       Charge ES:DI avec l'adresse de l'opérande spécifiée
lesi            macro   adrs
                mov     di, seg adrs
                mov     es, di
                mov     di, offset adrs
                endm

;LDXI ADRS-
;       Charge DX-SI avec l'adresse de l'opérande spécifiée
ldxi            macro   adrs
                mov     dx, seg adrs
                mov     si, offset adrs
                endm
```

Les macros lesi et ldxi sont particulièrement utiles pour charger des adresses dans es:di ou dx:si avant d'appeler diverses routines standard.

7.1.11 Encore plus !

La bibliothèque standard contient bien d'autres routines que ce chapitre ne mentionnera pas. Dès que vous en aurez le temps, lisez la documentation pour voir tout ce qui y est disponible. Nous n'avons mentionné ici que les routines que vous allez utiliser avec ce livre, de plus, il vous présentera de routines additionnelles au fur et à mesure qu'elles seront nécessaires.

7.2 Exemples de Programmes

Les programmes suivants montrent certaines opérations communes qui utilisent la bibliothèque standard.

7.2.1 Le fichier SHELL.ASM dans son essentiel

```
; Fichier d'exemple d'un SHELL.ASM de départ
;
; Randall Hyde
; Version 1.0
; 2/6/96
;
; Ce code source montre à quoi ressemble le fichier SHELL.ASM sans
; les commentaires superflus qui expliquent où placer les objets dans le
; fichier source. Vos programmes devraient commencer avec cette version
; du fichier SHELL.ASM. Les commentaires du SHELL que vous avez utilisé
; jusqu'à présent étaient utiles uniquement à *votre* compréhension
; et non pour quelqu'un voulant comprendre la logique des programmes que
; vous écrivez.

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

dseg            segment  para public 'data'

dseg            ends
```

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

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

Quit:           ExitPgm
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
```

7.2.2 E/S numériques

```
; Pgm7_2.asm - Numeric I/O.
;
; Randall Hyde
; 2/6/96
;
; La bibliothèque standard ne fournit pas de routines simples pour utiliser
; des entrées numériques. Ce code vous montrera comment lire des valeurs
; décimales et hexadécimales entrées par l'utilisateur à l'aide des
; routines Getsm, ATOI, ATOU, ATOH, IsDigit et IsXDigit.

                .xlist
                include stdlib.a
                includelib stdlib.lib
                .list
dseg            segment para public 'data'

inputLine       byte    128 dup (0)
SignedInteger   sword   ?
UnsignedInt     word    ?
HexValue                word    ?

dseg            ends

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

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

; Lire un entier signé du clavier.
InputInteger:   print
byte            "Input a signed integer value: ",0

                lesi    inputLine       ;es:di pointe sur inputLine
                gets                    ;Lit une ligne du clavier.

                mov     bx, -1
SkipSpcs1:      inc     bx
                cmp     inputLine[bx], ' ' ;Sauter tous les espaces.
```

```
                je      SkipSpcs1

                cmp     inputLine[bx], '-' ;Vérifier le signe moins.
                jne     NoSign
                inc     bx              ;Sauter si nombre négatif.

NoSign:         dec     bx              ;Reculer d'une place
TestDigs:               inc     bx      ;Passer au prochain caractère
                mov     al, inputLine[bx]
                IsDigit                 ;S'agit-il d'un nombre décimal ?
                je      TestDigs        ;Répéter si c'est le cas.

                cmp     inputLine[bx], ' ' ;Voir s'il termine par un
                je      GoodDec         ;caractère valide.
                cmp     inputLine[bx], ','
                je      GoodDec
                cmp     inputLine[bx], 0 ;La ligne d'entrée termine
                                        ;par zéro.
                je      GoodDec
                printf
                byte    "'%s' is an illegal signed integer. "
                byte    "Please re-enter.",cr,lf,0
                dword   inputLine
                jmp     InputInteger

; Ok, tous les caractères sont bons, faisons la conversion. Notez qu'ES:DI
; pointe encore sur inputLine.

GoodDec:                ATOI                    ;Effectuer la conversion
                mov     SignedInteger, ax       ;Enregistrer la valeur.

; Lire une valeur entière non signée.

InputUnsigned:  print
                byte    "Input an unsigned integer value: ",0
                lesi    inputLine
                gets            ;Lire une ligne de texte avec gets.

; Notez la ruse utilisée dans le code suivant. Il commence par un index de
; -2 et ensuite l'incrémente d'un. À chaque accès aux données dans cette
; boucle, le programme compare cet index avec l'emplacement inputLine[bx+1]
; qui initialise bx à zéro. Dans la boucle "TestUnsigned" ci-dessous, le
; code incrémente de nouveau bx si bien que bx contient à ce moment l'index
; dans la chaîne quand l'action prend place.

                mov     bx, -2
SkipSpcs2:      inc     bx
                cmp     inputLine[bx+1], ' ' ;Sauter les espaces.
                je      SkipSpcs2

TestUnsigned:   inc     bx              ;Passer au prochain caractère
                mov     al, inputLine[bx]
                IsDigit                 ;Voir si c'est un nombre décimal.
                je      TestUnsigned    ;Répéter si c'est le cas.
                cmp     inputLine[bx], ' '      ;Voir si on tombe
                je      GoodUnSigned            ;sur un caractère valide.
                cmp     inputLine[bx], ','
                je      GoodUnsigned
                cmp     inputLine[bx], 0 ;La ligne se termine par un zéro
                je      GoodUnsigned
                printf
                byte    "'%s' is an illegal unsigned integer. "
                byte    "Please re-enter.",cr,lf,0
                dword   inputLine
                jmp     InputUnsigned

; Ok, tous les caractères sont bons, on peut donc effectuer la conversion.
```

```
        ; Notez qu'ES:DI est encore en train de pointer sur inputLine.

        GoodUnsigned:   ATOU                    ;Effectuer la conversion
                        mov     UnsignedInt, ax ;Enregistrer la valeur.

        ; Lire une valeur hexadécimale du clavier.

        InputHex:               print
                        byte    "Input a hexadecimal value: ",0

                        lesi    inputLine       ;es:di pointe sur inputLine
                        gets                    ;Lire une ligne du clavier

                        ; La ligne suivante utilise la ruse qu'on connaît déjà
                        mov     bx, -2
        SkipSpcs3:      inc     bx
                        cmp     inputLine[bx+1], ' ' ;Sauter les espaces.
                        je      SkipSpcs3

        TestHex:                inc bx          ;Passer au prochain caractère
                        mov     al, inputLine[bx]
                        IsXDigit                ;Vérifier si c'est un nombre hexa.
                        je TestHex              ;Répéter si c'est le cas.
                        cmp     inputLine[bx], ' ' ;Tester s'il termine avec un
                        je      GoodHex         ;caractère valide.
                        cmp     inputLine[bx], ','
                        je      GoodHex
                        cmp inputLine[bx], 0    ;La ligne se termine par un zéro
                        je      GoodHex
                        printf
                        byte    "'%s' is an illegal hexadecimal value. "
                        byte    "Please re-enter.",cr,lf,0
                        dword   inputLine
                        jmp     InputHex

; Ok, tous les caractères sont corrects, faisons ici la conversion. Notez qu'ES:DI
; est encore en train de pointer sur inputLine.

        GoodHex:                ATOH            ;Effectuer la conversion
                        mov     HexValue, ax    ;Enregistrer la valeur.
                                                ;Afficher les résultats :
                        printf
                        byte    "Values input:",cr,lf
                        byte    "Signed: %4d",cr,lf
                        byte    "Unsigned: %4d",cr,lf
                        byte    "Hex: %4x",cr,lf,0
                        dword   SignedInteger, UnsignedInt, HexValue

        Quit:           ExitPgm
        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
```

7.3 Exercices de laboratoire

La Bibliothèque standard UCR pour les programmeurs d'assembleur est disponible et prête à l'usage, dans le cédérom qui accompagne ce livre[7]. Dans cet ensemble d'exercices de laboratoire, vous apprendrez comment installer cette bibliothèque standard et l'utiliser dans vos programmes.

7.3.1 Obtention de la bibliothèque

Une version récente de la bibliothèque accompagne ce livre. Elle est mise à jour périodiquement, donc il se peut que la version qui viene avec le livre soit déjà vieille. Pour la plupart des projets et des exercices que vous verrez dans ce livre, la version ici est probablement suffisante[8]. Cependant si vous voulez vous servir de la bibliothèque standard pour développer vos propres logiciels, vous devriez peut-être vous garder à jour.

L'endroit officiel où se trouve la dernière version de la bibliothèque est le site ftp.cs.ucr.edu[9] de l'université de Californie Riverside. Si vous avez un accès ftp via internet, vous pouvez télécharger la dernière copie de la bibliothèque standard à l'aide d'un compte anonyme ftp. Pour le faire, suivez ces étapes[10] :

- Exécutez votre programme ftp et connectez-vous à l'adresse ftp.cs.ucr.edu
- Quand le système vous demande un nom d'utilisateur, tapez *anonymous*.
- Quand le système vous demande un mot de passe, utiliser votre nom d'utilisateur complet (par exemple, quelque chose qui ressemble à *nom@machine.domaine*.
- Maintenant, ça y est. Passez au répertoire \pub\pc\pbmpcdir via la commande UNIX cd.
- Les fichiers de la bibliothèque sont compactés. Par conséquent, il faut les télécharger en mode binaire. Dans un programme ftp standard, il suffit d'entrer la commande *binary*. Si vous ne savez pas comment faire, consultez la documentation de votre programme ftp. **Le téléchargement par défaut se fait toujours en mode ASCII, mais si vous téléchargez le fichier de cette façon vous aurez probablement des problèmes à le décompacter ensuite.**
- Dans le répertoire \pub\pc\ibmpcdir vous trouverez plusieurs fichiers (généralement cinq, mais il peut y en avoir plus). En utilisant les commandes ftp appropriées (généralement get ou mget), copiez ces fichiers dans votre système.
- Quittez le serveur par la commande quit ou bye et quittez aussi votre programme ftp.
- Si vous avez téléchargé les fichiers en utilisant une machine UNIX, il vous faudra les transférer sur un PC travaillant avec DOS ou Windows. Pour les détails, consultez votre professeur ou votre administrateur UNIX.
- Voilà ! Vous venez de télécharger la dernière version de la bibliothèque standard.

Si vous n'avez pas d'accès internet ou vous rencontrez des problèmes en accédant au site ftp de l'UCR, vous trouverez une copie de la bibliothèque sur d'autres sites ftp, à partir d'autres postes ou chez un fournisseur de partagiciels (sharewares). Gardez à l'esprit cependant que les logiciels disponibles sur d'autres sites pourraient ne pas être à jour (vous pouvez parfois tomber sur des versions encore plus anciennes que celle venant avec ce livre).

Pour votre rapport de laboratoire : si vous avez réussi à télécharger la dernière version de la bibliothèque, décrivez les étapes que vous avez suivies. Décrivez également les fichiers que vous avez téléchargés. Si vous avez trouvé quelque fichier de type "readme", lisez-les et décrivez leur contenu.

7.3.2 Décompacter la bibliothèque standard

Afin d'occuper le moins d'espace disque possible et réduire le temps de téléchargement, la bibliothèque standard est compactée. Une fois que ses fichiers auront été téléchargés, vous devrez encore les décompacter pour pouvoir vous en servir. Notez aussi que même la version qui vient avec ce livre est compactée. Décompacter la bibliothèque standard est une action presque automatique. Il vous faudra simplement suivre les étapes suivantes :

[7] Évidemment le livre origianl n'était pas encore publié à l'époque de cette traduction, mais la bibliothèque se trouve actuellement ici https://dl.acm.org/doi/10.5555/134046.134047. Ce chapitre a été révisé en mai 2021, ndt.
[8] Une nouvelle version pourrait encore se révéler utile pour éliminer certaines bogues.
[9] Ce lien a été fourni en 1996 à l'heure de rédiger le livre en anglais. Cependant, le lien que vous avons donné à la note 7 est probablement le seul qui est actuellement à jour, car ftp.cs.ucr.edu ne répond plus en 2021, ndt.
[10] Ces instructions sont évidemment obsolètes, mais nous les incluons pour des questions de respect vers le texte original, ndt.

- Créez un répertoire dans votre disque dur et nommez-le "STDLIB".[11] Passez à ce répertoire en utilisant la commande CD.
- Copiez-y les fichiers que vous avez téléchargés ou que vous aviez déjà ailleurs.
- Exécutez la commande DOS "PATH=C:\STDLIB".
- Exécutez le fichier "UNPACK.BAT" en tapant "UNPACK" sur la ligne de commande du DOS.
- Attendez. Tout ce qui suit est automatique.
- Après avoir décompacté la bibliothèque, il vous faudra redémarrer votre machine, ou bien remettre la variable d'environnement PATH à sa valeur originale.

Si vous donnez pas à la variable PATH le nom du répertoire de la bibliothèque, le fichier UNPACK.BAT produira diverses erreurs et ne pourra pas bien décompacter les fichiers de la bibliothèque[12]. Il *effacera* même les fichiers compactés de votre disque. Par conséquent, assurez-vous de faire une copie de sauvegarde de ces fichiers dans une disquette ou dans un autre répertoire de votre disque avant de décompacter la bibliothèque. Ceci vous préservera de devoir éventuellement télécharger une autre fois les fichiers nécessaires.

Pour votre rapport : décrivez la structure des répertoires produite par le décompactage de la bibliothèque standard.

7.3.3 Utiliser la bibliothèque standard

Pendant le décompactage des fichiers de la bibliothèque standard, le programme UNPACK.BAT laisse une copie (complète) du fichier SHELL.ASM dans le répertoire STDLIB. Ce fichier devrait vous être familier, puisque vous l'avez déjà utilisé à maintes fois lors de projets antérieurs. Cette version particulière est une "version pleine", à savoir, elle contient différents commentaires expliquant où placer votre code et vos variables dans le fichier. Comme règle générale, le fait de laisser ces commentaires dans vos programmes dénote un très mauvais style de programmation. Une fois que vous avez lu ces commentaires et que vous avez compris la disposition du fichier SHELL.ASM vous devriez les effacer.

Pour votre rapport de laboratoire : incluez une version libre des commentaires superflus du fichier SHELL.ASM.

Au début de votre fichier SHELL.ASM vous trouverez les deux instructions suivantes :

```
include         stdlib.a
includelib      stdlib.lib
```

La première instruction indique à MASM de lire les définitions des routines de la bibliothèque standard à partir du fichier d'inclusion STDLIB.A (regardez le chapitre 8 pour une description des fichiers d'inclusion). La seconde instruction indique à MASM de passer le nom du fichier de code objet STDLIB.LIB à l'éditeur de liens de sorte qu'il peut lier votre programme avec le code de la bibliothèque standard. La nature exacte de ces deux instructions n'est pas très importante pour l'instant. Cependant, pour utiliser les routines de la bibliothèque standard, MASM a besoin de trouver ces deux fichiers au moment de l'assemblage et de l'édition de liens. Par défaut, MASM présume que ces deux fichiers se trouvent dans le répertoire courant chaque fois que vous assemblez un programme basé sur le modèle de SHELL.ASM. Puisque ce n'est pas le cas, vous aurez à exécuter deux commandes DOS spéciales pour indiquer à MASM où il peut trouver ces fichiers. Ces deux commandes sont :

```
set include=c:\stdlib\include
set lib=c:\stdlib\lib
```

Sans l'exécution de ces commandes, MASM produira une erreur (il ne pourra pas trouver ces fichiers) et plantera l'assembleur.

Pour votre rapport : Exécutez les commandes "SET INCLUDE=C:\" et "SET LIB=C:\"[13] et tentez ensuite d'assembler SHELL.ASM à l'aide de la commande DOS :

```
ml shell.asm
```

[11]Si vous êtes sur un ordinateur d'école, on pourrait vous demander d'utiliser un répertoire différent ou bien la bibliothèque standard pourrait simplement être déjà installée dans votre machine locale.
[12]Notez qu'ici, le fichier UNPACK.BAT utilise le programme LHA.EXE pour décompacter les fichiers. De nos jours, tout le processus de décompactage peut parfaitement être accompli avec WinZip, n.d.t.
[13]Ces commandes désactivent les valeurs courantes des chaînes LIB ou INCLUDE dans les variables d'environnement.

Signalez l'erreur dans votre rapport de laboratoire. Maintenant exécutez :

```
SET INCLUDE=C:\STDLIB\INCLUDE
```

Assemblez encore SHELL.ASM et notez toute erreur. Finalement, exécutez la commande

```
SET LIB=C:\STDLIB\LIB
```

et assemblez cette fois SHELL.ASM, sans erreur.

Si vous voulez éviter d'exécuter les commandes SET chaque fois que vous assemblez un programme, vous pouvez toujours ajouter ces commandes dans le fichier autoexec.bat. Ainsi, le système exécutera automatiquement ces commandes à chaque démarrage.

D'autres programmes (comme MASM ou Microsoft C++) peuvent également utiliser les commandes SET LIB et SET INCLUDE. Si ces commandes sont déjà présentes dans votre fichier autoexec.bat, il vous faudra ajouter vos commandes à la fin de celles déjà existantes, par exemple :

```
set include=c:\MASM611\include;c:\STDLIB\INCLUDE
set lib=c:\msvc\lib;c:\STDLIB\LIB
```

7.3.4 Les Fichiers de documentation de la bibliothèque standard

La bibliothèque standard UCR compte plusieurs centaines de routines ; sûrement plus de ce que ce chapitre peut raisonnablement documenter. La source "officielle" de la documentation est constituée par un ensemble de fichiers texte se trouvant dans le répertoire C:\STDLIB\DOC. Ces fichiers sont lisibles à l'aide de tout éditeur de texte et décrivent l'usage de chaque routine. Pour toute question à propos d'une sous-routine ou pour savoir quelles routines sont disponibles, vous devrez lire le fichier correspondant dans le sous-dossier.

La documentation consiste en une suite de fichiers organisés par classification de routines. Par exemple, un fichier décrit les routines de sortie, un autre décrit les routines d'entrée et un autre encore peut décrire les routines de traitement des chaînes. Le fichier SHORTREF.TXT fournit un guide rapide pour toute la bibliothèque. C'est un bon point de départ.

Pour votre rapport de laboratoire : incluez les noms des fichiers texte apparaissant dans le répertoire de documentation. Fournissez les noms de diverses routines documentées par chaque fichier.

7.4 Projets de programmation

1) Écrivez un programme se servant au moins de 15 routines de la bibliothèque standard UCR. Consultez l'annexe de ce livre et la documentation fournie avec la bibliothèque pour avoir des détails sur les différentes routines. Utilisez au moins cinq des routines qui ne sont pas documentées dans ce chapitre. Il vous faudra les apprendre par vous-même, à l'aide de la documentation.

2) Écrivez un programme qui démontre l'usage de chaque option de format que la routine PRINTF prévoit.

3) Écrivez un programme lisant 16 entiers signés entrés par l'utilisateur et gardez-les dans une matrice 4x4. Le programme devrait afficher la matrice 4x4 sous forme matricielle (par exemple, quatre lignes de quatre nombres avec chaque colonne soigneusement alignée).

4) Modifiez le programme du problème (3) de façon à pouvoir déterminer quel nombre demande le plus grand nombre majeur de positions d'affichage et affichez ensuite la matrice à l'aide de ce paramètre, plus une position supplémentaire, pour chaque nombre de la matrice. Par exemple, si la plus grande valeur est 1234, alors le programme affichera toutes les valeurs en se servant d'une largeur d'affichage de cinq positions.

7.5 Résumé

Ce chapitre a présenté plusieurs directives d'assembleur et pseudo-opcodes que MASM supporte. Il a aussi décrit brièvement certaines routines de la bibliothèque standard UCR pour les programmeurs 80x86. La

description de ce que la bibliothèque standard peut offrir n'est en aucun cas complète. Ce chapitre vous a tout de même fourni les informations nécessaires pour aller plus loin.

Dans ce chapitre, nous avons fait un usage étendu des diverses routines de la bibliothèque afin de vous aider à écrire des programmes assembleur de façon efficace. Bien que nous n'avons pas couvert toutes les routines disponibles, nous avons néanmoins décrit celles que vous utiliserez le plus souvent. Mais, au cours de ce, d'autres routines seront décrites, au fur et à mesure que le besoin se présentera. Voir :

7.6 Questions

1. Quel fichier devriez-vous utiliser pour écrire un programme qui utilise la bibliothèque standard ?
2. Quelle est la routine allouant de l'espace sur le tas ?
3. Quelle routine utiliseriez-vous pour afficher un caractère isolé ?
4. Quelle routine vous permettrait d'écrire une chaîne de caractères à l'écran ?
5. La bibliothèque standard ne fournit pas de routine pour lire un entier du clavier. Décrivez comment utiliser les routines GETS et ATOI pour accomplir cette tâche.
6. Quelle est la différence entre les routines GETS et GETSM ?
7. Quelle est la différence entre les routines ATOI et ATOI2 ?
8. Que fait la routine ITOA ? Décrivez les valeurs d'entrée et de sortie.