Prise en main de FASM sous Windows

par dap (Page personelle)

Date de publication : 12 septembre 2008

Dernière mise à jour :

Ce tutoriel est destiné aux programmeurs déjà expérimentés en assembleur x86 qui veulent passer à FASM sous Windows.

I - Pourquoi utiliser FASM ?

Autant le dire tout de suite : il n'y a pas de raison absolue pour préférer FASM à un autre assembleur. On peut quand même parler de ses avantages : FASM est multiplateforme et régulièrement mis à jour, contrairement à MASM et TASM par exemple. Il est aussi adapté à la programmation Windows, ce qui n'est pas le cas de NASM.
Du côté des désavantages on peut citer le fait qu'il n'est pas encore très populaire : je ne connais pas un seul tutoriel ou livre un peu complet qui utilise FASM. C'est pour cette raison que je ne conseillerais pas FASM à un débutant.

Personnellement j'ai longtemps utilisé MASM, puis j'ai décidé d'utiliser un autre assembleur qui serait multiplateforme et encore en développement actif. J'entendais souvent parler de NASM mais il ne me convenait pas à cause de son manque de mises à jour (il ne supportait pas encore le 64 bits) et son préprocesseur rigide. J'en suis finalement arrivé à FASM, qui avait tout ce qu'il me fallait mais semblait difficile à utiliser. La documentation générale est assez bien faite mais bizarrement elle reste très vague sur la création de programmes Windows alors que c'est sans doute la première chose sur laquelle un nouvel arrivant va vouloir se renseigner. Ce tutoriel a pour but d'aider ceux qui sont dans cette situation : après l'avoir lu vous devriez avoir les bases pour apprendre à utiliser FASM uniquement avec la documentation officielle.

II - Passer de MASM/TASM à FASM

Si vous venez de MASM ou TASM (1) il y a une différence de syntaxe avec FASM qui risque de vous poser des problèmes de compréhension : l'adressage de la mémoire se fait d'une façon qui peut paraitre tordue au premier abord. La définition d'une variable s'écrit comme avant :

```
var  dd  5  ; un DWORD initialisé à 5
```

Cependant lorsque vous utiliserez l'identifiant *var* FASM le remplacera par l'**adresse** de var, pas son contenu. Ce qui est intéressant à partir de là c'est que les crochets [] deviennent un opérateur simple à comprendre : ils servent à accéder à une donnée en mémoire. L'adresse de cette donnée est fournie entre les crochets. Avec MASM les crochets ne sont pas nécessaires pour faire un accès mémoire.
Concrètement :

```
mov eax, var    ; copie l'adresse de var dans EAX
mov eax, [var]    ; crochets => accès mémoire
                  ;             l'adresse du DWORD lu est celui de var, donc on
                  ;             copie dans EAX le contenu de var (5)

mov eax, [ebx+ecx*4]  ; crochets => accès mémoire
                      ;             l'adresse du DWORD lu est obtenue en
                      ;             calculant ebx + ecx*4
                      ; (c'est la même syntaxe qu'avec MASM ou TASM)
```

En comparaison la syntaxe MASM aurait donné :

```
mov eax, offset var
mov eax, var  ; mov eax, [var] est équivalent, mais c'est peu utilisé
mov eax, [ebx+ecx*4]
```

Il y a bien sûr d'autres différences mais la plupart ne sont pas vraiment importantes. Les instructions elles-mêmes s'écrivent le plus souvent de la même façon, il ne vous reste donc plus qu'à apprendre les directives utilisées pour créer les exécutables. C'est ce qu'on va voir dans la suite de ce tutoriel.

(1) TASM est surtout connu pour son mode de compatibilité avec MASM qui est utilisé par défaut, mais il supporte aussi un "ideal mode" où la syntaxe ressemble à celle de NASM/FASM.

III - Installer FASM

Téléchargez la version Windows ici : **http://flatassembler.net/download.php**.
Vous obtenez un .zip qui doit être décompressé dans le répertoire où vous voulez l'installer (pour l'exemple ce sera C:\FASM). À l'intérieur vous trouverez deux fichiers qui nous intéressent :

- FASM.EXE, utilisé pour lancer FASM en ligne de commande.
- FASMW.EXE, un IDE très minimaliste.

Ouvrez une console (Démarrer> Exécuter> Tapez "cmd" puis ENTER) et tapez simplement "fasm". Vous devriez obtenir une erreur dans ce genre-là :

```
'fasm' n'est pas reconnu en tant que commande interne ou externe, un programme exécutable ou un fichier
 de commandes.
```

C'est parce que vous devez d'abord ajouter le chemin de FASM.EXE à la variable d'environnement Path. Voilà le moyen le plus simple : clic droit sur Poste de travail> Propriétés> Avancé> Variables d'environnement> Cherchez la variable Path et cliquez sur Modifier. Ajoutez quelque part le chemin de FASM.EXE (par exemple C:\FASM) en le séparant des autres variables d'environnement avec des points-virgule. Si vous réessayez de taper "fasm" vous devriez maintenant avoir droit à un message d'erreur (il faut peut-être que vous ouvriez une nouvelle console pour que la modification de Path entre en compte) :

```
flat assembler  version 1.67.26
usage: fasm source [output]
```

Une autre solution est de taper en ligne de commande "path=C:\FASM;%Path%". Ça fonctionnera mais seulement jusqu'à ce que vous fermiez la console.

IV - Appeler FASM en ligne de commande

Créez un fichier de test, par exemple test.asm, et placez dedans un code de test comme "mov eax, [ebx]". Pour le compiler, tapez "fasm test.asm". Oui, c'est tout. Vous n'avez donné aucune information sur le format, donc FASM crée un fichier test.bin qui contient les instructions telles qu'elles, prévues pour tourner en mode 16 bits. Si vous voulez changer le nom du fichier de sortie vous devez donner un deuxième paramètre, ce qui fera par exemple "fasm test.asm test.toto". Évidemment l'intérêt de ces fichiers "binaires" est limité, donc on va voir quelques généralités sur les exécutables de Windows puis comment en créer avec FASM.

V - Aperçu d'un fichier PE

Sous Windows les exécutables suivent le format PE (Portable Executable). Normalement on n'a pas à se soucier de ça, mais utiliser FASM nécessite de savoir un peu comment ça fonctionne, et en contrepartie vous permet de contrôler facilement ce qui se trouve dans vos exécutables. Il existe en fait deux headers (win32ax.inc et win32wx.inc) qui fournissent des macros pour simplifier la tâche mais elles sont limitées et n'utiliser qu'elles vous priverait d'apprendre des choses intéressantes sur le fonctionnement de vos programmes.
Un fichier PE ressemble à ça :

```
 _____________________________
|           Entêtes           |
|-----------------------------|
| Section .text               |
|  (code)                     |
|-----------------------------|
| Section .data               |
```

```
|  (données)                  |
|-----------------------------|
| Section .rdata              |
|  (données en lecture seule) |
|-----------------------------|
| Section .idata              |
|  (imports des DLL)          |
|_____________________________|
```

Les entêtes sont créés automatiquement par FASM et donnent des informations générales sur l'exécutable.
Les sections fournissent le reste d'informations nécessaires à l'OS pour charger et exécuter le programme. Chaque section peut avoir différentes propriétés, c'est pour qu'on n'en utilise pas qu'une seule. Ici par exemple nous avons :

- .text contient le code et est donc exécutable.
- .data contient les données générales et est donc accessible en lecture comme en écriture.
- .rdata contient les données en lecture seule et est donc seulement accessible en lecture. Ça vous permet de détecter quand vous modifiez une variable globale qui est censée rester constante.
- .idata contient les imports des fonctions qu'on utilisera, comme printf(). Retenez simplement qu'au chargement de l'exécutable un tableau sera rempli avec les adresses des fonctions importées, et qu'il se trouve dans .idata.

Remarquez que les noms sont arbitraires et que vous pouvez très bien ajouter d'autres sections.

Quand vous assemblerez votre premier programme Windows vous serez peut-être surpris par sa taille (par exemple 1 Kio alors qu'il n'a l'air de nécessiter que quelques octets). FASM crée des sections dont la taille en octets est un multiple de 512 (c'est le plus petit alignement autorisé), sans doute parce que c'est la taille d'un secteur sur le disque. Si par exemple vous avez deux sections de 200 Kio, elles pourraient bien sûr tenir sur 400 Kio mais en les alignant Windows peut charger chacune des sections en lisant un seul secteur. Si elles avaient été comprimées sur 400 Kio il faudrait lire le secteur correspondant, puis filtrer ce qui appartient à la section voulue.
À l'exécution chaque section utilisée se retrouvera dans au moins une page mémoire de 4 Kio. Chaque page aura des caractéristiques qui dépendent de la section d'où elle vient. Il n'est pas forcément possible pour Windows de contrôler l'exécution des pages. Il faut d'abord que le processeur supporte le **NX bit**, et même si c'est le cas **ce n'est pas encore gagné**.

VI - Un exécutable minimal

```
format PE console
entry start

section '.text' code executable
start:
        xor eax, eax
        ret
```

Pour créer l'exécutable, tapez la même chose qu'avant : fasm fichier.asm. Ensuite exécutez "fichier.exe" en tapant fichier dans la console. La procédure sera la même pour compiler et lancer les programmes qui vont suivre.
La première ligne indique à FASM le format du fichier de sortie. Avec d'autres assembleurs on aurait donné cette information en ligne de commande mais l'auteur de FASM suit le principe "SSSO" (same source, same ouput), qui veut qu'un maximum d'options soient présent dans le fichier source lui-même pour simplifier la vie des gens qui l'assembleront.
La deuxième ligne donne le point d'entrée du programme, c'est-à-dire la première instruction qui sera exécutée. Ici elle n'est pas nécessaire mais elle permet d'être sûr de ne pas créer des bugs si vous ajoutez une section avec .text.
Ensuite nous créons une section qui s'appelle .text et qui a plusieurs attributs ou "flags". Je commence toujours par préciser si la section contient du code ou des données, .text tombe logiquement dans la première catégorie. On ajoute aussi le flag executable, ça vaut mieux.
L'étiquette "start" indique le point d'entrée du programme, comme expliqué plus haut.

Les deux dernières instructions correspondent à un return 0; en C. Elles paraissent logiques si on pense être dans la fonction main() ou WinMain(), qui est le point d'entrée d'un programme C. Ça fonctionne très bien chez moi (sous Windows XP SP2), mais c'est un comportement qui est (à ma connaissance) non documenté. Il vaudrait mieux utiliser une fonction Windows prévue pour ça, ce qui nous amène tout droit à la section suivante.

VI - Importer une fonction de DLL

Les macros que nous allons utiliser nécessitent d'utiliser un des headers fournis avec la version Windows de FASM. Les plus basiques sont win32a.inc et win32w.inc. Le premier est prévu pour l'encodage ANSI et l'autre pour Unicode (le "w" vient de "wide character"). Ensuite viennent win32ax.inc/win32wx.inc qui fournissent des macros plus avancées, et win32axp.inc/win32wxp.inc qui ajoutent la vérification du nombre d'arguments lors des appels de fonctions.
Pour ce que nous voulons faire (importer des fonctions et utiliser des chaînes ANSI), win32a.inc suffira largement.
Voilà à quoi ressemble le nouveau code :

```
format PE console
entry start

include 'win32a.inc'

section '.text' code executable
start:
        push 0
        call [ExitProcess]

section '.idata' data readable import
        library kernel32, 'kernel32.dll'
        import kernel32, ExitProcess, 'ExitProcess'
```

Tout d'abord quelques mots sur la fonction ExitProcess. Vous pouvez trouver sa documentation ici : **http://msdn.microsoft.com/en-us/library/ms682658.aspx**. Une fois que vous avez compris son fonctionnement, vous pouvez voir vers la fin à quelle DLL elle appartient :

Requirements

Client	Requires Windows Vista, Windows XP, or Windows 2000 Professional.
Server	Requires Windows Server 2008, Windows Server 2003, or Windows 2000 Server.
Header	Declared in Winbase.h; include Windows.h.
Library	Use Kernel32.lib.
DLL	Requires Kernel32.dll.

La directive include copie dans le fichier source le contenu de win32a.inc. Si vous ne voulez pas préciser à chaque fois l'adresse complète de ce fichier, la solution est de créer ou modifier une variable d'environnement Include dont la valeur sera un ou plusieurs répertoires où aller chercher les fichiers inclus. Le démarche est la même que pour la variable d'environnement Path vue dans la section III.

Le code de la section .text appelle simplement la fonction ExitProcess() en utilisant la convention stdcall (elle est normalement utilisée par toutes les fonctions Windows, à part celles qui implémentent les fonctions du C comme

printf() ou getchar()). La valeur passée en argument est zéro, ce qui veut dire que le programme se termine normalement. Ensuite la fonction ExitProcess() termine brutalement le processus.

Les crochets autour du label ExitProcess vous ont peut-être surpris. En fait ce label pointe sur une entrée dans la section .idata qui contient l'adresse de la fonction ExitProcess() (qui aura été mise là par le loader au chargement du programme). La valeur du label ExitProcess est donc une adresse quelque part dans notre propre programme, ce n'est pas ce que nous voulons ! Il faut accéder à la valeur pointée par ce label, c'est-à-dire [ExitProcess].

La section .idata est notre première section de données, d'où le flag data. Le flag suivant (readable) n'est pas nécessaire, mais je trouve plus logique de le mettre. Enfin import est nécessaire puisqu'il indique que les adresses des fonctions importées doivent être écrites dans la section.
Les dernières lignes utilisent deux macros contenues dans win32a.inc : library et import.
library sert à spécifier les DLL d'où vous allez importer. Pour chaque DLL vous devez ajouter un couple d'arguments, ce qui donne dans notre cas deux arguments puisque nous n'importons qu'une fonction depuis kernel32.dll. Le premier argument est le nom d'un label qui sera utilisé dans la macro suivante, et le deuxième une chaîne qui contient le nom exact de la DLL. Ça peut paraitre répétitif d'indiquer deux fois à peu près a même chose, mais le premier label peut avoir le nom que vous voulez (ce qui n'est pas non plus super utile si vous voulez mon avis).
La macro import doit être utilisée une fois pour chaque DLL que vous allez importer. Le premier argument est le label que vous avez associé au nom de la DLL avec library. Suivent une série de couples d'arguments du même style que ceux de library : le nom d'un label qui sera associé à la fonction importée, suivi d'une chaîne contenant le nom exact de cette fonction. Le nom du label est ici aussi personnalisable, par exemple vous auriez pu choisir Exit au lieu de ExitProcess si vous préférez taper moins de caractères.

VIII - Utiliser les fonctions du langage C

Les fonctions Windows permettent d'afficher du texte dans la console mais elles sont trop minimalistes pour être utilisées régulièrement. Heureusement Microsoft fournit un runtime C qui permet d'utiliser des fonctions beaucoup plus pratiques comme printf().
Nous allons utiliser msvcrt.dll. Cette DLL est idéale pour nous parce qu'elle existe sur tous les Windows et qu'elle aussi simple à utiliser que kernel32.dll. Il est quand même important de noter qu'un programme sérieux devrait utiliser une des versions mises à jour qui se trouvent quelque part dans WINDOWS\WinSxS (si vous avez un Visual C++ ou le **Visual C++ Redistributable Package**). Pour utiliser une de ces DLL dans votre programme assembleur vous devez ajouter un fichier XML dans une section de ressources, ce qui nous mènerait un poil hors du sujet et n'est pas nécessaire pour un programme de test comme celui-ci.
Voilà une nouvelle version du programme précédent qui affiche un message :

```
format PE console
entry start

include 'win32a.inc'

section '.text' code executable
start:
        push salut
        call [printf]
        pop ecx

        push 0
        call [ExitProcess]

section '.rdata' data readable
        salut db 'Salut tout le monde !', 10, 0

section '.idata' data readable import
        library kernel32, 'kernel32.dll', \
                msvcrt,   'msvcrt.dll'
        import kernel32, ExitProcess, 'ExitProcess'
        import msvcrt, printf, 'printf'
```

Il y a trois nouveautés à noter.

D'abord l'appel à printf() est suivi d'un pop : c'est parce que les fonctions du C ne suivent pas la convention stdcall, ce qui veut dire qu'elles laissent les arguments sur la pile. Vous pouvez enlever les arguments de la pile simplement en les resortant dans un registre libre comme ici, ou en ajoutant à ESP le nombre d'octets qui ont été poussés avant la fonction (dans notre cas ça aurait donné ADD ESP, 4).

Ensuite la nouvelle section .rdata qui va servir pour toutes les variables globales en lecture seule. La chaîne de caractères pointée par le label "salut" va être utilisée par une fonction qui vient du C, il est donc important qu'elle soit terminée par un zéro comme toutes les chaînes en C. Le 10 est le code du saut de ligne. Sous Windows on utilise traditionnellement les codes 13 et 10 mais le programmeur C doit être capable de manipuler le saut de ligne comme un seul caractère ('\n'). Le runtime s'occupe donc de faire la conversion : si avec un programme C vous écrivez le caractère '\n' dans un fichier et que vous l'ouvrez avec un éditeur hexadécimal, vous verrez que deux codes ont été écrits : 13 et 10 (D et A en hexadécimal).

La fonction printf() est importée de msvcrt.dll de la même façon qu'avec ExitProcess(). Pour aérer un peu le code il est courant d'ajouter des sauts de ligne entre les couples de DLL et de fonctions mais library et import n'ont pas été prévues pour ça : la solution est d'ajouter un \ avant le saut de ligne pour dire à FASM de l'ignorer.

IX - Astuces en tout genre

En voyant la tronche de la section d'imports vous vous êtes sûrement dit que ça devait être fastidieux d'ajouter une ligne pour chaque nouvelle fonction. Bonne nouvelle : il existe des fichiers que vous pouvez inclure pour importer automatiquement les fonctions que vous utilisez dans votre programme. Mauvaise nouvelle : il n'en existe pas encore pour msvcrt.dll.
Voilà à quoi ressemble notre section d'imports maintenant :

```
section '.idata' data readable import
        library kernel32, 'kernel32.dll', \
                msvcrt,   'msvcrt.dll'
        include 'api\kernel32.inc'
        import msvcrt, printf, 'printf'
```

Le fichier inclu contient en fait la même chose que ce que vous auriez tapé si vous aviez voulu importer toutes les fonctions de kernel32.dll. Ce n'est pas grave si le code source contient énormément d'imports, ils ne seront ajoutés à l'exécutable que si ils sont vraiment utilisés.

Quatre macros existent pour écrire les appels de fonction en une seule ligne. stdcall permet d'appeler les fonctions Windows un peu comme en C :

```
stdcall [MessageBox],0,szText,szCaption,MB_OK
```

ccall fait la même chose à part qu'elle est prévue pour nettoyer automatiquement les arguments laissés sur la pile par les fonctions du C.
invoke et cinvoke sont équivalentes aux deux macros précédentes mais elles vous évitent de devoir entourer de crochets le label de la fonction.

Vous devez aussi savoir que les fonctions Windows qui utilisent des chaînes de caractères existent en deux versions : une pour les chaînes ANSI et une pour Unicode. Pour les différencier on ajoute un suffixe à leur nom : par exemple  **WriteConsole()** n'existe pas vraiment dans kernel32.dll, vous devez importer WriteConsoleA (ANSI) ou WriteConsoleW (Wide/Unicode). C'est indiqué ici dans la page de documentation :

Requirements

Client	Requires Windows Vista, Windows XP, or Windows 2000 Professional.
Server	Requires Windows Server 2008, Windows Server 2003, or Windows 2000 Server.
Header	Declared in Wincon.h; include Windows.h.
Library	Use Kernel32.lib.
DLL	Requires Kernel32.dll.
Unicode/ANSI	Implemented as **WriteConsoleW** (Unicode) and **WriteConsoleA** (ANSI).

X - Et maintenant ?

Vous pouvez lire la **documentation de FASM** pour en apprendre plus sur ses fonctionnalités avancées et le langage assembleur. Si vous n'avez pas compris certaines choses dans ce tutoriel vous trouverez sans doute les réponses à vos questions dans le **tutoriel de Paul Carter**, sinon n'hésitez pas me contacter.

Si vous êtes intéressé par le format PE voilà tout ce qu'il faut pour le comprendre :

An In-Depth Look into the Win32 Portable Executable File Format

An In-Depth Look into the Win32 Portable Executable File Format, Part 2

Microsoft Portable Executable and Common Object File Format Specification