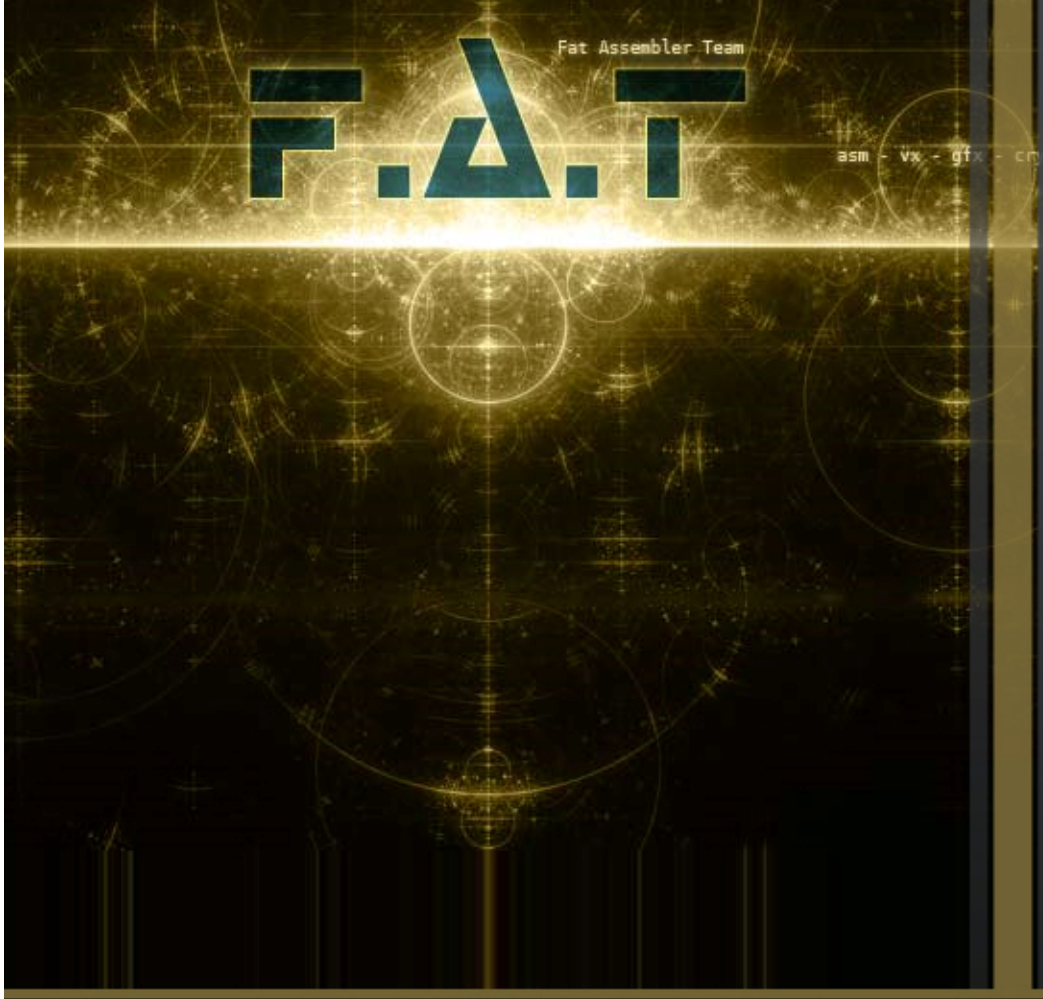


PAS A PAS VERS L'ASSEMBLEUR

```
.386
.model flat,stdcall
option casemap:none

include windows.inc

include user32.inc
includelib user32.lib

include kernel32.inc
includelib kernel32.lib

WinMain proto :DWORD,:DWORD,:DWORD,:DWORD

.data
ClassName    BYTE    "SimpleWinClass",0
AppName      BYTE    "L'assembleur Win32 est super !",0

.data?
hInstance HINSTANCE ?
CommandLine LPSTR ?

.code
start:
invoke GetModuleHandle, NULL
 mov hInstance,eax

invoke GetCommandLine
mov CommandLine,eax

invoke WinMain, hInstance,NULL,CommandLine, SW_SHOWDEFAULT
```

BY : LORD NOTEWORTHY

2K9

Table des matières

Bonjour et bienvenue dans ce Guide !

Je me nomme Lord Noteworthy et je suis l'auteur de ce Guide. Également le webmaster du site http://LordNoteworthy.c.la .

Alors ça y est ? Vous avez décidé de se lancer à la programmation en Assembleur mais vous ne savez pas trop par où commencer ? Bon, je vais essayer de vous donner quelques bases, ce qui croyez-moi, n'est pas évident ...

Mon obsession pour l'Assembleur m'a énormément incité à cerner ce qui est juste à savoir, car pour tout expliquer, cela demanderait quelque milliers de pages, écrites en tous petits caractères. Néanmoins, à partir de nombreux exemples, vous allez vous familiariser avec la syntaxe de ce langage et apprendre à travailler avec les instructions. Vous ferez connaissance de quelques principes de base sur l'architecture des systèmes informatiques, dans le cadre concret de la famille des processeurs IA-32 Intel et renforcerez vos compétences sur la syntaxe MASM. Enfin vous aurez un aperçu de l'architecture Win32. Il n'est nullement indispensable d'avoir une certaine expérience en programmation pour tirer parti de ce Guide, seulement de la patience, de la volonté, de la détermination d'apprendre et de comprendre. Certes si vous avez touché à tout autre langage de haut niveau que l'assembleur, vous allez vous sentir plus à l'aise, hors langages de balisage tels que l'HTML ne vous servira surtout pas.

Pour finir je tiens à préciser certaines petites choses. Tout d'abord ce texte, beaucoup de personne auraient pu l'écrire. De plus je ne suis pas parfait, il n'est pas improbable que ce Guide contienne quelques erratas, des incohérences ou d'autres petites choses qui m'auraient échappées, si vous en décelez une, merci de m'en faire part pour que je les corrige au plus vite possible.

Bon, eh bien sur ce, bonne lecture, et bon apprentissage à tous, en gardant à l'esprit la profonde maxime du regretté professeur Shadoko : « *La plus grave maladie du cerveau, c'est de réfléchir* ».

Introduction

Le début du document aborde des notions importantes en Assembleur et nécessaires pour bien comprendre la suite. Vous serez sans doute un peu déçu de ne pas faire des choses extrêmement puissantes immédiatement, mais patience : qui veut aller loin ménage sa monture. Ainsi, avant de nous plonger dans les arcanes de la programmation en Assembleur au sens large du terme, nous allons commencer par poser un certain nombre de bases.

Contrairement à ce qu'on dit souvent, un ordinateur ce n'est pas une machine très intelligente, c'est une machine qui donne l'illusion d'être intelligente car elle calcule très vite, à part ça, un ordinateur ça ne sait faire que des calculs très simple, encore pas avec tous les chiffres, mais uniquement deux chiffres, le **0** et le **1**. Le langage de chaque ordinateur est le langage **machine** où les instructions et les données sont représentées par des combinaisons de **bits**, les fameux zéros et uns. Lorsqu'un ordinateur traite du texte, du son, de l'image, de la vidéo, il traite en réalité des nombres.

Les langages de programmation ont considérablement évolué depuis les premiers calculateurs élaborés pour assister les calculs de trajectoire d'artillerie durant la seconde guerre mondiale. A cette époque là, les programmeurs travaillaient en langage machine, ce qui consistait à gérer des chaines très longues composées de 1 et de 0, ce fut un véritable travail de fourmi. Bientôt, les premiers assembleurs rendirent les instructions machine plus intelligibles et plus facile à utiliser. Dans les années soixante, pour faciliter encore la programmation, on a mis en place d'autres langages de programmation plus abordables, plus compréhensible par l'humain, ces langages sont dits **évolués** parce qu'ils donnent la possibilité aux programmeurs d'utiliser une syntaxe proche de la langue anglaise, avec des instructions et des mots comme *let variable = 10*.

Dans des discutions passionnées sur les compétences de chacun, il est rare que quelqu'un ne sorte pas l'idiotie suivante :

Le langage machine c'est plus rapide que l'assembleur ! Ou pire encore : L'assembleur, c'est génial, c'est plus rapide que le langage machine !

Rassurer vous, si vous avez à faire à ce genre de personnes, ne vous sentez pas ignorant, il s'agit de personnes qui ne savent pas de quoi elles parlent, et qui se permettent de porter des jugements. Le langage machine c'est exactement la même chose que l'assembleur, seule l'apparence diffère. Je m'explique. Si vous voulez mettre la valeur 5 dans EAX vous taperez :

```
mov EAX, 5  ; Cette instruction sert à placer la valeur 5 dans EAX
```

N'essayer pas d'en comprendre le contenu, vous n'avez même pas encore abordé la chapitre 1.

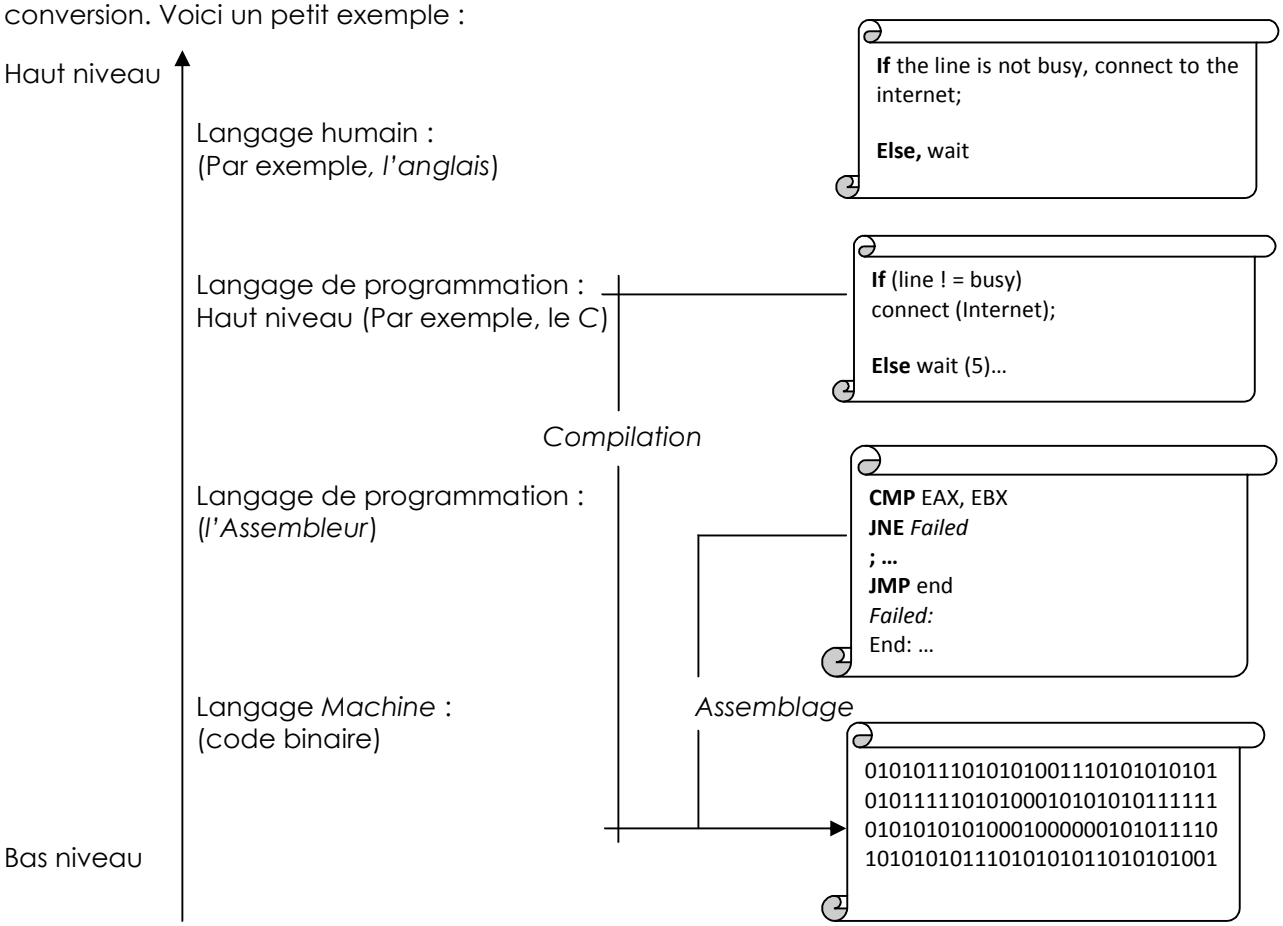
Cette instruction en assembleur sera quelque chose qui ressemble à ça en binaire : **1100110 10111000 101**

Quand votre microprocesseur rencontrera la valeur binaire **1100110 10111000**, il saura qu'il s'agit de l'instruction **MOV EAX, ?** Et que vous allez lui fournir à la suite une valeur immédiate qu'il devra mettre dans EAX. Si vous aviez tapé directement les bits un à un, le résultat aurait été exactement le même, vous auriez simplement beaucoup souffert pour rien, vous auriez alors programmé en langage machine. L'assembleur, se limite en fait à directement transcrire en code machine votre programme assembleur. L'assembleur ne modifiera en rien vos initiatives, la vitesse d'exécution est donc exactement la même, que le programme ait été programmé en langage machine bit par bit, ou en assembleur. Si par contre, vous programmez en *Pascal* ou en langage *C*, vous ne saurez pas ce que le compilateur fera de votre programme quand il le transformera en un programme machine directement compréhensible par le microprocesseur. Ces petites précision, devraient vous permettre de comprendre après pourquoi les instructions ont une adresse, elles sont pointées par **CS: EIP**, et c'est le microprocesseur qui les interprétera comme données ou instructions selon le contexte. Vous verrez, que, quand vous programmerez si par mégarde vous allez continuer l'exécution de votre programme dans des données, le microprocesseur se retrouvera alors avec des instructions incohérentes, et plantera assez vite.

Comme vous avez pu le constater, tout cela n'as rien de sorcier, le binaire que traite l'ordinateur avec facilité ne convient pas au programmeur qui commet souvent, par manque d'attention, des erreurs très difficiles ensuite à détecter, les nombres binaires se ressemblent, surtout après travaillé avec eux pendant plusieurs heures, prêtant à confusion et sans signification apparente. Il parait donc évident que ce type d'écriture est difficilement lisible pour nous, être humains.... Un programmeur pourra tenter de se souvenir de quelques codes binaires mais il pourrait investir ses efforts dans des tâches plus productives. L'idée

vient d'avoir un programme qui effectue la traduction langage assembleur ➔ langage machine, c'est ce programme qui est appelé **l'assembleur.**

Il est bien sur possible de traduire un programme écrit en assembleur à la main en remplaçant chaque instruction à son équivalente en binaire ; c'est ce qu'on l'appelle **l'assemblage manuel**, mais croyez moi nombreuses sont les difficultés associées à cette conversion. Voici un petit exemple :



D'emblée on comprend mieux l'intérêt de l'assembleur et des langages évolués qui soulagent les programmeurs d'un fardeau énorme, et qui présentent une facilité de programmation bien plus grande que le langage machine.

Assembleur, philosophie et atouts

L'assembleur abrégé **ASM** est le langage de programmation (un code de communication, permettant à un être humain de dialoguer avec sa machine) de plus **bas niveau**. Cela signifie qu'il est trop proche du matériel, qui oblige le programmeur à se soucier de concepts proches du fonctionnement de la machine, comme la mémoire.

Heureusement, ils existent des langages **hauts niveau,** à l'instar de *Visual Basic*, *Delphi*, *C++*, *Java*..., ce sont les langages les plus utilisés de nos jours, qui permettent au programmeur de s'abstraire de détails inhérents au fonctionnement de la machine. Il permet de manipuler des

concepts bien plus élaborés. En fait, les avantages qui offrent les langages haut niveau sont énormes: Ils assurent une meilleur portabilité, c'est-à-dire que l'on pourra les faire fonctionner sans guère de modifications sur des machines ou des systèmes d'exploitation différents, l'écriture d'un programme est beaucoup plus facile et prend donc beaucoup moins de temps, la probabilité d'y faire des fautes est nettement plus faible, la maintenance et la recherche des bugs sont grandement facilités.

Si nous résumons, le terme "Assembleur" désigne tour à tour deux choses différentes, mais apparentées : le langage de programmation de plus bas niveau accessible facilement à un humain et un logiciel transformant un fichier source contenant des instructions, en un fichier exécutable que le processeur peut comprendre.

Avantages et inconvénients de l'assembleur

Evidemment, rien n'est parfait. Jetons un petit coup d'œil sur le revers de la médaille:

- Le programme est long et fastidieux à écrire.
- Les programmes écrits en assembleur sont très peu portables vers une autre architecture, existante ou future.
- Programmer en assembleur est réservé aux développeurs chevronnés.
- Réaliser un programme complet avec demande énormément d'efforts et d'expérience.
- Difficulté de localisation d'erreurs au cours de la mise au point du programme.
- Pendant longtemps, la principale préoccupation des programmeurs était de concevoir des applications très courtes pouvant s'exécuter rapidement. Il faut dire que la mémoire coûtait chaire, tout comme le traitement de la machine. Avec la miniaturisation des ordinateurs, l'augmentation de leur performance et la chute des prix, les priorités ont changé. A l'heure actuelle, le coût de développement dépasse largement celui d'un ordinateur, l'important de faire des programmes performants, bien construis et faciles à mettre à jour, on n'a plus besoin d'utiliser l'assembleur…

Comme tout langage de programmation, l'assembleur a ses inconvénients, mais aussi ses avantages:

- Tous les langages de programmation sont des héritiers plus ou moins directs de ce langage élémentaire qu'est l'assembleur, ce dernier peut vous donner la logique nécessaire pour programmer en n'importe quels langages, ce langage trouve sa place dans l'ensemble des applications, sans oublier que n'importe quel programme, écrit dans n'importe quel langage est finalement traduit en langage machine pour être exécuté.
- On a la possibilité de faire tout et n'importe quoi avec la mémoire. L'ASM n'a pas de limite et de sécurité. Autant il peut être utile, autant il peut détruire ;)
- Les programmes faits en ASM sont plus petits, plus rapides et beaucoup plus efficaces que ceux fait avec des compilateurs, pourvu qu'il soit bien écrit. Une des raisons de cette rapidité d'exécution réside dans le compilateur. En fait, quand on code de quoi en C++, le compilateur doit convertir toutes les instructions en assembleurs, puis par la suite le convertir en langage machine tandis que si on code en ASM, le compilateur a juste besoin de le convertir en langage machine. Jusque là ça change rien puisque les deux finissent en langage machine mais les compilateurs de C++, quand ils convertissent des trucs en assembleur, ils rajoutent pleins d'instructions inutiles au fonctionnement de base du programme (c'est comme faire un document html avec un éditeur html). Tout le code qui rajoute va ralentir l'exécution du programme.

- C'est le seul langage permettant de modifier un programme compilé dont on n'a pas les sources (utiles pour le *Reverse Engineering* entre autres).
- On peut l'insérer dans des langages de plus haut niveau pour les parties nécessitant d'être optimiser, d'ailleurs bon nombre de langages de programmation permettent, afin de combler leurs lacunes (vitesse d'exécution, accès aux périphériques, etc.), d'appeler des routines écrites en assembleur.
- Vous voulez apprendre quelque chose de nouveau, que vous allez trouver du fun à le faire et l'applique et que vous n'allez jamais le déplorer …
- Et, cerise sur le gâteau, le fait que l'ASM est très compliquée n'est pas vraiment vrai.

L'assembleur se révèle être un langage bien à part, son apprentissage poursuit double objectif, d'une part avoir une compréhension plus profonde de la façon dont fonctionne l'ordinateur, d'autre part vous donnez des atouts considérables pour la compréhension et la maîtrise de tous les autres langages de programmation.

Que programmer en Assembleur ?

Il est important de savoir ou et quand il faut utiliser l'ASM afin d'avoir un programme le plus performant tout en conservant des temps de développement les plus courts possibles, l'ASM pourra vous être utile dans plusieurs cas :

- Tout ce qui doit être optimisé au niveau de la taille, par exemple pour programmer un virus...
- Lorsque la vitesse d'exécution est un facteur critique et que la moindre microseconde est précieuse, ou ne peut pas être fait avec autre chose que de l'assembleur, par exemple un driver de disque dur, un secteur de boot, ou une routine d'affichage de polygones/triangles en 3D...

Bref, l'assembleur est au programmeur ce que la clé de 12 est au mécano, le micro au chanteur, la plume à l'écrivain, ce que le sens de la chute était à Pierre Desproges... ^^

Chapitre 1 : Notions de base

Vu que l'assembleur est qualifié comme étant le langage de programmation le plus bas niveau, il dépend donc fortement du type de processeur. Ainsi il n'existe pas un langage assembleur, mais un langage assembleur par type de processeur. Il est donc primordial d'avoir un minimum de connaissances techniques. Nous verrons ensemble certains points essentiels à la bonne compréhension de la suite d'informations développées dans ce guide à savoir les termes techniques, les conventions, les bases numérales, les différents types de représentations, la mémoire et le microprocesseur. C'est là en effet que se trouvent les principales difficultés pour le débutant. Ne soyez pas rebuté par l'abstraction des concepts présentés dans les premiers paragraphes : il est normal que durant la lecture, beaucoup de choses ne soient pas claires à travers vos esprits. Tout vous semblera beaucoup plus simple quand nous passerons à la pratique dans le langage assembleur. Ces informations permettront de rafraîchir ou d'enrichir vos connaissances dans ce domaine.

Les systèmes de numération

Nous allons aborder un aspect très important de la programmation en assembleur : le système de numération. En informatique et en électronique numérique, on emploie plusieurs formes de numération. Les quatres bases les plus employées sont les suivantes : **binaire**, **octale**, **décimale**, **hexadécimale**. Voyons en détail ces différentes bases.

Décimale

Depuis la nuit des temps, l'homme a eu besoin de compter et de calculer. Selon les civilisations, divers systèmes de numération on été mis en place puis abandonnés.

A l'heure actuelle, nous utilisons le système de numération décimal. Ce système s'est imposé en Europe à partir du 10ème siècle. Aujourd'hui le système décimal est quasiment universel. L'idée de ce système réside dans le fait que nous possédons dix doigts, l'écriture décimale nécessite donc l'existence de 10 chiffres qui sont *0, 1, 2, 3, 4, 5, 6, 7, 8, et 9*.Lorsque nous écrivons un nombre en mettant certains de ces chiffres les uns derrière les autres, l'ordre dans lequel nous mettons les chiffres est capital. Ainsi, par exemple, 4678 n'est pas du tout le même nombre que 8647. Et pourquoi ? Quelle opération, quel décodage mental effectuons nous lorsque nous lisons une suite de chiffres représentant un nombre ?

Le problème est que nous sommes tellement habitués à faire ce décodage de façon instinctive que généralement nous n'en connaissons plus les règles. Mais ce n'est pas très compliqué de les reconstituer. Considérons par exemple le nombre 4678. Il est composé des chiffres des milliers (4), des centaines (6), des dizaines (7), et des unités (8).

On peut donc écrire :

- 4687 = 4000 + 600 + 80 + 7

Ou bien encore :

- 4687 = 4 × 1000 + 6 × 100 + 8 × 10 + 7

D'une manière légèrement différente, même si cela paraît un peu bébête :

- 4687 = (4 × 1000) + (6 × 100) + (8 × 10) + 7

Arrivé à ce stade de la compétition, les matheux notent la ligne ci-dessus à l'aide du symbole de puissance**.** Cela donne :

- $4687 = (4 \times 10^3) + (6 \times 10^2) + (8 \times 10^1) + (7 \times 10^0)$

En rappel, tout nombre élevé à la puissance 0 est égal à 1.

Et voilà, nous y sommes. Un nombre peut donc être décomposé sous la forme de la somme de chacun des chiffres qui le composent, multipliés par la dimension de la base à l'exposant de leur rang. Cette méthode n'est pas valable uniquement pour les nombres décimaux.

Binaire

Un ordinateur n'ayant pas de doigts, compter en base décimale ne lui est pas particulièrement adapté. La base **2**, plus connue sous le nom de la base **binaire**, l'est par contre beaucoup plus. Pour des raisons technologiques, l'ordinateur ne peut traiter qu'une **information** codée sous forme **binaire**.

Le courant électrique **passe** ou **ne passe pas**, qu'on peut interpréter en **0** ou **1**.
Dans la réalité physique il n'y a pas de 0 de 1 qui se balade au cœur de l'ordinateur. Le choix du 0 et de 1 est une pure convention, une image commode, que l'on utilise pour parler de toute **information binaire**, et on aurait pu choisir n'importe quelle paire de symboles à leur place, quelque chose qui ne peut avoir que de états : par exemple, ouvert ou ferme, vrai ou faux, militaire ou civil, libre ou occupé…

En base 2, tout nombre s'écrit à l'aide des deux chiffres 0 et 1, appelés **bits** (**BI**nary digi**T**). C'est la plus petite unité d'information manipulable par une machine numérique.
Or un bit n'est pas suffisant pour coder toutes les informations que l'on souhaitera manipuler. On va donc employer des groupements de plusieurs bits.

- Un groupe de **4 bits** s'appelle un **quartet (Nibble** en anglais**)**.
- Un groupe de **6 bits** s'appelle un **hexet.**
- Un groupe de **8 bits** s'appelle un **octet (Byte)**.
- Un groupe de **16 bits**, soit **deux octets**, est appelé un **mot (Word)**.
- Un groupe de **32 bits**, soit **deux mots**, est appelé un **double mot (Dword)**.
- Un groupe de **64 bits**, soit **deux doubles mots**, est appelé un **(Qword)**.

Notez l'utilisation d'un B majuscule pour différencier Byte et bit.

Il existe des unités de manipulation pour les très grandes valeurs. Même si elles ne sont pas employées par les micro-ordinateurs actuels, il est bon d'en connaître l'existence.

- Un **kilo-octet** (Ko) = 1024 octets.
- Un **Mégaoctet** (Mo) = 1024 Ko = 1048576 octets.
- Un **Gigaoctet** (Go) = 1024 Mo = 1073741824 octets.
- Un **Téraoctet** (To) = 1024 Go = 1 099511627776 octets.

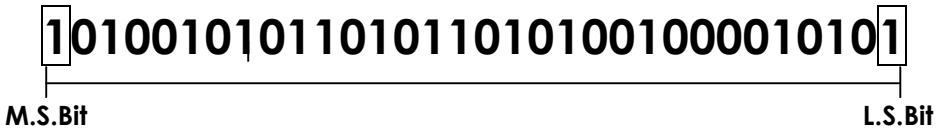
Les bits sont généralement regroupés par 4 ou 8 bits. Dans la représentation binaire, on distingue les deux bits d'extrémité d'un nombre :

- Le **bit de poids fort** (**Most Significant Bit**, ou **MSB**) est le bit, dans une représentation binaire donnée, ayant la plus grande valeur, celui le plus à gauche. Exactement comme dans le nombre 189, c'est le chiffre le plus à gauche qui à le plus de poids, la valeur la plus forte.
- Le **bit de poids faible** (**Least Significant Bit**, ou **LSB**) est pour un nombre binaire le bit ayant dans une représentation donnée la moindre valeur, celui le plus à droite. Exactement comme dans le nombre 189, c'est le chiffre le plus à droite qui à le moins de poids, la valeur la plus faible.

De même, si l'on découpe un nombre en deux, le premier paquet à gauche est appelé paquet de poids faible, le second paquet à droite est appelé paquet de poids fort. Ainsi un **mot** est composé de **2 octets :**

- 1 octet de poids fort (**Least Significant Byte**).
- 1 octet de poids faible (**Most Significant Byte**).

Si on fait un schéma de ce qu'on vient de dire, on se retrouve avec ce qui suit :



Les opérations arithmétiques simples telles que l'**addition**, la **soustraction**, la **division** et la **multiplication** sont facile à effectuer en binaire.

Octale

*« **Compter en octal, c'est comme compter en décimal, si on n'utilise pas ses pouces** »* - ***Tom Lehrer***. Comme vous le devinez sûrement, tous comme on peut dire que le binaire constitue un système de codage en base 2, le décimal un système de base 10, vous l'aurez compris l'octal est un système de base 8. Ce système utilise 8 symboles : **0, 1, 2, 3, 4, 5, 6, 7**. Il n'est plus guère employé aujourd'hui, puisqu'il servait au codage des nombres dans les ordinateurs de première génération. Le système octal a cédé la place au système hexadécimal. Etant donne que ce système de numération n'est plus vraiment employé, on ne s'attardera pas d'avantage sur le sujet !

Hexadécimale

La notation hexadécimale consiste à compter non pas en base 2, 8, ou 10, mais en base **16**. Cela nécessite d'avoir 16 chiffres.
En décimale vous comptez comme ça :

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, …

Et bien en hexadécimale on compte de cette manière :

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, …

Pour les dix premiers, on n'a été pas cherché bien loin : on a recyclé les dix chiffres de la base décimale, là il nous manque encore 6 chiffres, plutôt qu'inventer de nouveaux symboles, on a alors convenu de les représenter par les premiers lettres de l'alphabet. Ainsi par convention, A vaut 10, B vaut 11 et ainsi de suite jusqu'à F qui vaut 15.
La raison pour laquelle l'hexa est utile est que la conversion entre l'hexa et le binaire est très simple. Avec l'avancement de la technologie, les nombres binaires sont devenus de plus en plus longs et rapidement incompréhensibles. Le système hexadécimal est venu pour freiner l'accroissement en longueur des nombres binaires, il fournit une façon beaucoup plus compacte pour représenter le binaire, vous le voyez l'hexadécimal comporte de nombreux avantages, on peut représenter **16** valeurs avec seulement **un chiffre**, alors qu'il en faut **deux** pour la décimal, et **quatre** pour le binaire. Voyons voir cela de plus près, avec 4 bits, nous pouvons coder **2 × 2 × 2 × 2 = 16** nombres différents. En base seize, 16 nombres différents se représentent avec un seul chiffre, de même qu'en base 10, dix nombres se représentent avec un seul chiffre.

Afin d'éviter tout risque d'erreur entre bases, il est recommandé d'écrire un :

- "**b**" à la fin d'un nombre binaire.
- "**d**" à la fin d'un nombre décimal (base par défaut).
- "**o**" à la fin d'un nombre octal.
- "**h**" à la fin d'un nombre hexadécimal, toujours en minuscules, afin de ne pas confondre avec les "chiffres" hexadécimaux B et D.
  Par exemple : 11011**b** = 27**d** = 33**o** =1B**h**

Et enfin, la numération binaire, comme les autres systèmes de numération, n'oblige pas la représentation des valeurs nulles ou inutiles, par exemple :

- Le nombre 220**d** = 11011100**b** est représentable sur 8 bits.

- Le nombre 16**d** = 00010000**b** est représentable sur 5 bits, ce qui donne réellement en omettant les chiffres zéro inutiles 10000**b**.

Le tableur ci-dessous montre l'équivalence de représentation de nombre allant de 0 à 17 :

Décimal	Binaire	Octale	Hexadécimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
17	10001	21	11
…	…	…	…

Les conversions entre bases numérales

Il est indispensable de savoir faire ce genre de conversion pour être capable de se débrouiller avec un simple crayon à pointe de graphite et une feuille de papier (on n'a pas toujours sous la main une calculatrice équipée de ce genre de fonction)…

Décimale ➔ Binaire

Prenons un nombre au hasard : **13***d* = **1101***b*.
Bain, comment le 13 devient t'il 1101? Et bien c'est un petit peu comme si l'ordinateur rangeait treize boules dans une boite, une boite avec des colonnes, dans la première colonne, en peut mettre qu'une seule boule, dans la deuxième : deux boules, dans la troisième : 4 boules, dans la quatrième : 8 boules.

_____		_____		_____		_____
8	*4*	*2*	*1*

Attention l'ordinateur commence toujours par la plus grande colonnes et il ne fait que des colonnes pleines.

- Chaque fois qu'on a une colonne **pleine** c'est un **1**.
- Chaque fois qu'on a une colonne **vide** c'est un **0**.

Voyons ce que ca donne avec 13 boules :

- La colonne de 8 pas de problème, on peut la remplir, ca donne un 1, il nous reste cinq boules.
- La colonne de 4 on peut la remplir, c'est un 1, il nous reste une boule.
- La colonne de 2 on ne peut pas la remplir, c'est un 0.
- La colonne de 1 on peut la remplir c'est un 1.

1		**1**		**0**		**1**
8	*4*	*2*	*1*

Donc, le nombre **13***d* correspond à l'information binaire **1101***b*.

Jusqu'à présent, nous avons vu différents moyens de codage de nombres. Mais il est nécessaire de savoir également comment coder des caractères alphanumériques. Par exemple lors de leur entré à partir d'un clavier. La mémoire de l'ordinateur conserve toutes les données sous forme numérique. Il n'existe pas de méthode pour stocker directement les caractères. Dans les années 60, le code ASCII (*American Standard Code for Information Interchange*), qui est actuellement le plus connue et le plus largement compatible, est adopté comme standard, chaque caractère possède donc son équivalent en code numérique, par exemple la lettre 'M' correspond au nombre 77*d*. Un nouveau code, plus complet, qui supplante l'ASCII est l'**Unicode**. Une des différences clés entre les deux codes est que l'ASCII utilise un octet pour encoder un caractère alors que l'Unicode en utilise deux (ou un mot). Par exemple, l'ASCII fait correspondre l'octet 41*h* (65*d*) au caractère majuscule 'A'; l'Unicode y fait correspondre le mot 0041*h*. Comme l'ASCII n'utilise qu'un octet, il est limité à 256 caractères différents au maximum. L'Unicode étend les valeurs ASCII à des mots et permet de représenter beaucoup plus de caractères. C'est important afin de représenter les caractères de tous les langages du monde. J'ai inclus la liste des différents codes ASCII ainsi que leur signification en annexe

Mais comment tu fais pour faire 77 !! Regarde bien ton boulier, il n'a que (8 + 4 + 2 + 1) ça fait 15. Bien cette fois ci, il suffit d'agrandir le boulier et voilà:

128	*64*	*32*	*16*	*8*	*4*	*2*	*1*

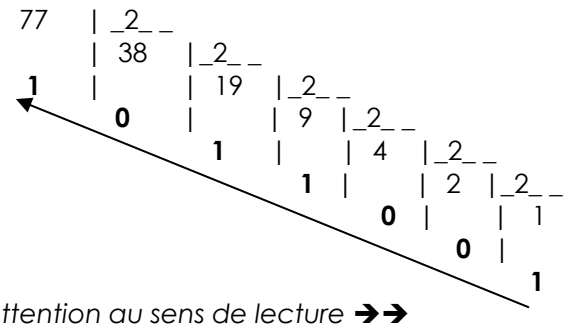
- Avec 77 boules, on ne peut pas remplir la colonne de 128, c'est un 0.
- On rempli en revanche la colonne de 64, c'est un 1, il nous reste 13 boules.
- Pas assez pour remplir la colonne de 32, c'est un 0.
- Pas assez non plus pour rempli la colonne de 16, c'est un 0.
- Assez pour remplir la colonne de 8, c'est un 1, il nous reste deux boules.
- Je remplis la colonne de 4, là c'est un 1.
- Pas assez pour remplir la colonne de 2, c'est un 0.
- Et comme j'ai plus qu'une seule boule, je remplis la dernière colonne, c'est un 1.

0		**1**		**0**		**0**		**1**		**1**		**0**		**1**
128	*64*	*32*	*16*	*8*	*4*	*2*	*1*

Donc, la lettre 'M' correspond à l'information binaire **01001101***b*.

Il existe une autre méthode classique pour transformer un nombre du système décimale dans un autre système, par **divisions successives.**
La méthode de décomposition par divisions successives consiste à diviser le nombre plusieurs fois (si nécessaire) dans la **base choisie** jusqu'à obtenir un quotient nul. Tant qu'il s'agit d'une conversion décimale ➔ binaire, on divise par 2. Les restes successifs des divisions, pris dans leur ordre inverse, forment le nombre désiré.

Reprenons le même exemple. La division entière de 77 par 2 donnes :



Attention au sens de lecture ➔➔

L'écriture binaire de **77***d* est donc : **01001101***b*.

Binaire ➔ Décimale

Une fois que le système binaire est bien compris, il est facile de transposer ses principes pour comprendre le système binaire. Je rappelle, pour trouver la valeur d'un nombre binaire, il faut, à l'instar des bits décimaux, multiplier la valeur du bit par la valeur de 2 exposé par la position du bit moins un. Cependant, dans ce cas-ci, puisque le bit peut seulement être 1 ou 0, le calcul revient plus à une décision d'inclure la valeur ou non dans le nombre binaire qu'à une multiplication. Ainsi, pour trouver la valeur d'un nombre binaire vous pouvez utiliser le tableau suivant :

0	1	0	0	1	1	0	1
2^7	**2^6**	**2^5**	**2^4**	**2^3**	**2^2**	**2^1**	**2^0**

D'où **01001101***b* correspond à **: $(0 \times 2^7) + (1 \times 2^6) + (0 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (0 \times 2^0) = 77$***d*.

En suivant les principes énoncés dans la conversion binaire. Cependant il ne s'agit plus, de puissances de 2 mais de puissances de 16 puisque la base est 16.

Binaire ➔ Hexadécimale

Pour convertir un nombre en hexadécimale, il y'en a deux méthodes l'une consiste à faire un grand détour, en repassant par la base décimale. L'autre méthode consiste à faire le voyage direct du binaire vers l'hexadécimale.

- **La première méthode :**

Prenons un octet au hasard : 1011 1101. A partir de ce qu'on vient de voir, notre 1011 1101 deviendront : **$(1 \times 2^7) + (0 \times 2^6) + (1 \times 2^5) + (1 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) =$ 189***d*. De là, il faut repartir vers la base hexadécimale.

Décimale ➔ hexadécimale

On utilise la méthode **des divisions successives**, même principe que pour le système binaire, sauf que l'on divise par 16.
On note les restes des divisions successives puis on lit ces restes en remontant.

Exemple :

```
189   | _16_
      |  11  |_16_
 13   |      |  0
      |  11  |
```

On y est le nombre s'écrit donc en hexadécimal : **BD***h*

Hexadécimale ➔ Décimale :

Inversement il est aisé de passer d'un nombre en hexadécimal à un nombre décimal par **multiplications successives**, en suivant les principes énoncés dans la conversion binaire. Cependant il ne s'agit plus, de puissances de 2 mais de puissances de 16 puisque la base est 16.

Exemple : **BD***h* = **$(11 \times 16^1) + (13 \times 16^0)$**
= 176 + 13
= 189*d*

- **La deuxième méthode :**

Plus rapide consiste à découper le nombre binaire en quartets, à partir de la droite, puis remplacer chaque quartet par le symbole hexadécimal correspondant

Dans l'exemple précédent, on peut remarquer que **1011** **1101** en binaire, nous conduit à **B** **D** en hexadécimal.

1011, c'est 8+2+1, donc 11.

1101, c'est 8+4+1, donc 13.

Le nombre s'écrit donc en hexadécimal : **BD**. C'est la même conclusion qu'avec la première méthode.

Hexadécimale ➔ Binaire

Le passage de l'hexadécimal en binaire peut se faire de la manière inverse, en convertissant les chiffres qui composent le nombre hexadécimal en leur équivalent binaire. Notez que les 0 de tête des 4 bits sont importants ! Si

Exemple : **BD***h* = **B** **D** On trouve donc que **BD***h* = **10111101***b*.
1011 **1101**

Y'a t'il des nombres négatifs en binaire ?

Bien sûr que oui, jusqu'à présent nous avons parlé de mots binaires sans en spécifier le signe. Un nombre tel qu'on à appris à l'écrire est appelé un nombre **non signé** (**unsigned** en anglais), il est toujours positif. Au contraire, maintenant nous allons apprendre à écrire des nombres qui peuvent représenter des valeurs soit positifs, soit négatifs. On dira qu'ils sont signés (**signed**). Un nombre signé n'est donc pas forcément négatif.

Chaque bit d'un octet peut occuper deux états, il y'a donc dans un octet :

2 × 2 × 2 × 2 × 2 × 2 × 2 × 2 = 2^8 = 256 possibilités. Donc il est possible de coder des nombres allant de **00000000** à **11111111** en binaire, soit **00** à **FF** en hexadécimal et **0** à **255** en décimal.

Considérons l'opération 0 – 1 = –1 en décimal, elle consiste à retrancher 1 de 0, et tentons de la réaliser en binaire.

```
  0                0  0  0  0  0  0  0  0
- 1              - 0  0  0  0  0  0  0  1
------     =     ---------------------------------------------------------------------------
= -1               1  1  1  1  1  1  1  1
```

Donc **–1***d* sera représenté par **FF***h*.

Recommençons et soustrayons 1 à -1 :

```
- 1                1  1  1  1  1  1  1  1
- 1              - 0  0  0  0  0  0  0  1
------     =     ---------------------------------------------------------------------------
= -2               1  1  1  1  1  1  1  0
```

So, **-2***d* sera représenté par **FE***h***.**

Ceci dit cette méthode de détermination de la représentation binaire d'un nombre négatif n'est pas très commode, c'est pourquoi nous allons introduire la notion de notation en **complément à 2**. La procédure est la suivante :

- Si le nombre est positif, conversion identique.
- Si le nombre est négatif:
  1. On convertit d'abord sa valeur absolue en binaire.
  2. On inverse tous les bits du nombre: les 0 deviennent des 1, les 1 deviennent des 0 **(complément à 1)**.
  3. On ajoute 1 au résultat.

Par exemple, pour représenter -2 sur 8 bits:

- 2 s'écrit **00000010***b*
- -2 s'écrit donc **11111101***b* + **1***b* = **11111110***b*.

Le résultat trouvé est bien le même que précédemment. Nous pouvons vérifier que l'opération +2 - 2 donne bien zéro sur **8 bits**. Le 1 situé à gauche n'est pas pris en compte. Par cette méthode il est donc possible de coder les nombre décimaux compris entre **-128 et 127**. Le bit du poids fort de chaque octet est égal à :

- **0** si le nombre est **positif**.
- **1** si le nombre est **négatif**.

Notons que cette notation en complément à deux n'est pas obligatoire. C'est au programmeur de décider si les nombres qu'il utilise sont compris entre 0 et 256 ou bien entre -128 et + 127.

Opérations Arithmétiques

On peut tout comme avec le décimal effectuer des opérations standards tout comme l'**addition**, la **soustraction** la **division** et la **multiplication.**

L'addition

Un rappel sûrement inutile mais qui clarifie des choses... Quand vous faites une addition en décimal, vous faites la somme des chiffres se trouvant dans une même colonne :

- Si la somme est inférieure à 10, alors vous posez le résultat obtenu et passez à la colonne suivante.
- Si la somme est supérieure à 10, alors vous posez le chiffre des unités et gardez en retenue le chiffre des dizaines.
- Si vous faites la somme de 2 nombres, alors la retenue ne pourra être supérieure à 1.

Le principe est exactement le même en binaire, on commence à additionner les bits de poids faible, les bits de droite, puis on a des retenues, lorsque la somme de deux bits poids dépasse la valeur de l'unité la plus grande (dans le ca binaire : 1), cette retenu est reporté de poids plus fort suivant...
En suivant ce principe, on obtient en binaire la table d'addition suivante :

	Résultat	**Report**
0 + 0 =	0	0
0 + 1 =	1	0
1 + 0 =	1	0
1 + 1 =	0	1

Pour illustrer ce principe, effectuons par exemple les opérations suivantes :

```
          1                        1   1   1   1
    1     0     0     1                0   1   1   1
+   0     1     0     1          +     1   0   0   1
-------------------------      -------------------------
    1     1     1     0          1   0   0   0   0
```

De même en hexadécimal, il suffit de procéder comme dans le système décimale, ainsi nous obtenons :

```
    A     2          { Ah = 10d
+   3     C          { Ch = 12d
-----------------
=   D     E
```

Désolé cet exemple ne comporte pas de retenues, les nombres obtenus étant plus grand que 16 nous additionnons 4 et nous effectuons un report, en voici un qui en comporte:

```
    2     F          { Fh = 15d
+   5     3
-----------------
          1
=   7     18
            4
-----------------
=   8     22
-----------------
=   8     2
```

Ainsi **2F***h* + **53***h* = **82***h*. Nous pouvons constater qu'une table d'addition en hexadécimal ou une calculatrice avec la fonction BIN, DEC, OCT, ET HEX se relève un outil précieux.

La soustraction

Pour effectuer une soustraction en binaire, nous procédons de la même façon que pour une soustraction en décimale, il y'a quatre cas possibles :

	Résultat	**Emprunt**
0 – 0 =	0	0
0 - 1 =	1	1
1 - 0 =	1	0
1 - 1 =	0	0

Voyons ce que ça donne :

```
          1                                  1   1
    1     0     0     0                1     0   1   0
```

```
-       0       1       1       1                       -       1       0       0       1
-----------------------------------------               -----------------------------------------
        0       0       0       1                               0       0       0       1
```

En hexadécimal, ça marche comme en décimal. La seule difficulté provient de ce que l'on n'apprend pas la table d'addition en hexadécimal. B+F=1A par exemple. Il faut donc réfléchir un peu plus qu'en décimal

```
        F       9
-       A       2
-------------------------
        5       7
```

Vous avez vu ? Ce n'est pas si dur ! On pourrait, de la même façon, illustrer le fonctionnement d'autres opérations arithmétiques en binaire, mais là ce n'est pas notre objectif. C'est tout ce qu'il faut retenir jusqu'ici.

Astuce

Dorénavant, ne vous vous souciez pas de faire ces calculs à la main. La solution est beaucoup plus simple que cela. Procurez vous de votre calculatrice Windows. Pour ce faire, le moyen le plus rapide est de cliquer sur touche "*Windows*" + *R*, tapez "*calc*" et cliquer sur *OK*. Un raccourci est aussi disponible, pour les amants de la souris, dans le menu *tous les programmes*, dans le sous-menu *accessoires*. Une fois que vous avez démarré la calculatrice, rassurez-vous que vous soyez en mode scientifique, *(Affichage/Scientifique)*. Vous pouvez choisir le système numérique à l'aide des 4 boutons radios encerclés (**Hex** = Hexadécimale, **Déc** = décimal, **Oct** = Octale et **Bin** pour binaire, ensuite, cliquez sur le système numéral dans lequel vous voulez avoir la conversion.

Cela fait déjà une bonne base pour commencer, vous avez appris les systèmes de numération. Vous devriez être en mesure d'effectuer de simples conversions facilement.

Chapitre 2 : Organisation de l'Ordinateur

Et c'est parti pour l'assembleur, je rassemble mon cerveau et prend mon courage à deux pattes... en effet, il va m'en falloir, car après les explications de bases, assez facile, et que vous connaissiez sans doute déjà, on va s'attaquer à un gros morceau très important dans la programmation en assembleur, puisqu'il s'agit de la gestion de la mémoire. Il est vital de comprendre comment elle est organisée, et comment on y accède efficacement. D'ou le titre.

Nous y voilà. Encore un chapitre barbant... Donc comme je l'ai dit plus haut, ce chapitre est extrêmement important. Car après ça, on entrera de plein pied dans la programmation en apprenant des instructions en plus grosse quantité, puis dans les chapitres suivants, nous aborderont la structure des données. Donc je le souligne bien, il faut comprendre ce chapitre.

Pour décrire brièvement ce chapitre, je peux déjà vous dire qu'il traitera de l'organisation de la mémoire, de la façon dont le processeur communique avec, les différents registres des processeurs Intel de la famille *80x86*, les modes d'adressage de ces processeurs, ensuite nous

expliquerons le fonctionnement des registres et des drapeaux ainsi que la pile. Enfin, nous verrons les premières instructions d'assembleur, liées donc à la gestion de la mémoire.

Un microprocesseur, en deux mots

Sans entrer dans les détails qui seront vus en cours, nous présentons le minimum à savoir sur ce qu'est un microprocesseur.

Le microprocesseur, noté **CPU** (*Central Processing Unit*), est un élément indispensable dans un ordinateur. Il s'agit d'un élément semi-conducteur au silicium dont la fabrication nécessite une précision extrême. En effet, le microprocesseur regroupe un certain nombre de transistors élémentaires interconnectés, caractérisé par une très grande intégration et doté des facultés fonctionnelles d'interprétation et d'exécution des instructions d'un programme. Le microprocesseur n'est pas uniquement utilisé au sein des PC. De plus en plus les objets qui nous entourent sont truffés de microprocesseurs, de la voiture au téléphone portable en passant par la chaudière et la télévision.

Un microprocesseur constitue le cœur de tout ordinateur: il exécute les instructions qui composent les programmes que nous lui demandons d'exécuter. Les instructions sont stockées en mémoire (en dehors du microprocesseur). Ces instructions (dont l'ensemble compose le langage assembleur) sont très simples mais n'en permettent pas moins, en les combinant, de réaliser n'importe quelle opération programmable. Pour exécuter un programme, le processeur lit les instructions en mémoire, une par une. Il connaît à tout instant l'adresse (l'endroit dans la mémoire) à laquelle se trouve la prochaine instruction à exécuter car il mémorise cette adresse dans son *compteur ordinal*.

Avant de nous intéresser plus particulièrement à l'architecture interne d'un microprocesseur, il convient de connaître l'évolution impressionnante de ces composants.

Historique

Le concept de microprocesseur a été créé par la Société Intel. Cette Société, créée en 1968, était spécialisée dans la conception et la fabrication de puces mémoire. À la demande de deux de ses clients — fabricants de calculatrices et de terminaux — Intel étudia une unité de calcul implémentée sur une seule puce. Ceci donna naissance, en 1971, au premier microprocesseur du monde, le 4004, qui était une unité de calcul 4 bits fonctionnant à 108 kHz. Il résultait de l'intégration d'environ 2300 transistors. A partir de cette date, un rythme très rapide d'évolution s'est installé. De nombreux autres modèles sont apparus, de plus en plus puissants et de plus en plus complexes. Ce rythme d'évolution s'est maintenu sans fléchir jusqu'à aujourd'hui. Ces évolutions concernent les techniques d'intégrations des nombres de transistors et la fréquence d'horloge des processeurs. Il y a également d'autres améliorations, comme la largeur de bus mémoire, la taille des registres ou la taille de la mémoire cache. Dans le même intervalle de temps, leur puissance de traitement est passée de 60 000 instructions exécutées par seconde par l'Intel 4004 à plusieurs milliards par les machines actuelles les plus puissants. L'histoire des microprocesseurs sur les trente dernières années est certainement la plus formidable évolution technologique de l'histoire humaine, tant en durée qu'en ampleur. Aujourd'hui, le multimédia puis le 3D et le temps réel. Demain, le monde virtuel !

Ce rythme d'évolution effréné est appelé *loi de Moore* (*Gordon E. Moore*, du non du président de la compagnie Intel qui l'a formulé dans les années 70). Il prévoyait que les performances des processeurs (par extension le nombre de transistors intégrés sur silicium) doubleraient tous les 12 mois. Cette loi a été révisée en 1975, portant le nombre de mois à 18. La loi de Moore se vérifie encore aujourd'hui. Moore estime que cette évolution se poursuivra jusqu'en 2017, date à laquelle elle devrait rencontrer des contraintes liées à la physique des atomes.

Le tableau si dessous recense l'évolution de la gamme des processeurs compatibles Intel x86, avec les dates des premières versions de chaque modèle.

Date	Nom	Nombre de transistors	Finesse de gravure (µm)	Fréquence de l'horloge	Largeur des données	MIPS
1971	4004	2 300		108 kHz	4 bits/4 bits bus	
1974	8080	6 000	6	2 MHz	8 bits/8 bits bus	0,64
1979	8086 - 8088	29 000	3	5 MHz	16 bits/8 bits bus	0,33
1982	80286	134 000	1,5	6 MHz	16 /16 bits bus	1
1985	80386	275 000	1,5	16 à 40 MHz	32 bits/32 bits bus	5
1989	80486	1 200 000	1	25 à 100 MHz	32 bits/32 bits bus	20
1993	Pentium	3 100 000	0,8 à 0.28	60 à 233 MHz	32 bits/64 bits bus	100
1997	Pentium II	7 500 000	0,35 à 0.25	233 à 450 MHz	32 bits/64 bits bus	300

1999	Pentium III	9 500 000	0,25 à 0.13	450 à 1400 MHz	32 bits/64 bits bus	510
2000	Pentium 4	42 000 000	0,18 à 0.065	1,3 à 3.8 GHz	32 bits/64 bits bus	1 700
2004	Pentium 4D « Prescott »	125 000 000	0,09 à 0.065	2.66 à 3.6 GHz	32 bits/64 bits bus	9 000
2006	Core 2™ Duo	291 000 000	0,065	2,4 GHz (E6600)	64 bits/64 bits bus	22 000
2007	Core 2™ Quad	2*291 000 000	0,065	3 GHz (Q6850)	64 bits/64 bits bus	2*22 000 (?)
2008	Core 2™ Duo (Penryn)	410 000 000	0,045	3,16 GHz (E8500)	64 bits/64 bits bus	~24 200
2008	Core 2™ Quad (Penryn)	2*410 000 000	0,045	3,2 GHz (QX9770)	64 bits/64 bits bus	~2*24 200

Cette miniaturisation a permis:

- D'augmenter les vitesses de fonctionnement des processeurs, car les distances entre les composants sont réduites.
- De réduire les coûts, car un seul circuit en remplace plusieurs.
- De créer des ordinateurs bien plus petits : les micro-ordinateurs.

Comme dans tout circuit intégré la technologie de fabrication impose au microprocesseur des caractéristiques :

- Le **jeu d'instructions** qu'il peut exécuter. Tout microprocesseur contient en lui-même un jeu d'instructions. Ces instructions, très basiques, se résument à une tâche simple, par exemple :
  - Mettre telle valeur dans la mémoire.
  - Ou additionner telle valeur avec telle autre valeur et mettre le résultat dans la mémoire.
  - Comparer deux nombres pour déterminer s'ils sont égaux, comparer deux nombres pour déterminer lequel est le plus grand, multiplier deux nombres...

- La **complexité de son architecture**. Cette complexité se mesure par le nombre de transistors contenus dans le microprocesseur. Plus le microprocesseur contiendra de transistors, plus il pourra effectuer des opérations complexes, et/ou traiter des chiffres des grande taille.
- Le **nombre de bits** que le processeur peut traiter ensemble, autrement la longueur des données que peut manipuler un microprocesseur. Les premiers microprocesseurs ne pouvaient traiter plus de 4 bits d'un coup. Ils devaient donc exécuter plusieurs instructions pour additionner des nombres de 32 ou 64 bits. Les microprocesseurs actuels (en 2007) peuvent traiter des nombres sur 64 bits ensemble. Le nombre de bits est en rapport direct avec la capacité à traiter de grands nombres rapidement, ou des nombres d'une grande précision (nombres de décimales significatives).
- La **vitesse de l'horloge**. Le rôle de l'horloge est de cadencer le rythme du travail du microprocesseur. L'horloge tourne à une fréquence fixée (appelée vitesse d'horloge). Lorsque vous achetez un ordinateur à 1,6GHz, 1,6GHz est la fréquence de cette horloge. L'horloge ne décompte pas les minutes et les secondes. Elle bat simplement à un rythme constant. Les composants électroniques du processeur utilisent les pulsations pour effectuer leurs opérations correctement comme le battement d'un métronome aide à jouer de la musique à un rythme correct. Le nombre de battements (ou, comme on les appelle couramment cycles) que requiert une instruction dépend du modèle et de la génération du processeur. Le nombre de cycles dépend de l'instruction. Plus la vitesse de l'horloge n'augmente, plus le microprocesseur complète de calculs en une seconde.

Notations

x86 ou **i86** : est la dénomination de la famille de microprocesseurs compatibles avec le jeu d'instructions de l'Intel 8086. Les différents constructeurs de microprocesseurs pour PC se doivent de maintenir une compatibilité ascendante afin que les anciens logiciels fonctionnent sur les nouveaux microprocesseurs. L'architecture de la série x86 à partir du Pentium a été nommée *IA-32* par Intel.

IA-32 (*Intel architecture 32 bits*) ou **i386** : désigne l'architecture à partir du Intel 80386, 32 bits, qui a enfin permis de sortir du mode réel correctement, et de faire un bon multitâches sur les Intel. Et d'avoir un adressage mémoire de 4 Go.

IA-64 *(Intel Architecture 64 bits)* désigne l'architecture des nouvelles est une architecture de processeurs Intel destinées à remplacer les x86. Il s'agit d'une rupture totale avec la série x86, le jeu d'instructions n'ayant plus rien à voir, ni les éléments de l'architecture du processeur. Le résultat est quelque chose de globalement nettement plus simple, donc de bien plus rapide, en donnant un contrôle plus fin au logiciel sur le matériel.

Il faut savoir que les familles de processeur ce sont succédés en ajoutant leurs lots d'améliorations, et pas uniquement en matière de rapidité. Voici les principaux processeurs x86: 8088/8086, 80188/80186, 80286, 80386, 80486, 80586/pentium.

8088, 8086: Ces processeurs, du point de vue de la programmation sont identiques. Ils étaient les processeurs utilisés dans les tous premiers PC. Ils offrent plusieurs registres 16 bits : AX, BX, CX, DX, SI, DI, BP, SP, CS, DS, SS, ES, IP, FLAGS. Ils ne supportent que jusqu'à 1Mo de mémoire et n'opèrent qu'en mode réel. Dans ce mode, un programme peut accéder à n'importe quelle adresse mémoire, même la mémoire des autres programmes ! Cela rend le

débogage et la sécurité très difficiles ! De plus, la mémoire du programme doit être divisée en segments. Chaque segment ne peut pas dépasser les 64Ko.

80286: Ce processeur était utilisé dans les PC de type AT. Il apporte quelques nouvelles instructions au langage machine de base des 8088/86. Cependant, sa principale nouvelle fonctionnalité est le mode protégé 16 bits. Dans ce mode, il peut accéder jusqu'à 16Mo de mémoire et empêcher les programmes d'accéder à la mémoire des uns et des autres. Cependant, les programmes sont toujours divisés en segments qui ne peuvent pas dépasser les 64Ko.

80386: Ce processeur a grandement amélioré le 80286. Tout d'abord, il étend la plupart des registres à 32 bits (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP) et ajoute deux nouveaux registres 16 bits : FS et GS. Il ajoute également un nouveau mode protégé 32 bits. Dans ce mode, il peut accéder jusqu'à 4Go de mémoire. Les programmes sont encore divisés en segments mais maintenant, chaque segment peut également faire jusqu'à 4Go !

80486/**Pentium**/**Pentium Pro**: Ces membres de la famille 80x86 apportent très peu de nouvelles fonctionnalités. Ils accélèrent principalement l'exécution des instructions.

Le **80586**, appelé **Pentium** pour des raisons de protection commerciale, dispose d'un bus de données de 64 bits et est muni d'un dispositif de prévision des branchements. Il est constitué de 2 processeurs en pipe-line parallèles lui permettant d'exécuter deux instructions en même temps. Son cadencement est envisagé (en 1994) jusqu'à 150 MHZ.

Pentium MMX: Ce processeur ajoute les instructions MMX (*MultiMedia eXentions*) au Pentium. Ces instructions peuvent accélérer des opérations graphiques courantes.

Pentium II: C'est un processeur Pentium Pro avec les instructions MMX (Le Pentium III est grossièrement un Pentium II plus rapide).

Pourquoi les différents microprocesseurs d'Intel sont-ils alors compatibles entre eux ?

Tout simplement parce que chaque fois que Intel sort un nouveau processeur, toutes les instructions des processeurs précédents sont incluses. En fait, il n'y a généralement pas beaucoup de nouvelles instructions ajoutées. Ceci nous assure que les vieux programmes (ceux écrits pour de plus vieux microprocesseurs compatibles) peuvent parfaitement rouler (fonctionner) sur les nouveaux processeurs. Par contre, les nouveaux programmes, s'ils emploient les nouvelles fonctions plus performantes des nouveaux microprocesseurs, ne pourront pas fonctionner sur les microprocesseurs plus anciens. C'est ce qu'on appelle la **compatibilité ascendante**. Les nouveaux PC sont compatibles avec les anciens, mais pas le contraire. Les nouveaux Macintosh sont compatibles avec les anciens, mais pas le contraire. Les PC ne sont pas compatibles avec les Macintosh.

Le mode de fonctionnement des x86

Le 8086, 8088, 80186 et 80188 n'ont qu'un mode de fonctionnement, le mode réel :
1 Mo au maximum et ils sont monotâches. A partir du 80286 et surtout du 80386, il y a eu trois modes de fonctionnement. En voici le détail :

- **Le mode réel** : mode par défaut du microprocesseur, celui dans lequel est le processeur au démarrage, et dans lequel s'exécute DOS. Il fournit les mêmes

fonctionnalités que le 8086. Cela a plusieurs conséquences, dont l'une est qu'il n'est pas possible d'adresser plus que 1 Mo mémoire.

- **Le mode protégé** : exploité par Windows à partir de la version 3.1. Il tire son nom de "protégé" de ses nombreuses fonctions de protection. Le terme protection signifie ici "protection contre les bugs ou les programmes malveillants". Eh oui, en mode protégé, les programmes sont soumis à des règles très strictes, qui ne sont pas réellement astreignante pour des programmeurs normaux, mais qui sont terribles dès que le programme tente de sortir de l'espace qui lui est réservé. Pourquoi donc ? Le mode protégé à été conçu pour permettre l'existence de systèmes multitâches stables. Imaginons qu'une des tâches commette une "erreur de protection", le système d'exploitation la ferme, et le système de plante pas - du moins en théorie...
  Par exemple, il est n'est pas possible à une tâche de faire des E/S sur certains ports, donc de modifier anarchiquement l'état du matériels, ce qui pourrait entraîner un plantage - seul le système d'exploitation est autorisé à le faire, mais il a pour mission de contrôler scrupuleusement le processus... (Et là, on voit que Windows 95 a été fait légèrement à la va-vite...). Il n'est pas non plus possible de lire des portions de mémoires réservés à d'autres programmes ou au système d'exploitation, question de confidentialité et de sécurité.
  Ce qui est très appréciable, c'est que toutes les erreurs qui pourraient survenir déclenchent ce que l'on nomme une Exception, c'est à dire une interruption déclenchée par le processeur, interceptée par le système d'exploitation, qui prend les mesures nécessaires (fermer l'application en question). Conséquence immédiate : les plantages, c'est du passé !

- Le **mode virtuel** est un mode qui permet à des programmes réalisés pour le 8086 de tourner sur un système multiutilisateur, comme s'il y avait plusieurs processeurs dans le même.

Organisation interne du microprocesseur

Maintenant que nous savons ce qu'est un microprocesseur, penchons-nous sur son organisation intérieure. En effet, un nombre très important de divers éléments est compris dans la puce de silicium que vous avez pu observer ci-dessus.

En fait, un microprocesseur x86 n'est pas uniquement un bloc de *n* millions de transistors, main bien un assemblage intelligent et fortement optimisé de blocs de transistors nommés unités. Caque unité est dédiée à une tache précise. On peut les regrouper en trois parties principales. Il s'agit de :

- l'**UAL** (**Unité Arithmétique et Logique**) ;
- de **l'Unité de commande** ;
- et du **jeu de registres** ;

L'unité arithmétique et logique est un élément particulièrement important au cœur du microprocesseur. L'unité arithmétique et logique est commandée par l'unité de commande. Son rôle et d'effectuer des opérations mathématiques de base, comme des additions, des soustractions, des multiplications et des divisions. L'unité arithmétique et logique est également capable d'effectuer des opérations logiques, comme les fonctions NON, ET-Logique, OU-inclusif, OU-Exclusif, etc. Les éléments que l'UAL doit traiter proviennent du jeu de registres.

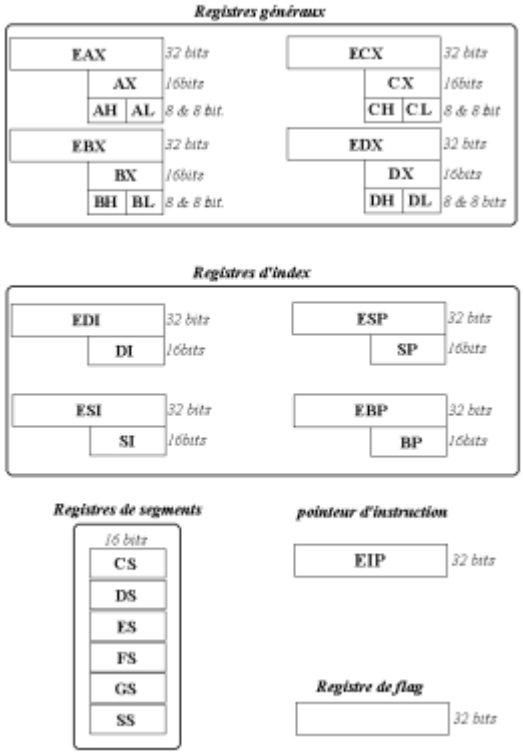
Le jeu de registre contient l'ensemble des registres du microprocesseur. On appelle registres des cellules mémoires qui sont logées non pas dans la mémoire RAM de l'ordinateur, mais directement sur le processeur lui même dans le but de recevoir des informations spécifiques, notamment des adresses et des données stockées durant l'exécution d'un programme. Il existe plusieurs types de registres, mais ils sont en nombre très limitée. Certains d'entre eux sont affectés à des opérations d'ordre général et sont accessibles au programmeur à tout moment. Nous disons alors qu'il s'agit de registres généraux. D'autres registres ont des rôles bien plus spécifiques et ne peuvent pas servir à un usage non spécialisé. Enfin, d'autres registres sont invisibles et par conséquent inaccessible au programmeur. Ces registres ne sont accessibles qu'au microprocesseur. Lorsque nous exécutons un programme, l'UAL à toujours accès à ces registres. Nous verrons plus loin qu'il est possible d'affecter des valeurs à notre guise aux registres généraux.

Les registres présentent l'avantage de permettre un accès beaucoup plus rapide qu'un accès à la RAM. Le microprocesseur et la mémoire RAM constituent en effet deux éléments distincts d'un système informatique et le processeur doit tout d'abord chargé à travers un canal toutes les données auxquelles il veut accéder. Cela entraîne naturellement un certain délai qui peut être évité en stockant les données directement dans le microprocesseur. C'est justement à cela que servent les registres.

Les instructions agissent sur des données qui sont situées soit en mémoire, soit dans des registres du processeur. Pour accéder une donnée en mémoire, il faut spécifier son **adresse**. Pour accéder une donnée dans un registre, il faut spécifier son nom (chaque registre possède un nom qui est une chaîne de caractères). On peut les regrouper en quatre catégories.

- **Registres généraux** (ou de travail).
- **Registres d'offset** (de déplacement, de pointeur, ou d'index).
- **Registres de segment**.
- **Registres de flag** (d'états, des indicateurs, d'exceptions ou de drapeaux).

Si l'on schématise :



Registres généraux

Chaque génération étant un amalgame et une amélioration des précédents, exceptés pour les 8088, 8086, 80188, 80186, et 80286 pour lesquels le registre est identique. C'est pourquoi quand je parlerai de 8086, il sera en fait question de ces 5 processeurs.

Le processeur 8086 original fournissait quatre registres généraux de 16 bits:

- **AX (Accumulateur)** ;
- **BX (Base)** ;
- **CX (Compteur) ;**
- et **DX (Données)**.

Ils ne sont pas réservés à un usage très précis, aussi les utilise-t-on pour manipuler des données diverses. Ce sont en quelque sorte des registres à tout faire. Chacun de ces quatre registres peut servir pour la plupart des opérations, mais ils ont tous une fonction principale qui les caractérisent.

Le registre **AX** sert souvent de registre d'entrée-sortie : on lui donne des paramètres avant d'appeler une fonction ou une procédure. Il est également utilisé pour de nombreuses opérations arithmétiques, telles que la multiplication ou la division de nombres entiers. Lorsque vous devez utiliser un registre pour une opération quelconque et que vous ne savez pas lequel utiliser, privilégiez celui-ci — c'est le plus optimisé au niveau de la rapidité d'exécution des opérations. Il est appelé "accumulateur".

Exemple :

```
MOV AX, 13        ; place l'entier 13 dans AX.
ADD AX,  7        ; ajoute à AX le nombre 7 et place le résultat dans AX.
```

Le registre **BX** peut servir de base. Il est utilisé pour l'adressage indirect, nous verrons plus tard ce que ce terme signifie.

Le registre **CX** est utilisé comme compteur dans les boucles. Par exemple, pour répéter 10 fois une instruction en assembleur, on peut mettre la valeur 10 dans CX, écrire l'instruction précédée d'une étiquette qui représente son adresse en mémoire, puis faire un **LOOP** à cette adresse. Lorsqu'il reconnaît l'instruction LOOP, le processeur "sait" que le nombre d'itérations à exécuter se trouve dans CX. Il se contente alors de décrémenter CX, de vérifier que CX est différent de 0 puis de faire un saut "jump" à l'étiquette mentionnée. Si CX vaut 0, le processeur ne fait pas de saut et passe à l'instruction suivante.

Exemple :

```
MOV  CX, 10      ; place l'entier 10 dans CX
Toto :
; Ecrire les instructions à répéter ici.
LOOP Toto
; (…)
```

Chacun de ces registres peut être décomposé en deux sous registres de 8 bits. Par exemple, le registre AX pouvait être décomposé en **AH** et **AL** comme le montre la Figure si dessus. Le registre AH contient les 8 bits de poids fort de AX, et AL contient les 8 bits de poids faible. (**H** pour **High** et **L** pour **Low**).

Au lancement des processeurs 80386 et plus récents, tous ces registres ont été étendus à 32 bits. Ainsi, le registre AX est devenu **EAX** (*Extended* AX). Pour la compatibilité ascendante, AX fait toujours référence au registre 16 bits et on utilise EAX pour faire référence au registre 32 bits. AX représente les 16 bit de poids faible de EAX tout comme AL représente les 8 bits de poids faible de AX (et de EAX). Il n'y a pas de moyen d'accéder aux 16 bit de poids fort de EAX directement. Mais alors, pour travailler sur la partie haute des 32 bits, il faut employer le registre EAX ou faire une **rotation** des bits pour la ramener sur les 16 bits du bas accessibles via AX. Il est évident que ces quatre registres sont liés. Il l'on modifie par exemple AH, cela changera AX, et par conséquent EAX. Seuls AH et AL sont indépendant l'un de l'autre. Pour modifier la partie haute des 32 bits du registre EAX, il faut procéder via le registre EAX. Par contre, pour modifier la partie basse des 32 bits, on peut procéder à l'aide du registre AX. Il est également possible de modifier la partie haute du registre de 16 bits AX avec le registre AH, et la partie basse avec le registre AL. Il est intéressant de travailler avec ces registres (AH et AL) pour la manipulation de caractères ou des données dont la taille ne dépasse pas un octet.

Dans le tableau suivant, vous pouvez visualisez cette décomposition :

Registre(s)	Taille
EAX	32 bits
AX	16 bits
AH AL	8 bits chacun

Les autres registres étendus sont EBX, ECX, EDX, ESI et EDI. La plupart des autres registres sont également étendus. BP devient EBP; SP devient ESP; IP devient EIP; et
FLAGS devient EFLAGS.

Registres de segments

Les registres 16 bits **CS**, **DS**, **SS** et **ES** sont des registres de segment. Ils indiquent quelle zone de la mémoire est utilisée pour les différentes parties d'un programme. Contrairement aux registres généraux, ces registres ne peuvent servir pour les opérations courantes : ils ont un rôle très précis. On ne peut d'ailleurs pas les utiliser aussi facilement que AX ou BX, et une petite modification de l'un d'eux peut suffire à « planter » le système. Eh oui !

- **CS** pour *segment de code*, registre permettant de fixer ou de connaitre l'adresse du début des instructions d'un programme.
- **DS** pour *segment de données*, ce registre est dédié à la localisation de vos données.
- **SS** pour *segment de pile*, ce registre pointe sur la pile. Comme je ne vous ai pas encore expliqué ce qu'est la pile, j'y reviendrai plus tard.
- **ES** pour *segment extra*, ce registre est utile pour la manipulation de données entre plusieurs segments en parallèle des registres DS, FS et GS.
- **FS** et **GS** pour *segment extra,* ces registres ont les mêmes fonctionnalités que le registre DS. Il est disponible à partir des processeurs 80386.

Dans le registre CS est stockée l'adresse de segment de la prochaine instruction à exécuter. La raison pour laquelle il ne faut surtout pas changer sa valeur directement est évidente. De toute façon, vous ne le pouvez pas. Le seul moyen viable de le faire est d'utiliser des instructions telles que des sauts "**JMP**" ou des appels "**CALL**" vers un autre segment. CS sera alors automatiquement actualisé par le processeur en fonction de l'adresse d'arrivée. Le registre DS est quant à lui destiné à contenir l'adresse du segment des données du programme en cours. On peut le faire varier à condition de savoir exactement pourquoi on le fait. Par exemple, on peut avoir deux segments de données dans son programme et vouloir accéder au deuxième. Il faudra alors faire pointer DS vers ce segment. ES est un registre qui sert à adresser le segment de son choix. On peut le changer aux mêmes conditions que DS. Par exemple, si on veut copier des données d'un segment vers un autre, on pourra faire pointer DS vers le premier et ES vers le second. Le registre SS adresse le segment de pile. Il est rare qu'on doive y toucher car le programme n'a qu'une seule pile.

On ne peut pas mettre directement une valeur immédiate dans un registre de segment, le microprocesseur ne le permet pas:

```
MOV DS, 10  ; est incorrect
```

Par contre :

```
MOV AX, 10
MOV DS, AX  ; est correct
```

Registres d'offset

Il y a deux registres d'index et trois de pointeurs de 16 bits :

SI (Index de source) et **DI (Index de destination)**. Ils sont souvent utilisés comme des pointeurs, mais peuvent être utilisés pour la plupart des mêmes choses que les registres généraux. Cependant, ils ne peuvent pas être décomposés en registres de 8 bits.

Les registres 16 bits **BP (Pointeur de base)** et **SP (Pointeur de pile)** sont utilisés pour pointer sur des données dans la pile du langage machine et sont appelés le pointeur de base et le pointeur de pile, respectivement. Nous en reparlerons plus tard. Le registre **IP (Pointeur d'instruction)** contient le déplacement entre le début du registre CS et la prochaine instruction que doit exécuter le processeur (Il peut être dangereux de manipuler ce registre car, de par sa fonction, une mauvaise utilisation risque d'entraîner un plantage de votre programme).

Le registre de pointeur d'instruction (IP) est utilisé avec le registre CS pour mémoriser l'adresse de la prochaine instruction à exécuter par le processeur. Normalement, lorsqu'une instruction est exécutée, IP est incrémenté pour pointer vers la prochaine instruction en mémoire.

Pour ces registres, on trouve également la capacité de découpage des registres de 32 à 16 bits précédemment énoncé dans sur la section des registres généraux.

Registre de flag

Un programme doit pouvoir faire des choix en fonction des données dont il dispose. Pour cela, il lui faut par exemple comparer des nombres, examiner leur signe, découvrir si une erreur a été constatée, etc.... Il existe à cet effet de petits indicateurs, les flags qui sont des bits spéciaux ayant une signification très précise. De manière générale, les flags fournissent des informations sur les résultats des opérations précédentes. Ils sont tous regroupés dans un registre: le registre des indicateurs. Comprenez bien que chaque bit a un rôle qui lui est propre et que la valeur globale du registre ne signifie rien. Le programmeur peut lire chacun de ces flags et parfois modifier leur valeur directement. En mode réel, certains flags ne sont pas accessibles. Nous n'en parlerons pas. Nous ne commenterons que les flags couramment utilisés. Nous verrons quelle utilisation on peut faire de ces indicateurs dans la troisième partie de ce tutoriel. Le registre flags permet de fixer et ce connaître l'état du processeur grâce aux différents bits qui le composent, ce qui permet ainsi d'avoir à tout instant l'état résultant de l'exécution d'une opération par le microprocesseur. La plupart des instructions affectent ce registre. Chaque bit relate l'état spécifique de la dernière instruction exécuté. Après l'utilisation de certaines instructions, il est affecté et l'état de ses différents bits permet de prendre des décisions, par exemple choix lors de branchement conditionnel (boucle et saut) pour vérifier s'il y a une retenue après une opération arithmétique. Le registre FLAGS est composé de 32 bits également appelé indicateur :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Bit
0	NT	IOPF		OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF	Appellation

Les 16 premiers bits (0 à 15) sont considérés comme les bots de drapeau.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16

										ID	VIP	VIF	AC	VM	RF

Les case qui sont grisées correspond aux bits réservés ou non définis par Intel.
EFLAGS est un registre en deux groupes de bits indicateurs.

- **CF (Carry Flag),** en français indicateur de retenue. Ce bit est positionné à l'état haut 1 s'il y a une retenue à l'issue de l'exécution d'une opération arithmétique.

- **PF (Party Flag),** indicateur de parité. Ce bit est positionné à l'état haut 1 si le résultat d'une opération est un nombre pair de bit mis à 1.

- **AF (Auxiliary Flag),** retenue auxiliaire. Ce bit est positionné à l'état haut 1 si, après une opération, il y a une retenue.

- **ZF (Zero Flag),** indicateur de zéro. Ce bit est positionné à l'état haut 1 si, après une opération, le résultat est nul.

- **SF (Sign Flag),** indicateur de signe. Ce bit est positionné à l'état haut 1 si, après une opération, le résultat est négatif.

- **IF (Interrupt Flag),** sert à empêcher les appels d'interruptions lorsqu'il est positionné à 1. Cependant, toutes les interruptions ne sont pas masquables.

- **OF (Overflow Flag),** indicateur de débordement de capacité. Ce bit est positionné à l'état haut 1 si, après une opération, le résultat déborde la capacité du registre. Prenons, par exemple, l'addition de la valeur 0FFFF*h* maximale qui peut être stockés dans un mot de 16 bits avec une valeur 1 quelconque :
  0FFFF*h* + 1h = 10000*h*. Le résultat de cette addition est la valeur 10000h et ce nombre ne tient pas sur 16 bits ! Il y a donc un dépassement ! Cet indicateur sert également à signaler des débordements lors de l'utilisation d'arithmétique signé (-32767 à 32768 contre 0 à 65535).

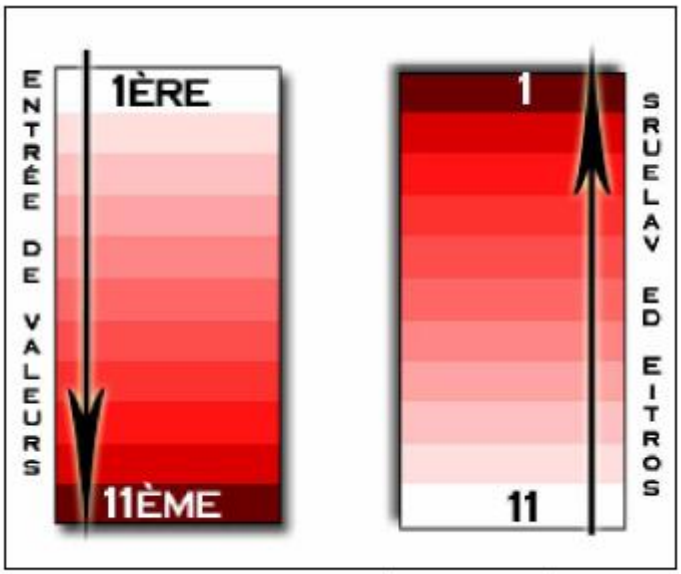
Il en existe d'autres, mais nous ne les utiliserons pas ici.

La pile

La programmation d'un ordinateur nécessite l'usage (d'au moins) une pile. Une pile est eune région de la RAM où l'on va stocker ou retirer des informations, elle est structurée linéairement et dont on ne peut manipuler que le sommet, à l'image d'une pile d'assiettes : vouloir retirer une assiette du milieu conduit à la catastrophe. La seule méthode d'accès consiste à empiler ou dépiler des assiettes à partir du sommet de la pile. Une telle structure permet de conserver en mémoire l'ordre selon lequel on y a stocké des informations. En les dépilant, l'ordinateur se retrouve dans l'état ou il y était avant les opérations qui ont conduit à des empilements successifs. L'assembleur vous fera manipuler une pile qui est stockée "en fond de panier", c'est à dire dans les adresses les plus hautes de la mémoire. (Correspondant aux 640 Ko). Une petite particularité : La base de la pile se trouvé à l'adresse maximale, et elle s'accroît vers les adresses basses. A l'inverse, les programmes se développent sur le "tas", en bas de mémoire, juste au dessus de la zone réservée pour le système. Pile et Tas croissent donc à l'inverse l'un de l'autre. L'utilisation de la pile est très fréquente pour la sauvegarde temporaire du contenu des registres et pour le passage de paramètres lorsqu'un langage de haut niveau fait appel à une routine en assembleur. La taille de la pile varie en fonction de la quantité d'informations qu'on dépose ou qu'on en retire.

Dans l'architecture x86 32bits, le registre ESP sert à indiquer l'adresse du sommet d'une pile dans la RAM. Les instructions "**PUSH**" et "**POP**" permettent respectivement d'empiler et de dépiler des données. Les instructions "**CALL**" et "**RET**" utilisent la pile pour appeler une fonction et la quitter par la suite en retournant à l'instruction suivant immédiatement l'appel.

En cas d'interruption, les registres EFLAGS, CS et EIP sont automatiquement empilés. Dans le cas d'un changement de niveau de priorité lors de l'interruption, les registres SS et ESP le sont aussi.



Les couleurs représentent des valeur, de la 1ère valeur → 11ème valeur.

La mémoire

La mémoire est la partie de l'ordinateur dans laquelle les programmes et les données sont rangés. Sans mémoire, dans laquelle le processeur lit et écrit de l'information, il n'y aurait pas d'ordinateurs tels que nous les connaissons. Une mémoire est formé d'un certain nombre de cellules (ou cases), chacune de ces cellules contenant une certaines quantité d'informations. Chaque cellule a un numéro, que nous appellerons son adresse, qui permet à un programme de la référencer.

Une analogie consiste à comparer la mémoire à une longue rangée de tiroirs alignes les un derrière les autres. Si on donne à chaque tiroir un numéro, en commençant par 0 pour le 1èr tiroir, on dira que ce numéro est l'adresse de la mémoire, dans la suite on parlera d'adresse mémoire. La coutume est (de nombreux avantages la justifie) de noter les adresses mémoires en hexadécimal.
L'unité mémoire de base est l'octet. Un ordinateur avec 3 Go de mémoire peut stocker jusqu'à environ 3 milliards d'octets d'informations.

0	1	2	3	4	5	…	…	N - 1
3A	11	B1	8A	EF	6A	…	…	AB

Adresses Mémoire

Ce qu'il est important d'avoir à l'esprit, c'est que le microprocesseur a besoin de mémoire pour travailler : c'est dans la mémoire qu'il va lire son programme, écrire des valeurs...

La pagination mémoire

Les processeurs IA-32 sont dotés d'une fonction appelée *pagination*. Elle permet à un segment d'être divisé en blocs mémoire, appelés *pages* et mesurant tous 4096 octets. Le mécanisme de pagination permet l'exploitation par tous les programmes qui fonctionnent
En même temps en espace mémoire apparemment plus vaste que la mémoire physiquement installé sur l'ordinateur. L'ensemble des pages est parfois appelé la *mémoire* virtuelle. Un système d'exploitation moderne dispose normalement d'un programme appelé le *gestionnaire de mémoire virtuelle* (**VMM**, *Virtual Memory Manager*).

La pagination constitue une solution technique de première importance à un problème rencontrent sans cesse les programmeurs et les concepteurs de matériels. En effet, pour pouvoir exécuter un programme, il faut le charger en mémoire, mais la mémoire est coûteuse. L'utilisateur veut pouvoir lancer plusieurs programmes à la fois et basculer entre l'un et l'autre. En revanche, l'espace disque est beaucoup mois coûteux. La pagination donne l'illusion de disposer d'une quantité quasi illimitée de mémoire vive. En revanche, une partie de la mémoire étant remisée sur disque dur, les accès sont beaucoup moins rapides qu'avec la mémoire réelle.

Pendant l'exécution d'une tâche, il est possible de stocker certaines parties du programme sur disque si elles ne sont pas en cours d'exécution. Le résultat de cette opération est le remisage disque (*swapping*) de pages. La portion de la tâche qui est en cours d'exécution doit rester en mémoire. Lorsque le processeur à besoin d'exécuter les instructions qui se trouvent dans une page qui a été remisée sur disque, il provoque une faute de page, ce qui force à relire depuis le disque la page de code ou de données pour la placer en mémoire. Pour voir comment fonctionne ce système, vous devez disposer d'un ordinateur dans la quantité de mémoire vive n'est pas suffisante (32 ou 64 Mo) puis lancer entre cinq et dix applications. Vous remarquerez un ralentissement des performances lorsque vous basculez d'un programme à l'autre, car le système d'exploitation doit déporter des portions des applications de la mémoire vers le disque. C'est pour cette raison qu'un ordinateur fonctionne plus vite lorsqu'il est doté d'une plus grande quantité de mémoire vive : cela évite de recouvrir trop souvent au mécanisme de pagination en laissant une plus grande portion de chaque application en mémoire.

Organisation de la mémoire

La mémoire des x86 est organisée selon deux principes :

- Le mode segmenté.
- Le mode protégé.

Le premier style est le mode par défaut de tous les processeurs x86. Ce style est contraignant dans son fonctionnement, mais il n'est pas nécessaire, contrairement au mode protégé, de se lancer dans de lourdes routines d'initialisation mal maîtrisées. Ce style par défaut pourrait être nommé **mode d'adressage segmenté**. En effet, la mémoire dans ce mode n'est pas un long et unique bloc, mais une suite de **"segment"** Les segments ont une taille limite de 64Ko. Cette capacité d'adressage de 65 535 octets est due à la limite des 16 bit des premiers

processeurs 8086. A l'époque, en 1981, un processeur qui était capable d'adresser plus de 64 ko, c'était très fort, et le prix de la RAM était prohibitif ! Pour trouver l'adresse réelle de la mémoire, il faut une valeur de positon appelé en anglais **"offset"** (décalage). Cet offset est donc le déplacement qui existe entre le premier octet du segment et l'octet de la mémoire qui est pointée.

A partir de ces deux valeurs **SEGMENT: OFFSET**, on peut localiser n'importe quel octet dans la mémoire adressable par le processeur. Exemple A000h:0000h

Le second style d'adressage de la mémoire de ces nouveaux processeurs est spécifique du mode protégé. Une fois que le processeur est initialisé en mode protégé, la mémoire est adressable d'un seul bloc de 4Go.

Lorsqu'ils ont conçu le mode protégé, les ingénieurs d'Intel en ont profité pour se débarrasser d'une sérieuse limitation de leurs processeurs jusqu'alors en mode protégé : La limitation d'adressage mémoire de 1Mo a disparu. Il devient possible d'adresser 16Mo sur 80286 et 4Go à partir du 80386. Par ailleurs, le 80386, doté d'un bus sur 32 bits, dispose de nouveaux modes d'adressage mémoire.

A partir du 80286, le processeur dispose d'instructions pour la commutation des tâches; cette opération devient dès lors très simple à programmer et ainsi plus fiable. Ces processeurs comportent aussi une gestion automatique de la mémoire virtuelle, pas très intéressante sur 80286, mais relativement performante sur 80386. Le processeur a aussi la capacité de simuler autant de processeurs en mode réel que l'on souhaite et de leur allouer à chacun 1 Mo de mémoire. C'est ce qui se passe lorsqu'on lance un programme DOS sous Windows. Les programmes exécutés ont l'impression de tourner sur un processeur indépendant, avec un mémoire indépendante.

Les programmes Win32 fonctionnent en mode protégé depuis le 80286. Mais le 80286 c'est maintenant de l'histoire ancienne. Donc nous devons seulement nous intéresser au 80386 et à ses descendants.

Windows dirigent chaque programme Win32 séparément. Ceci signifie que chaque programme Win32 aura à sa disposition ses propres espaces de mémoire. Cependant, cela ne signifie pas que chaque programme win32 a 4GB de mémoire physique, mais seulement que le programme peut adresser n'importe quelle adresse dans cette gamme. Windows fera tout le nécessaire pour adresser les vraies références (physiques) du programme aux adresses mémoires valables 00401000… qu'il utilise. Bien sûr, le programme doit respecter les règles de Windows, autrement il causera une Erreur de Protection Générale tant redoutée. Chaque programme est seul dans son espace d'adresse. C'est la différence avec les programmes Win16. Tous les programmes Win16 peuvent « malheureusement plus ou moins se chevaucher » les uns les l'autres. Mais pas sous Win32. Ces gardes fous réduisent la chance d'une écriture du code d'un programme par dessus les données d'un autre.

Le modèle de mémoire est aussi résolument différent des vieux jours du monde du 16 bits. Désormais, sous Win32, nous ne sommes plus concernés par le modèle de mémoire en plusieurs segments! (0001.xxxx puis 0002.xxxx puis 0003.xxxx…) L'assembleur 32 bits est à la fois plus clair et plus simple que le DOS et il n'est pas handicapé par l'arithmétique des segments. Vous n'avez plus à vous soucier des paires de registres comme AX:DX pour les entiers longs et il n'y a plus la limitation des segments de mémoire à 64 Ko qui existait dans la segmentation 16 bits. Il y a seulement un seul modèle de mémoire : le modèle de mémoire uniforme (**Flat**). La mémoire est un seul espace continu. Ça veut dire aussi que vous n'avez plus besoin de jouer avec les différents registres concernant les segments [ES et SS]. Vous pouvez employer n'importe quel registre de segment pour adresser n'importe quel point dans l'espace de

mémoire (ouf !). C'est une grande aide pour les programmeurs. C'est ce qui fait de l'assembleur Win32 une programmation aussi facile que le C.

Quand vous programmez sous Win32, vous devez respecter quelques règles importantes. Une règle majeure est que Windows emploie lui-même ESI, EDI, EBP et EBX et les valeurs de ses registres changent, en même temps qu'une autre application tourne en parallèle avec Windows. Rappelez-vous donc cette règle primordiale d'abord : si vous employez chacun de ces quatre registres à l'intérieur d'une procédure, n'oubliez jamais de les reconstituer "rétablir" avant le contrôle de retour à Windows. L'exemple évident ce sont les Call qui appellent une **API** de Windows. Cela ne signifie pas que vous ne pouvez pas employer ces quatre registres, vous le pouvez. Mais seulement que vous devez rétablir ces registres après un call.

Chapitre3 : Instruction du microprocesseur

Les instructions des x86 sont très nombreuses et il n'est pas question de les développer toutes dans ce guide. Nous détaillerons dans chapitre uniquement les instructions classiques, c'est-à-dire celles qui sont les plus importantes pour une initiation à la programmation courante en assembleur. Dans ce chapitre, nous développerons le fonctionnement des instructions du microprocesseur et nous agrémenterons ces descriptifs par quelques courts exemples de mise en situation. Nous approfondirons un certain nombre de ces instructions.
Préalablement, nous verrons quelques précautions d'usage liées à l'emploi et au maniement des instructions, des directives et de certains mots réservés. Ces avertissements engloberont les conventions de lecture et d'écriture des différents champs développés.

Anatomie d'un programme en assembleur

Et oui, comme toute "langue vivante", l'assembleur possède des propres règles de "grammaire" et d'"orthographe". Et gare à vous si, dans un désir d'indépendance tout à fait mal à propos, vous décidez de ne pas y plier!

Ceci dit, ces règles sont très simples et il faudra qu'un peu d'habitude pour les connaître. Mieux même, ces programmes que jusqu'à maintenant vous considériez être du chinois en seront réduit à n'être finalement que… de l'anglais: on tombe vraiment dans la facilité! Nous pourrons distinguer deux sortes deux règles dans la programmation en assembleur :

- Les règles absolues: si par malheur vous les transgresser, vous vous ferez froidement « jeter » par la machine avec en guise de compliment de petits mots d'amitié de type "**ILLEGAL FORMAT**", "**MISSING OPERAND**", le tout dans le plus pur anglais de *Shakespeare*;

- Les règles conseillées: ce sont celles qui sont laissé à l'appréciation du client. En gros, si vous ne les saviez pas, vous risquez :

    - d'une part de passer beaucoup de temps que nécessaire pour faire tourner un programme.
    - d'autre part de devoir vous reporter au chapitre consacré au code erreur par suite d'une étourderie ce qui est très désagréable.

Structure des instructions

Un programme en assembleur a une forme bien particulière. Chaque ligne d'un code source assembleur comporte une instruction. Chaque ligne est composée de champs. De gauche à droite, on a :

Etiquette	Mnémonique	Opérande destination	Opérande source	Commentaire
Début:	MOV	EAX	EBX	; EAX <- EBX

- Le **champ étiquette**, qui peut être vide.
- Le **champ mnémonique** (obligatoire)
- Le **champ opérande**, par exemple EAX, BX, DH, CL, MonCompteur, 10… (normalement obligatoire)
- Et d'un **champ commentaire**, qui peut être vide.

Découvrons chaque partie d'une instruction, en commençant par le champ d'étiquette.

Étiquette

Une instruction peut être précédée d'un identificateur qui représente l'adresse de stockage de cette instruction. On appelle cet identificateur une *étiquette* (label en anglais). Il label permet un repérage plus aisé à l'intérieur du programme. Le nom du label est arbitraire. Vous pouvez le nommer comme ça vous chante tant qu'il est unique et ne viole pas la convention de nom de l'assembleur.

- Par exemple vous ne pouvez pas prendre le mot 'MOV' comme nom de label.
- N'utilisez pas de labels qui peuvent être confondus avec d'autres labels.
- Evitez les lettres I, O, Z, et les chiffres 0, 1, 2.
- Evitez aussi des labels tels que **XXXX** et **XXXXX** car on ne les différencie pas aisément.

Example:

```
Toto:
mov CH, 00h
inc AX
jmp Toto
```

Mnémonique

Un mnémonique d'instruction est un mot court qui identifie une instruction. Les mnémoniques du langage assembleur sont d'origine anglaise. Les lecteurs français retrouvent néanmoins de nouveaux éléments communs avec leur langue.

Opérandes

La syntaxe assembleur **I**ntel x86 définit l'opérande de gauche comme **opérande destination**, celui dans lequel on trouve le résultat de l'instruction. L'opérande de droite est appelé **opérande source**. Le champ opérande est un champ optionnel selon l'instruction (Une

instruction peut avoir entre zéro et trois opérandes). Dans la plupart des assembleurs, il ne peut y avoir qu'une seule instruction par ligne. Chaque opérande être le nom d'un registre, une constante, une expression constante ou un emplacement mémoire.

Exemple	Type d'opérande
13	Constante
2 + 6	Expression constante
eax	Registre
Compteur	Emplacement mémoire

```
; Exemple d'instructions simples :
mov EAX, 9h      ; Charge le registre EAX avec la valeur 13h.
mov EBX, 1d      ; Charge EBX avec la valeur décimale 9.
Debut:
dec EAX          ; Décrémente le registre EAX (EAX = EAX - 1).
cmp EAX, EBX     ; Comparer EAX & EBX et positionne les flags
                 ; en fonction du résultat.
jne Debut        ; Saut à "debut" si EAX est différent d'EBX.
. . .            ; On est sûr que dans cette instruction EAX
                 ; sera  égale à EBX.
```

Ecrire des commentaires dans le programme

Aujourd'hui, vous savez exactement comment votre programme fonctionne. Néanmoins, si quelqu'un d'autre le lit ou si vous le relisez vous-même qu'après quelque temps (vous aurez peut-être oublié le rôle de telle ligne ou telle fonction), vous serez l'un comme l'autre d'y trouver des commentaires explicatifs. Ces commentaires expliquent (en français ou dans toute autre langue que vous parlez) ce que se passe dans le programme écrit en langage informatique.

Ainsi, un commentaire est un texte ajouté au code source d'un programme servant à décrire le code source, facilitant sa compréhension par les humains. Il est donc séparé du reste du code grâce à une syntaxe particulière. La norme est le point virgule (;). Le positionnement de se sigle dans une ligne implique que tout le texte qui se trouve à sa suite est considéré comme un commentaire. Ce texte ne sera donc pas traité à l'assemblage. Certains assembleurs tolèrent la syntaxe d'autres langages comme le C, c'est-à-dire les couples de caractères : **//** et **/* */**. Mais je vous conseille de respecter la norme assembleur dans un souci de compatibilité entre assembleurs. Vous avez déjà vu des exemples des commentaires dans les exemples de programmes que nous avons présentés, voici un exemple de code incluant des commentaires :

```
   ; La phrase qui se trouve ici n'est pas prise en compte.
mov AL, 00000010b  ; Copier la valeur binaire dans AL.
sub Al, 00000001b  ; Soustraire le contenu de Al de 00000001b.
                   ; Et stocke le résultat dans AL
inc CX             ; Incrémenter CX (CX = CX + 1)
```

Lisibilité et présentation

L'assembleur ne différencie pas majuscules et minuscules, et ne tient pas compte du nombre d'espaces. Les instructions suivantes sont identiques et correctes :

Exemple :

```
Mov Ah, 1           ;  Charge AH avec la valeur 1.
mOV  aH   ,  1      ;  Charge AH avec la valeur 1.
MOV AH, 1           ;  Charge AH avec la valeur 1.
mov ah,1            ;  Charge AH avec la valeur 1.
```

La lisibilité du code source a un impact direct sur la façon dont un développeur comprend un système logiciel. La maintenance de code désigne la facilité de modifier un système logiciel pour ajouter de nouvelles fonctionnalités, modifier des fonctionnalités existantes, corriger des bogues ou améliorer les performances. Bien que la lisibilité et la maintenance résultent de plusieurs facteurs, la technique de codage représente une partie spécifique du développement de logiciel sur laquelle tous les développeurs peuvent avoir une influence, il est donc fortement conseillé, voire absolument indispensable, d'être un minimum régulier et strict dans la présentation des programmes sources. Aussi, il suffit de respecter une présentation uniforme et régulière, et d'appliquer pour cela des règles élémentaires, qui ne sont pas spécifiques à l'assembleur :

- Commenter correctement le fichier du code source. Ne pas tomber dans la facilité en commentant ce que fait l'instruction, mais dire à quoi elle sert. Il ne faut non plus être avare dans les commentaires ainsi qu'au niveau de l'aération du code source, Ne pas hésiter à sauter des lignes pour mieux visualiser les blocs d'instructions. Toutes ces modifications ne seront pas assemblées et n'occultent donc pas les performances de l'exécutable, et elles feront la différence au moment de la relecture du code source.

- Appliquer les tabulations pour améliorer la lisibilité du code source, Ainsi, essayer de respecter des colonnes imaginaires dédiées aux étiquettes, mnémoniques, opérandes et commentaires.

- User des majuscules à bon escient. Il est préférable de réserver par exemple les mots en majuscules aux déclarations de constantes.

- Utiliser les bonnes bases numéraires afin d'améliorer la lecture du code. En effet, l'homme ne lit pas naturellement et aussi facilement le binaire ou l'hexadécimal que le décimal. Surtout pour les grandes valeurs. Il est possible de mettre ces valeurs lisibles dans les commentaires.

- Essayer de casser la linéarité du code source car il est relativement lourd d'analyser un fichier source qui n'est qu'une longue suite d'instructions. Pour cela, il faut essayer de faire des modules, de commenter par blocs. Ce qui se traduit par des séparations des commentaires grâce à des lignes de tirets. Nous verrons cela en détail dans le paragraphe qui traite la programmation modulaire.

Notes et abréviations

Chaque instruction listée dans ce chapitre est décrite selon un même modèle, composé de différents champs faisant référence aux notions : drapeaux et opérandes.

Comme nous l'avons déjà vu, le microprocesseur possède un registre particulier qu'est le registre flag. La plupart des instructions modifient les indicateurs du registre EFLAGS. Un tableau indique l'effet de l'instruction couramment décrite sur ces indicateurs. Il a la forme suivante ;

OF	DF	IF	TF	SF	ZF	AF	PF	CF

Pour indiquer l'état de chaque bit composant les flags de ces tableaux, nous avons choisi les signes suivants :

- **0** indique un bit est positionné à 0.
- **1** indique un bit est positionné à 1.
- ***** le bit est modifié en fonction du résultat de l'exécution de l'instruction (0 ou 1).
- **?** indique un bit positionné de manière non prévisible et mis à 0 ou à 1 par l'instruction.
- Une case **vide** indique que l'indicateur n'est pas modifié par l'instruction.

Par exemple, dans le tableau suivant, résultat de passage de l'instruction **OR**, on peut en déduire qu'il est composé de :

OF	DF	IF	TF	SF	ZF	AF	PF	CF
0				*	*	?	*	0

A la lecture de ce tableau, on peut dire que la direction est interne, il n'y a pas d'interruption, il n'y a pas d'exécution pas à pas, le signe, le zéro, la parité sont fonction de l'exécution. Enfin, la retenue auxiliaire est variable. Le débordement et la retenue sont annulés.

Opérandes

Pour chaque instruction, on utilisera des opérandes afin de visualiser les paramètres de fonctionnement de chacune. On trouve des mémoires et des registres classiques.

- **Reg8** désigne un registre 8 bits : AL, AH, BL, BH, CL, CH, DL, DH.
- **Reg16** désigne un registre 16 bits : AX, BX, CX, DX, DI, SI, BP, SP.
- **Reg32** désigne un registre 32 bits : EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP.
- **Accum** désigne l'accumulateur qui peut être AL, AX ou EAX selon la taille de l'opération.
- **Mem16** désigne un emplacement mémoire de 16 bits.
- **Mem32** désigne un emplacement mémoire de 32 bits.
- **Immed8** désigne une valeur immédiate de 8 bits.
- **Immed16** désigne une valeur immédiate de 16 bits.

- **Immed32** désigne une valeur immédiate de 32 bits.

Liste des instructions par fonctionnalités

Cette liste récapitule les instructions que nous connaissons déjà et en présente de nouvelles. Elle n'est pas exhaustive mais vous sera amplement suffisante pour la plupart de vos programmes.

Instructions de déplacement et d'affectation

Les instructions de déplacement et d'affectation de données permettent de manipuler des blocs de données d'un endroit à l'autre de la mémoire. La taille des données déplacées peut être de type : octet (8 bits), mot (16 bits), double mot (32 bits), quadruple mot (64 bits), et même des groupes d'un des types précédents.

Ce transfert peut être effectué de plusieurs manières :

- D'un registre vers la mémoire ;
- De la mémoire vers un registre ;
- D'un registre vers un registre ;
- De la mémoire vers la mémoire ;

Voici la liste des instructions de déplacement et d'affectation :
BSWAP, XMPWCHG, CAMPXHG8B, IN, LDS, LES, LFS, LSS, LEA, LODSB, LODSW, MOV, MOVSB, MOVSW, MOVSD, MOVSX, MOVZX, PUSH, PUSHA, PUSHAD, POP, POPA, POPAD, STOSB, STOSW, STOSD, XLAT.

Instructions logiques et arithmétiques

Les instructions arithmétiques et logiques sont effectuées par l'unité arithmétique et logique. Il s'agit d'opérations directement effectuées sur les bits de la donnée que l'on traite. Sont comprises dans cette appellation :

- Les instructions d'addition ;
- Les instructions de soustraction ;
- Les instructions de multiplication ;
- Les instructions de division ;
- Les instructions logiques (ET, OU, ...) ;

Les opérations arithmétiques et logiques modifient l'état des indicateurs. Voici la liste des ces instructions :
AAA, AAD, AAM, ADC, ADD, SUB, SBB, CBW, CWD, CWDE, CQD, DAA, DAS, DEC, INC, MUL, DIV, IMUL, IDIV, AND, OR, XOR, NOT, NEG.

Faites attention à l'emploi des instructions liées à l'usage des nombres signés, faites la distinction avec les instructions dédiées aux nombres non signés. Cela ne s'applique qu'aux opérations de multiplication et de division. En effet, les manipulations de nombres signés négatifs à partir d'addition ou de soustraction ne sont en fait que des traitements de compléments à 2.

Instructions de manipulation de bits

Ces instructions permettent une manipulation directe des bits, un à un ou par groupe. Il faut remarquer que ces instructions fonctionnent sans l'emploi des opérations logiques (AND, not, or, etc.) Cette capacité à travailler de cette manière sur les bits est une spécificité de l'assembleur.

Instructions de décalage

On distingue trois groupes d'instructions dans cet ensemble. Le premier groupe permet le décalage des bits présents dans un groupe (octet, mot ou double mot). La direction du décalage peut être fixée vers la droite ou la gauche ou de plusieurs bits.
SAL, SHL, SAR, SHAR, SHLD, SHRD.

Le deuxième groupe est spécialisé dans les rotations de bit dans un même octet, mot ou double mot.
ROR, ROL, RCL, ROR.

Instructions de traitement

Le troisième groupe est constitué des instructions de traitement. C'est-à-dire les instructions permettant de manipuler les bits directement, de changer leu état (0, 1), d'effectuer des tests.

BT, BTC, BTR, CLC, STC, CMC, CLD, STD, CLI, STI.

Instructions de contrôle et de test

Nous sommes sur le point d'aborder une partie intéressante, dans la mesure ou elle permet de comprendre la réaction du programme suite à la valeur du résultat (0 ou 1). Nous allons maintenant nous intéresser aux conditions. Sans conditions, nos programmes informatiques feraient un peu toujours la même chose, ce qui serait carrément barbant à la fin.

Dans la vie, on a souvent besoin de décider du déroulement des choses en fonction de certaines conditions. Par exemple : "S'il pleut, je prends mon parapluie" (ceci n'a d'ailleurs de sens que si je souhaite sortir !). De même en Assembleur, il est possible de décider de n'exécuter une action que si une condition particulière est remplie.

Pour tester si une condition est vraie avant d'exécuter une action, on dispose des instructions permettent d'activer des branchements "saut", avec ou sans fonction de test et de choix. Le déroulement du programme est alors dévié de son chemin courant vers un autre, spécifique du choix. On distingue ces instructions de saut en deux catégories suivant que :

- Le saut est effectué quoi qu'il arrive « **saut inconditionnel** ».
- Le saut est effectué ou non selon l'état d'un registre « **saut conditionnel** ».

Saut inconditionnel

Ce sont des branchements obligatoires, le processeur saute directement à l'adresse destination. Il procède aux sauts sans aucun test préliminaire. Cela comprend également l'utilisation des interruptions.
JMP, CALL, RET, INT, IRET

Saut conditionnel

Ces branchements sont effectifs après validation d'une condition. Il saute à l'adresse seulement si la condition est vraie. Dans le cas contraire, l'exécution se poursuit avec l'instruction suivante. Ces instructions sont liées aux instructions de test.

LOOP, JCC, REP.

Test

Ces instructions permettent la comparaison de deux opérandes passés en paramètre et, en fonction du résultat du test, positionnent les flags (CF, SF, AF, etc.). Ces instructions sont liées aux instructions de saut conditionnel.

CMP, TEST.

Pour les exemples qui vont suivre, je vais vous décrire l'instruction la plus utilisée dans le microprocesseur: **MOV**

MOV registre1, registre2 a pour effet de copier le contenu du registre2 dans le registre1, le contenu préalable du registre1 étant écrasé. Cette instruction vient de l'anglais « move » qui signifie « déplacer » mais attention, le sens de ce terme est modifié, car l'instruction MOV ne déplace pas mais place tout simplement. Cette instruction nécessite deux opérandes qui sont la destination et la source. Ceux-ci peuvent être des registres généraux ou des emplacements mémoire. Cependant, les deux opérandes ne peuvent pas être toutes les deux des emplacements mémoire. De même, la destination ne peut pas être ce qu'on appelle une valeur immédiate (les nombres sont des valeurs immédiates, des valeurs dont on connait immédiatement le résultat) donc pas de **MOV 10, AX**. Ceci n'a pas de sens, comment pouvez-vous mettre dans le nombre 10, la valeur d'AX ? 10 n'est pas un registre.

Exemples :

```
mov  ax, bx     ; Transfert d'un registre de 16 bits vers un registre de 16 bits
mov  ah, cl     ; Transfert d'un registre de 8 bits vers un registre de 8 bits
mov  ax, Val1   ; Transfert du contenu d'une case mémoire 16 bits vers AX
mov  Val2, AL   ; Transfert du contenu du AL vers une case mémoire d'adresse Val2
mov  esi, edi   ; Transfert d'un registre de 32 bits vers un registre de 32 bits
```

Remarque :

Il est strictement interdit de transférer le contenu d'une case mémoire vers une autre case mémoire comme suit :

```
mov  Val1, Val2
```

Pour remédier à se problème on va effectuer cette opération sur deux étapes :

```
mov al, Val2
mov Val1, al
```

On n'a pas le droit aussi de transférer un registre segment vers un autre registre segment sans passer par un autre registre :

```
mov ds, es
```

On va passer comme la première instruction :

```
mov ax, es
mov ds, ax
```

Le **CS** n'est jamais utilisé comme registre destination. Une autre règle à respecter, les opérandes doivent être de la même taille donc pas de **MOV AX, AL** ou **MOV EBX, DX** cela me semble assez logique. On ne peut pas utiliser une valeur **immédiate** avec un registre de segment (**MOV ES, 5** : impossible mais **MOV ES, AX** : possible).

Exemples :

MOV AL, CL ; place le contenu de CL dans AL ; Donc si AL=5 et CL=10, nous aurons AL=10 et CL=10

MOV CX, ES: [DI] (place dans CX, le contenu 16 bits de l'emplacement ES:[DI]). Donc si le

Word (le word est l'unité correspondant à 16 bits) ES: [DI]=34000, nous aurons CX=34000.

MOV CL, DS: [SI] (place dans CL, le contenu 8 bits de l'emplacement DS:[SI]).

Donc si DS:[SI] (attention en 8 bits) = 12, CL vaudra 12.

Au sujet de ces emplacements mémoires, il faut que vous vous les représentiez comme des "cases" de 8 bits, comme le plus petit registre fait 8 bits, on ne peut pas diviser la mémoire en emplacements plus petits. On peut prendre directement 16 bits en mémoire (un word) ou carrément un Dword (32 bits). Il est très important de saisir cette notion. Un exemple concret : vous avez des livres sur une étagère, le livre se trouvant toute à gauche est le 1er de la liste, nous nous déplaçons de gauche à droite.
Un livre est un byte (8 bits), 2 livres sont un word (16 bits), 4 livres font un Dword
(32 bits). Si nous faisons MOV AX, ES:[DI], nous allons prendre 2 livres, en commençant par le livre se trouvant à l'emplacement DI et en prenant ensuite le livre se trouvant
À l'emplacement DI+1 (l'emplacement suivant dans votre rangée de livre, 1 byte plus loin en mémoire).

Voici un autre exemple :

MOV EAX, ES:[DI] (place un Dword depuis ES:[DI] dans EAX). C'est comme si on copiait
4 livres dans EAX.

Et quelques exemples incorrects :

MOV 1,10 (impossible et pas logique)

MOV ES:[DI], DS:[SI] (incodable malheureusement)

MOV ES, 10 (incodable, il faut passer par un registre général donc MOV ES, AX ou MOV ES, CX mais pas de MOV ES, AL).Ce qu'il faut retenir c'est que l'instruction MOV est comme le signe '='
MOV ES, CX c'est comme faire ES=CX.

Voilà ! Une partie pénible de terminer. Par contre, je n'en ai pas détaillé pour le moment les instructions car je pense que vous comprendrez mieux le moment venu, avec un exemple concret d'utilisation.

Je suis bien conscient que ces chapitres étaient riches en vocabulaire, en nouveautés mais bon, voilà : votre calvaire... ne fait que commencer. Eh oui : tout ceci était un amuse-bouche ! Maintenant, nous rentrons dans le vif du sujet. Cela fait déjà une bonne base pour commencer. Si vous connaissez déjà tous ça sur le bout des doigts, vous êtes bien parti. D'ailleurs j'en profite pour vous rappeler qu'il est nécessaire de connaitre ces chapitres pour pouvoir passer aux suivants. En fait, je dirais même que vous n'y parviendrez pas avant un bon bout de temps... Mais tout arrive avec le temps ;) Donc patience... Et surtout bonne chance ! Et n'oubliez pas que pour apprendre, rien ne vaut la pratique ! Mais rassurez-vous, à partir de maintenant tout ce que nous allons faire va apparaître à l'écran. J'espère que vous avez encore de l'énergie, car, dans le chapitre suivant, nous aborderons quelque chose de bien plus passionnant...

Bon ! Tenez-vous prêts pour faire vos premiers codes Assembleur.

Les outils nécessaires au programmeur

Pour développer des programmes, il faut des outils. Ces outils sont de plusieurs catégories et plus ou moins performants et complexes en fonction de leur usage. Nous allons expliquer l'utilité de chacun d'entre eux. Mais avant ça, faisons un petit récapitulatif.

Chaque type de processeur comprend son propre langage machine. Les instructions dans le langage machine sont des nombres stockés sous forme octets en mémoire. Chaque instruction à son propre code numérique unique appelé code d'opération ou Opcode. Les instructions des processeurs x86 varient en taille. L'Opcode est toujours au début de l'instruction. Beaucoup d'instructions comprennent également les données (des constantes ou des adresses) utilisées par l'instruction. Le langage machine est très difficile à programmer directement. Déchiffrer la signification d'instructions codées numériquement est fatigant pour des humains. Par exemple, l'instruction qui dit d'ajouter les registres EAX et EBX et de stocker le résultat dans EAX est encodée par les codes hexadécimaux suivants :

03 C3

C'est très peu clair. Heureusement, un programme appelé un assembleur peut faire ce travail laborieux à la place du programmeur.

Un programme en langage d'assembleur est stocké sous forme de texte (comme un programme dans un langage de plus haut niveau). Chaque instruction assembleur représente exactement une instruction machine. Par exemple, l'instruction d'addition décrite ci-dessus serait représentée en langage assembleur comme suit :

ADD EAX, EBX

Ici, la signification de l'instruction est beaucoup plus claire qu'en code machine. Le mot **ADD** est une mnémonique pour l'instruction d'addition.

Un assembleur est un programme qui lit un fichier texte avec des instructions assembleur et convertit l'assembleur en code machine. Les compilateurs sont des programmes qui font des conversions similaires pour les langages de programmation de haut niveau. Un assembleur est beaucoup plus simple qu'un compilateur. "Cela a pris plusieurs années aux scientifiques de l'informatique pour concevoir le simple fait d'écrire un compilateur, En effet, à votre titre d'information, la conversion du code source en code machine est très complexe. Elle implique de comprendre comment transcrire un ensemble d'instructions de haut niveau en instructions machine de bas niveau, très spécifiques. Lorsque le processus est complet, un exécutable est crée. C'est ce programme que vous pouvez exécuter. Des livres entiers existent sur la façon de convertir les langages de programmation de haut niveau en langage machine. Heureusement pour vous, les vendeurs de compilateurs ont lu tous ces livres : vous n'avez pas besoin de comprendre comment cela fonctionne Vous avez juste à assembler votre code avec un assembleur les programmes que vous avez écrits" Si vous désirez approfondir vos connaissances dans le domaine de la compilation, vous devez essayer : ***Engineering a Compiler****, Keith Cooper, Linda Torczon.* ***Advanced Compiler Design and Implementation****, Morgan Kaufmann.*

Chaque instruction du langage d'assembleur représente directement une instruction machine. Les instructions d'un langage de plus haut niveau sont beaucoup plus complexes et peuvent requérir beaucoup d'instructions machine. Une autre différence importante entre l'assembleur et les langages de haut niveau est que comme chaque type de processeur à son propre langage machine, il a également son propre langage d'assemblage. Porter des programmes assembleur entre différentes architectures d'ordinateur est beaucoup plus difficile qu'avec un langage de haut niveau.

Comme vous avez sûrement dû constater, chaque famille de processeur utilise un jeu d'instructions différent. Il existe beaucoup de langages assembleurs, qui changent selon l'architecture (*Intel x86, MIPS, ARM, Motorola, Zilog, Transmeta, Texas Instruments, VIA, Atmel* ...) et dans une moindre mesure selon les systèmes d'exploitation (Linux, Windows...).

J'ai choisis de se concentrer sur le langage assembleur des processeurs x86 de la famille Intel parce qu'il est utilisé sur chaque PC dans le monde et largement le plus populaire de l'architecture des processeurs.

Alors à votre avis, de quels outils un programmeur a-t-il besoin ? Si vous avez attentivement suivi cette introduction, vous devez en connaître au moins un !

Vous voyez de quoi je parle ? Vraiment pas ?

Eh oui, il s'agit de l'incontournable assembleur, sujet principal de ce guide, ce fameux programme qui permet de traduire votre langage assembleur en langage machine. Mais cela ne suffit pas ! Il faut d'autres outils pour mettre au point les logiciels, qu'ils soient petits ou grands.

Bon, de quoi d'autre a-t-on besoin ?

Je ne vais pas vous laisser deviner plus longtemps. Voici le strict minimum pour un programmeur :

- Un **éditeur de texte** qui vous permet de modifier votre code source sans quitter l'environnement.
- Un **assembleur** permettant de faire la traduction du langage assembleur en langage machine.
- Un **éditeur de lien « linker »** permettant la création du fichier exécutable.
- Un **éditeur de ressources** permettant de créer des ressources Windows telles que des bitmaps, des icônes, des boites de dialogues et des menus.
- Un **débuggeur** pour vous aider à traquer les erreurs dans votre programme.

Détaillons ce baratin :

Le programme doit être saisi dans un fichier texte d'extension .asm non formaté « c'est-à-dire sans caractères en gras, souligné, avec des polices de caractères de différentes tailles, ... » appelé fichier source. En théorie un logiciel comme le Bloc-notes sous Windows fait l'affaire. L'idéal, c'est d'avoir un éditeur de texte intelligent proposant des fonctions de glisser-déposer, doté d'une mise en couleur automatique des mots clés, des commentaires et d'autres éléments, d'une mise en retrait automatique et « intelligente », et d'une saisie semi-automatique affichant les paramètres de fonctions Windows en cours de frappes, ce qui vous permet de vous repérer dedans bien plus facilement.

L'assembleur transforme le fichier source assembleur en un fichier binaire dit « fichier objet » (code en langage machine, non exécutable car incomplet). Il s'agit d'un fichier avec l'extension .obj. Ce fichier contient des instructions et des données en langage machine, et chaque donnée a un nom symbolique par lequel elle est référencée.

L'éditeur de lien lie les codes objets et résout les références. C'est lui qui produit le fichier exécutable. Il li les modules objets, va chercher les fonctions situées dans des bibliothèques de fonctions ou de routines. Il permet également de créer par exemples les librairies .lib, ou les .dll.

Si vous avez passé votre nuit sur un programme et que, au matin, rien ne fonctionne encore, vous avez quatre possibilités :

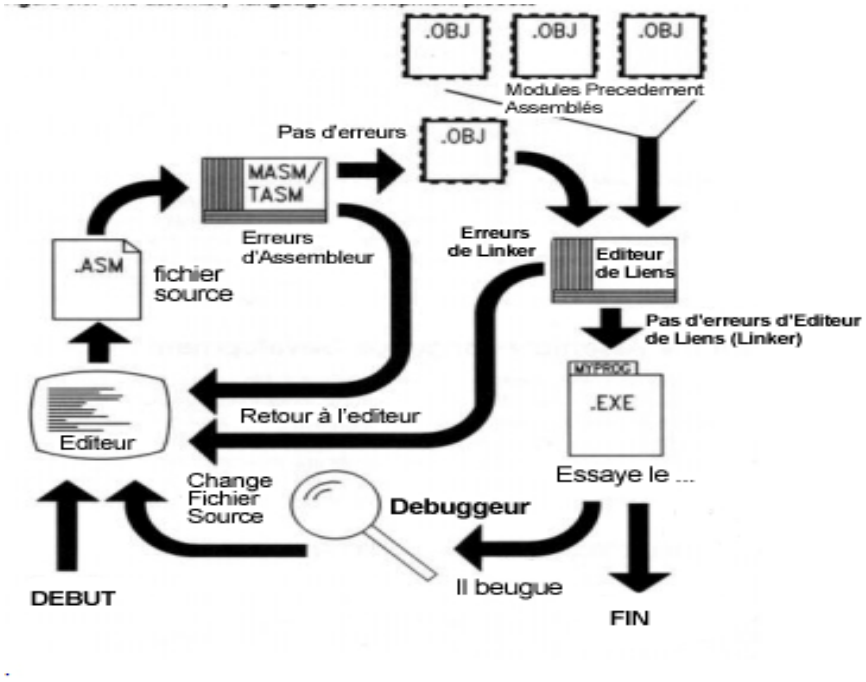
- Abandonner (dommage).
- Vous procurer un ordinateur très très lent pour essayer de voir ce qui ne va pas durant l'exécution du programme (pas bon pas bon).
- Remplir votre programme d'instructions d'affichage de façon qu'un petit message s'affiche après l'exécution de chaque ligne de code, pour vous annoncer que telle ligne vient d'être exécuté, ce qui vous permettra de déterminer l'endroit où quelque chose se déroule mal (ça fonctionne mais c'est ce qu'on appelle une galère).
- Vous employer un débogueur (cela fonctionne et ce n'est pas une galère).

Ban, dès que votre programme dépassera quelques lignes, il est presque certain que vous vous heurtez à quelques problèmes lors du premier assemblage. Si ce n'est pas le cas, vous êtes un super programmeur ou vous avez recopié la source dans un livre. Un **débuggeur** est un outil qui vous permet d'exécuter votre programme ligne par ligne. Cela facilite le contrôle de la logique de votre programme et la compréhension de son fonctionnement, la

localisation et la correction des erreurs. Bien que le choix du débogueur soit personnel. Je suggère à tous les débutants de commencer avec un débogueur comme OllyDbg, qui est un analyser, debugger et assembleur 32 bits avec une interface intuitive. Il est bien puissant et ses plugins lui permettent même de rivaliser avec le grand *SoftIce*.

Les **ressources** définissent l'interface utilisateur du programme Windows. Ce sont elles qui distinguent les programmes Windows des programmes DOS. Les programmes Windows comportent des fenêtres que l'on appelle parfois juste des dialogues, des menus, des bitmaps et toutes sortes de choses qui facilitent l'exploitation d'un programme. Mais les ressources ne sont pas le propre de Windows : Les systèmes d'exploitation Macintosh, Unix, OS/2 et bien d'autres emploient également les ressources.

Pour faire bref, un programme écrit en assembleur doit subir un certain nombre de transformations avant de pouvoir être exécuté. La figure suivante présente les différentes étapes du traitement d'un programme en langage d'assemblage.



A partir de maintenant on a 2 possibilités :

Soit on récupère chacun de ces programmes séparément. C'est la méthode la plus compliquée, mais elle fonctionne. D'ailleurs bon nombre de programmeurs préfèrent, "utiliser ces programmes séparément" et utiliser un simple fichier .bat pour lancer l'assemblage, l'artillerie lourde n'est pas indispensable et les logiciels avec surenchères de fonctionnalités inutiles, c'est juste bon à perdre du temps. Une simple fonction "Go To Line" à la rigueur est tout ce qu'il faut pour eux. Je ne détaillerai pas cette méthode ici, je vais plutôt vous parler de la méthode simple.

Soit on utilise un programme "Tout-en-1" qui combine un éditeur de texte, un assembleur, un éditeur un lien, un éditeur de ressource et un débuggeur. Ces programmes "Tout-en-1" sont appelés IDE, ou encore **Environnements de développement Intégré**.

L'environnement de développement intégré a été conçu pour vous permettre de vous déplacer de fenêtre en fenêtre et de fichier en fichier de différentes façons, selon vos préférences ou impératifs de projet. Vous pouvez choisir de parcourir tous les fichiers ouverts de l'éditeur ou de parcourir toutes les fenêtres Outil actives dans l'IDE. Vous pouvez aussi basculer directement vers tout fichier ouvert dans l'éditeur, indépendamment de son dernier ordre d'accès. Ces fonctionnalités peuvent contribuer à accroître votre productivité lorsque vous travaillez dans l'IDE.

Bien que des IDE pour plusieurs langages existent. Bien souvent (surtout dans les produits commerciaux) un IDE est dédié à un seul langage de programmation. On peut également trouver dans un IDE un système de gestion de versions et différents outils pour faciliter la création de l'interface graphique.

Il existe plusieurs environnements de développement. Vous aurez peut-être un peu de mal à choisir celui qui vous plaît au début. Une chose est sûre en tout cas: vous pouvez faire n'importe quel type de programme, quel que soit l'IDE que vous choisissez.

Il m'a semblé intéressant de vous montrer quelques IDE parmi les plus connus. Tous sont disponibles gratuitement. Personnellement, je navigue un peu entre tous ceux-là et j'utilise l'IDE qui me plaît selon l'humeur du jour.

- Un des IDE les plus connus : **RadAsm** est une petite merveille, convivial, riche en fonctionnalité, rapide, et d'une taille ridicule par rapport à d'autre EDI, il vous permettra de développer de vraies applications avec tout le confort que l'on peut espérer d'une application de qualité. C'est celui là que nous utiliserons tout au long de ce guide.
- Il existe **WinAsm Studio** qui ressemble très fortement à RadAsm avec peut-être un peu moins de fonctionnalités mais dont l'interface et beaucoup plus agréable et facile à utiliser. De plus, il se charge beaucoup plus vite et explique clairement comment fabriquer ses propres plugins.
- **AsmEditor** est un EDI permettant la création et la compilation de projets en langage assembleur. Il inclut un contrôle des compilateurs permettant de gérer autant de chaînes de compilations que nécessaire (pour différentes plateformes par exemple).
- **Easy Code** sous une apparence semblable à celle de Visual Basic, me parait aussi qu'il ferra parfaitement l'affaire.

Je vous recommande RadAsm, mais ce n'est pas une obligation. Quel que soit l'IDE que vous choisissez vous serez capables de faire autant de choses. Vous n'êtes pas limités. Nous avons fait le tour des IDE les plus connus. N'oubliez pas cependant qu'il en existe d'autres et que rien ne vous empêche de les utiliser si vous les préférez. Quel que soit l'IDE choisi, vous pourrez suivre sans problème la suite du cours.

On va encore devoir faire un choix sur l'assembleur qu'on devra implémenter dans RadAsm, en effet il existe plus d'une douzaine de différents assembleurs disponibles pour le processeur x86 fonctionnant sur PC. Ils différent en terme de fonctionnalités et de syntaxe. Certains sont adaptés pour les débutants, certains sont destinés uniquement aux programmeurs confirmés. Certains sont très bien documentés, d'autres ont peu ou pas de documentation. Certains sont enrichis par beaucoup d'exemples de programmation, certains ont très peu d'exemples de code. Certains assembleurs sont étoffés de tutoriels et des livres disponibles qui utilisent leur syntaxe, d'autres en manquent. Certains sont très basic, d'autres sont très complexes. Quel est le meilleur assembleur, alors?

Comme bon nombre de questions posées dans la vie, il n'y a pas de réponse simple à la question « Quel est le meilleur assembleur ? » La raison en est que différentes personnes ont des critères différents pour évaluer ce qui est « le meilleur ». A l'absence d'une échelle universelle pour juger les différents assembleurs, il n'y a aucun moyen de choisir un seul assembleur et le désigner le meilleur.

Historiquement, MASM, l'assembleur de Microsoft, est une évolution du premier assembleur 8088 développé pour IBM. MASM a été pendant longtemps la seule solution pour le développeur en assembleur. La dernière version s'appelle maintenant ML. Pendant longtemps, la seule vraie alternative a été le *Turbo Assembleur* (*TASM*) de Borland, qui fut plébiscité dès sa mise à disposition par une grande majorité d'utilisateurs pour sa rapidité et sa souplesse. Avec le passage à Windows 9x, les outils de développement de type macro-assembleurs n'ont plus été une priorité pour les éditeurs. Ces solutions étaient performantes mais payantes. Maintenant, et depuis quelque année, grâce à l'émancipation des logiciels du monde libre, de nouveaux assembleurs tout aussi performants sont apparus. Le plus connu de ces assembleurs libres est NASM (*The Netwide Assembler)*. Mais il existe également d'autres solutions comme FASM (*Flat Assembler*), GoAsm, RosAsm, HLA (*High level assembly*), Gas (*GNU Assembler*)… Autant le dire tout de suite : il n'y a pas de raison absolue pour préférer MASM à un autre assembleur. Tous les exemples qui sont développés dans ce guide seront spécifiques à MASM en particulier. MASM reste encore aujourd'hui un des assembleurs phare de la communauté des programmeurs en assembleur pour la plateforme Win32, mais d'autres assembleurs rassemblent chacun une communauté importante.

- MASM supporte une grande variété de macros aidant à la programmation en langage assembleur ainsi que des idiomes de programmation structurée, incluant des constructions de haut niveau pour les boucles, les appels de procédures, les branchements, etc. ce qui fait de MASM un assembleur à programmation de haut niveau.
- De nombreux projets supportant MASM ont vu le jour. Ainsi des environnements de développement intégré permettent un développement plus aisé avec MASM.
- De nombreux forums ou sites internet proposent des codes sources de la documentation ou de l'aide concernant cet assembleur qui malgré son ancienneté reste un des assembleurs les plus supportés.
- Le support officiel de MASM par Microsoft se résume aujourd'hui à ajouter des instructions lorsque de nouveaux processeurs voient le jour et à améliorer la prise en charge du 64 bits.

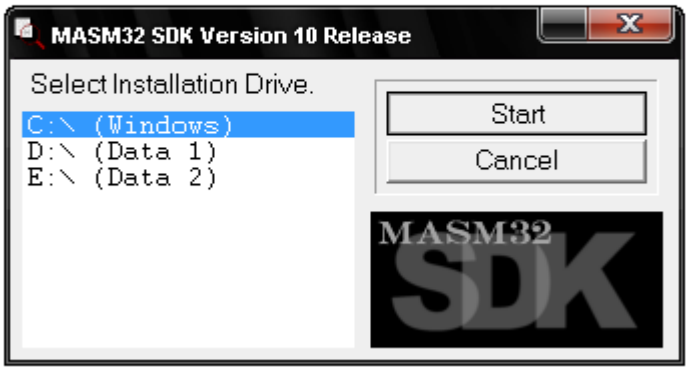
La dernière des exigences qui s'ajoutera à celles ci-dessus sera remplie, soit en possèdent l'ancien fichier d'aide *Win32.hlp* ou par la mise à jour MSDN (**Microsoft Developer Network**) des bibliothèques disponibles en ligne dans le cadre de la Plate-forme SDK (**Software Development Kit**) disponible en téléchargement libre chez Microsoft.

Toutes les Informations que vous pourrez tirer profit dans la suite de ce guide vont se faire en deux catégories principales:

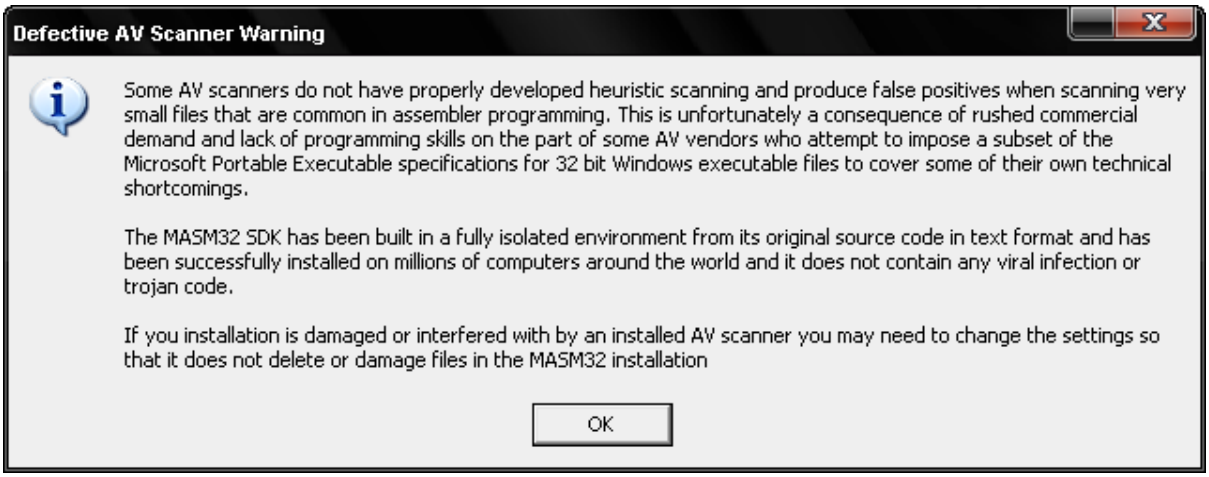
- MASM32 syntaxe - Comment écrire correctement les instructions en ASM sous MASM.
- L'architecture Windows et ses APIs.

Installation de Masm

Une fois que vous avez récupéré Masm, décompresser où vous voulez, l'installation de Masm ne pose pas de problème mais, pour que chacun soit bien rassuré, nous en montrons ici les étapes. Pour démarrer, lancez *install.exe,* cette page devrait s'afficher :



Cliquer sur « Start », pour indiquer que vous acceptez d'installer Masm dans la racine C:\ et cliquer sur « OK ». Ensuite, cliquer de nouveau sur « OK », La présence d'un virus dans le package de RadASM est hautement improbable.



Rien de particulier à signaler. Cliquez sur « OK « autant de fois que nécessaire en laissant les options par défaut, peu de temps après, vous devriez avoir une fenêtre comme celle-ci :

L'installation est maintenant terminée avec succès, cliqué sur n'importe quel bouton pour continuer …

Pour s'assurer que l'installation s'est passée sans encombre, vous devriez apparaitre le dossier masm32 dans C:\.

Configuration de RadAsm

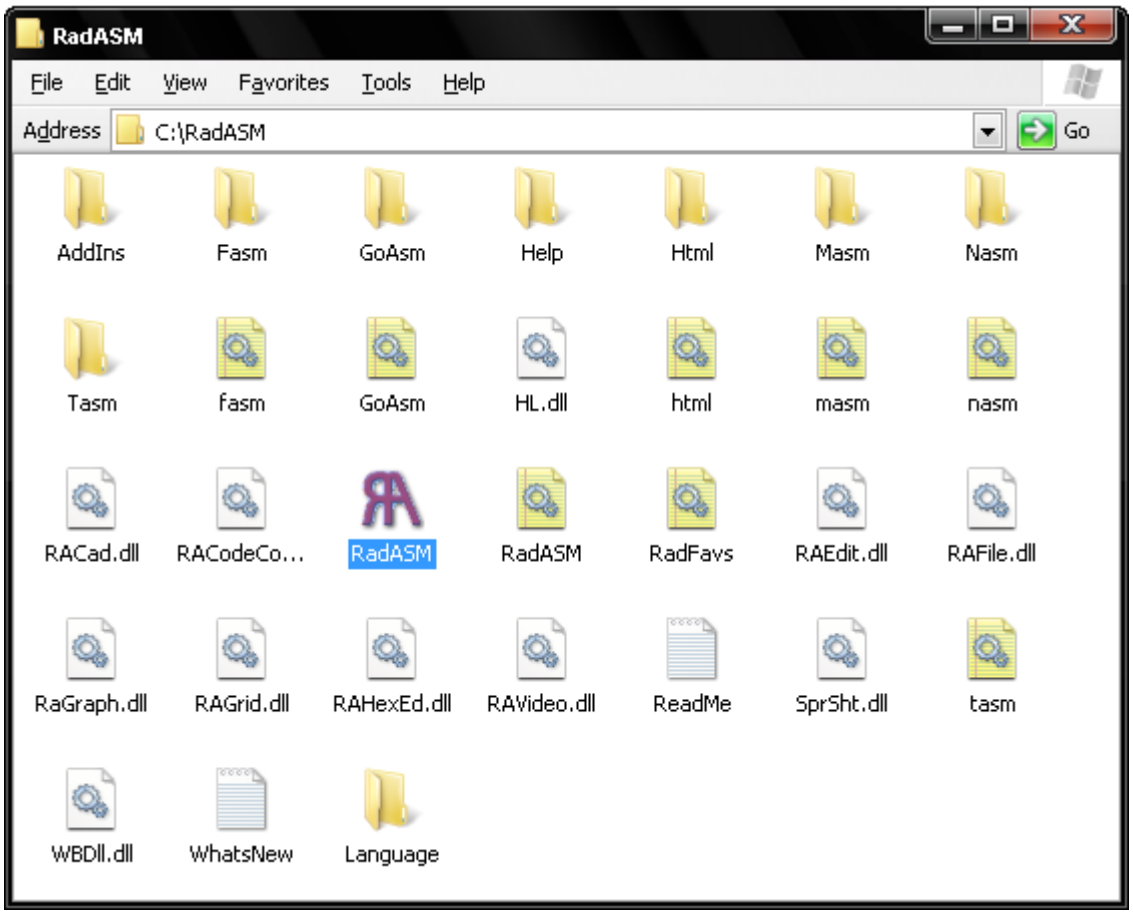
D'abord, n'hésitez pas à consulter le site de l'auteur http://www.radasm.com/ afin d'être sûr de bien récupérer tous les fichiers archive constituant l'application. Ceux qui nous intéressent sont les suivants :

- **RadAsm 2.2.1.6**: contient l'IDE (nécessaire).
- **RadAsm Assembly programming** : si vous utilisez les langages type assembleur (nécessaire).
- **RadAsm language pack** : si vous souhaitez traduire l'interface (facultatif).

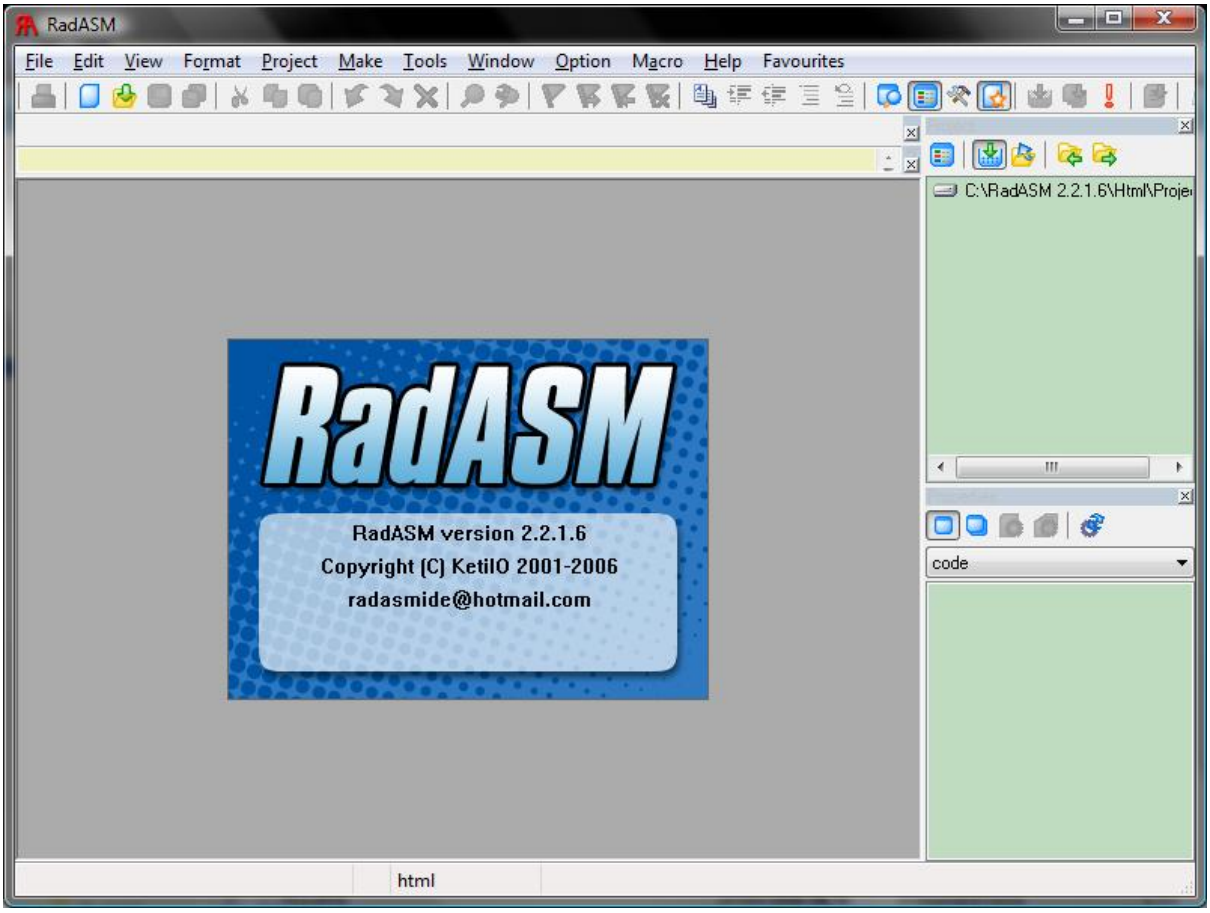
Vous devez maintenant avoir une archive contenant les trois fichiers. Décompressez-les où vous voulez, puis vous devriez avoir ceci :

Maintenant, il va falloir faire une petite manipulation pour simplifier la suite.

Dans le fichier RadLNG, copier le fichier « **Language** » dans le fichier « **RadAsm** ». De même mettez le contenu du fichier « **Assembly** » dans le fichier « **RadAsm** ».



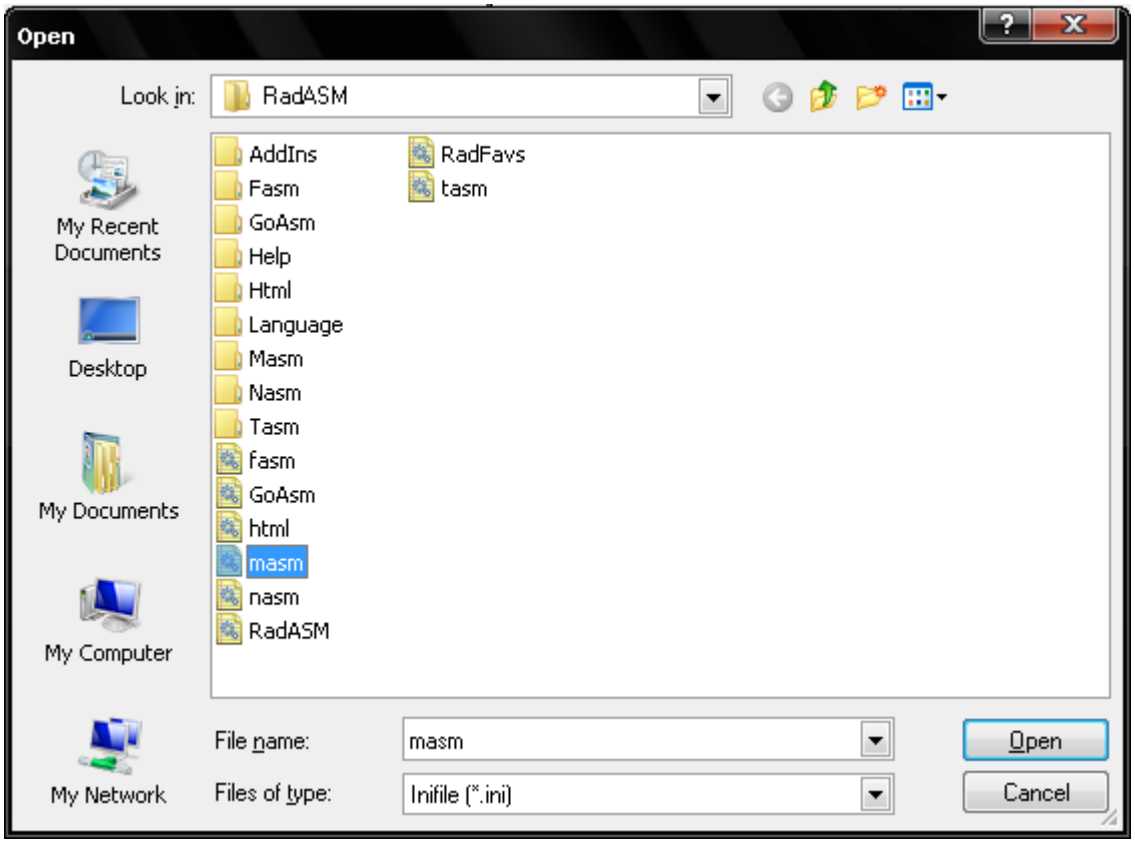
Lancez RadAsm, un fichier sera ouvert automatiquement.



Comme tout programme, il y en a eu plusieurs versions. Les captures d'écran que je fais sont sur la version 2.2.1.6 comme vous pouvez le voir. Ce genre de programme évolue rapidement, mais si vous avez une version supérieure ne vous inquiétez pas. Le fonctionnement du programme ne change pas d'une version à l'autre.

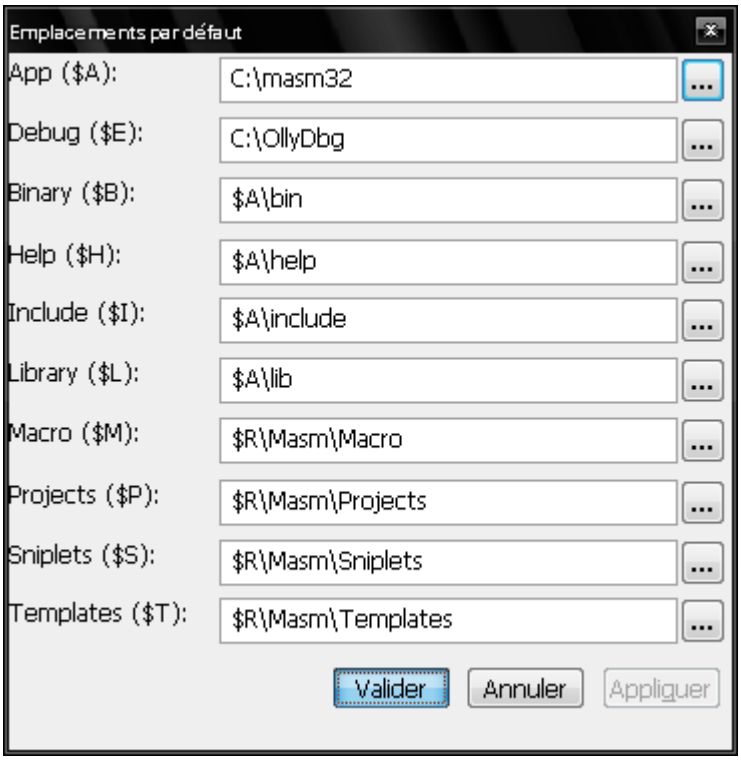
Oui, par défaut RadAsm est en anglais. Vous pouvez le modifier en allant en haut dans le menu, **Option / Language**, choisissez « **Français** », et enfin « **Appliquer** », **OK**.

Rendez-vous encore sur **Option / Langage de programmation**, supprimez **HTML** cala ne servira absolument à rien. Cliquez cependant sur le bouton "..." en bas à droite. Une nouvelle fenêtre s'ouvre : choisissez **masm**.



Enfin, confirmer votre modification en cliquons sur **Valider**.

Enfin, il nous reste plus qu'à indiqué à RadAsm l'emplacement de Masm, alors pour ce faire, dans **Option / Emplacement par défaut**, vous devez simplement remplir les champs suivants. Dans mon cas, c'est :

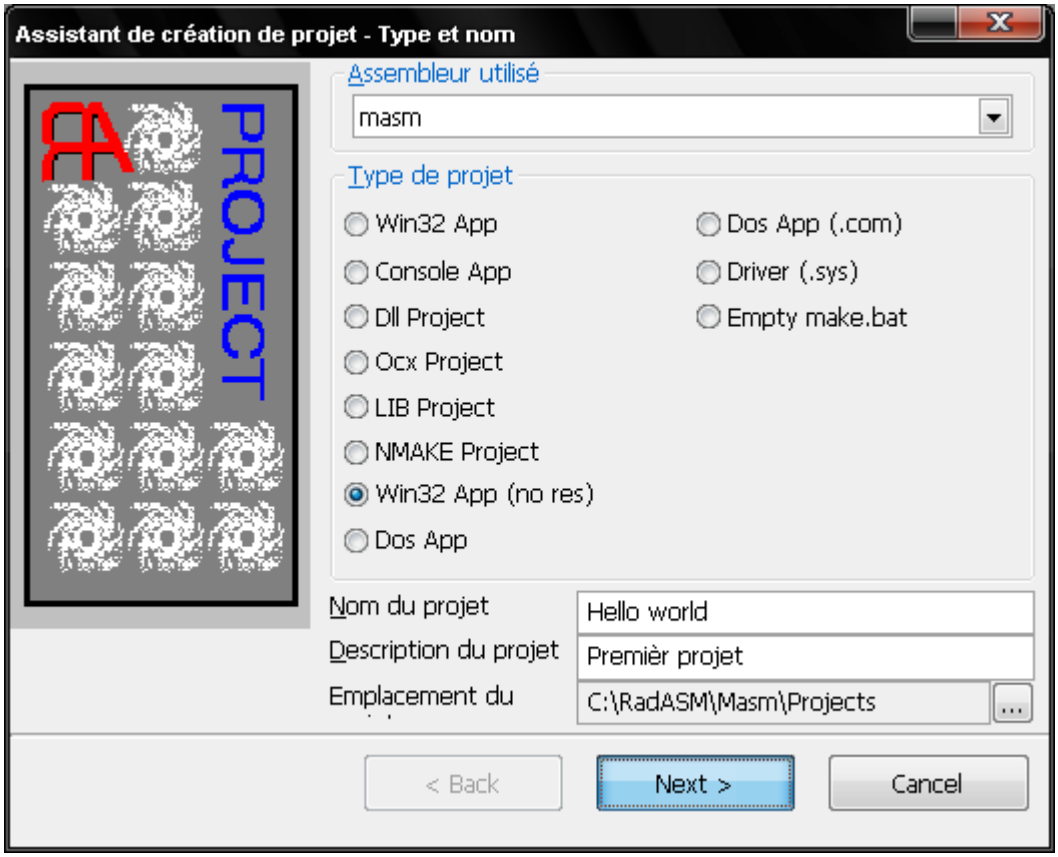


Cliquez ensuite sur **Valider**, puis **Appliquer**.

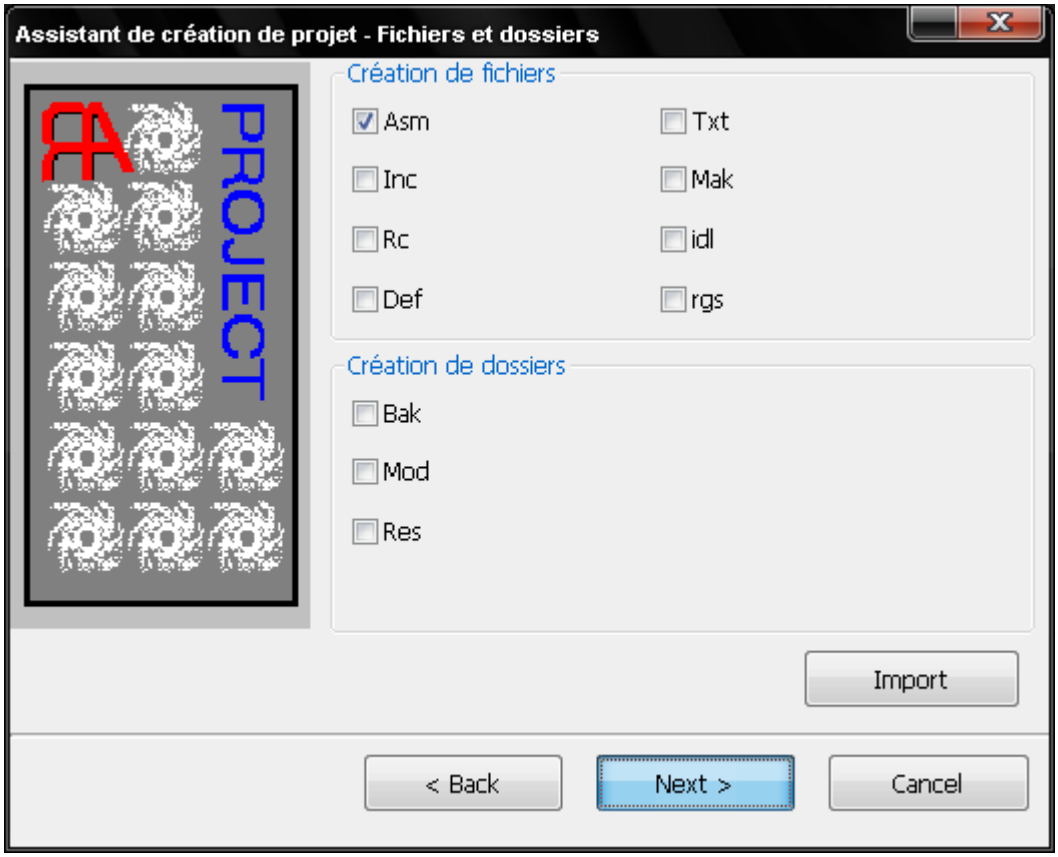
Créer son premier projet

Maintenant que notre environnement est tout à fait prêt à être utilisé. On peut passer à l'utilisation. Au départ, rien ne s'affiche. Il va falloir demander à RadAsm de créer un nouveau projet. Un projet c'est l'ensemble de tous les fichiers source du programme. En effet, quand on programme, on sépare souvent notre code dans plusieurs fichiers différents. Ces fichiers seront ensuite combinés par l'assembleur qui en fera un exécutable. Justement, la gestion de projet fait partie des principaux caractéristiques d'un IDE.

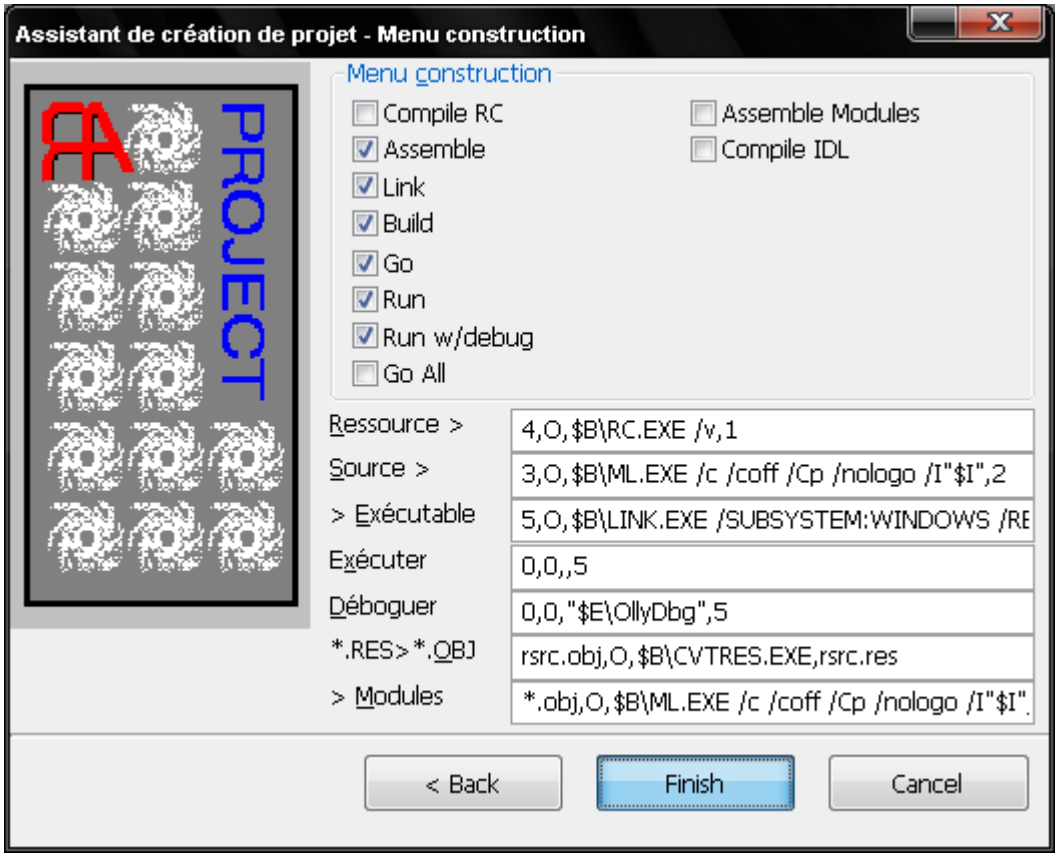
Retenez bien la marche à suivre, car vous devrez faire cela la plupart du temps. Pour créer un nouveau projet c'est simple : allez dans le menu **Fichier / Nouveau Projet**. Vous devriez voir quelque chose qui ressemble à ça :



On vous demande le nom et la description de votre projet, et dans quel dossier les fichiers source seront enregistrés, que du choix n'est ce pas ? Cliquer sur « Next » deux fois.



Veillez à ce que « Asm » soit coché, ensuite cliquez sur « Next ».



Cliquez sur « Finish », eh voilà, le tour est joué et le projet créé. Vous allez pouvoir commencer à écrire du code !

Maintenant que ces quelques mots ont été dits, je vais maintenant vous faire faire un tour rapide de l'interface que vous propose RadAsm, en gros, des éléments dont nous allons nous servir dans ce guide.

Présentation rapide de l'interface

Avant de commencer, regardez bien les raccourcis clavier présents dans les menus... Ils sont très utiles et peuvent vous faire gagner beaucoup de temps. On n'est pas obligés bien sûr, mais croyez-moi, quand on y a goûté, on ne peut plus s'en passer…

Les raccourcis à retenir sont :

- **CTRL + SHIFT + N** : Nouveau projet
- **CTRL + SHIFT + S** : Enregistrer tous les fichiers
- **CTRL + F5** : Assembler puis exécuter
- **CTRL + S** : Enregistrer
- **CTRL + C** : Copier la sélection
- **CTRL + X** : Couper la sélection
- **CTRL + V** : Coller la sélection

- **CTRL + Z** : Annuler l'action précédente
- **CTRL + A** : Sélectionner tout
- **CTRL + F** : Chercher / Remplacer

Voyons voir plus en détail comment RadAsm est organisé :

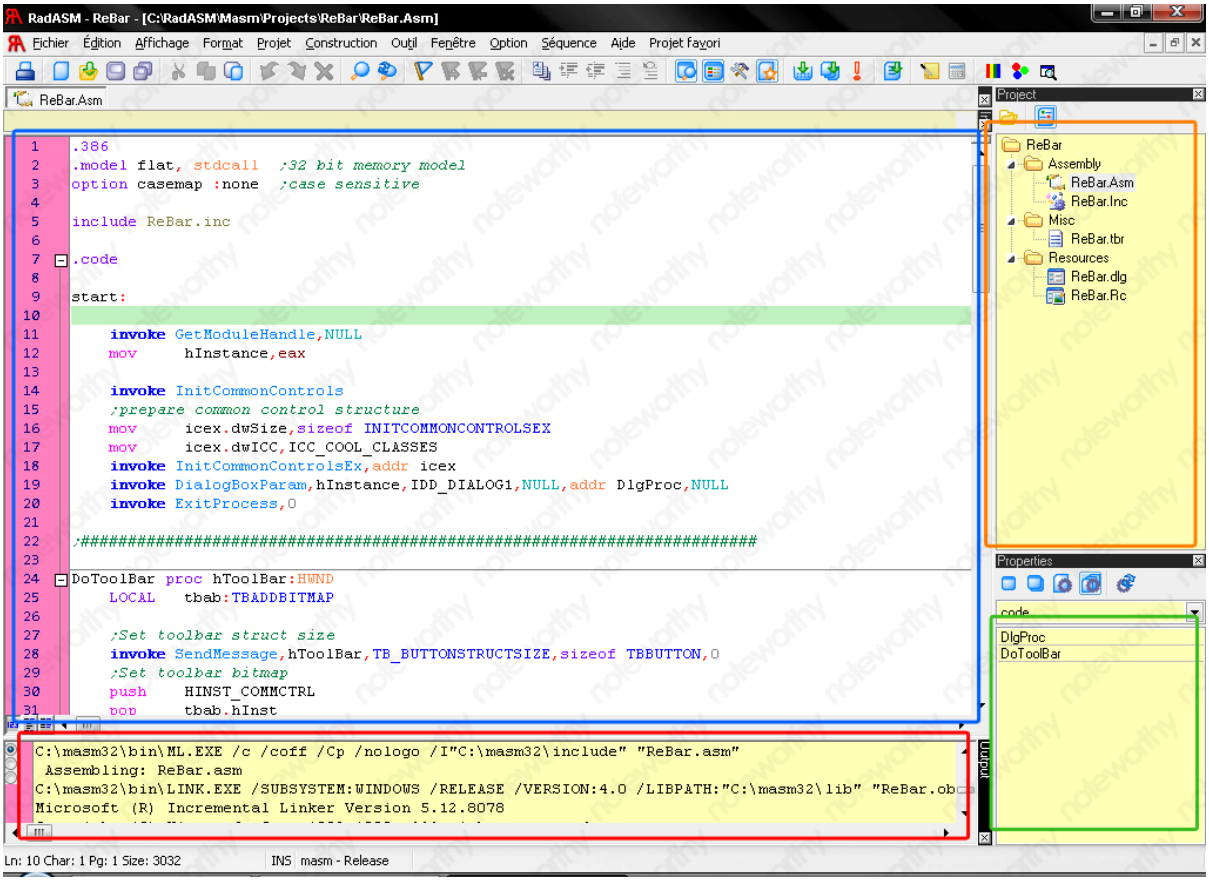
Ces 4 icônes sont sans aucun doute les plus utilisées, et pour cause : ce sont elles qui permettent d'appeler l'assembleur pour créer un exécutable de votre projet. Dans l'ordre, de gauche à droite, ces icônes signifient :

Assembler : tous les fichiers source de votre projet sont envoyés au compilateur qui va se charger de créer un exécutable. S'il y a des erreurs, l'exécutable ne sera pas créé et on vous indiquera les erreurs en bas RadAsm.

Construire : quand vous faites **Assembler**, RadAsm n'assemble en fait que les fichiers que vous avez modifiés et pas les autres. Parfois, je dis bien parfois, vous aurez besoin de demander à RadAsm de vous recompiler tous les fichiers. On verra plus tard quand on a besoin de ce bouton, et vous verrez plus en détail le fonctionnement de la compilation dans un chapitre futur. Pour l'instant, on se contente de savoir le minimum nécessaire pour ne pas tout mélanger.

Exécuter : cette icône lance juste le dernier exécutable que vous avez assemblé. Cela vous permettra donc de tester votre programme et voir ainsi ce qu'il donne.

Construire & Exécuter : évidement, ce bouton est la combinaison des deux boutons précédents. C'est d'ailleurs ce bouton que vous utiliserez le plus souvent. Notez que s'il y a des erreurs pendant l'assemblage (pendant la génération de l'exécutable), le programme ne sera pas exécuté.



Dans l'encadré orange, vous trouverez le dossier de votre projet ainsi que son contenu. Ici, vous pourrez gérer votre projet comme bon vous semble (ajout, suppression...). Nous verrons un peu plus tard quels sont les différents types de fichiers qui constituent un projet

Dans l'encadré bleu, je pense que vous avez deviné... C'est ici que nous allons écrire nos codes sources.

Dans l'encadré rouge, c'est là que vous verrez apparaître le contenu de vos programmes ainsi que les erreurs éventuelles.

Et enfin pour finir, **dans l'encadré vert**, dès lors que nous aurons appris à coder nos propres fonctions, c'est ici que la liste des méthodes et des variables sera affiché

Selon vos goûts, vous pourrez ultérieurement personnaliser un certain nombre d'autres éléments de l'interface et de l'éditeur, en actionnant les commandes du menu Outils : Par exemple, vous pouvez faire en sorte que les numéros de ligne apparaissent dans la gouttière (la marge gauche).

Notez enfin que vous pouvez alléger la barre d'outils en choisissant les groupes de boutons qui y apparaissent. Pour cela, cliquez avec le bouton droit de la souris dans la barre d'outils (mais pas sur un bouton) :

Ce description ne prétend pas être complète, il s'agissait tout d'abord d'effectuer un tour d'horizon de cet EDI qui je l'espère vous aura permis de mieux le connaître et d'en apprécier

la puissance. En outre, certaines fonctionnalités ou options m'ont sûrement échappé et vous ne manquerez pas de les découvrir par vous-même.

Squelette d'un programme en Assembleur

Nous en savons à présent suffisamment pour commencer à mettre pour de bon les mains dans le cambouis. Ouvrons donc un nouveau projet appelé Skeleton. Ce programme nous fournira une base de départ pour développer n'importe quel programme. Si vous ne comprenez pas quelques-uns des codes, ne paniquez pas. J'expliquerai chacun d'entre eux plus tard.

```
.386
.model flat, stdcall

.data

.data?

.const

.code
start:


end start
```

Nous analyserons maintenant ligne par la ligne. La plupart de ce code représente des directives.

Certains langages, dont l'assembleur, comportent un grand nombre de directives ou instructions dédiées à la gestion et au contrôle des processus de compilation ou d'assemblage. Ces directives dont le rôle est de contrôler et de mettre, place les bons paramètres en vue des différentes phases de l'assemblage, sont rassemblés en plusieurs groupes selon leur fonction. Il faut noter que la prise en charge et la syntaxe de ces directives varient d'un assembleur à l'autre, même si historiquement la plupart s'inspirent de Masm. Les directives font partie de la syntaxe du langage assembleur mais ne dépend pas du jeu d'instructions d'Intel. Des programmes assembleurs différents peuvent générer du code machine identique en ce qui concerne les processeurs Intel, tout en offrant un jeu de directives différent.

La différence entre majuscules et minuscules dans les noms des directives ne sont pas prises en considération. Par exemple, l'assembleur traitera dans la même façon .include et .include.

.386

Ces directives permettent de choisir le processeur cible. Elles déterminent ainsi comment doit être analysé le code qui le suivra. L'usage le plus fréquent de ces directives est d'en écrire une seule au début du programme source, en privilégiant le dernier processeur de la série.

Cette directive n'influence pas le codage des mnémoniques, mais restreint l'usage de certaines instructions. Par exemple, si la directive .386 est utilisé, vous pourrez employer les instructions spécifiques du 80386. Par contre, soumettant à l'assemblage le même code source, mais la directive .8086, il y aurait une erreur à chacune des instructions spécifiques du 80386 ! Mais, si dans le même code source vous mettez des instructions spécifiques au 80586, il y aura également erreur car ces instructions ne sont supportées ni par le 8086, ni par le 80386 ! Ces directives permettent d'assurer un contrôle lors d'assemblage descendant, c'est à dire de permettre le filtrage par rejet des instructions inexistantes sur d'anciens processeurs. Il y a en réalité deux formes presque identiques pour chaque modèle de directive. .386/.386p .486/.486p … Ces versions « p » sont nécessaires seulement quand votre programme emploie des instructions privilégiées. Les instructions privilégiées sont des instructions réservées par le système du CPU / fonctionnement en mode protégé.

.model flat, stdcall

.model est une directive d'assembleur a pour but de déterminer plusieurs caractéristiques importante dans un programme : le type de mémoire, la convention d'appel des fonctions. La syntaxe de la directive .model est la suivante :

.model MemoryModel ModelOptions

MemoryModel spécifie le type de mémoire de votre programme. Sous Win32, il y a seulement un type, le modèle **FLAT** « plat », vous vous souvenez ? On utilise un seul segment de 4 GB, bref, c'est une manière d'organiser la mémoire pour ne plus qu'il y ait de segments, et juste des adresses mémoire 32bits.

ModelOptions signifie : appel standard, c'est-à-dire la façon dont sont transmis les paramètres à la fonction, autrement l'ordre de passage des paramètres de la pile vers un Call, de « gauche à droite » ou « de droit à gauche » et aussi ça équilibrera l'encadrement de la pile après l'appel de la fonction. La plate-forme Win32 emploie exclusivement **STDCALL**, les paramètres sont transmis de la droite vers la gauche car les APIs utilisent ce format.

.data

Cette section contient les données initialisées de votre programme. Les variables que l'on donne une valeur dès le départ.

.data?

Cette section contient les données non initialisées de votre programme. Parfois vous voulez juste prévoir un espace mémoire pour contenir de futurs données, mais ne voulez pas l'initialiser (lui donner une valeur initiale). C'est le but de cette section. L'avantage des données non initialisées est que l'espace est alloué quand le programme est chargé dans la mémoire mais la taille de fichier sur le disque n'est pas augmentée.

.const

Il arrive parfois que l'on ait besoin d'utiliser une variable dont on voudrait qu'elle garde la même valeur pendant toute la durée du programme. C'est-à-dire qu'une fois déclarée, vous voudriez que votre variable conserve sa valeur et que personne n'ait le droit de changer ce qu'elle contient. Ces variables particulières sont appelées constantes, justement parce que leur valeur reste constante.

Vous n'êtes pas forcément obligé d'employer ces trois sections dans votre programme. Déclarez seulement la (ou les) section (s) que vous voulez utiliser. Par contre, il n'y a seulement qu'une section pour le code : **.code,** cette section représente les instructions proprement-dites du programme.

<label>

…

end <label>

Où <label> représente n'importe quelle étiquette arbitraire, et est employé pour marquer le début et la fin de votre code. Les deux "label" doivent être identiques. Toutes vos lignes de code doivent se trouver entre <label> et end <label>. Il remplace le main des langages **C / C++**.

Rien de bien compliqué ! Mais il a le mérite d'être compris aisément.

Chapitre 5 : L'environnement Windows

Contrairement à DOS qui est un environnement mono-tâche, mono-utilisateur par excellence, Windows depuis la version 95 est un vrai un système d'exploitation graphique et multitâche pouvant ainsi gérer plusieurs "tâches" simultanément. La programmation de logiciels sur cet OS nécessite donc quelques notions de l'API WIN32.

L'interface de programmation Win32 (en anglais, Win32 *Application Programming Interface* ou Win32API) est commune aux systèmes d'exploitation Windows 9x et Windows NT/2000. Les API Windows offrent aux programmeurs la possibilité d'interagir avec le système d'exploitation. Elles offrent des possibilités presque infinies, et dépassent de très loin les possibilités apportées par les environnements de développement (Visual Basic, Windev, ...). Par exemple, elles vous permettront de contrôler une application, d'accéder au registre Windows, de jouer des sons, mais aussi l'ouverture de connections vers Internet, l'écriture sur votre disque dur, sortir un beau listing sur votre imprimante …

Les API ne sont en fait que des fonctions semblables à celles que vous pouvez créer dans votre environnement de développement : en règle générale, on leur fournit un certain nombre de paramètres, et elles renvoient quelque chose, ou réalisent une action précise. Ces fonctions sont contenues dans des fichiers dll, tels "user32.dll", "kernel32.dll", ou bien d'autres encore. Les fonctions les plus couramment utilisées sont celles qui constituent Microsoft Windows lui-même. Ces procédures sont toutefois écrites en langage C, et doivent donc être déclarées avant de pouvoir les utilisées avec d'autres langages.

Les API Windows sont plutôt faciles à utiliser, une fois que l'on connaît leur déclaration et leurs paramètres. Leurs difficultés sont autres : les problèmes se posent généralement lorsqu'on cherche l'API qui nous rendrait service, puisqu'on se trouve alors confronté à des milliers de fonctions aux noms pas toujours très explicites. Lorsqu'enfin on a trouvé celle qui convient, on découvre qu'on est incapable de l'utiliser, car on ne connaît ni sa déclaration, ni ses paramètres, ni son utilisation ! Pour résoudre ce problème, il n'y a pas cinquante solutions : la première est de chercher des exemples utilisant cette API, la deuxième est d'utiliser la MSDN Online sur le site de Microsoft (qui contient entre autres cette documentation), ou encore le

fichier d'aide win32.hlp, disponible en téléchargement gratuitement. Ce fichier d'aide contient une description très complète des APIs de Windows, des structures à utiliser, etc.

Les programmes Win se lient dynamiquement avec ces DLLs, c'est-à-dire : Les codes des fonctions API ne sont pas inclus dans le programme Win exécutable. Dans votre programme, pendant son exécution, pour accéder à l'API désirée, vous devez déclarer cette information dans le fichier exécutable. L'information est dans une bibliothèque d'importation. Vous devez lier vos programmes avec les bibliothèques d'importation correctes sinon ils ne seront pas capables de retrouver la fonction de l'API souhaitée.

Quand un programme Win est chargé dans la mémoire, Windows lit l'information stockée dans le programme. Cette information inclut les noms des fonctions et les DLLs où résident ces fonctions. Ex : la DLL "User32.dll" contient la fonction **MessageBox** (ainsi qu'une multitude d'autres fonctions). L'information que cherchera Windows dans votre exécutable ce sera quelque chose comme **« Call USER32 ! MessageBox ».** Dès que Windows tombera sur un tel renseignement dans votre programme, il chargera la DLL et exécutera la fonction.

Il y a deux catégories de fonctions API : les uns pour ANSI et les autres pour Unicode. Les noms des fonctions API pour ANSI sont suivis du suffixe "A", ex : **MessageBoxA**. Ceux pour Unicode ont le suffixe "W". Windows 95 traite naturellement ANSI, et Windows NT Unicode.

D'habitude on se sert de la convention ANSI, qui pour l'ensemble des caractères proposés par votre ordinateur sont terminés par le caractère NULL. ANSI représente chaque caractère sur 1 octet. Il est suffisant pour les langues européennes, mais il ne peut pas manipuler la plupart des langues orientales qui ont plusieurs milliers de caractères uniques. C'est pourquoi UNICODE entre en jeu. Un caractère UNICODE a une taille de 2 octets, ce qui lui permet d'avoir une série de code de 65536 caractères uniques.

Mais la plupart du temps, vous emploierez un fichier INCLUDE (*.inc) qui peut déterminer et choisir les fonctions API appropriées à votre plate-forme suivant ce que doit faire votre programme. Référez-vous juste aux noms de fonction API sans le suffixe.

Traditionnellement, les tutoriaux pour la programmation, dont le but est de faire la démonstration rapide d'un langage de programmation commence, avec une application qui affiche le message "Hello World" dans une console. La fenêtres d'interface graphique équivalente à cela est la fonction **MessageBox**. Dans RadAsm pour, ouvrez un nouveau projet nommé **MessageBox**.


```
1   .386
2   .model flat, stdcall
3   option casemap:none
4
5   include windows.inc
6
7   include user32.inc
8   includelib user32.lib
9
10  include kernel32.inc
11  includelib kernel32.lib
12
13
14  .data
15  MsgBoxCaption BYTE "Hello World!", 0
16  MsgBoxText    BYTE "Un bon langage aujourd'hui vaut mieux qu'un langage parfait demain.", 0
17
18
19  .code
20  start:
21
22  invoke MessageBox, NULL, addr MsgBoxText, addr MsgBoxCaption, MB_OK + MB_ICONASTERISK
23  invoke ExitProcess,0
24
25  end start
```

Cliquez sur "**Go**" ou "**CTRL + F5**", puis coucou petite fenêtre, fait cheez à la caméra.

Maintenant nous allons analyser les nouvelles parties.

option casemap : none

.option est une directive d'assemblage qui indique que la différence entre majuscule et minuscule est importante, ainsi par exemple, **MessageBox** et **messagebox** seront complètement différent.

Notez la nouvelle directive, ***include***. Cette directive est suivie par le nom du fichier que vous voulez insérer à sa place. Dans cet exemple, quand Masm traite la ligne include windows.inc, Il ouvrira windows.inc qui se trouve dans le répertoire C:\MASM32\include, comme tout les autres fichiers **.inc** d'ailleurs, et fera en sorte que son contenu (celui de windows.inc) soit collé dans votre programme win32. Il ne contient pas de **prototype de fonction**. windows.inc n'est en aucun cas complet (c'est compréhensif car on peut toujours y rajouter de nouvelles choses). windows.inc est un fichier maître, celui-ci contient toutes les déclarations de constante, les appels de fonctions, les structures de données Win32, il contient notamment :

- **MB_OK**
- **NULL**

Celles-ci peuvent être utilisées par leur nom pour améliorer la lisibilité de votre code source.

T'es bien gentil, mais c'est quoi, un **prototype de fonction** ? Celle-ci appelle immédiatement une autre, c'est quoi une **fonction** ?

En programmation, on désigne par une fonction un **morceau de code** qui sert à faire quelque chose de précis. Une fonction exécute des actions et renvoie un résultat. On dit qu'une fonction possède une entrée et une sortie. Schématiquement, ça donne quelque chose comme ça :

Je vous l'ai dit dès le tout début de ce chapitre du cours, et je ne vous ai pas menti : les APIs Windows sont aussi des fonctions. Ce sont en quelque sorte des fonctions toutes prêtes qu'on utilise très souvent. Ces fonctions ont été écrites par des programmeurs avant vous, elles vous évitent d'avoir à réinventer la roue à chaque nouveau programme.

Lorsqu'on appelle une fonction, il y a 3 étapes :

1. L'entrée: on fait "rentrer" des informations dans la fonction (en lui donnant des informations avec lesquelles travailler), on appelle ça, ses **arguments** ou ses **paramètres**.
2. Les calculs : grâce aux informations qu'elle a reçues en entrée, la fonction travaille.
3. La sortie : une fois qu'elle a fini ses calculs, la fonction renvoie un résultat. C'est ce qu'on appelle la sortie, ou encore le **retour**.

Le prototype d'une fonction correspond simplement à son en-tête. On place donc le prototype en début de programme (Si vous avez déjà programmé en C++, vous connaissez sans doute les prototypes de fonctions, qui sont utilisés dans les déclarations des classes). Cette description permet à l'assembleur de « vérifier » la validité de la fonction à chaque fois qu'il la rencontre dans le programme, en lui indiquant :

- Le type de valeur renvoyée par la fonction.
- Le nom de la fonction.
- Les types d'arguments.

Contrairement à la **définition** de la fonction, le prototype n'est pas suivi du corps de la fonction (contenant les instructions à exécuter), et ne comprend pas le nom des paramètres (seulement leur type). Un prototype de fonction ressemble donc à ceci :
FunctionName PROTO [ParameterName]:DataType, [ParameterName]:DataType,...

En résumé, la directive **PROTO** crée un prototype de procédure. C'est une sorte de pré-déclaration qui mentionne le nom de la procédure et la liste de ses paramètres. Grâce à un prototype, vous pouvez insérer des appels à une procédure avant le corps de définition de cette procédure (Si vous avez déjà programmé en C++, vous connaissez sans doute les prototypes de fonctions, qui sont utilisés dans les déclarations de classes.). Le mot lé proto est précédé du nom de la fonction puis suivi de la liste des paramètres, séparés par des virgules. Dans l'exemple, ci-dessous, on définit **ExitProcess** comme une fonction qui prend seulement un seul paramètre de type **DWORD**.

```
ExitProcess PROTO :DWORD
```

includelib ne fonctionne pas comme include. C'est seulement une façon de dire à Masm quelles bibliothèques d'importation sont employées par vos programmes. Quand Masm voit une directive includelib, il met une commande de linker dans le fichier d'objet pour que le linker sache avec quelles bibliothèques d'importation votre programme a besoin de se lier. Vous n'êtes pas forcé d'employer includelib quoique. Vous pouvez spécifier les noms des bibliothèques d'importation dans la ligne de commande du linker, mais croyez-moi, c'est ennuyeux et la ligne de commande ne peut contenir que 128 caractères.

Le programme que nous avons entre les mains fait appel à deux apis MessageBox et ExitProcess fessons partie respectivement aux librairies user32.lib et kernel32.lib, donc nous avons besoin d'inclure le prototype de fonction de user32.dll et kernel32.dll. Ces fichiers sont user32.inc et kernel32.inc. Si vous l'ouvrez avec un éditeur de texte, vous verrez que c'est plein de prototypes de fonction. Si vous n'incluez pas kernel32.inc par exemple, vous pouvez toujours appeler ExitProcess, mais seulement avec la syntaxe d'appel simple (Push 0 puis Call ExitProcess). Vous ne serez pas capables d'invoker la fonction (ExitProcess).

Humm, invoke ! Qu'est-ce qui se cache derrière cet obscur jargon ?

Avec l'utilisation de la directive Masm **invoke** (High Level Syntax), les APIs Windows peuvent être utilisés d'une manière très similaire aux langages "C" ou « Pascal »

```
invoke NomDeFonction, par1, par2, par3, ...
```

Ceci est la forme moins lisible de la même fonction :

```
...
push par3
push par2
push par1
call NomDeFonction
```

Permet non seulement de diminuer le nombre de lignes lors des appels de proc (push, push...), mais aussi de vérifier automatiquement la syntaxe de l'appel. Le nombre de paramètres passés à la fonction doit correspondre au nombre d'arguments attendus par la fonction. La taille des paramètres (dword, byte...) doit correspondre à la taille des arguments attendus par le proc. Si une des deux conditions n'est pas remplie, il se produit une erreur à l'assemblage.

Aucune de ces erreurs ne se produit lorsqu'on utilise les push suivis d'un call. La sanction de la non utilisation de invoke peut fort bien être des heures et des heures de débogage passés à chercher les arguments manquant ou mal dimensionnés (bonjour la galère s'il existe plus d'une centaine d'appels à contrôler, ce qui n'a rien d'énorme).

Par exemple, si vous faites :

```
call ExitProcess
```

Sans pousser un dword sur la pile (Push 0, avant d'appeler la fonction ExitProcess), l'assembler ne sera pas capable de comprendre cette erreur à votre place. Vous vous en apercevrez plus tard quand votre programme plantera. Mais si vous employez :

```
invoke ExitProcess
```

Le linker vous informera que vous avez oublié de pousser un **DWORD** sur la pile évitant ainsi l'erreur. Je vous recommande donc d'employer invoke au lieu d'un simple 'Call'.

```
18  .code
19  start:
20
21  push MB_OK + MB_ICONASTERISK
22  push offset MsgBoxCaption
23  push offset MsgBoxText
24  push NULL
25  call MessageBox
26
27  push NULL
28  call ExitProcess
29
30  end start
```

Une fois compilé, le programme aura la même taille que si on avait utilisé la syntaxe de haut niveau de Masm.

Pour des problèmes plus complexes, nous obtenons ainsi de longues listes d'instructions, peu structurées et par conséquent peu compréhensibles. En plus, il faut souvent répéter les mêmes suites de commandes dans le code source, ce qui entraîne un gaspillage de mémoire interne et externe. Vous pouvez ainsi créer des prototypes de fonction pour vos **propres** fonctions. Nous verrons ça dans les chapitres suivants.

Si vous êtes intéressé par la programmation orientée objet, pensez à toutes les fonctions d'une même classe. C'est à peu près l'équivalent d'une collection de procédures et de données définis dans le même module de code source assembleur. Rappelons que le langage assembleur à été crée bien avant la programmation orientée objet. Il ne peut donc pas proposer une structure formelle du niveau du celle du C++ ou de Java. C'est à vous d'imposer un peu de rigueur dans la structure de rédaction de vos programmes.

Chapitre 6 : Structure de données

Les variables et les constantes

Pour fonctionner, un programme à généralement besoin de pouvoir utiliser des données. Pour cela, il doit pouvoir stocker les valeurs, et pouvoir y accéder par la suite. Le programme pourra ensuite les contrôler, et pourquoi pas, les modifier.

Pour cela, il existe un moyen simple qui est l'utilisation des **variables**.

Il y a toutes sortes de choses que l'on peut stocker au sein de variables : les informations concernant un employé, la longueur d'un morceau de musique, le nombre de bicyclettes commandées aujourd'hui, etc.

La variable est un concept important en programmation informatique. Intuitivement, il s'agit d'un nom qui se réfère à une chose. Informatiquement, c'est une référence à une **adresse** mémoire. Il s'agit donc d'une entité créée dans l'ordinateur à laquelle on donne un nom et qu'on utilise en évoquant ce nom dans le code source du programme, capable de contenir des informations dont la valeur peut varier au cours du temps selon les manipulations que le programme lui fait subir. Les déclarations introduisent les variables qui sont utilisées, fixent leur type et parfois aussi leur valeur de départ. Les opérateurs contrôlent les actions que subissent les valeurs des données. Pour produire de nouvelles valeurs, les variables et les constantes

peuvent être combinées à l'aide des opérateurs dans des expressions. Les variables peuvent être très utiles si vous n'avez pas suffisamment de registres.

Par exemple, le nom **NombreDeChansons** vous permet de savoir tout de suite que cette variable représente le nombre de chansons à écouter dans le juke-box. En revanche, **C3PO** n'est pas un nom génial pour une variable. Ce peut être un numéro de série, un nom de robot ou qui sait quoi.

Masm définit plusieurs types de données intrinsèques. Chaque type permet d'exploiter un ensemble de valeurs et de les affecter à des variables et à des expressions du même type. Par exemple, une variable de type intrinsèque **DWORD** permet de stocker n'importe quelle valeur entière sur 32 bits. Il existe des types plus contraignants, et notamment REAL4, qui ne peuvent accepter qu'une constante numérique réelle. L'image si dessous présente tous les types de données pour les valeurs entières, excepté les trois dernières. Pour celle-ci, la notation **IEEE** correspond au standard des formats de nombres réels (à virgule flottante) tels que publiées par l'association scientifique **IEEE** Computer society.

```
.data
; Entièr sur 8 bits (1 octet).
MonByte     BYTE    12h

; Entièr sur 16 bits (2 octets).
MonWORD     WORD    567Bh

; Entièr sur 32 bits (4 octets).
MonDWORD    DWORD   1ABF0FFFh

; Entièr sur 48 bits (6 octets).
MonFWORD    FWORD   1FFFFF0F01Ah

; Entièr sur 64 bits (8 octets).
MonQWORD    QWORD   0FFFFFAFFFFF00120h

; Entièr sur 80 bits (10 octets).
MonTBYTE    TBYTE   0FFAFFAFF0701FFF00120h
```

Les directives **BYTE, WORD, DWORD, FWORD, QWORD, TBYTE** servent à définir des emplacements de données, vous pouvez encore utiliser les directives **DB** (**Define BYTE**), **DW** (**Define WORD**), **DD** (**Define DWORD**), **DF** (**Define FWORD**), **DQ** (**Define QWORD**), et **DT** (**Define TBYTE**) pour déclarer vos variables.

Pour laisser une variable sans valeur initiale (non nationalisé), il suffit d'indiquer pour l'initialiser le point d'interrogation. Il en résultera que la variable recevra une valeur lors de l'exécution.

```
.data?
Score       BYTE    ?
```

En informatique, les rationnels sont souvent appelés des « flottants ». Ce terme vient de « en virgule flottante » et trouve sa racine dans la notation traditionnelle des rationnels. La directive **REAL4** permet de réserver un espace pour une variable réelle sur 4 octets à simple précision. La directive **REAL8** définit un réel à double précision sur 8 octets, et la directive **REAL10** définie un réel à précision étendue sur 10 octets. Chaque directive suppose la mention d'un initialiseur constant réel dont la largeur correspondant :

```
.data
rVal1        REAL4    -2.1
rVal2        REAL8    3.2E-260
rVal3        REAL10   4.6E+4096
```

Pour déclarer une donnée de type chaîne (un nom programmatiquement correct pour désigner du texte), vous devez délimiter les caractères qui la constituent au moyen d'apostrophes ou des guillemets. Les chaînes les plus répandues en programmation se terminent par un octet dont la valeur est 0. Ces chaînes sont appelées chaînes à zéro terminal, ou chaînes AZT. Elles sont très utilisées dans les langages C, C++ et Java, et par les fonctions de Windows.

A chaque caractère correspond un octet de stockage. Les chaînes constituent une exception à la règle qui veut que les valeurs de type octet soient séparées les unes des autres par des virgules. Si cette exception n'existait pas, la variable dans le code source aurait dû être définie comme ceci :

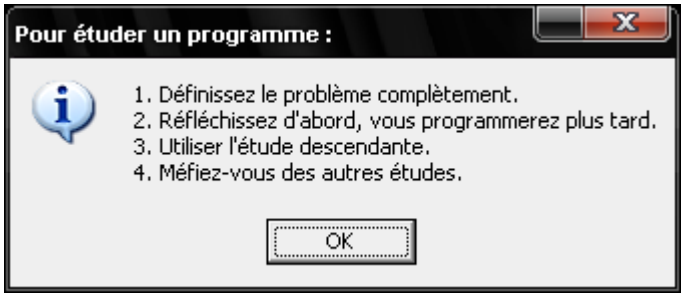
```
MsgBoxCaption BYTE 'H','e','l','l','o',' ','W','o','r','l','d','!', 0
```

Ce qui serait vite fatiguant.

Vous pouvez distribuer une chaine de caractères en plusieurs lignes sans être forcé de répéter le label au début de chaque ligne, comme le montre cet exemple :

```
.data
MsgBoxCaption    BYTE    "Pour étuder un programme : ", 0
MsgBoxText       BYTE    "1. Définissez le problème complètement. ", 0Dh, 0Ah
                 BYTE    "2. Réfléchissez d'abord, vous programmerez plus tard.",0Dh, 0Ah
                 BYTE    "3. Utiliser l'étude descendante.", 0Dh, 0Ah
                 BYTE    "4. Méfiez-vous des autres études.", 0
```

Hourra !



Rappelons que les octets possédant les valeurs décimales **0Dh (13d)** et **0Ah (10d)** correspondent au couple Retour chariot/Saut de ligne (CR/LF) qui permettent de marquer la fin d'une ligne de texte. Lorsque ces deux caractères sont envoyé pour affichage, il provoque le renvoie du curseur dans la première colonne de la ligne suivante.

Le caractère d'extension de la ligne Masm est la barre oblique inverse « \ ». Il permet de tronquer visuellement une ligne qui reste au niveau logique. Le symbole \ ne peut donc être placé qu'à la fin d'une ligne. Au niveau du programme assembleur, les deux instructions suivantes sont absolument identiques :


```
Bienvenue    BYTE     "Vive l'assembleur ;)", 0

; et

Bienvenue \
BYTE "Vive l'assembleur ;)", 0
```

Portée des variables

La portée d'une variable désigne la zone du code dans laquelle la variable sera connue et donc utilisable. On distingue deux portées différentes :

- Une portée locale à une fonction.
- Une portée globale à un programme.

Si vous déclarez une variable ou une constante en dehors d'une définition de fonction (dans la section .data, data?, ou .const), il s'agit d'une variable **globales statiques**. Le terme « statique » signifie que la durée de vie de la variable est égale à celle de l'exécution du programme. Le terme « global » signifie que la valeur de la variable est accessible et modifiable dans tout le programme. En revanche, si vous déclarez une variable au sein d'une définition de fonction, il s'agit d'une variable **locale**. Elle est créée et détruite chaque fois que la fonction est exécutée ; il est impossible d'y accéder en dehors de la fonction. Sa durée de vie est donc bien inférieure à la durée d'exécution du programme. Attention : les variables locales ne peuvent être déclarées qu'en tête de la fonction. Vous connaissez sans doute déjà les variables locales si vous avez programmé avec un langage de haut niveau.

Voyons comment exploiter les variables locales en assembleur. Rappelons d'abord les avantages du « local » par rapport au « global » :

- L'accès limité à une variable local simplifie le débogage, car les origines éventuelles d'un disfonctionnement sont moins nombreuses.
- Une variable local utilise la mémoire de manière efficace, en l'occurrence uniquement pendant l'exécution de la routine dans laquelle elle est déclarée. Cela permet de réutiliser l'espace pour de nouvelles variables locales.
- Une variable locale peut porter le même nom qu'une variable globale, mais elle n'en reste pas moins distincte. En conséquence, si vous modifiez la valeur d'une variable, cela n'a aucun effet sur l'autre. Au sein de la fonction dans laquelle la variable locale est déclarée, seule la version locale est significative. Ce concept porte le nom de **visibilité**.

Les variables locales sont crées sur la pile d'exécution. Vous ne pouvez pas leur donner de valeur initiale lors de l'assemblage, mais vous êtes libres d'écrire une valeur initiale en début d'exécution.

La directive LOCAL

La directive **LOCAL** déclare une ou plusieurs variables locales dans une procédure. Elle doit être mentionnée sur la ligne qui suit immédiatement celle qui constitue l'ouverture du bloc de définition d'une procédure. Elle doit être introduite par la directive **PROC**. Voici sa syntaxe formelle :

LOCAL ListeVar

ListeVar correspond à une liste de noms de variables séparés par des virgules et pouvant s'étendre sur plusieurs lignes. Chaque définition de variable respecte la syntaxe suivante :

Label : type

Label doit être un identificateur autorisé et type doit être un des types standard **WORD**, **DWORD**, etc. ou un type défini par le programmeur. (Nous aborderons les structures de données et les types définis par le programmeur directement après ce paragraphe).

La directive LOCAL alloue de la mémoire sur la pile pour des variables locales employées par la fonction. La totalité des directives LOCALES doivent être placées juste au-dessous de la directive PROC. Les variables locales ne peuvent pas être employées à l'extérieur de la fonction, elles sont créés et seront automatiquement détruites quand la fonction retourne à l'appel. Un autre inconvénient est que vous ne pouvez pas initialiser des variables locales automatiquement parce qu'elles sont uniquement allouées dans la mémoire de la pile au moment seulement où la fonction est entrée.

Exemple :

La procédure **TriABulle** déclare la variable local **temp** de taille double-mot (dword) et la variable **SwapFlag** de taille **BYTE** :

```
TriABulle proc
    LOCAL temp:DWORD, SwapFlag:BYTE
```

On a appris que pour déclarer un prototype de procédure on utilise la directive **PROTO**, maintenant on va voir comment déclarer la procédure elle-même en utilisant la directive **PROC.** Au niveau élémentaire, une procédure est un bloc contigu d'instructions portant un nom et se terminant par une instruction de retour d'exécution. Vous déclarer une procédure avec le couple de directives **PROC** et **ENDP**. La directive **PROC** est essentielle : Elle peut comporter une liste de paramètres, comme le montre sa syntaxe formelle simplifiée :

```
label proc,
    paramètre_1,
    paramètre_2,
    .
    .
    paramètre_n,
    ret

label endp
```

Vous devez indiquer comme dernière instruction celle de retour **RET**. Elle oblige le processeur à poursuivre l'exécution à l'instruction qui suit celle qui a constitué l'appel au sous-programme. Les paramètres peuvent être placés sur une seule ligne :

```
label proc, paramètre_1, paramètre_2, ..., paramètre_n,
```

Chaque paramètre doit obéir à la syntaxe suivante :

Nomparam: type

Nomparam est un nom choisi par le programmeur. Le paramètre est accessible uniquement dans la procédure courante (portée locale). Vous pouvez donc utiliser le même nom de paramètre dans plusieurs procédures, à condition que ça soit ni variable globale, ni un label

de code. Pour le type, choisissez l'un des mots réservés suivants : **BYTE, SBYTE, WORD, SWORD, DWORD, SDWORD, FWORD, QWORD** ou **TBYTE**, ou indiquez un type qualifié, qui peut être un pointeur sur l'un des types existants. Voici quelques exemples de types qualifiés PTR BYTE, PTR WORD, ...

Voyons quelques définitions de procédure avec différents types de paramètres. Ne vous souciez pas du corps de la fonction pour l'instant.

Exemple 1

La procédure suivante attend en entrée deux valeurs de taille double-mot :

```
AjouterDeux proc
    val1:DWORD,
    val2:DWORD
    ...

ret
AjouterDeux endp
```

Exemple 2

La procédure suivante attend un pointeur sur un octet :

```
RemplirTableau proc,
    pTableau:ptr DWORD
    ...

    ret
RemplirTableau endp
```

Exemple 3

La procédure suivante attend deux pointeurs sur des doubles-mots :

```
Permuter proc,
pValX:ptr DWORD,
pValY:ptr DWORD
    ...

    ret
Permuter endp
```

Exemple 4

La procédure suivante attend le pointeur sur un octet **pTampon**. Elle déclare localement la variable **fileHandle** :

```
LireFichier proc,
    pTampon: ptr BYTE

    LOCAL fileHandle:DWORD
    ...

    ret
LireFichier endp
```

La directive ALIGN

La directive ALIGN permet d'aligner une variable sur une frontière d'octet, de mot, de double-mot ou de paragraphe. Voici sa syntaxe :

ALIGN frontiere

frontiere peut être égale à 1, 2 ou 4. Si elle vaut 1, la prochaine variable est alignée sur une frontière d'octet (valeur par défaut). Si la valeur est 2, la variable est alignée sur la prochaine adresse paire. Si la frontière vaut 4, la prochaine variable est stockée à la prochaine adresse multiple de 4 en partant du début du segment. Pour assurer ce recadrage, l'assembleur insère des octets vides avant la variable. Vous pouvez vous demander à quoi sert cet alignement. C'est parce que le processeur peut lire et écrire plus rapidement des objets stockés à des adresses paires.

Prenons l'exemple suivant. Nous savons que **bVal** est automatiquement stocké à une adresse offset paire. Si nous ajoutons **ALIGN 2** avant la directive **wVa**l, cette variable sera placée à un décalage pair elle aussi. Nous allons par exemple choisir d'implanter la première variable à l'adresse **00404000** :

```
bVal    BYTE    ?   ; 00404000
ALIGN 2
wVal    WORD    ?   ; 00404002
bVal2   BYTE    ?   ; 00404004
ALIGN 4
dVal    DWORD   ?   ; 00404008
dVal2   DWORD   ?   ; 0040400C
```

Si **dVal** avait été placée à l'offset **00404005**, la directive **ALIGN 4** l'aurait repoussée jusqu'à l'offset **00404008**.

Directive d'égalité (=)

La directive symbolisé par le signe = permet d'associer un nom de symbole à une expression numérique entière. Voici sa syntaxe générique :

nom = expression

Normalement, **expression** est une valeur entière sur 32 bits. Lors de l'assemblage du programme, toutes les occurrences de **nom** sont remplacées par expression dans une première étape de prétraitement. Par exemple, lorsque l'assembleur rencontre les deux lignes suivantes :

```
COMPTEUR = 100
mov al, COMPTEUR
```

Il génère une seule instruction machine :

```
mov al, 100
```

Dans l'exemple précédant, nous aurions pu éviter de définir le symbole **COMPTEUR**, et ajouter directement la valeur littérale 100 dans l'instruction **MOV**. L'expérience montre que l'utilisation de symboles rend les programmes plus simples à lire et à maintenir. Imaginez que **COMPTEUR** soit utilisé dix fois dans le même programme. Si vous avez besoin plus tard de modifier la valeur de **COMPTEUR** en 200, vous n'avez qu'une seule ligne à modifier :

```
COMPTEUR = 200
```

Lorsque vous réassemblez le programme, tous les occurrences du symbole **COMPTEUR** sont automatiquement remplacées par la valeur 200. Si ce symbole n'avait pas été défini, il aurait fallu modifier dix fois la valeur, au risque d'en oublier une. Cela provoquerait certainement un bogue.

L'opérateur PTR

L'opérateur **PTR** permet de passer outre la taille déclarée à l'origine pour un opérande. Vous n'en aurez besoin que lorsque vous devrez accéder à une variable avec un attribut de taille différent de celui utilisé lors de la déclaration de cette variable.

Par exemple, supposons que nous ayons besoin de copier les 16 bits de poids faible d'une variable de taille double-mot appelée **monDouble** dans le registre **AX**. Le programme assembleur refuse la copie suivante, car les tailles des deux opérandes ne correspondent pas :

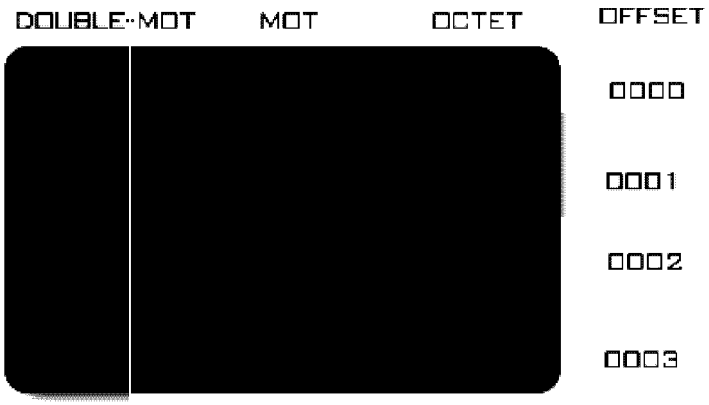
```
.data
monDouble   DWORD   12345678h

.code
mov ax, monDouble   ; Erreur
```

Si nous insérons la mention **WORD PTR**, nous pouvons copier le mot de poids faible **(5678*h*)** dans **AX** :

```
mov ax, WORD PTR monDouble  ; Correct
```

Pourquoi n'est ce pas 1245h qui a été copié dans AX ? La cause en est l'ordre de stockage *little endian* dont nous avons parlé précédemment. Dans la figure qui suit, la structure du stockage mémoire de monDouble est présentée de trois manières : tout d'abord en tant que double-mot, puis sous la forme de deux mots distincts (1234h), et enfin sous la forme de deux mots distincts (1234h), et enfin sous la forme de quatre octets distincts (78h, 56h, 34h, 12h) ;



Le processeur peut accéder à la mémoire selon n'importe laquelle des trois approches, quelle que soit la manière dont la variable a été déclarée à l'origine. Par exemple, si monDouble est situé à l'offset 0000, la valeur sur16 bits stockée à cette adresse est 5678h. Pour lire le mot suivant, 1234h, il faut utiliser la mention monDouble+2, comme ceci :

```
mov ax, WORD PTR [monDouble+2]   ; 1234h
```

Nous pouvons utiliser l'opérateur **BYTE PTR** pour récupérer un seul octet de monDouble vers BL, par exemple :

```
mov bl, BYTE PTR monDouble   ; 78h
```

Vous avez remarqué que **PTR** doit toujours être combiné avec l'un des noms de types de données standard : **BYTE, SBYTE, WORD, SWORD, DOWRD, SDWORD, FWORD, QWORD** ou **TBYTE**.

Copie d'une petite valeur dans une grande

Supposons que nous ayons besoin de copier deux valeurs de petite largeur vers un opérande de grande taille. Dans notre exemple, le premier mot est copié dans la partie basse d'**EAX,** et le second dans la partie haute. Cela n'est pas possible que grâce à l'opérateur **DWORD PTR** :

```
.data
Bloc2Mots    WORD     5678h, 1324h

.code
mov eax, DWORD PTR Bloc2Mots     ; EAX = 12346578h
```

L'opérateur TYPE

L'opérateur **TYPE** permet de connaître la taille, en octets, d'un élément isolé d'une variable complexe ou d'une variable simple. Par exemple, **TYPE** appliqué à un octet renvoie 1 ; appliqué à un mot, il renvoie 2 ; appliqué à un double-mot, il renvoie 4 ; et appliqué à un quadruple-mot, il renvoie 8. Voici des exemples :

```
.data?
var1    BYTE    ?   ; "TYPE var1" renvoie 1
var2    WORD    ?   ; "TYPE var2" renvoie 2
var3    DWORD   ?   ; "TYPE var3" renvoie 4
var4    QWORD   ?   ; "TYPE var4" renvoie 8
```

Les tableaux

Les tableaux sont certainement les variables structurées les plus populaires. Ils sont disponibles dans tous les langages de programmation et servent à résoudre une multitude de problèmes. Imaginons que dans un programme, nous ayons besoin simultanément de 12 valeurs (par exemple, des notes pour calculer une moyenne). Evidemment, la seule solution dont nous disposons à l'heure actuelle consiste à déclarer douze variables, appelées par exemple Notea, Noteb, Notec, etc. Bien sûr, on peut opter pour une notation un peu simplifiée, par exemple N1, N2, N3, etc. Mais cela ne change pas fondamentalement notre problème, car arrivé au calcul, et après une succession de douze instructions « Lire » en algorithme distinctes. C'est tout de même bigrement laborieux. Et pour un peu que nous soyons dans un programme de gestion avec quelques centaines ou quelques milliers de valeurs à traiter, alors là c'est le suicide direct. C'est pourquoi la programmation nous permet de rassembler toutes ces variables en une seule, au sein de laquelle chaque valeur sera désignée par un numéro. En bon français, cela donnerait donc quelque chose du genre « la note numéro 1 », « la note numéro 2 », « la note numéro 8 ». C'est largement plus pratique, vous vous en doutez.

Un ensemble de valeurs portant le même nom de variable et repérées par un nombre, s'appelle un **tableau**, ou encore une variable indicée. Le nombre qui, au sein d'un tableau, sert à repérer chaque valeur s'appelle – ô surprise – **l'indice**. Chaque fois que l'on doit désigner un élément du tableau, on fait figurer le nom du tableau, suivi de l'indice de l'élément, entre parenthèses. Comme dit ci-dessus, un tableau est une collection séquentielle de variables, toutes de la même taille et de même type, appelées des éléments. Vous pouvez avoir accès à tous les éléments dans un tableau autant qu'au premier élément. Tandis qu'un tableau est identifié d'un nom, chaque élément dans un tableau en fait référence avec un nombre d'indice, commençant par le zéro. L'indice de tableau apparaît entre parenthèses carré après le nom. Par exemple, définissant un tableau d'octets a appelé "Tab":

```
.data
Tab         WORD    1, 3, 5, 7, 11, 13, 17
```

Donne une valeur de 1 à **Tab [0],** une valeur de 3 à **Tab [1],** et ainsi de suite. Cependant, dans des tableaux avec des éléments plus grands que 1 octet, les nombres d'indice (sauf le zéro) ne correspondent pas à la position d'un élément. Vous devez multiplier la position d'un élément de sa taille pour déterminer l'indice de l'élément. Ainsi, pour le tableau

```
wTab        WORD    1, 3, 5, 7, 11, 13, 17
```

Tab [4] représente le troisième élément (5), qui est de 4 octets depuis le début du tableau. De même l'expression Tab [6] représente le quatrième élément (7) et Tab [10] représente le sixième élément (13).

Si vous devez déclarer un grand tableau, vous pouvez utiliser l'opérateur **DUP**, par exemple :

```
Tab     BYTE    5   DUP(9)
```

Qui est équivalent à écrire :

```
Tab     BYTE        9, 9, 9, 9, 9
```

Plus encore :

```
Tab2    BYTE    5   DUP(1, 2)
```

Qui est équivalent à écrire :

```
Tab2    BYTE    1, 2, 1, 2, 1, 2, 1, 2, 1, 2
```

A noter qu'ne chaine de caractère est aussi un tableau de caractères qui prend 1 octet de mémoire pour chaque caractère.

L'opérateur LENGTHOF

L'opérateur **LENGTHOF** permet de compter le nombre d'éléments d'un tableau, à partir des valeurs indiqués sur la même ligne que le label. Comme exemple, nous allons définir les données suivantes :

```
.data
octet1  BYTE    10, 20, 30
tab1    WORD    30 DUP (?),0 ,0
tab2    WORD    5  DUP (3 DUP (?))
tab3    DWORD   1, 2, 3, 4
digitst BYTE    "13245678", 0
```

Le tableau suivant présente les valeurs qui sont renvoyées par chacune des expressions LENGHTOF.

Expression	Valeur
LENGTHOF octet1	**3**
LENGTHOF tab1	**30 + 2**
LENGTHOF tab2	**5 * 3**
LENGTHOF tab3	**4**
LENGTHOF digitSt	**9**

Remarquez que lorsque nous utilisons des opérateurs de duplication **DUP** imbriqués dans une définition de tableau, **LENGTHF** envoie le produit des deux compteurs d'éléments.

LENGHTOF est "myope" : lorsque vous déclarez un tableau sur plusieurs lignes sources, **LENGHTOF** ne voit que les données de la première ligne. Dans l'exemple suivant, la mention **LENGHTOF monTablo** renvoie la valeur 5 :

```
monTablo    BYTE    10, 20, 30, 40, 50
            BYTE    60, 70, 80, 90, 100
```

Pour pallier à cette myopie, vous pouvez terminer la première ligne par une virgule et poursuivre la liste des valeurs sur la ligne suivante. Dans l'exemple suivant, **LENTGHOF monTablo** renvoie la valeur 10 :

```
monTablo    BYTE    10, 20, 30, 40, 50,
                    60, 70, 80, 90, 100
```

Les structures

Jusqu'à présent, vous avez découvert les types de données simples et vous avez vu que l'on peut stocker une information dans une variable. Mais supposons que vous désirez stocker quelque chose de plus complexe qu'un simple type de donnée : par exemple, le nom de l'utilisateur, son adresse et son numéro de téléphone.

Vous pouvez certes le faire en stockant ces informations dans trois variables séparées : une pour le nom, une autre pour l'adresse, et une autre pour le numéro de téléphone. Néanmoins, il y a plus pratique : dans la vie, il est naturel de regrouper les choses. Par exemple, vous désirez lire ou imprimer l'enregistrement concernant un employé. Bien que cet enregistrement comporte de nombreux éléments (nom, adresse, numéro de téléphone, salaire, etc.), vous le pensez probablement comme une entité complète. Il est d'ailleurs bien plus facile de dire : « *imprimer l'enregistrement de l'employé* » que de dire « *imprimer le nom, l'adresse, le numéro de téléphone, le salaire* », etc.

Le groupement d'un ensemble de variables apparentées en une seule entité est appelé **structure**. Les structures sont une caractéristique très puissante de Masm32, et facilité l'organisation et le traitement des informations.

Pour déclarer une structure, utilisez le couple de directives **STRUCT** précédé par le nom de la structure et **ENDS**. Par exemple :

```
Date struct
Jour    BYTE    ?
Mois    BYTE    ?
Annee   WORD    ?
Date ends
```

Définissez ensuite la variable (un cas de la structure de **Date** appelée "Lundi") l'un ou l'autre avec des données non initialisées « .data?» :

```
.data?
Lundi Date <>
```

Ou avec des données initialisées :

```
.data
Dimanche Date <19, 08, 1990>
```

Ou encore :

```
.data
Dimanche Date  (19, 08, 1990)
```

Finalement vous pouvez avoir accès à la structure et ses **champs** directement du nom comme cela :

```
mov al, Dimanche.mois
```

Ou indirectement via un indicateur comme cela :

```
mov ebx, offset Dimanche     ; Adresse de chargement de la
                             ; variable dimanche dans ebx.
mov al, [ebx].Date.mois      ; Copie du champ mois d'ebx dans Al
```

Les Unions

Dans une structure, les données membres sont placées de manière contigüe en mémoire. Supposons que dans notre ordinateur, un **BYTE** prenne un octet, un **WORD** en prenne 2 et un **DWORD** en prenne 4. Définissons alors notre struct comme ceci :

```
struct RepereCartesien
x    BYTE    ?
y    WORD    ?
z    DWORD   ?
RepereCartesien ends
```

Voici alors le schéma de sa répartition en mémoire :

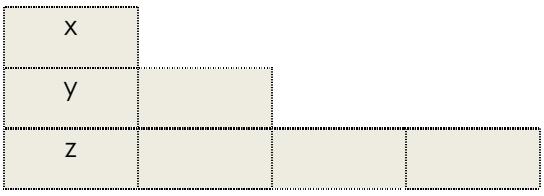
Elle occupe donc en mémoire la somme des tailles de ses membres. D'ailleurs, vous pouvez avoir la taille de votre struct avec l'opérateur **sizeof**.

Il existe un autre type de **structure de données**, appelé **union**. Une union se comporte extérieurement de manière très similaire à la struct, mis à part qu'elle ne peut être passée comme argument à une fonction. En matière de syntaxe et d'accès aux données membres, rien ne change. Ce qui change, c'est la répartition en mémoire et les conséquences qui en découlent quant à l'utilisation des membres.

Voici notre premier exemple d'union :

```
union RepereCartesien
x    BYTE    ?
y    WORD    ?
z    DWORD   ?
RepereCartesien ends
```

Vous constatez qu'il n'y a pas grand chose qui change. Par contre, voici sa répartition en mémoire :



Les trois membres commencent tous à la même case! Donc la taille de l'union est égale à la taille du membre le plus grand. Le rôle principal de l'union est d'économiser de l'espace en mémoire. En effet, au lieu de mettre les membres bout à bout en mémoire, vous les superposez, si bien que vous pourriez utiliser 50 DWORD dans une fonction sans prendre plus de 4 octets en mémoire. Cependant, si c'était aussi simple que ça, on ne travaillerait qu'avec des unions. En fait, il y a une contrainte : on ne peut travailler qu'avec un seul membre à la fois. Je m'explique.

Du fait de cette superposition, vous ne pouvez charger qu'un seul type à la fois. Si vous effectuez une affectation à l'un, les valeurs des autres membres ne sont plus retenues... ce n'est ni facile à expliquer, ni facile à comprendre. Un bon moyen de retenir ce qui se passe, c'est de se dire qu'une union est comme un ensemble de variables que vous ne pouvez jamais utiliser en même temps : l'utilisation de l'une de ces variables provoque la "disparition" de toutes les autres, jusqu'à ce que vous ré-effectuiez une affectation avec une autre.

Je vous ai présenté ici les unions car elles existent. Cependant, je crois bien que je n'en ai utilisé qu'un nombre limité de fois dans ma vie, et tenter de les utiliser en pensant économiser de la mémoire, c'est se compliquer la vie pour rien. Tout ce qu'il vous faut savoir, c'est qu'elles existent, histoire que vous ne soyez pas complètement déboussolés si vous en croisez une dans un programme.

Les pointeurs

Tout le monde sait « en gros » ce qu'est un pointeur, mais parfois certaines ambiguïtés persistent et je préfère rendre les choses claires dès le début. Avant de voir la notion de pointeur sur une variable, peut-être serait-il utile de bien se souvenir de ce qu'est une variable, justement. Alors une variable, c'est un emplacement quelque part en mémoire qui sert à stocker des valeurs ou des objets (ou des pommes, des poires). Il faut éviter de confondre le contenant, ou conteneur (la variable) du contenu (la pomme). Si je dis ça, c'est parce que souvent la syntaxe même du langage les confond.

En algorithme
Var : c, d en entier **c <- 4** **d <- c**

La première affectation place 4 (la pomme) dans c (le contenant). La seconde affectation copie le contenu de c dans le contenant de d. Ca n'est pas du tout la même chose ! Après la deuxième affectation, d contient 4, pas c. Et quand je dis « pas c », ça n'est ni « c la valeur », ni « c la variable » : si je modifie c après-coup, d contient toujours 4. Je ne vous prends pas pour un idiot en vous expliquant tout ça : ici c'est totalement évident, mais vous verrez qu'un peu plus tard ça va le devenir soudainement beaucoup moins, alors autant être clair dès le début.

Donc, à présent que nous faisons bien la distinction entre le contenant et le contenu, et que nous savons qu'une affectation depuis une autre variable ne fait que copier le contenu d'une variable vers l'autre variable, intéressons-nous aux pointeurs proprement dits.

On se rappelle que la mémoire, c'est une suite d'octets, et que chaque octet peut être numéroté par un nombre allant de 0 à beaucoup. Du coup, une variable (le conteneur) peut être désignée par l'indice du premier octet où se trouve ce conteneur, si l'on suppose que le conteneur se trouve placé d'une manière « continue » en mémoire, ce qui sera toujours le cas. C'est ce qu'on appelle l'adresse de la variable. Une adresse désigne donc le conteneur qu'est une variable.

Un pointeur, c'est une variable dont le contenu est une adresse. Donc, c'est une variable dont le contenu désigne une autre variable. J'insiste sur le fait que c'est le contenu du pointeur qui désigne cette seconde variable, et pas le pointeur lui-même.

Les opérateurs ADDR & OFFSET

L'**OFFSET** renvoie à l'adresse d'une variable. Il est utilisé pour spécifier la localisation plutôt que le contenu de la variable:

```
.data
Var     db      23h

.code
mov eax, Var
mov ebx, offset Var
```

L'**OFFSET** peut aussi passer l'adresse d'une variable à une fonction en invoquant une instruction. Cependant, ça fonctionnera seulement pour des variables globales déclarées. Cela échouera avec des variables. Ceux-ci n'ont aucune compensation comme ils sont créés sur la pile au moment de l'exécution. .

L'opérateur **ADDR** résout ce problème. Il est utilisé exclusivement avec invoke pour passer l'adresse d'une variable à une fonction. Pour des variables globales il résume à une simple instruction **PUSH**, pareil que si l'**OFFSET** avait été utilisée :

L'opérateur **ADDR** transmet un pointeur lors de l'appel à une procédure avec **INVOKE**. La valeur de ce pointeur est une adresse qui désigne un emplacement mémoire, ce qui correspond au passage de paramètre par référence et non par valeur. C'est valable seulement dans le contexte de la directive **INVOKE**. Vous ne pouvez pas l'employer pour assigner l'adresse d'une donnée à un registre ou une variable, par exemple. Vous devez employer **OFFSET** au lieu d'ADDR dans ce cas. Cependant, il y a quelques différences entre les deux :

- **ADDR** ne peut pas manipuler à l'avance la référence tandis qu'offset le peut. Par exemple, si la ligne où est rendu le programme, invoke un label qui est défini quelque part plus loin dans le code source, **ADDR** ne marchera pas.

```
MsgBoxCaption        BYTE    "Hello World !", 0
MsgBoxText           BYTE    "Vive l'assembleur ;)", 0

....

invoke MessageBox,NULL, offset MsgBoxText, offset MsgBoxCaption, MB_OK
```

MASM indiquera une erreur. Si vous utilisez **OFFSET** au lieu d'ADDR dans ce petit bout de code, alors MASM l'assemblera avec succès.

- **ADDR** peut manipuler des variables locales tandis qu'**OFFSET** ne le peut pas. Une variable locale est seulement un petit espace réservé sur la pile. Vous ne connaîtrez seulement son adresse que le temps de l'exécution. **OFFSET** est interprété pendant le déroulement du programme par l'assembleur. Donc il est naturel qu'**OFFSET** ne travaille pas avec des variables locales. **ADDR** est capable de manipuler des variables locales grâce au fait que l'assembleur vérifie d'abord si la variable mentionnée par **ADDR** est globale ou locale. Si c'est une variable globale, il met l'adresse de cette variable dans le fichier d'objet. À cet égard, il travaille comme offset. Si c'est une variable locale, il produit un ordre d'instruction qu'il appelle en réalité avant la fonction :

```
lea eax, VariableLocale
push eax
```

Puisque **LEA** peut déterminer l'adresse d'un label pendant l'exécution, ça fonctionne très bien.

Les crochets

Les crochets indiquent généralement la valeur (le contenu) d'une variable par opposition à son adresse. Cependant, la syntaxe MASM diffère légèrement à celle d'autres assembleurs à cet égard. Dans MASM tous ceux-ci produisent la même instruction:

```
mov eax, 1
mov eax, [1]
mov eax, DWORD PTR 1
mov eax, DWORD PTR [1]
```

MaVariable et **[MaVariable]** les deux signifient **la valeur** de **MaVariable**.

Beaucoup de programmeurs utilisent par habitude des crochets avec des variables pour dénoter le contenu ainsi cela rend le code source un peu plus lisible et il le rend plus facile au code d'être supporté par d'autres assembleurs.

Comme il a été dit ci-dessus, l' « **offset MaVariable »** & « **addr MaVariable »** les deux signifient **l'adresse** de **MaVariable**. Quand c'est utilisé avec les registres, cependant, les crochets font vraiment la différence et ont tendance à pointer l'adresse mémoire :

```
mov ebx, eax     ; Copie la valeur de EAX dans EBX.
mov ebx, [eax]   ; Copie la valeur à l'adresse mémoire de EAX dans EBX.
mov [ebx],eax    ; Copie la valeur de EAX en mémoire à l'adresse dans EBX.
```

La fonction MessageBox

Faites sortir votre référence d'APIs (le fichier Win32.hlp), sinon, l'MSDN est mise à votre disposition. On constate alors que l'API MessageBox prend 4 paramètres comme nous le voyons.

MessageBox Function

Displays a modal dialog box that contains a system icon, a set of buttons, and a brief application-specific message, such as status or error information. The message box returns an integer value that indicates which button the user clicked.

Syntax

```
int MessageBox(
    HWND hWnd,
    LPCTSTR lpText,
    LPCTSTR lpCaption,
    UINT uType
);
```

La fonction MessageBox sert généralement à afficher un petit message à l'utilisateur comme pour l'informer du succès ou de l'échec d'une opération par exemple ou pour demander une action à effectuer lorsque le programme ne peut prendre de décision tout seul. Donc la solution la plus simple pour afficher du texte depuis un programme graphique consiste à la placer dans une boîte de message qui, une fois ouverte, attend que l'utilisateur clique un bouton.

hWnd est le « handle » ou la poignée de la fenêtre parente. Vous pouvez vous imaginer qu'un « Handle » est un numéro qui représente la fenêtre à laquelle vous faîtes référence. Sa valeur n'a pas d'importante pour vous. Vous vous rappelez seulement qu'il représente la fenêtre. Quand vous voulez faire quelque chose avec la fenêtre, vous devez vous y référer par son 'handle' ou tout simplement NULL si on ne veut lui associer aucune fenêtre.

En ce qui concerne les deux paramètres suivants de type texte, aucun problème. **lpText** pointe sur une chaîne à zéro terminal à afficher dans la boîte, **lpCaption** pointe sur une chaîne à zéro terminal qui sert de titre à la boîte. **uType,** le rôle de cet argument est double : il doit indiquer d'une part quel est le jeu de boutons souhaité, parmi celle qui sont six disponibles. Il doit également stipuler quelle illustration viendra éventuellement égayer votre boîte à messages, parmi les quatre possibles (Critical, Exclamation, Information, Question). Il est possible d'activer de surcroît quelques autres options, mais nous les laisserons de côté provisoirement. Ce qui compte, c'est de comprendre comment ça marche.

Il a y d'autres styles de boutons comme ci-dessous

IDABORT	**Abort** button was selected.
IDCANCEL	**Cancel** button was selected.
IDCONTINUE	**Continue** button was selected.
IDIGNORE	**Ignore** button was selected.
IDNO	**No** button was selected.
IDOK	**OK** button was selected.
IDRETRY	**Retry** button was selected.
IDTRYAGAIN	**Try Again** button was selected.
IDYES	**Yes** button was selected.

Ceux-ci affichent un style d'icône:

Icon	Flag values
	MB_ICONHAND, MB_ICONSTOP, or MB_ICONERROR
	MB_ICONQUESTION
	MB_ICONEXCLAMATION or MB_ICONWARNING
	MB_ICONASTERISK or MB_ICONINFORMATION

Et le bouton qui est sélectionné par défaut :

Value	If the message box has more than one button, the default button would be
MB_DEFBUTTON1	The first button
MB_DEFBUTTON2	The second button
MB_DEFBUTTON3	The third button
MB_DEFBUTTON4	The fourth button

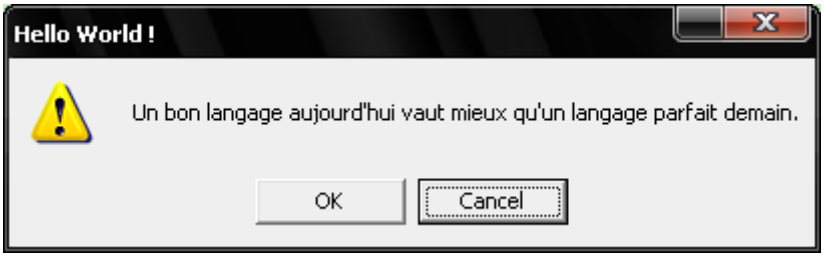
Le contrôle de ces constantes est défini dans le fichier windows.inc qui peut aussi être ouvert dans RadAsm comme indiqué ci-dessous:

```
3385  MB_OK                   equ 0h
3386  MB_OKCANCEL             equ 1h
3387  MB_ABORTRETRYIGNORE     equ 2h
3388  MB_YESNOCANCEL          equ 3h
3389  MB_YESNO                equ 4h
3390  MB_RETRYCANCEL          equ 5h
3391  MB_ICONHAND             equ 10h
3392  MB_ICONQUESTION         equ 20h
3393  MB_ICONEXCLAMATION      equ 30h
3394  MB_ICONASTERISK         equ 40h
3395  MB_USERICON             equ 80h
3396  MB_ICONERROR            equ MB_ICONHAND
3397  MB_ICONINFORMATION      equ MB_ICONASTERISK
3398  MB_ICONSTOP             equ MB_ICONHAND
3399  MB_ICONWARNING          equ MB_ICONEXCLAMATION
3400  MB_DEFBUTTON1           equ 0h
3401  MB_DEFBUTTON2           equ 100h
3402  MB_DEFBUTTON3           equ 200h
3403  MB_DEFBUTTON4           equ 300h
3404  MB_APPLMODAL            equ 0h
3405  MB_SYSTEMMODAL          equ 1000h
3406  MB_TASKMODAL            equ 2000h
```

Plusieurs constantes peuvent être combinées en un paramètre pour une fonction en les séparant avec "**or**" ou "**+**". Par exemple Modifier le code en:

```
invoke MessageBox, NULL, addr MsgBoxText, addr MsgBoxCaption,
MB_OKCANCEL or MB_ICONEXCLAMATION or MB_DEFBUTTON2
```

Ca vous donnera une fenêtre comme ci-dessous:



Chapitre 6 : Principe de base de la programmation Windows

Pour se lancer dans le développement d'applications pour Windows, il est indispensable de bien connaître les spécificités de la programmation sous Windows, qui se distingue nettement de la programmation traditionnelle. La programmation Windows passe obligatoirement par l'**API Win32**, qui rassemble tous les types de données et les fonctions qui s'utilisent pour créer des programmes Windows. L'**API Win32** est importante, et il faut consentir quelques efforts pour se sentir à l'aise avec ses nombreuses facettes.

Certains aspects de la programmation Win32 sont communs à tous les programmes Windows qu'il s'agisse d'un traitement de texte ou d'un jeu. Ce sont surtout ces caractéristiques communes qui sont exposées dans ce chapitre. Une fois que vous l'aurez lu et assimilé, vous aurez des bases solides qui vous permettront d'acquérir des connaissances valables pour toute sorte de programme.

Programmation événementielle

Un des aspects les plus déroutants de la programmation Windows pour les débutants est la nature événementielle de cette programmation, plus simplement « événementielle » signifie que les programmes Windows rependent à leur environnement. Le flux du programme est dirigé par les événements externes au programme, comme des clics de souris et des frappes au clavier. Au lieu d'écrire du code détectant les frappes du clavier, le programmeur Windows écrit du code qui accomplit une action chaque fois qu'une frappe au clavier est rapportée au programme. Au fur et à mesure que vous vous familiarisez avec le concept d'événements, vous pouvez constater que toute modification de l'environnement peut déclencher un événement. Vous pouvez ainsi concevoir et écrire vos programmes de manières qu'ils ignorent ou réagissent à des événements spécifiques.

La nature événementielle de Windows a beaucoup à voir avec le fait que Windows est un système d'exploitation graphique. Windows a fort à faire et effectue un traitement très complexe pour permettre à plusieurs programmes d'être exécutés simultanément et de partager le même espace à l'écran, de la mémoire, des périphériques d'entrée et, virtuellement, toute autre ressource système. L'approche événementielle de la gestion des interactions avec l'utilisateur et des modifications système est pour beaucoup dans la flexibilité et la puissance de Windows. Dans le contexte de la programmation d'applications, l'approche événementielle permet de décomposer les tâches en fonction des événements qui se produisent. En d'autres termes, un clic avec le bouton gauche de la souris, avec le bouton droit de la souris, une écriture dans une zone de texte, un choix dans une case d'option ou une case à cocher, le déplacement d'un objet et une frappe au clavier sont traités comme des événements distincts.

Communication par le biais de messages

Dans les programmes non graphiques traditionnels, le programme invoque une fonction système pour accomplir une tâche donnée. Dans les programmes Windows graphiques, ce scénario est poussé plus avant, puisque Windows peut invoquer une fonction du programme. En effet, Windows peut envoyer un *message* au programme, dans lequel sont contenues des informations relatives à un événement. Il envoie des messages à chaque fois que se produit quelque chose dont un programme souhaite être informé, comme le glissement de la souris ou le redimensionnement de la fenêtre principale du programme par l'utilisateur.

Les messages sont adressés à une fenêtre spécifique, qui est généralement la fenêtre principale. Dans une session Windows, plusieurs fenêtres cohabitent généralement à l'écran, et chaque fenêtre reçoit du temps à autre des messages informant des changements survenus. A chacune de ces fenêtres est associée *une procédure de fenêtre*, qui est une fonction spéciale chargée de traitement des messages envoyées par le système d'exploitation à la fenêtre. Windows se charge d'adresser les messages aux procédures de

fenêtre appropriées. Il appartient au programmeur d'écrire le code de la procédure de fenêtre pour que le programme réponde à certains messages.

La notion d'indépendance vis-à-vis des périphériques

A l'âge d'or des jeux vidéo par exemple, les développeurs de jeux écrivaient souvent leurs programmes d'une manière telle qu'ils utilisaient directement la mémoire de la carte graphique. Cette technique rendait l'exécution des jeux extrêmement rapide, parce que les images des jeux étaient directement traites par la carte graphique. Windows n'offre pas cette possibilité. De manière générale, Windows a été volontairement conçu pour empêcher les utilisateurs d'interagir directement avec les composantes de l'ordinateur, comme la carte graphique. De ce fait, les applications Windows sont dites *indépendantes vis-à-vis des périphériques*, c'est-à-dire qu'elles dessinent les images de manière indépendante des périphériques utilisés à cette fin. L'intérêt de cette approche réside dans le fait que les applications Windows peuvent fonctionner avec une vaste gamme de matériel sans aucune modification particulière. Une telle compatibilité était impensable dans l'univers DOS.

L'indépendance vis-à-vis de périphériques sous Windows est rendue possible par l'interface de périphérique graphique, ou **GDI** (Graphical Device Interface), qui est la partie de l'**API Win32** qui gère l'aspect graphique. L'interface GDI intervient entre le programme et le matériel graphique physique. Au niveau du GDI, les images sont dessinées sur un périphérique virtuel de sortie. C'est ensuite à Windows qu'il appartient d'opérer la transition entre le périphérique de sortie virtuel et un périphérique réel spécifique, *via* la configuration système, les pilotes des périphériques et ainsi de suite. Bine que l'interface **GDI** a été expressément conçue pour vous garder à distance du matériel graphique, elle reste très performante. En fait, toutes les applications présentées dans ce guide sont uniquement fondées sur cette interface.

Stockage des informations des programmes sous forme de ressources

Quasiment chaque programme Windows utilise des *ressources*, c'est-à-dire des éléments d'informations relatifs au programme situés hors du code du programme lui-même. Par exemple, les icônes sont des ressources, tout comme les images, les sons ou les menus. Toutes les ressources d'un programme Windows sont spécifiées dans un script de ressource spécial, appelé fichier **RC**. Celui-ci est un fichier texte contenant une liste de ressources et qui est compilé et lié au programme exécutable final. La plupart des ressources sont créées et modifiées au moyen d'outils de ressource visuels. Comme l'éditeur d'icône intégré dans l'environnement de développement RadAsm.

Des types de données étranges

La plus grande difficulté dans l'apprentissage de la programmation Windows est peut être de se sentir à l'aise avec les types de données étranges qui sont utilisé dans tout programme Windows. Un des types de données les plus communes est appelé *descripteur* (*handle*). Sous Windows, un descripteur est un nombre qui fait référence à un objet Windows graphique, comme une fenêtre ou une icône. Les descripteurs sont importants, parce que Windows déplace souvent les objets dans la mémoire et que les adresses mémoire, de ce fait, sont des cibles « mouvantes ». Les descripteurs sont utilisés dans toute l'API Win32 pour manipuler

les objets Windows. Ainsi formulée, cette notion de descripteur vous semble peut-être abstraite et difficile à saisir. Mais vous la comprendrez bien mieux dès lors que vous la verrez mise en pratique dans un programme Windows.

Outre les descripteurs, l'**API Win32** introduit toute une gamme de nouveaux types de données, qui peuvent sembler étrange au novice. Tous les types de données Win32 étant écrit en majuscules, ils sont faciles à identifier. Par exemple, **RECT** est un type de données assez simple et couramment utilisé. **RECT** représente une structure rectangulaire. **HWND** est également un type de données assez courant, qui représente la poignée d'une fenêtre. Compte tenu du grand nombre de types de données utilisés dans l'**API Win32,** nous ne les passerons par tous symétriquement en revue dans ce chapitre. En revanche, nous décrirons les types de données de cette API au fur et à mesure que nous les rencontrons dans les programmes de ce guide.

Convention spéciale de nommage

Si vous avez déjà jeté un coup d'œil sur un programme Windows, vous vous êtes peut-être interrogé sur les noms des variables. Les programmeurs cherchent depuis toujours à trouver le moyen d'écrire du code qui soit facile à comprendre. Windows rend cette tâche particulièrement complexe, du fait du grand nombre et de la diversité de ses types de données. Le programmeur Microsoft Charles Simonyi, aujourd'hui légendaire, proposa une solution plutôt astucieuse à ce problème : la *notation hongroise* (M. Simonyi est hongrois). Dans cette notation, les noms de variables commencent par une lettre ou des lettres minuscules qui sont indicatives du type de données de la variable. Par exemple, tous les noms, les variables entières commencent par la lettre **i**. Voici quelques-uns des préfixes de notation hongroise couramment utilisés dans les programmes Windows :

- **i** : entier;
- **b** : booléen (**BOOL**);
- **c** : caractère;
- **s** : chaîne;
- **sz** : chaîne terminé par zéro;
- **p** : pointeur;
- **h** : descripteur (handle);
- **w** : entier court non signé (**WORD**);
- **dw** : entier long non signé (**DWORD**);
- **l** : entier long.

BOOL, **WORD** et **DWORD** sont des types de données Win32 couramment utilisés.

La notation hongroise est facile à appliquer. Par exemple, un compteur entier pourrait être nommé **iCompt** et un descripteur d'icône, **hIcone**. De la même manière, une chaine terminée par un zéro stockant le nom d'une équipe sportive pourrait être appelée **szNomEquipe**. Notez bien que cette notation est totalement optionnelle, mais très pratique. Nous l'appliquerons dans tous les programmes de ce guide, et vous pourrez ainsi juger rapidement de sa commodité.

La programmation Windows en pratique

Nous pourrions poursuivre ainsi notre exposé théorique sur l'**API Win32**. Toutefois, étant donné que notre objectif est donc arrivé le plus rapidement possible à la pratique de l'API Win32 en assembleur, nous allons directement à l'étude d'un programme Windows. Dans la prochaine section nous étudierons les principaux aspects d'un programme Windows, en examinant le

code correspondant. A ce stade, il n'est pas capital que vous compreniez chaque ligne du code.

Une simple fenêtre

Ci-dessous voici le code source de notre programme ouvrant une simple fenêtre. Avant une plongée dans les entrailles de la programmation Win32 assembleur, regardons quelques points importants qui soulageront votre programmation.

Vous devez mettre tous les constants Windows, les structures et les prototypes de fonction dans un fichier includeE (*.inc) et l'inclure au début de votre fichier .asm. Cela vous évitera de gros efforts de réécriture par vous même. Actuellement, la plupart des fichiers INCLUDE ,tel que windows.inc, sont complets. Vous pouvez aussi définir vos propres constants et structures mais vous devez les mettre dans un fichier INCLUDE (*.inc) différent de 'windows.inc'.

Les constants Windows, les structures et les prototypes de Fonction sont toutes des instructions assembleur. Donc si vous n'avez pas compris, on écrit seulement des bouts de programme en ASM dans des fichiers *.inc et quand vous en avez besoin pour un de vos programmes, vos y faites référence grâce à une instruction spéciale :'INCLUDE…'

Utilisez la directive includelib pour indiquer quelle bibliothèque d'importation est employée par votre programme. Par exemple, si votre programme appelle la fonction MessageBox, vous devez mettre la ligne :

includelib user32.lib

Au début de votre fichier .asm. Cette directive informe MASM que votre programme utilisera certaines fonctions dans cette bibliothèque d'importation. Si vos fonctions d'appels de programme ne font pas parti de la même bibliothèque, ajoutez juste un IncludeLib pour chaque bibliothèque que vous employez. En employant la directive IncludeLib, vous n'avez plus à vous inquiétez des bibliothèques d'importations.

En déclarant des prototypes de fonction API, des structures, ou des constants pour les inclure dans vos propres fichiers, essayez de respecter les noms d'origines employés par les fichiers INCLUDE de Windows. Ça vous évitera bien des maux de tête à chercher quelque article dans la référence Win32 API.


```
;-------------------------------------------------------------
;   HelloWin.asm -- Affiche "L'assembleur Win32 est super !"
;   dans titre de la fenêtre -- (c) Lord Noteworthy, 2009
;-------------------------------------------------------------

.386
.model flat,stdcall
option casemap:none

include windows.inc

include user32.inc
includelib user32.lib

include kernel32.inc
includelib kernel32.lib

WinMain proto :DWORD,:DWORD,:DWORD,:DWORD

.data
ClassName    BYTE     "SimpleWinClass",0
AppName      BYTE     "L'assembleur Win32 est super !",0

.data?
hInstance HINSTANCE ?
CommandLine LPSTR ?

.code
start:
invoke GetModuleHandle, NULL
 mov hInstance,eax

invoke GetCommandLine
mov CommandLine,eax

invoke WinMain, hInstance,NULL,CommandLine, SW_SHOWDEFAULT
```


```
invoke ExitProcess, eax

-------------------------------------------------------------------
WinMain proc hInst:HINSTANCE,hPrevInst:HINSTANCE,
             CmdLine:LPSTR,CmdShow:DWORD
     LOCAL wc:WNDCLASSEX
     LOCAL msg:MSG
     LOCAL hwnd:HWND

     mov   wc.cbSize,SIZEOF WNDCLASSEX
     mov   wc.style, CS_HREDRAW or CS_VREDRAW
     mov   wc.lpfnWndProc, OFFSET WndProc
     mov   wc.cbClsExtra,NULL
     mov   wc.cbWndExtra,NULL
     push  hInstance
     pop   wc.hInstance
     mov   wc.hbrBackground,COLOR_WINDOW+1
     mov   wc.lpszMenuName,NULL
     mov   wc.lpszClassName,OFFSET ClassName
     invoke LoadIcon,NULL,IDI_APPLICATION
     mov   wc.hIcon,eax
     mov   wc.hIconSm,eax
     invoke LoadCursor,NULL,IDC_ARROW
     mov   wc.hCursor,eax
     invoke RegisterClassEx, addr wc
     invoke CreateWindowEx,NULL,\
                 ADDR ClassName,\
                 ADDR AppName,\
                 WS_OVERLAPPEDWINDOW,\
                 CW_USEDEFAULT,\
                 CW_USEDEFAULT,\
                 CW_USEDEFAULT,\
                 CW_USEDEFAULT,\
                 NULL,\
                 NULL,\
                 hInst,\
                 NULL
```

```
    mov   hwnd,eax
    invoke ShowWindow, hwnd,CmdShow
    invoke UpdateWindow, hwnd

    .WHILE TRUE
        invoke GetMessage, ADDR msg,NULL,0,0
        .BREAK .IF (!eax)
        invoke TranslateMessage, ADDR msg
        invoke DispatchMessage, ADDR msg
    .ENDW
    mov     eax,msg.wParam
    ret
WinMain endp

-------------------------------------------------------------------------
WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
    .IF uMsg==WM_DESTROY
        invoke PostQuitMessage,NULL

    .ELSE
        invoke DefWindowProc,hWnd,uMsg,wParam,lParam
        ret

    .ENDIF
    xor eax,eax
    ret
WndProc endp
end start
```

Un tel listing a de quoi effrayer le profane. Nous le donnons ici d'un bloc pour que vous ayez une idée de contenu d'un programme Windows complet. Heureusement, la plupart de ces codes sont juste une ossature commune pour tous les programmes. Ces codes que vous pouvez les copier d'un fichier de code source à un autre. Ou si vous préférez, vous pourriez assembler quelques-uns de ces codes dans une bibliothèque pour être employés comme une sorte d'introduction dans votre programme. Le prochain chapitre dissimulera une grande partie de ce code, dont vous n'aurez plus alors à vous soucier. Mais, avant cela, examinons ce programme dans ses grandes lignes.

Les trois premières lignes sont nécessaires. .386 dit à MASM que nous avons l'intention d'employer le jeu d'instruction de processeur 80386 dans ce programme. .model flat, stdcall dit à MASM que notre programme emploie la mémoire plate adressant le modèle. C'est pourquoi nous employions le paramètre stdcall prenant par défaut cette convention dans notre programme.

Ensuite c'est au tour du prototype de fonction de WinMain. Puisque nous appellerons WinMain plus tard, nous devons définir son prototype de fonction d'abord, pour que nous puissions l'invoquer par la suite.

Nous devons inclure windows.inc au début du code source. Il contient les structures importantes et les constants qui sont employés par notre programme. Le fichier INCLUDE, windows.inc, est simplement un fichier texte. Vous pouvez l'ouvrir avec n'importe quel éditeur de texte. Notez s'il vous plaît que windows.inc ne contient pas toutes les structures et constants (encore).

Notre programme appelle les fonctions API qui résident dans user32.dll (CreateWindowEx, RegisterWindowClassEx, par exemple) et kernel32.dll (ExitProcess), donc nous devons lié notre

programme avec ces deux bibliothèques d'importation. La question suivante : comment puis-je savoir qui importe la bibliothèque qui doit être liée avec mon programme ? La réponse : Vous devez savoir où résident les fonctions API appelées par votre programme. Par exemple, si vous appelez une fonction API dans gdi32.dll, vous devez vous lier avec gdi32.lib.

C'est l'approche spécifique de MASM. Pour TASM, la bibliothèque d'importation qui est liée à votre programme est uniquement : import32.lib. C'est beaucoup, beaucoup plus simple.

```
.data
ClassName    BYTE     "SimpleWinClass",0
AppName      BYTE     "L'assembleur Win32 est super !",0

.data?
hInstance    HINSTANCE    ?
CommandLine  LPSTR        ?
```

Ci-dessus on trouve les deux sections pour les "DATA".

Dans .data, on déclare deux données terminées chacune par zéro :

- **ClassName** désigne la classe de fenêtre à partir de laquelle la fenêtre va être créée.
- **AppName** désigne le nom de la fenêtre, c'est-à-dire, dans le cas d'une fenêtre, le string qui va être affichée en guise de titre. Notez que les deux variables sont initialisées.

Dans .data?, deux variables sont déclarées :

- **HINSTANCE** désigne l'instance Handle de notre programme.
- **CommandLine** désigne la ligne de commande de notre programme. Ces deux données sont extrêmement courantes, **HINSTANCE** et **LPSTR**, sont vraiment de nouveaux noms (de type: DWORD). Vous pouvez les voir dans windows.inc. Notez que chacune des variables se trouvant dans la section .data? ne sont pas initialisées, c'est-à-dire qu'elles ne doivent pas contenir de valeur spécifique en démarrage, mais nous réservons de l'espace pour notre future utilisation.

```
.code
start:
invoke GetModuleHandle, NULL
mov hInstance,eax

invoke GetCommandLine
mov CommandLine,eax

invoke WinMain, hInstance,NULL,CommandLine, SW_SHOWDEFAULT
invoke ExitProcess, eax
```

.code contient toutes vos instructions ASM. Vos codes doivent résider entre le label « start » et le label "End Start ". Notre première instruction est l'appel GetModuleHandle pour aller chercher l'Instance handle de notre programme. Sous Win32, l'Instance Handle et Module Handle sont les mêmes. Vous pouvez vous représenter l'Instance Handle comme le n° d'identification de votre programme. Il est employé comme paramètre par plusieurs des fonctions API que notre programme doit appeler, donc c'est généralement une bonne idée de le déclarer dès le début de notre programme.

Notez : En réalité sous win32, l'Instance Handle est l'adresse linéaire de votre programme dans la mémoire.

Après avoir fait appel à une fonction de Win32 (une API), la valeur de retour de cette fonction, est placée dans eax. Toutes les autres valeurs sont renvoyées par des variables que vous avez vous-même défini avant le Call qui appel cette fonction.

Une fonction Win32 (ou une procédure) que vous appelez (Call) souvent préservera les registres de segment ebx, edi, esi et le registre ebp. Au contraire, ecx et edx sont considérés comme des registres de travail et sont toujours indéfinis après la sortie d'une fonction de Win32 (sortie d'une procédure).

Notez : n'espérez pas que les valeurs d'eax, ecx, edx soient préservés après un appel de fonction API.

La ligne qui suit un appel (un call) à une fonction d'API, attend en retour une valeur dans eax. Dès que vous appelez une fonction API de Windows, vous devez utiliser la règle suivante : préservez puis rétablissez les valeurs des registres de segment ebx, edi, esi et ebp après le retour de la fonction sinon votre programme plantera peu de temps après, ceci est valable pour vos procédures Windows et pour les fonctions de rappel de service Windows.

L'appel 'GetCommandLine' est inutile si votre programme ne traite pas de ligne de commande. Dans cet exemple, je vous montre comment l'appeler dans le cas où vous en auriez besoin dans votre programme.

Vient ensuite l'appel de WinMain. Ici il reçoit quatre paramètres : l'Instance Handle de notre programme, l'Instance Handle du précédent Instance de notre programme (un programme de plus haut niveau qui aurait appelé le notre), la Command Line et l'état de la fenêtre à sa première apparition. Sous Win32, il n'y a aucun Instance précédent (premier). Chaque programme est seul dans son espace d'adresse, donc la valeur d'hPrevInst vaut toujours 0. C'est un des restes de Win16 quand tous les programmes étaient encore exécutés dans un même espace d'adresses pour savoir si c'était le premier Instance ou non. Donc sous win16, si hPrevInst est NULL, alors c'est que c'est le premier Instance. le petit dernier ne vous servira probablement pas souvent : il s'agit du mode d'affichage (visible, invisible, minimisé etc.) de la fenêtre principale.

Notez : Vous n'êtes pas obligé de déclarer le nom de fonction en tant que : 'WinMain'. En fait, vous avez une totale liberté à cet égard. Et même, vous pouvez ne pas employer de fonction WinMain-équivalent du tout. Le but est de placer votre code (votre véritable programme) à l'intérieur de la fonction WinMain à côté de GetCommandLine et votre programme sera toujours capable de fonctionner parfaitement.

A la sortie de WinMain, eax est rempli du code de sortie. Nous passons ce code de sortie comme paramètre à ExitProcess qui sert à terminer notre programme.

```
WinMain proc hInst:HINSTANCE,hPrevInst:HINSTANCE,
            CmdLine:LPSTR,CmdShow:DWORD
```

La susdite ligne est la déclaration de la fonction WinMain. Notez l'ensemble des paramètres qui suivent la directive PROC. Ce sont les paramètres que WinMain reçoit de la fonction qui l'appelle. Vous pouvez utiliser les noms de ces paramètres au lieu de vous servir de la pile pour pousser les valeurs de vos paramètres.

```
LOCAL wc:WNDCLASSEX
LOCAL msg:MSG
LOCAL hwnd:HWND
```

Nous aurons besoin pour afficher la fenêtre de plusieurs variables :

- **wc** est la classe de fenêtre nécessaire à la construction de la fenêtre principale. C'est en fait une liste d'informations permettant par la suite de créer la fenêtre selon notre bon vouloir.
- **msg**, quant à lui, est comme son nom l'indique un message système, utilisé pour communiquer entre l'utilisateur et les fenêtres : il s'agit en fait du descriptif d'un événement.
- **hwnd** est de type **HWND**, qui veut dire handle de fenêtre.

```
mov    wc.cbSize,SIZEOF WNDCLASSEX
mov    wc.style, CS_HREDRAW or CS_VREDRAW
mov    wc.lpfnWndProc, OFFSET WndProc
mov    wc.cbClsExtra,NULL
mov    wc.cbWndExtra,NULL
push   hInstance
pop    wc.hInstance
mov    wc.hbrBackground,COLOR_WINDOW+1
mov    wc.lpszMenuName,NULL
mov    wc.lpszClassName,OFFSET ClassName
invoke LoadIcon,NULL,IDI_APPLICATION
mov    wc.hIcon,eax
mov    wc.hIconSm,eax
invoke LoadCursor,NULL,IDC_ARROW
mov    wc.hCursor,eax
invoke RegisterClassEx, addr wc
```

Les lignes ci-dessus sont vraiment simples par leur concept. Il s'agit juste de plusieurs lignes d'instruction regroupées. Le concept qui se cache derrière toutes ces lignes c'est d'initialiser la Windows class.

La classe de fenêtre

Une 'Windows Class' ou « classe de fenêtre » n'est rien de plus que la première ébauche d'une fenêtre. On définit donc plusieurs caractéristiques importantes d'une fenêtre comme son icône, son curseur, la fonction qui la contrôle, sa couleur etc. Vous définissez la fenêtre grâce à sa 'Windows Class'. C'est en quelque sorte le but de ce concept. Si vous voulez créer plusieurs fenêtres avec les mêmes caractéristiques, il serait temps de penser à stocker toutes ces caractéristiques dans un unique endroit (donc dans une procédure par exemple) et de s'y référer quand c'est nécessaire. Cet arrangement économisera pas mal de mémoire en évitant la duplication d'informations. Windows doit être très efficace dans l'utilisation des ressources mémoires, qui restent encore rares.

Faisant le point : toutes les fenêtres générées par un programme Windows sont crées à partir d'une classe de fenêtre, qui est un modèle définissant tous les attributs d'une fenêtre. Plusieurs fenêtres peuvent être générées à partir d'une même classe de fenêtre. Par exemple, il existe une classe de fenêtre Win32 standard qui définit les attributs d'un bouton et qui est utilisée pour créer tous les boutons dans Windows. Pour créer une fenêtre de votre

cru, vous devez enregistrer une classe de fenêtre, puis créer une fenêtre à partir de cette classe. Pour pouvoir créer de nouvelles fenêtres à partir d'une classe de fenêtre, la classe de fenêtre doit être enregistrée auprès de Windows, en utilisant la fonction de l'API Win32 **RegisterClass** ou **RegisterClassEx**. Une fois qu'une classe de fenêtre a été enregistrée, vous pouvez l'utiliser pour créer autant de fenêtres qu'il vous sied.

Dans l'API Win32, les classes de fenêtre sont représentées par une structure de donnée appelée **WNDCLASS** ou **WNDCLASSEX**, cette structure va nous permettre de définir un type de fenêtre à part entière dont on pourra créer des exemplaires à volonté. Vous pouvez, par exemple, voir cette structure comme un patron sur papier pour une maison définissant complètement les caractéristiques de votre maison et permettant de créer autant d'exemplaire de cette maison que vous le désirez. **WNDCLASSEX** est définie de la manière suivante :

WNDCLASSEX Structure

The **WNDCLASSEX** structure contains window class information. It is used with the RegisterClassEx and GetClassInfoEx functions.

The **WNDCLASSEX** structure is similar to the WNDCLASS structure. There are two differences. **WNDCLASSEX** includes the **cbSize** member, which specifies the size of the structure, and the **hIconSm** member, which contains a handle to a small icon associated with the window class.

Syntax

```
typedef struct _WNDCLASSEX {
    UINT    cbSize;
    UINT    style;
    WNDPROC lpfnWndProc;
    int     cbClsExtra;
    int     cbWndExtra;
    HANDLE  hInstance;
    HICON   hIcon;
    HCURSOR hCursor;
    HBRUSH  hbrBackground;
    LPCTSTR lpszMenuName;
    LPCTSTR lpszClassName;
    HICON   hIconSm;
} WNDCLASSEX;
```

Quelques-uns des types de données étranges évoqués précédemment dans ce chapitre apparaissent dans ce code. A ce stade, nous n'avons pas besoin d'expliquer ce morceau de code ligne par ligne. Nous nous contenterons d'examiner quelques unes de ces éléments les plus intéressants :

- **cbSize** : La taille de la structure **WNDCLASSEX** en octets. Nous pouvons employer l'opérateur **SIZEOF** pour obtenir (initialiser) cette valeur.
- **style** : correspond à une combinaison de plusieurs options de style comme **WS_CAPTION** et **WS_BORDER** qui déterminent l'aspect et le comportement de la fenêtre.
- **cbWndExtra**: Indique le nombre d'octets supplémentaires à allouer en fonction de la 'Windows Instance'. Le système d'exploitation initialise les octets à zéro.

- **cbClsExtra**: Indique le nombre d'octets supplémentaires à allouer en fonction de la structure (ses éléments) de la Windows Class choisie. Le système d'exploitation initialise les octets à zéro. Vous pouvez stocker des données spécifiques à la Windows Class ici.
- **lpfnWndProc** : est un pointeur sur une fonction du programme qui reçoit et traite des messages d'événements déclenchés par l'utilisateur.
- **hIcon** et **hCursor** : contiennent les handles vers l'icône et le curseur de programme. Obtenez-les après l'appel de **LoadIcon** et **LoadCursor**.
- **lpszMenuName** : pointe sur une chaine définissant le menu associé à cette fenêtre, ici, il n'y en aura aucun.
- **lpszClassName** : pointe sur une chaîne à zéro terminal contenant le nom de la classe de la fenêtre.

Ces éléments sont relatifs aux parties d'une fenêtre les plus évidentes. **lpfnWndProc** est probablement le plus subtil à comprendre, parce qu'il s'agit d'un pointeur sur la procédure de fenêtre de la classe de fenêtre. Nous reviendrons un peu plus loin sur cette procédure de fenêtre. Les éléments **hIcon** et **hIconSm** sont utilisés pour définir les icônes de la classe de fenêtre : ils correspondent aux icônes de programme qui sont visibles lorsqu'un programme est en cours d'exécution sous Windows. **hCursor** s'utilise pour définir un pointeur de souris spécial pour la classe de fenêtre, qui se substitue au pointeur flèche standard. Et pour finir, **hbrBackground** s'utilise pour définir l'arrière-plan de la classe de fenêtre. Le blanc est utilisé comme couleur d'arrière plan pour la majorité des fenêtres, mais vous pouvez très bien choisir une autre couleur.

Une fois encore, il n'est pas nécessaire à ce stade que vous compreniez en détail la structure d'une classe de fenêtre. L'idée est seulement ici de vous familiariser suffisamment avec la programmation Win32 pour que nous puissions assembler un programme complet. Un peu plus loin dans ce chapitre, nous ferons usage de la structure de classe de fenêtre pour créer un programme Windows minimal.

Création d'une fenêtre

Après l'enregistrement de la classe de fenêtre, la création d'une fenêtre principale représente la partie critique d'un programme Windows. Pour créer une fenêtre, vous devez en passer par une classe de fenêtre. Bien que les classes de fenêtre définissent les caractéristiques générales d'une fenêtre. Ces attributs doivent être spécifiés sous la forme d'arguments de la fonction **CreateWindowEx**. Celle-ci est la fonction de l'API Win32 dédiée à la création de la fenêtre. L'exemple qui suit montre comment créer une fenêtre au moyen de la fonction **CreateWindowEx**, Remarquez qu'il y a 12 paramètres à cette fonction, beuurr ! ! !

CreateWindowEx Function

The **CreateWindowEx** function creates an overlapped, pop-up, or child window with an extended window style; otherwise, this function is identical to the CreateWindow function. For more information about creating a window and for full descriptions of the other parameters of **CreateWindowEx**, see **CreateWindow**.

Syntax

```
HWND CreateWindowEx(
    DWORD dwExStyle,
    LPCTSTR lpClassName,
    LPCTSTR lpWindowName,
    DWORD dwStyle,
    int x,
    int y,
    int nWidth,
    int nHeight,
    HWND hWndParent,
    HMENU hMenu,
    HINSTANCE hInstance,
    LPVOID lpParam
);
```

Il n'est pas vital que vous compreniez chacune des lignes de ce fragment du code. Nous allons nous contenter d'en étudier les aspects les plus intéressants. Tout d'abord, **dwExStyle** c'est le style de fenêtre supplémentaire. C'est le nouveau paramètre qui est ajouté à la vieille fonction 'CreateWindow'. Vous pouvez mettre des nouveaux styles de fenêtre pour Windows 9x & NT ici. En temps normal, vous pouvez spécifier votre style de fenêtre ordinaire dans dwStyle, mais si vous voulez certains styles spéciaux pour une fenêtre 'tip top', vous devez les spécifier ici. Vous devez employer un NULL si vous ne voulez pas de styles de fenêtre supplémentaires. Le nom de la classe de fenêtre est spécifié comme argument, **ClassName**. Après vient le titre de la fenêtre, « **L'assembleur Win32 est super !** ». Ce titre apparaîtra dans la barre de titre de la fenêtre à l'exécution du programme. **WSOVERLAPPEDWINDOW** est un style Win32 standard qui identifie une fenêtre traditionnelle redimensionnable. Les quatre styles C**W_USEDDEFAULT** indiquent la position XY de la fenêtre à l'écran, ainsi que la hauteur et la largeur de la fenêtre. Vous pouvez régler ces paramètres sur des valeurs numériques, mais **CW_USEDDEFAULT** indique à Windows de choisir une valeur par défaut raisonnable. **hWndParent**, c'est l'Handle de la fenêtre parente de votre fenêtre (si celle-ci existe). Ce paramètre indique à Windows si cette fenêtre est un enfant (Window Child) d'une autre fenêtre et, si elle l'est, qui est la fenêtre parente. Notez que ce n'est pas la même relation Parent-Enfant que les 'multiple document interface' ou (MDI). Les 'Window Child' n'ont pas forcément les mêmes caractéristiques que leur 'Window Parent'. Ces caractéristiques sont uniquement spécifiques à l'utilisation interne de chaque fenêtre. Si la fenêtre parente est détruite, toutes ses fenêtres enfant seront détruites automatiquement. C'est vraiment très simple. Dans notre exemple, il y a seulement une fenêtre, donc ce paramètre reste NULL. **hMenu**: C'est l'Handle du menu de la fenêtre. Il est NULL si la Class Menu est utilisée. Revenez en arrière à un des membres de l'instruction WNDCLASSEX, soit 'lpszMenuName'. lpszMenuName indique le menu par *default* pour la 'Window Class'. Chaque fenêtre de cette Window Class aura le même menu par défaut. À moins que vous

ne spécifiiez un menu *overriding* pour une fenêtre spécifique via son paramètre hMenu. hMenu est en réalité un paramètre à double usage. Si la fenêtre que vous souhaitez créer utilise un 'Type de Fenêtre' (Style) prédéterminé (c'est-à-dire un Control), un tel Control ne peut pas posséder de menu. hMenu est employé comme ID de ce Control au lieu de cela. Windows peut décider si hMenu est vraiment un Handle de menu ou un Control ID en regardant le paramètre lpClassName. Si c'est le nom d'une Window Class prédéterminée, hMenu est un Control ID. Si ce n'est pas, ce sera alors l' Handle du menu de votre fenêtre. hInstance: l'Instance Handle pour le module (la partie) du programme créant la fenêtre. lpParam: Le pointeur facultatif pour des données a passer à la fenêtre. Il est employé par la fenêtre MDI pour passer les données **CLIENTCREATESTRUCT**. Normalement, cette valeur est mise au NULL, signifiant que l'on ne passe aucunes données via CreateWindow. La fenêtre peut prendre la valeur de ce paramètre après l'appel à la fonction **GetWindowLong**.

Bon d'accord, elles sont lourdes... mais non, vous n'êtes pas obligés de les retenir. Mais à force de les utiliser (vous verrez), vous les connaîtrez par cœur. "C'est en forgeant qu'on devient forgeron" à ce qu'il paraît.

```
mov    hwnd,eax
invoke ShowWindow, hwnd,CmdShow
invoke UpdateWindow, hwnd
```

En retour couronné de succès de **CreateWindowEx**, l'Handle de la fenêtre est placée dans **eax**. Nous devons garder cette valeur pour une utilisation future. La fenêtre que nous venons de créer n'est pas automatiquement affichée. Mais la fenêtre ne s'affichera pas pour autant. Elle est créée, mais c'est tout. Pour l'afficher, il faut appeler la fonction **ShowWindow** ! Celle-ci prend en premier paramètre le handle de la fenêtre dont on veut changer le mode d'affichage, et un DWORD déterminant son mode. Vous pouvez soit utiliser le paramètre de WinMain modeDAffichage, soit utiliser la valeur qu'il prend d'habitude : SW_SHOW. Maintenant que la fenêtre est en mode "affichée", il faut rafraîchir l'écran afin de la montrer à l'utilisateur : c'est le rôle d'**UpdateWindow**, prenant comme seul paramètre le handle de la fenêtre. Vous l'utiliserez sûrement beaucoup après avoir créé des contrôles enfants d'une fenêtre, afin de rafraîchir l'écran juste après leur création.

Je souhaite que nous commencions par quelques explications sur la notion du « test » et des « boucles » en MASM avant de continuer à analyser notre code.

Les Tests

La directive conditionnelle .IF

MASM définit une directive de haut niveau, nommé .IF (n'oubliez pas le point en préfixe), qui simplifie la rédaction des instructions IF composées par rapport à l'emploi de CMP et sauts conditionnels; voici la syntaxe formelle:

```
  .IF condition1
  ; instructions
[ .ELSEIF condition2
  ; instructions ]
[ .ELSE
  ; instructions ]
  .ENDIF
```

Les crochets montrent que **.ELSEIF** et **.ELSE** sont facultatives alors que **.IF** et **.ENDIF** sont obligatoires. Chaque condition est une expression booléenne qui utilise les mêmes opérateurs que ceux du C++ et de Java. L'expression n'est pas évaluée que pendant l'exécution. Voici quelques exemples de conditions acceptables :

```
eax  >  10000h
val1 <= 100
val2 == eax
val3 != ebx
```

Voici maintenant quelques conditions composées. Nous supposons que **val1** et **val2** sont des variables de type double-mot :

```
(eax > 0) && (eax > 10000h)
(val1 <= 100) || (val2 <= 100)
(val2 != ebx) && !CARRY?
```

Le tableau suivant présente tous les opérateurs relationnels et logiques utilisables.

Opérateur	Description
expr1 = *expr2*	Renvoie Vrai si *expr1* est égale à *expr2*
expr1 != *expr2*	Renvoie Vrai si *expr1* n'est pas égale à *expr2*
expr1 > *expr2*	Renvoie Vrai si *expr1* est plus grande que *expr2*
expr1 >= *expr2*	Renvoie Vrai si *expr1* est plus grande ou égale à *expr2*
expr1 < *expr2*	Renvoie Vrai si *expr1* est plus petite que *expr2*
expr1 <= *expr2*	Renvoie Vrai si *expr1* est plus petite ou égale à *expr2*
!*expr*	Renvoie Vrai si *expr1* est fausse
expr1 && *expr2*	Réalise un AND logique entre *expr1* et *expr2*
expr1 || *expr2*	Réalise un OU logique entre *expr1* et *expr2*
expr1 & *expr2*	Réalise un AND binaire entre *expr1* et *expr2*
CARRY?	Renvoie Vrai si le drapeau **Carry** est armé
OVERFLOW?	Renvoie Vrai si le drapeau **Overflow** est armé
PARTY?	Renvoie Vrai si le drapeau **Parity** est armé
SIGN?	Renvoie Vrai si le drapeau **Sign** est armé
ZERO?	Renvoie Vrai si le drapeau **Zero** est armé

Génération automatique du code ASM

Avec les directives de haut niveau telle que .IF et .ELSE, c'est le compilateur d'assembleur qui va écrire les instructions assembleur à votre place. Partons par exemple d'une directive .IF qui cherche à comparer le contenu de EAX à la variable **val1**:

```
mov eax, 6
.IF eax > val1
    mov resultat, 1
.ENDIF
```

Nous supposons que val1 et resultat sont des entiers non signés sur 32 bits. Lorsque le compilateur lit les lignes précédentes, il est considère comme des instructions assembleur :

```
mov eax, 6
cmp eax, val1
jbe @C0001
mov resultat, 1
@C0001:
```

Le nom du label **@C0001** a été crée par l'assembleur (c'est pourquoi il est peu lisible). Cette convention permet de garantir tous les noms de labels sont uniques dans chaque procédure.

Comparaison signées et non signés

Lorsque vous utiliser .IF pour comparer des valeurs, vous devez vous méfiez de la manière dont sont générés les sauts conditionnels. Si c'est la comparaison concerne une variable non signée, c'est une instruction de saut conditionnel non signée qui sera insérée dans le code généré. Voici un exemple très proche du précédant avec comparaison entre eax et **val1** qui est un double mot non signé :

```
.data
val1        DWORD       5
resultat    DWORD       ?

.code
mov eax, 6
.IF eax > val1
    mov resultat , 1
.ENDIF
```

L'assembleur comprend ces lignes en choisissant d'utiliser l'instruction **JBE** (saut pour non signé) :

```
mov eax, 6
cmp eax, val1
jbe @C0001
mov resultat, 1
@C0001:
```

Comparaisons d'entiers signés

Essayons une comparaison de même style avec **val2**, un double-mot signé :

```
.data
val2 SDWORD -1

.code
mov eax, 6
.IF eax > val2
    mov resultat, 1
.ENDIF
```

A partir de ces lignes l'assembleur va générer du code utilisant l'instruction **JLE** qui permet un branchement en comparant des valeurs signées :

```
mov eax, 6
cmp eax, val2
jle @C0001
mov resultat, 1
@C0001:
```

Comparaisons de registres

Que se passe-t-il si on veut comparer deux registres ? Il est évident que l'assembleur ne pas en déduire l'état signé ou non signé des valeurs :

```
mov eax, 6
mov ebx, val2
.IF eax > ebx
    mov resultat, 1
.ENDIF
```

Dans ce cas le programme assembleur opte pour la comparaison non signé. Autrement dit la directive .IF utilisée pour comparer deux registres donne lieu à l'injection d'une instruction **JBE**.

Expressions composées

Nous avons déjà rencontré les expressions composées au début de chapitre. La plupart des expressions booléennes composées s'articulent autour des deux opérateurs **OR** et **AND**. Si vous utilisez la directive .IF, le symbole || représente l'opérateur **OR** :

```
.IF expression1 || expression2
    ;instructions
.ENDIF
```

Pour l'opérateur **AND**, son homonyme est && :

```
.IF expression1 && expression2
    ;instructions
.ENDIF
```

Les boucles

Et ça y est, on y est, on est arrivés, la voilà, c'est Broadway une autre structure : ça est les **boucles**. Si vous voulez épater vos amis, vous pouvez également parler de **structures répétitives**, voire carrément de **structures itératives.** Ca calme, hein ? Bon, vous faites ce que vous voulez, ici on est entre nous, on parlera de boucles.

Bon, alors nous avons vu jusqu'ici les notions de base, les variables, les opérateurs, ainsi que les structures conditionnelles. Mais il manque encore quelques points importants afin de faire un programme digne de ce nom. En effet, les programmes ont parfois besoin de répéter une ou plusieurs instructions pour effectuer une certaine tâche. C'est même encore plus fréquent que vous ne pourriez l'imaginer.

Les boucles servent à répéter une fonction un certain nombre de dois ou jusqu'à ce qu'une condition soit vraie. En programmation, on veut souvent pouvoir faire quelque chose un certain nombre de fois. Ce genre de boucle apparaît dans toutes les applications possibles et imaginables : les graphismes, les calculs mathématiques, la cryptologie, l'accès aux fichiers … la liste est trop longue.

Directives .REPEAT et .WHILE

Les directives **.REPEAT** et **.WHILE** offrent une solution alternative pour mettre en place des boucles avec **CMP** et des instructions de saut conditionnel. Elles acceptent les expressions de saut conditionnelles vues au tableau *des opérateurs relationnels et logique d'exécution*.

La directive **.REPEAT** exécute les instructions du corps de la boucle avant de tester la condition de boucle qui est placée derrière la directive **.UNTIL**

```
.REPEAT
    ;instructions
.UNTIL condition
```

Le test de condition étant réalisé à la fin, les instructions qui font partie de la boucle **.REPEAT** sont toujours exécutées au mois une fois. Au contrario, dans la boucle **WHILE**, si le teste est faux dès le début, les instructions ne seront jamais exécutées.

La directive **.WHILE** est un peu l'inverse de la précédente : elle teste la condition avant d'exécuter le premier tour de boucle :

```
.WHILE condition
    ;instructions
.ENDW
```

Exemples

L'extrait suivant fait la somme des 100 premiers nombres entiers au moyen de la directive **.WHILE :**

```
mov eax, 0          ; eax = 0
mov ebx, 0          ; ebx = 0
.WHILE eax < 100
    inc eax         ; eax = eax + 1
    add ebx, eax    ; ebx = ebx + eax
.ENDW
```

L'extrait suivant fait la somme des 100 premiers nombres entiers au moyen de la directive **.REPEAT :**

```
mov eax, 0          ; eax = 0
mov ebx, 0          ; ebx = 0
.REPEAT
    inc eax         ; eax = eax + 1
    add ebx, eax    ; ebx = ebx + eax
.UNTIL eax == 100
```

Autre exemple : une boucle contenant l'instruction .IF

L'extrait suivant fait la somme des 100 premiers nombre entiers au moyen de la directive **.WHILE** en sauvegardant la somme des 50 premiers dans une variables **res**.

```
mov eax, 0          ; eax = 0
mov ebx, 0          ; ebx = 0
.WHILE eax < 100
    .IF eax == 50
      mov res, ebx  ; res = ebx
    .ENDIF
    inc eax         ; eax = eax + 1
    add ebx, eax    ; ebx = ebx + eax
.ENDW
```

Sortir de la boucle

Il peut être préférable, dans certains cas, de sortir de la boucle avant que la condition d'arrêt n'ait eu lieu. Si vous voulez sortir d'une boucle dans laquelle vous êtes directement (c'est-à-dire que vous n'êtes pas dans une sous-boucle, boucle **imbriqué**), vous avez à votre disposition la directive **.BREAK**. Cette directive sort directement de la boucle dans laquelle vous êtes, autrement dites, elle permet de briser la structure de contrôle qui l'encadre.

Sachez cependant que la directive **.BREAK** ne sort que de la boucle dans laquelle vous vous trouvez directement. Il vous faudra avoir recours à d'autres astuces pour sortir de toutes les boucles emboîtées d'un coup. Mais je pense que le jour où vous aurez besoin de faire cela, vous trouverez vous-même la solution. Je ne vais donc pas en rajouter pour le moment; voici la syntaxe formelle:

```
.BREAK [ .IF condition ]
```

Il existe également une autre directive pour contrôler une boucle **: .CONTINUE**. Comme la directive **.BREAK**, **.CONTINUE** ne peut être placé que dans le corps d'une boucle. Cette instruction saute directement après la dernière instruction de la boucle, mais sans en sortir. On passe donc directement à l'itération suivante ; voici la syntaxe formelle:

```
.CONTINUE [ .IF condition ]
```

La directive Goto

Cette directive exécute un saut **inconditionnel** ou un **branchement**. Rencontrée au cours de l'exécution d »un programme, la directive **goto** provoque la transfert ou le branchement immédiat vers l'emplacement désigné. Ce branchement est dit inconditionnel, car il ne dépend d'aucune condition.

La directive goto est son étiquette peuvent se trouver dans des blocs de code différents, mais doivent toujours faire partie de la même fonction.

Syntaxe :

```
Test1:
;instructions
...

goto Test2
;instructions
...

Test2:
;instructions
...

goto Test1
```

Oups ! Désolé ! Je parle, je parle et j'oublie l'essentiel, revenons à nos moutons.

Cette fois-ci, notre fenêtre est sur l'écran. Mais elle ne peut pas recevoir d'informations extérieures (comme une saisie de texte ou bien la détection de la pression d'un bouton). Donc nous devons l'informer de ce qui se passe. Nous faisons ça avec une boucle de message.

Traitement des messages

Précédemment dans ce chapitre, nous avons vu que Windows communique avec les programmes en leur transmettant des messages. Nous allons maintenant étudier de plus en détail le fonctionnement de ces messages Windows. Trois éléments information sont associés à un message :

- une fenêtre ;
- un identificateur de message ;
- des paramètres de message.

La fenêtre associée à un message est la fenêtre à laquelle est adressé le message. L'*identificateur de message* est un nombre qui identifie la nature du message envoyé. Dans l'API Win32, une constante numérique est associée à chaque message. Par exemple, **WM_CREATE**, **WM_PAINT** et **WM_MOUSEMOVE** sont trois constantes numériques définies dans l'API Win32, qui identifient respectivement les messages associés à la création de la fenêtre, au tracé de fenêtre et au mouvement de la souris.

Les paramètres de message consistent en deux éléments d'information, qui sont entièrement spécifiques au message envoyé. Ces paramètres à 32 bits sont appelés **wParam** et **lParam**. La signification de ces paramètres dépend du message en cours de traitement. Par exemple, le paramètre **wParam** du message **WM_SIZE** contient des informations relatives au type de dimensionnement possible sur la fenêtre, tandis que le paramètre **lParam** contient les nouvelles valeurs de la largeur et de la hauteur de la zone interne de la fenêtre (également appelée *zone client* de la fenêtre). La largeur et la hauteur sont stockées dans les mors bas et haut de la valeur **lParam** à 32 bits. Ce mode de stockage est l'approche classiquement utilisée dans Win32 pour stocker deux éléments d'information en un même emplacement. Si la notion du mot haut et bas ne vous est pas familière, ne vous inquiétez

pas : nous verrons un peu plus loin, quand nous en aurons besoin, comment extraire du paramètre les informations **lParam** utiles.

Lorsqu'un message est transmis par Windows à un programme, il est traité dans ma fonction **WndProc ()**. Si **WndProc ()** est chargé de traiter les messages pour une classe de fenêtre donnée, c'est au programme qu'il appartient d'acheminer les messages vers les procédures de fenêtre appropriées. Pour cela, on utilise une *boucle de message* au cœur de la fonction **WinMain ()**:

```
.WHILE TRUE
        invoke GetMessage, ADDR msg,NULL,0,0
        .BREAK .IF (!eax)
        invoke TranslateMessage, ADDR msg
        invoke DispatchMessage, ADDR msg
.ENDW
```

Il y a seulement une boucle de message pour chaque module. Cette boucle de message vérifie continuellement les messages de Windows grâce à un 'Call **GetMessage**'. **GetMessage** passe une donnée à une structure de MESSAGE de Windows. Cette structure de messagesera remplie de l'information du message que Windows veut envoyer à une fenêtre dans le module. La fonction **GetMessage** ne retournera pas (ne reviendra pas à votre programme mais au contraire restera dans Kernel ou User) tant qu'un message (ex : appui sur un bouton) ne sera pas transmis pour une fenêtre dans le module. Pendant ce temps-là, Windows peut donner le contrôle à d'autres programmes. C'est en quelque sorte le fonctionnement en Multigestion de la plate-forme Win16. **GetMessage** renverra une erreur si le Message **WM_QUIT** est reçu dans la boucle de message, et ainsi nous termineront la boucle et irons vers la sortie du programme.

TranslateMessage est une fonction utile qui saisie les entrées de clavier (à la volée) et produit un nouveau Message **WM_CHAR** qui est placé sur la file d'attente des messages. Le message avec **WM_CHAR** contient la valeur ASCII pour la touche pressée, c'est bien plus facile de procéder ainsi. Vous pouvez omettre cet appel si votre programme ne traite pas de frappes.

'DispatchMessage' redirige les données d'un message (ex : envoi : on vient d'appuyer sur le bouton) à la procédure responsable de la fenêtre spécifique.

```
    mov     eax,msg.wParam
    ret
WinMain endp
```

Si la boucle de message se termine, le code de sortie est stocké dans le membre wParam de la structure de MESSAGE. Vous pouvez stocker ce code de sortie dans eax pour le rendre à Windows. Actuellement, Windows ne se sert pas de la valeur de retour, mais c'est mieux de respecter les règles.

La procédure de fenêtre

```
WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
```

C'est notre procédure Win. Vous ne devez pas la nommer WndProc. Le premier paramètre, hWnd, est l'Handle de la fenêtre pour laquelle le message est destiné. uMsg est le message. Notez qu'uMsg n'est pas une structure de message. C'est juste un numéro (nombre). Windows définit des centaines de messages, dont la plupart ne seront pas intéressants pour

vos programmes. Windows enverra un message approprié à une fenêtre seulement si quelque chose d'intéressant arrive à cette fenêtre. La procédure qui traite vos fenêtres reçoit le message et y réagit intelligemment. **wParam** et **lParam** sont juste des paramètres supplémentaires pour l'utilisation de certains messages. Certains messages envoient des données d'accompagnement en plus du message lui-même. On passe ces données à la procédure de fenêtre au moyen de **wParam** et **lParam**.

```
    .IF uMsg==WM_DESTROY
        invoke PostQuitMessage,NULL

    .ELSE
        invoke DefWindowProc,hWnd,uMsg,wParam,lParam
        ret

    .ENDIF
    xor eax,eax
    ret
WndProc endp
```

Vient ici la partie cruciale. C'est l'endroit où la plupart de l'intelligence de votre programme réside. Les codes qui répondent à chaque message de Windows sont dans la procédure de fenêtre (la procédure qui s'occupe du contrôle de votre fenêtre). Votre code doit vérifier le message de Windows pour voir si c'est un message intéressant. S'il l'est, faites tout ce que vous voulez faire en réponse à ce message et retourner ensuite la valeur zéro dans eax. S'il ne l'est pas, vous devez appeler l'API 'DefWindowProc', pour lui passer tous les paramètres que vous y avez reçu pour les traiter par défaut. 'DefWindowProc' est une fonction API qui traite les messages qui ne sont pas intéressant pour votre programme. Le seul message que vous pouvez et devez envoyer vers votre procédure qui traite les messages est **WM_DESTROY**. Ce message est envoyé à votre procédure de fenêtre pour la refermée. Au moment où votre procédure de fenêtre reçoit ce message, votre fenêtre est déjà enlevée de l'écran. C'est juste un renseignement comme quoi votre fenêtre a bien été détruite, vous devez vous préparer à retourner à Windows. Si vous utilisez cela, vous n'avez d'autres choix que de quitter. Si vous voulez avoir une chance d'arrêter la fermeture de votre fenêtre, vous devez utiliser le message **WM_CLOSE**. Maintenant revenons à **WM_DESTROY**, après cette fermeture radicale, vous devez appeler PostQuitMessage qui postera **WM_QUIT** en retour à votre module (traitement du message). **WM_QUIT** fera le retour de GetMessage avec la valeur zéro dans eax, qui à son tour, terminera la boucle de message et quittera Windows. Vous pouvez envoyer le message **WM_DESTROY** à votre propre procédure de fenêtre en appelant la fonction DestroyWindow.

Et voilà : vous êtes prêts pour créer votre première fenêtre ! En combinant le tout, vous obtenez cette œuvre magnifique :

Enfin, presque. Vous aurez sûrement une barre de titre différente, à cause du thème de Windows.

Voilà, ce dernier chapitre est terminé, j'espère qu'il vous a plu. Ne vous découragez surtout pas face à la longueur du code, car cela en vaut la peine, et cela va rentrer progressivement en tant qu'automatisme ;)

Conclusion et Remerciements

C'est ici ou s'achève notre petite introduction à l'assembleur, merci de l'avoir lue! Il vous reste tout à découvrir, l'API Win32 en ASM est loin d'être ce que vous avez lu, mais un peu de volonté et de motivation vous serez capable d'être un programmeur ASM confirmé ;) seule la pratique permettant de faire des progrès et c'est comme ceci que les choses rentrent le mieux, à vos claviers !

Je tiens à vous dire que "écrire" est quelque chose de très dur, pour cela je remercie inconcevablement tous ceux qui rédigent des tutoriaux et qui les mettent à notre disposition, tous ceux qui combattent pour vulgariser l'information et tous ceux qui partagent leurs connaissances par la seule et unique motivation : Le plaisir.

J'avoue que je me suis inspiré de certains livres/ezines/ebooks principalement ce des trois auteurs Kip Ivrine « Assembleur x86 », Jean-Bernard Emond « Le tout en poche Assembleur x86 », Michaël Morrison, « Programmation des jeux ». Ce guide est également inspirée de quelques billets à savoir notre chère ami la Wiki, du site : http://win32assembly.online.fr/, merci à Iczelion's, son créateur.

Je tiens à remercier aussi toute personne ayant contribué à la réalisation de ce guide que ce soit de manière directe ou indirecte. J'entends par là : Benoît-M, Rémi Coquet, Obsidian, Alcatîz, Alucard, MrGh0st, N@bilX, DaBDouB-MoSiKaR, Alucard, marksman, Metal-, l'équipe **FAT** (Kaze, Baboon, Tuna, Ed, Silmaril, Poco, Codasyl), **CiM** (Jester [mes resptecs], r007, Mr L, X-Crk, Willingness, BlackPirate, FiRe, xsp!der, DrNitro, Le_Maudit, …), FC(BeatriX2004, Squallsurf, Virtualabs, rAsM, …), AT4RE (Mouradpr, GamingMaster, Zeak47, …), Logram-Project (Youscef, Azerty, SteckDennis …) et la liste est longue donc tous ceux que j'ai oublié.

Vos remarques, suggestions, conseils et critiques me seraient très utiles pour l'améliorer encore ; vous pouvez les envoyer à Noteworthy@live.fr.

Sweet dreams are made of this …

Lord Noteworthy, le 22 Avril 2009 // *FAT Assembler Team*

Liens utiles

• http://www.masm32.com/masmdl.htm MASM32 Version 10 Downloads
•http://www.oby.ro/rad_asm/ RadASM Assembler IDE
• HelpPC Reference Library.
• Win32 Programmer's Reference.
• Ralf Brown's Interrupt List
• Assembly Programming Journal
• Lots of good information about x86 processors
• X86 Opcode and Instruction Reference
• MSDN Online Library
• Intel Assembler Code Table 80x86 - Overview of instructions
• Intel Hex Opcodes And Mnemonics
• Intel Architecture Software Developer's Manual, Volume 1 Basic Architecture.
• Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual.
• Intel Architecture Software Developer's Manual, Volume 3 System Programming Guide.
• Intel 64-bit Pentium Documentation.
  - http://webster.cs.ucr.edu/Page_TechDocs/Intel64.pdf *Volume 1*
  - http://webster.cs.ucr.edu/Page_TechDocs/Intel64_2.pdf *Volume 2*

Références bibliographiques

• The Art of Assembly Language.
• Assembly Language: Step-by-Step.
• Assembly Language Step-by-Step: Programming with DOS and Linux, 2d edition.
• Write Great Code Understanding the Machine, Volume I.
• Write Great Code Thinking Low-Level, Writing High-Level, Volume II.
• 80x86 Assembly Language and Computer Architecture.
• Professional Assembly Language.
• Guide to Assembly Language Programming in Linux.
• The Zen of Assembly Language.
• Linux Assembly Language Programming.

Windows (32-bit) Assembly Tutorials

• The Microsoft® Macro Assembler Programmer's Guide.
• Learn Microsoft Assembler In A Day.
• The Assembly Programming Master Book.
• 32/64-Bit 80x86 Assembly Language Architecture.
• Win32 Assembler Coding for Crackers.
• The Art of Assembly Language 32-Bit Edition.
• Masm: Microsoft Assembler http://www.masm32.com/
• Nasm: The Netwise Assembler http://www.nasm.us/
• Fasm: Flat Assembler http://flatassembler.net/index.php Documentation, FAQ, Exemples...
• Tasm : Turbo Assembler http://info.borland.com/borlandcpp/cppcomp/tasmfact.html
• RosAsm: The Bottom-Up Assembler http://betov.free.fr/RosAsm.htm
• Yasm Modular Assembler: http://www.tortall.net/projects/yasm/

DOS (16-bit) Assembly tutorials

• Mastering Turbo Assembler, 2d edition.
• Adams asm tutorials. http://www.programmersheaven.com/d/click.aspx?ID=F15526

[French]

• Assembleur Pratique de Bernard Fabrot chez Marabout.
• Assembleur x86 par Jean Bernard Emond - Le Tout en Poche.
• Introduction à la programmation en Assembleur Processeurs 64 bits de la famille 80x86.
• Une introduction en Assembleur.
• Apprendre l'assembleur Intel DOS 16bit avec Tasm.
• Initiation à l'assembleur.
• Assembleur i8086.
• Méthodologie de Programmation en Assembleur.
• http://www.haypocalc.com/wiki/Assembleur_Intel_x86
• Nasm Langage Assembleur PC
• http://frenchezines.free.fr/tries/counters...ssembleur1.html
• http://frenchezines.free.fr/tries/counters...ssembleur2.html
• http://edouard.fazenda.free.fr/pub/ftp/Ezines/asmtutorial/
• http://edouard.fazenda.free.fr/pub/ftp/Ezi...asmbeginner.txt

Forum incontournable

• http://fat.next-touch.com/forum/: le forum du site, où règne une excellente ambiance, qui vous aidera lorsque vous rencontrez des problèmes, vous pourrez également y demander des conseils …

• http://www.developpez.net/forums/ : la plus importante communauté francophone dédiée au développement informatique.

- http://www.forumcrack.com/forum/
- http://www.winasm.net/forum/
- http://www.masm32.com/board/
- http://www.asmcommunity.net/

Code Source

• http://win32assembly.online.fr/source.html **Masm32**
• http://www.magma.ca/~wjr/
• http://asmsource.cjb.net/
• http://www.asmfr.com/
• http://www.geocities.com/SiliconValley/Heights/7394/
• http://rs1.szif.hu/~tomcat/win32/ Win32Nasm
• http://pagesperso-orange.fr/vombonjour/myprogs.html
• http://www.team-x.ru/guru-exe/index.php?path=Sources/

Autres cours

- http://www.csc.depauw.edu/~bhoward/asmtut/ **Nasm 16bit**
- http://thsun1.jinr.ru/~alvladim/man/asm.html#5
- http://www.drpaulcarter.com/pcasm/pcasm-book-french-pdf.zip **Nasm 32bit [FR]**
- http://www.xs4all.nl/~smit/asm01001.htm#index1 **Tasm 16bit**
- http://web.cecs.pdx.edu/~bjorn/CS200/linux_tutorial/ **Nasm**
- http://www.btinternet.com/~btketman/tutpage.html **Tasm 16bit**
- http://www.doorknobsoft.com/tutorial/asm-t...rning-assembly/ **Tasm 16bit**.
- http://www.chez.com/asmgges/index.htm **32bit Fasm, Nasm, GoAsm, RosAsm tuts,**
- http://www.asm32.motion-bg.com/
- http://www.osdata.com/topic/language/asm/asmintro.htm
- http://www.jorgon.freeserve.co.uk/
- http://www.int80h.org/bsdasm/ **Nasm 32bit**
- http://burks.bton.ac.uk/burks/language/asm/asmtut/asm1.htm
- http://www.deinmeister.de/wasmtute.htm
- http://www.hochfeiler.it/alvise/index.html
- http://www.geocities.com/emu8086/index.html
- http://www.gentle.it/alvise/w32p.htm
- http://www.geocities.com/technosoftt/
- http://pagesperso-orange.fr/vombonjour/
- http://members.a1.net/ranmasaotome/main.html
- http://games-creators.org/wiki/PureBasic#T...x_ASM_.28x86.29
- http://www.avr-asm-tutorial.net/avr_en/
- http://webster.cs.ucr.edu/Page_TechDocs/asmstyle.html
- http://assembly.ifrance.com/assembly