Jeremy Gordon

avec la collaboration de *Wayne J. Radburn*

Assembleur 32/64 bits

GoAsm Version 0.61

A "Go" development tool: http://www.GoDevTool.com

Contacts :
Jeremy Gordon – **jg@jgnet.co.uk**
GoAsm Assembler and Tools forum
(dans le Forum MASM)

Volume 1

Traduction française
Robert Cordonnier – Orléans
(alias Asmou sur www.developpez.net)
Version 1.8.3D – septembre 2017
robert.cordonnier@wanadoo.fr

Sommaire

Introduction

Ce document constitue la première partie de la traduction française intégrale du manuel de l'assembleur 32/64 bits GoAsm développé par Jeremy Gordon avec la collaboration de Wayne J. Radburn. Outre l'assembleur proprement dit, il propose de nombreuses annexes à destination des débutants ainsi que quelques exemples de programmation. Le volume 2 s'intéresse aux autres outils Go, notamment l'éditeur de liens GoLink, le compilateur de ressources GoRC et le débogueur GoBug et propose un panorama de ce qu'il convient de connaitre en matière de DLL et de standard Unicode.

1 Comment utiliser ce manuel

Si vous voulez savoir **pourquoi j'ai écrit** GoAsm, connaître les **aspects juridiques** et les conditions de **licence** relatifs à ce produit, la suite de cette introduction vous est destinée.

Si vous voulez un aperçu de certaines des caractéristiques de GoAsm, alors cliquez **ici**.

Si vous **débutez** et que vous voulez apprendre comment faire un programme Windows simple, cliquez **ici**.

Si vous souhaitez voir quelques exemples de code GoAsm pour **plateforme 32 bits**, activez les liens suivants :

- **HelloWorld1.asm** : programme de console Windows 32 bits (voir également **ici**)**,**
- **HelloWorld2.asm** : programme pour Windows GDI 32 bits dessinant une ellipse dans une fenêtre,
- **HelloWorld3.asm** : version plus élaborée de HelloWorld2.asm avec usage intensif de trames de pile, de structures, de variables locales, INVOKE et de définitions (macros),
- **HelloDialog.asm** : dialogue utilisant la fonction DialogBoxIndirectParam avec création de contrôles par modèle interne (tables décrivant les contrôles),

Si vous souhaitez voir quelques exemples de code GoAsm pour **plateforme 64 bits**, activez les liens suivants :

- **Hello64World1.asm** : programme de console Windows 64 bits**,**
- **Hello64World2.asm** : programme Windows 64 bits dessinant une ellipse dans une fenêtre**,**
- **Hello64World3.asm** : programme Windows dessinant une ellipse dans une fenêtre avec commutation de compilation sur plateforme 32 ou 64 bits**.**

Les **programmes Unicode** ainsi que certains aspects de programmation afférents ont été intégrés dans un document séparé qui constitue le volume 2.
Si vous voulez en savoir plus sur les lignes directrices qui structurent GoAsm, alors cliquez **ici**.
Si vous êtes simplement intéressé par la façon d'utiliser GoAsm, alors cliquez **ici** pour acquérir les bases de cet assembleur, **ici** pour en connaître ses fonctionnalités avancées ou **ici** pour découvrir les points divers – mais néanmoins importants – le concernant.
Cliquez enfin **ici** si vous désirez tout connaitre sur la programmation en 64 bits permise par cet assembleur.

Si vous débutez en assembleur

Bienvenue aux joies de la programmation assembleur ! Ecrivez des programmes de travail rapides et compacts. L'assembleur fonctionne très bien avec Windows. Et, s'il est vrai que nous sommes en présence d'un langage de bas niveau, il n'en demeure pas moins que l'API Windows (Applications Programming Interface) lui adjoint des fonctionnalités de très haut niveau. Les deux sont parfaitement compatibles aussi bien en 64 bits qu'en 32 bits. Ce document vous aidera à appréhender la programmation en assembleur. Consultez plus particulièrement, dans votre parcours initiatique, le **chapitre 2** et les **annexes**. On lira enfin avec le plus grand intérêt les tutoriels qui n'auraient pas fait l'objet de traduction et qui figurent sur le site **http://www.godevtool.com/**.

2 En quoi un nouvel assembleur est-il nécessaire ?

Il existe un certain nombre d'assembleurs sur le marché tels que le très populaire MASM de Microsoft, NASM (issu d'une équipe dirigée à l'origine par Simon Tatham et Julian Hall), TASM de Borland et enfin, A386 de Eric Isaacson. De mon point de vue, aucun de ces assembleurs ne peut être considéré comme parfait dans le cadre de la programmation Windows. Certains ont même des défauts gênants. En écrivant GoAsm je me suis efforcé de construire un assembleur qui produise toujours un code de taille minimale avec une syntaxe claire et évidente, qui n'impose que de faibles exigences au niveau du script source et propose des extensions pour aider à la programmation en Win32 et Win64. Cela m'a également donné l'occasion d'écrire l'éditeur de liens GoLink, qui est finement réglé pour travailler avec GoAsm.
D'autres que moi ont également essayé d'engager une démarche similaire, notamment René Tournois qui a écrit le fabriquant d'exécutable Spasm (maintenant appelé RosAsm) et Tomasz Grysztar avec son assembleur flat (FASM).

3 Versions et mises à jour

Mon intention est de préserver GoAsm de tout bug connu. Donc, je travaille habituellement sur des corrections de bugs dès que je les découvre (à moins d'être en vacances). Je produis généralement un correctif à destination de ceux qui signalent des bugs en leur envoyant (ou en postant) une copie de GoAsm avec un numéro de version affecté d'une lettre suffixe. Les bugs relativement mineurs sont généralement traités de cette façon et puis donnent finalement lieu à la publication d'une mise à jour formelle. Ces mises à jour peuvent être obtenues à partir de mon site Web à l'adresse **http://www.godevtool.com/**. Un bug grave peut entraîner la publication immédiate d'une mise à jour de GoAsm. Je travaille également à son amélioration de temps en temps : cela se traduit par la mise à disposition d'une version bêta de GoAsm qui est disponible pour tests. Ces versions d'essai sont souvent également disponibles à partir de mon site web. C'est n'est seulement qu'à l'issue de ces tests et des éventuelles modifications induites que les versions bêta se transforment en mise à jour officielle.

4 Forum de discussion

Il existe un forum consacré à l'**assembleur GoAsm et ses outils** à l'intérieur du forum MASM géré par Hutch. Vous pouvez y exprimer vos idées sur les outils "Go", me poser des questions ou faire de même avec d'autres utilisateurs et vérifier les mises à jour. Le forum est aussi l'occasion pour moi de vous consulter sur les améliorations à apporter à GoAsm et aux autres outils "Go".

5 Environnements de Développement Intégré (IDEs)

Les IDEs sont des éditeurs qui vous aident à utiliser la syntaxe de programmation correcte, puis à exécuter les outils de développement en vue de créer les fichiers de sortie. En voici quelques uns :
Easy Code pour GoAsm : excellent IDE de Visual Assembler écrit par Ramon Sala.
Téléchargez ECGo.zip sur le site **http://www.godevtool.com/** – incluant par ailleurs les versions les plus récentes des outils "Go" et des fichiers d'inclusion pour l'utilisation de Easy Code – 792K.
Les tutoriels de Bill Aitken pour l'utilisation de GoAsm et de l'IDE.
RadAsm : excellent IDE de Visual Assembler pour Windows conçu par Ketil Olsen.
Vous pouvez aller sur Donkey's stable pour Radasm, les fichiers d'inclusion, les macros, des exemples et projets GoAsm.
NaGoa : Visual Assembler (utilisant GoRC seulement).

6 Aspects juridiques

Copyright

GoAsm - licence et distribution

Vous pouvez utiliser GoAsm à toutes fins, y compris des programmes commerciaux. Vous pouvez le redistribuer librement (mais sans contrepartie financière, ni l'utilisation avec un programme ou tout autre maté-

riau pour lequel l'utilisateur est invité à payer). Vous n'êtes pas habilités à masquer ou à contester mes droits d'auteur.

Avertissement

J'ai fait tous les efforts possibles pour faire en sorte que GoAsm et de son programme d'accompagnement AdaptAsm soient au point, mais vous les utilisez entièrement à vos risques. Je ne peux accepter la moindre responsabilité concernant leur fonctionnement, le travail produit, ni les conséquences d'erreurs entachant éventuellement ce manuel.

7 Remerciements

Je dois des remerciements particuliers à Wayne J. Radburn, de Gatineau, au Québec, qui a entrepris et conduit avec succès tout récemment un ensemble d'améliorations et de corrections de bugs dans les versions les plus récentes de GoAsm (0,57 à 0,61). Je voudrais également remercier Edgar Hansen de Kelowna, en Colombie-Britannique, Canada («Donkey») pour son soutien continu et ses encouragements à Wayne et moi-même et, d'une manière générale, remercier tous les utilisateurs de GoAsm. Nous sommes trois, désormais, à détenir le code source de GoAsm et de GoLink, et cela contribuera à garantir l'avenir du projet "Go". Je suis également très reconnaissant à toutes ces autres personnes qui m'ont encouragé à écrire ces programmes et m'ont éclairé par d'utiles commentaires, des rapports et des conseils avisés. Je me dois de citer, en particulier :
Leland M. George de West Virginia, **Daniel Fazekas** de Budapest, **Greg Heller** du Congo ("Bushpilot"), **René Tournois** de Louisville, Meuse, France ("Betov"), **Ramon Sala** de Barcelone, Espagne, **Bryant Keller** de Cartersville, Géorgie, **Emmanuel Zacharakis** (Manos), et **Brian Warburton** de Weybridge, au Royaume-Uni.
Merci aussi pour le soutien, les suggestions et les rapports de bogues de grv, Jeff Aguilon, Jonne Ahner, Thomas Hartinger, Martyn Joyce, Kazó Csaba, Dmitry Ilyin, Patrick Ruiz, et de tous les contributeurs du **forum GoAsm et outils associés**, ainsi que d'autres forums que j'aurais omis de mentionner ici.

Chapitre 1
Les concepts de GoAsm

1.1 Les caractéristiques de GoAsm en bref

- GoAsm est un assembleur 32 bits pour les processeurs 86 et Pentium et un assembleur 64 bits pour les processeurs AMD64 et EM64T.
- GoAsm produit un fichier objet dans le format Portable Executable COFF approprié pour un éditeur de liens tel que GoLink ou ALINK. Le format COFF est de loin supérieur à l'OMF (Module Object Format) produit par certains assembleurs plus anciens parce que, dans le format OMF, la taille des fichiers objets est limitée à environ 55K. Dans tout projet d'envergure vous vous situez au-delà de cette limite.
- GoAsm fonctionne seulement **en mode flat**. Cela signifie qu'il n'y a pas de segmentation du code et des données. Cela rend le script source beaucoup plus propre et plus facile à écrire. Fondamentalement, dans GoAsm, vous pouvez déclarer la section, puis commencer à coder. En programmation 32 bits, vous pouvez utiliser les registres 32 bits pour y entreposer et manipuler des données (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP) et aussi leurs subdivisions 8 bits et 16 bits (AL, AH, BL, BH, CL, CH, DL, DH et AX, BX, CX, DX, SI, DI, BP et SP). Vous pouvez adresser des données de toute taille dans la mémoire, mais, dans la mesure où GoAsm fonctionne uniquement en mode flat vous ne pouvez utiliser que des adresses 32 bits pour ce faire. Il en résulte que vous ne pouvez pas utiliser, par exemple, des instructions telles que `ADC W[BX], 6` ou `MOV AX,[SI]` pour traiter des données en mémoire ; vous devez impérativement leur préférer `ADC W[EBX], 6` ou `MOV AX,[ESI]`.
- GoAsm a un certain nombre de fonctionnalités pour vous aider à écrire des programmes Unicode ou pour utiliser le même script source pour des programmes Unicode et ANSI. GoAsm peut lire les fichiers Unicode (UTF-16 ou format UTF-8) et peut recevoir ses commandes et produire sa sortie en Unicode. Voir le support Unicode pour un aperçu et l'écriture de programmes Unicode pour plus de détails.
- GoAsm fonctionne également comme **assembleur 64 bits**. Bien que le code exécutable en 64 bits soit tout à fait différent, le code source est très similaire et se révèle tout aussi facile à écrire. Vous pouvez même, à partir d'un même code source, produire des exécutables en 32 ou 64 bits à l'aide d'un commutateur approprié. Voir le chapitre 6 relatif à l'écriture de programmes 64 bits pour plus de détails.
- GoAsm prend en charge tous les mnémoniques standard (autres que ceux utilisés uniquement pour la programmation 16 bits), les instructions en virgule flottante x87, MMX, 3DNow! (avec les extensions), les instructions SSE, SSE2, SSE3 et SSSE4, ainsi que AES (cryptage selon l'algorithme de Rijndael), ADX, et quelques autres instructions nouvelles diverses. Voir à ce sujet **mnémoniques pris en charge** et **syntaxe des registres FPU, MMX et XMM**.
- J'ai essayé de prendre le meilleur de la syntaxe des assembleurs utilisée en général et du "C", de mon point de vue. Voir la section **Syntaxe et compatibilité avec d'autres assembleurs** pour plus de détails.
- Plus de flexibilité et de simplicité sont obtenues en renonçant au contrôle du type de variable ou des paramètres d'API. **Voir mon explication pour cette décision**.
- Tous les labels (sauf ceux réutilisables) sont supposés être **global** et **public**, en ce sens qu'ils sont accessibles à d'autres fichiers sources (via l'éditeur de liens). Ceci est très simplement réalisé en utilisant un flag dans le fichier objet et évite la nécessité de déclarer de tel ou tel label en GLOBAL et PUBLIC. D'où une grande économie de temps et d'effort au bénéfice du programmeur ! (Les labels réutilisables dans le cadre des sauts de code courts sont traités différemment).
- Pour des raisons de certitude et de clarté dans votre script, les **crochets** sont obligatoires pour l'écriture et la lecture de mémoire. Voir **mon explication de ce choix**.
- GoAsm fournit un moyen très simple de **mettre en pile (PUSH) le pointeur d'une chaîne terminée par un zéro** pour les appels d'API. Vous pouvez également **pousser en pile le pointeur vers des données brutes ordinaires**. Voir plus d'informations à ce sujet.

- Vous pouvez également charger les pointeurs chaînes terminées par 0 et des pointeurs vers les données brutes dans les registres de la même manière. Par exemple **MOV EAX, ADDR 'Bonjour'**. Voir le **paragraphe 3.11** sur ce point.
- Contrairement à MASM, GoAsm **ne renverse pas l'ordre de mémorisation** des caractères de valeurs immédiates. Par exemple, la saisie de MOV EAX, 'The ' à destination de GoAsm doit s'écrire MOV EAX, ' ehT' avec MASM. La première syntaxe offre une bien meilleure lisibilité du code source. Elle est également plus cohérente avec les chaînes de caractères d'octets encadrées de guillemets, qui sont toujours chargées caractère par caractère. En théorie, il est discutable qu'un assembleur inverse l'ordre de mémorisation. La raison en est que le processeur, du fait de sa structure, inverse l'ordre des octets chargés dans un registre lorsque ceux-ci proviennent de la mémoire, cette inversion se produisant naturellement dans le sens registre vers mémoire. On voit bien que ces 2 inversions s'annulent dans les faits d'où l'idée qu'il serait imprudent que l'assembleur ne casse cette logique au détriment de la lisibilité du code-source. NASM avait pris le contre-pied sur cette question et GoAsm s'est rallié à ce choix. Penchez-vous sur cette pratique respectivement pour le **code** et pour les **données**. Voir aussi le mécanisme de **mémorisation inversée**.
- GoAsm offre un système flexible et pratique en matière de **labels de code**. Je crois que cela est extrêmement important. GoAsm offre 3 possibilités : des labels **uniques** («globaux»), labels **réutilisables de portée locale** ainsi que des labels **réutilisables de portée non-limitée**.
- A ce système de label de GoAsm, s'ajoute un système de notation très explicite pour les **sauts de code courts et longs**.
- Les appels de type "C" sont disponibles en utilisant **INVOKE**.
- GoAsm vous permet d'appeler une fonction dans une **bibliothèque de code statique** et d'en charger le code et les données directement dans le fichier de sortie au moment de l'assemblage.
- GoAsm et le linker GoLink sont les seuls fournissant le moyen d'appeler une fonction dans un autre exécutable **directement par ordinal**, en utilisant une syntaxe simple telle que, par exemple : CALL MyDll:6. Et vous pouvez importer des pointeurs de données sans aucune formalité.
- GoAsm vous permet de spécifier les **EXPORTS** dans votre fichier source. Vous pouvez utiliser le nom seulement, spécifiez une valeur ordinale, et vous assurer qu'aucun nom n'apparaît dans l'exécutable final si vous le souhaitez.
- Lors de l'édition des liens des fichiers objets GoAsm, GoLink est en mesure d'identifier les labels de code et de données qui n'auraient pas été utilisés ou référencés. En utilisant cette fonctionnalité, vous pouvez facilement repérer **ces déclarations de données et zones de code redondants** dans votre programme.
- Les instructions PUSH, POP, ARG, INC et DEC peuvent respectivement être **répétées sur plusieurs opérandes successifs** en séparant ces derniers par des virgules.
- L'utilisation de **FLAGS** comme opérande de PUSH, POP et ARG, et INVOKE et USES.
- **sauvegarde et restauration automatiques des registres et des flags** à l'aide de l'**instruction USES**.
- Pour simplifier les procédures de callback de Windows, GoAsm fournit une **structure automatisée de trame de pile** utilisant FRAME ... ENDF. Des sous-routines peuvent partager les données stockées sur la pile en utilisant USEDATA ... ENDU. Les données locales peuvent être déclarées *dynamiquement* sur une base de message spécifique.
- **Simplification de la forme de l'indicateur de type**. Par exemple. D au lieu de DWORD PTR.
- GoAsm fournit un support complet pour les **structures** et **unions**. Les membres de structures d'unions sont traitées comme des labels de plein droit afin qu'ils puissent être traités en utilisant un point et apparaissent comme des symboles de débogage.
- GoAsm fournit également un support complet pour les **equates**, les **macros** et les **définitions** y compris les définitions de portée limitée.
- GoAsm fournit un support complet pour les inclusions de fichiers (**include**), et vous pouvez également charger un fichier directement dans une section GoAsm en utilisant **INCBIN**.
- Au lieu d'utiliser INCBIN vous pouvez charger des **blocs de données** déclarés dans le fichier source lui-même en utilisant **DATABLOCK**.
- GoAsm est sensible à la casse des caractères (majuscules/minuscules) dans le cas des noms de labels et des noms de définition. Cette fonctionnalité n'est pas escamotable car, permettre l'utilisation de la-

bels de casse mixte peut induire de la confusion. Dans toutes les autres situations GoAsm n'est pas sensible à la casse. Ainsi, par exemple, vous pouvez écrire indifféremment MOV [ESI], EAX, #INCLUDE, #IF, #DEFINE, STRUCT, DB, ou mov [ESI], eax, #include, #if, #define, struct, db.

- **GoAsm ne supporte pas les commandes de run-time "if"**. MASM vous permet de tester les conditions et/ou des instructions de répétition dans une boucle utilisant les commandes .IF / .ELSE / .ELSEIF / .ENDIF et .WHILE / .BREAK. Celles-ci testent les conditions à l'exécution. L'assembleur A386 offre la forme #IF pour cette commande, laquelle teste les flags au moment de l'exécution. Cependant, dans MASM, les séries de commandes IF / ELSE / ELSEIF / ENDIF / IFDEF sont utilisées au moment du processus d'assemblage pour structurer le code objet en fonction des paramètres testés. A386 utilise la forme #IF pour ce faire, tandis que NASM lui préfère %IF. La syntaxe "C" est #IF pour les tests de compilation. Pour ma part, ayant utilisé toutes ces syntaxes pour les tests d'exécution qui sont si semblables à la syntaxe établie tout en signifiant quelque chose de complètement différent, j'affirme que c'est une excellente recette pour un désastre assuré. Pour le moment, j'ai décidé de ne pas soutenir toute forme de tests d'exécution ou en boucle. Je suis prêt à reconsidérer cette décision radicale si quelqu'un peut suggérer une syntaxe appropriée. Pour le moment les utilisateurs GoAsm devront donc se contenter des classiques CMP, TEST, LOOP et autres mnémoniques de saut conditionnel. Pour autant, GoAsm soutient pleinement l'**assemblage conditionnel** au moment de la compilation utilisant les commandes de type "C" #if / #else / #elseif / # endif, etc..
- GoAsm n'impose pas que l'adresse de départ de votre programme soit un nom réservé comme dans l'assembleur A386, bien que cette possibilité soit néanmoins offerte avec le mot réservé START. De plus, vous ne devez pas définir un label puis la directive END pour définir le **point d'entrée** (comme dans MASM). Avec GoAsm vous utilisez tout simplement un label et indiquez à l'éditeur de liens ce qu'est ce label. Ou, si vous utilisez GoLink, START est présumé en l'absence d'indication contraire. Consulter également sur ce point la section relative à l'**utilisation de GoAsm avec différents linkers**.
- Vous pouvez essayer GoAsm sur vos scripts source existants. Pour vous soulager de quelques travaux fastidieux pour adapter ces fichiers initialement destinés à d'autres assembleurs, j'ai écrit le programme **AdaptAsm.exe**, qui effectue ce travail pour vous (sans écraser, pour autant, le fichier d'origine !).
- Et si vous désirez connaître précisément la rapidité d'exécution de GoAsm lors de l'assemblage de vos fichiers ou de parties d'entre eux, vous disposez pour cela de la directive **GOASM_REPORTTIME**.

1.2 Syntaxe et compatibilité avec d'autres assembleurs

La syntaxe acceptable pour l'assembleur est d'une importance capitale pour tout programmeur en assembleur. Elle varie selon les assembleurs. GoAsm ne crée pas de code 16 bits et fonctionne uniquement en mode «flat» (absence de segments). Pour cette raison, sa syntaxe est très simple. J'ai choisi ce que je considère être la meilleure syntaxe avec, pour principal objectif, la clarté et la cohérence. Vous pouvez être en désaccord avec moi sur ce point. Si oui, je serais intéressé par vos points de vue.
Lors de l'écriture initiale de GoAsm j'ai réfléchi à la possibilité de construire une syntaxe entièrement compatible avec celle d'autres assembleurs, mais j'ai dû rapidement y renoncer en raison d'écarts trop importants susceptibles de se traduire par d'importantes incohérences. J'ai également renoncé à rendre GoAsm entièrement compatible avec un quelconque autre assembleur.
Vous reconnaîtrez la syntaxe d'autres assembleurs. Lorsque cela était possible, j'ai essayé de rester proche de ce que je considère être la meilleure syntaxe de l'assembleur d'usage général. Vous reconnaîtrez également certaines syntaxes empruntées à la programmation en "C". J'ai suivi principalement la syntaxe "C préprocesseur" lorsqu'il me semblait inutile de procéder autrement. Cela rend également l'utilisation du préprocesseur commandes de GoAsm totalement compatible avec mon compilateur de ressources GoRC.

1.3 Pourquoi GoAsm ne vérifie pas les types et paramètres

Après réflexion, j'ai décidé que GoAsm ne devait pas vérifier les types ou les paramètres. Ceci, dans le but de réduire substantiellement la taille du script source et d'ajouter à sa flexibilité et à sa lisibilité. Je conclus que même vérifier sommairement le type dans la programmation assembleur pour Windows n'est pas du tout essentiel, et génère plus d'inconvénients que d'avantages.

Permettez-moi de m'en expliquer ici.

Dans la **vérification de type**, l'assembleur doit s'assurer que les références aux zones de mémoire sont faites avec la bonne taille et le bon type de données en fonction de l'usage qui doit être fait de ces zones de mémoire. Ce résultat est obtenu grâce à un processus en deux étapes. Premièrement, lorsque la zone de mémoire est déclarée, le programmeur doit lui allouer un certain "type". Ensuite, lorsque la zone de mémoire est utilisée, le programmeur a encore pour tâche d'indiquer le type de la mémoire appelée à être utilisée. S'il y a discordance, l'assembleur ou le compilateur afficheront une erreur.

Certains assembleurs, comme NASM, ne font aucune vérification de type. D'autres, comme A386, ne font que des vérifications sommaires sur les types BYTE, WORD, DWORD, QWORD et TWORD. MASM et TASM, comme "C", vous permettent de spécifier vos propres types en utilisant TYPEDEF puis en assurent la vérification.

La **vérification de paramètre** s'assure que le nombre correct de paramètres est passé à une API et en contrôle individuellement le type. La plupart des assembleurs ne vérifient pas les paramètres, mais MASM permet de le faire si le pseudo-mnémonique INVOKE est utilisé.

Les requis pour parvenir à une vérification parfaite de type et de paramètre correspondant au niveau du compilateur "C" sont énormes. Il suffit de regarder l'en-tête d'un programme Windows et de voir les longues listes de différents types alloués aux différentes structures et les paramètres d'API. Sans compter évidemment les efforts du programmeur qui sont nécessaires dans le script source pour veiller à ce qu'aucune erreur ne soit renvoyée par l'assembleur ou le compilateur.

Pour toutes ces raisons, j'ai décidé de suivre l'exemple de NASM sans même proposer la vérification élémentaire de type implémentée dans A386. J'ai utilisé ce dernier de nombreuses années et j'ai apprécié sa syntaxe propre, mais j'ai plutôt ressenti sa vérification sommaire de type comme un obstacle lors de la programmation sous Windows. Ceci, parce que l'on est souvent confronté à des situations où il est nécessaire d'écrire ou de lire des données en utilisant une taille différente de celle utilisée pour les déclarer en premier lieu.

J'ai également renoncé au contrôle de paramètre, estimant qu'il complique inutilement les choses. Il exige d'énormes listes d'API et des paramètres qui doivent être fournis à l'assembleur ou au compilateur afin qu'ils puissent vérifier que ceux-ci correspondent aux besoins de l'API. Oubliez-en ne serait-ce qu'un seul et votre programme ne compile pas. Considérons l'exemple suivant :

```
PUSH 40h,EDX,EAX,[hwnd]
CALL MessageBoxA
```

Voici un appel d'API qui met en œuvre 4 paramètres. Vous seriez tenté d'attendre de l'assembleur qu'il en contrôle le nombre et qu'il vous alerte en cas d'erreur de votre part sur ce point. Mais vous n'avez pas besoin de cet avertissement car votre programme sera tout simplement planté si tel est le cas. Alors, pourquoi effectuer un tel test dans la mesure où il n'y a rien de sournois ici, en tout cas rien qui ne puisse être appréhendé au stade de l'expérimentation ? Au-delà du nombre de paramètres on peut également s'interroger sur la nécessité de tester le type de chacun d'eux. En effet, quel est l'intérêt d'un tel contrôle dans la mesure où *tous* les paramètres à destination des API sont de type Dword (avec une ou deux exceptions sur des milliers) ? Donc, le risque de taille erronée des données à destination d'une API est quasiment nul.

Je conviens qu'il puisse être possible d'envoyer le mauvais *type* de données à une API. Par exemple, vous pourriez envoyer une constante là où il devrait y avoir un handle ou le contenu d'une adresse mémoire en lieu et place d'un *pointeur* vers une adresse mémoire. Cependant, l'API ne fonctionnera tout simplement pas dans ce cas – et, encore une fois, il n'y a rien qui ne puisse être remarqué au stade de l'expérimentation.

L'abolition du contrôle de paramètres et de type ne libère pas seulement l'assembleur de beaucoup de travail, le rendant plus rapide en fonctionnement ; elle épargne aussi au programmeur les interrogations qui accompagnent inévitablement la manipulation d'entête et d'inclusion de fichiers. Enfin, elle garantit une plus grande fluidité dans l'adressage mémoire, car les notifications d'erreur vous seront épargnées si, d'aventure, vous voulez utiliser des données selon une taille qui ne correspond pas à celle qui a été préalablement déclarée. Donc, en GoAsm même si *lParam* a été déclarée comme une valeur DWORD,

```
MOV [lParam],AL
```

est encore permis. Et si LOGFONT est une structure simple de dwords, GoAsm se satisfait pleinement, par exemple, de

```
MOV B[LOGFONT+14h],1
```

que vous pouvez utiliser pour définir une police en italique.

En m'exemptant du contrôle de type et de paramètres, j'ai été en mesure d'abolir également EXTRN. GoAsm n'a pas besoin de connaître le type de symboles qui sont déclarés en dehors du fichier source (c'est-à-dire découverts pendant la phase d'édition de liens). J'espère que vous conviendrez que cela vous épargnera beaucoup de travail acharné et l'angoisse d'avoir à ajouter ces EXTRNs dans les programmes liés.

La contrepartie de la suppression du contrôle du type et des paramètres est que vous devez indiquer à GoAsm la taille des données à exploiter, dans les cas où celle-ci n'est pas implicite.

Ainsi, par exemple, `MOV [MemThing],23h` est-il incorrect. Pour charger `23h` en tant qu'octet en `MemThing` vous devez coder `MOV B[MemThing],23h` (équivalent à `MOV Byte Ptr [MemThing],23h` avec MASM). Ceci est parce GoAsm ne saura pas au moment de l'assemblage si la valeur `23h` doit être chargée en tant qu'octet, mot ou dword, formats qui sont tous acceptés par l'instruction MOV.

À certains égards, l'exigence d'un indicateur de type (lorsque celui-ci n'est pas évident) est utile. Ceci, ne serait-ce que parce que vous pouvez voir immédiatement sur le libellé de l'instruction elle-même la taille de mémoire affectée par son action. Vous n'avez pas à vous reporter à une déclaration de données antérieure pour rechercher son type pour déterminer ce que l'instruction fera. Ainsi, par exemple :

```
MOV B[MemByte],23h    ; réconfortant de voir cela se limite à une opération sur un octet
FLD Q[NUMBER]         ; utile de savoir que c'est nombre réel Qword chargé en double précision
INC B[COUNT]          ; essentiel de savoir que ce comptage est limité à 256
```

Un autre avantage découlant de l'absence de vérification de tout paramètre est qu'il n'y a pas besoin que GoAsm mette en relief les noms des appels vers d'autres modules ou ceux destinés à l'importation. Lors de l'utilisation GoLink, c'est un avantage considérable puisqu'il n'y a pas besoin de fichiers LIB à l'étape d'édition de liens. Mais cela signifie aussi que les fichiers objets GoAsm seront différents de ceux fabriqués par un compilateur "C" ou MASM parce que ces fichiers contiennent des symboles qui seront mis en relief tandis que GoAsm ne fera pas rien de tel. Depuis sa version 0.26.10, GoLink est cependant en mesure d'accepter des fichiers objets des deux ensembles d'outils précités et de les lier aux fichiers objets GoAsm (il suffit juste d'utiliser le commutateur de GoLink /mix – voir l'aide de GoLink).

1.4 Pourquoi GoAsm utilise les crochets pour l'écriture et la lecture de la mémoire

Les programmeurs en assembleur ont longtemps débattu sur l'usage de crochets dans le libellé de l'adressage mémoire. L'argument dominant est que, puisque vous devez utiliser des crochets lorsque l'adresse est contenue dans un registre – par exemple `MOV EAX,[EBX]` –, alors vous devez également utiliser des crochets lorsque l'adresse est matérialisée par un label – par exemple `MOV EAX,[lParam]`. Evidemment, j'ai suivi ce débat avec intérêt. MASM et A386 se sont abstenus de trancher, de sorte que les deux instructions qui suivent font exactement la même chose :

```
MOV EAX,lParam
MOV EAX,[lParam]
```

Cependant, A386 différencie les labels suivis ou non de deux points d'où il résulte que la remarque qui précède est vraie si `lParam` a été déclaré par :

```
lParam DD 0
```

et fausse si `lParam` a été déclaré par :

```
lParam: DD 0
```

Dans ce dernier cas, `MOV EAX,lParam`, toujours selon l'assembleur A386, agirait à l'identique de `MOV EAX,OFFSET lParam`. Très déroutant !
NASM a fait le grand saut en en faisant une condition pour tout adressage mémoire libellé entre crochets. Toutefois, preuve que le débat reste indéterminé, `MOV EAX,lParam` y demeure permis. Dans cet assembleur, cette formulation équivaut au `MOV EAX, OFFSET lParam` utilisé par d'autres assembleurs.

Donc, quand on regarde le code assembleur, sans connaître la syntaxe de l'assembleur concerné, on ne peut jamais être vraiment sûr de ce que `MOV EAX,lParam` fait. La même instruction peut faire deux choses totalement différentes selon l'assembleur utilisé.
Le TASM de Borland, lorsqu'il passe en mode "Idéal", proscrit complètement `MOV EAX,lParam` et permet seulement

```
MOV EAX,[lParam]
```

ou

```
MOV EAX,OFFSET lParam
```

J'approuve cette approche. L'objectif principal, ici, est de s'assurer que le codage est sans ambiguïté. Pour cette raison, j'ai décidé que GoAsm devait être strict sur cette question. Par conséquent, dans GoAsm :

```
MOV EBX,wParam
```

est complètement interdit, à moins que wParam ne soit un mot défini. Afin d'obtenir l'offset dans GoAsm vous devez utiliser

```
MOV EBX,ADDR wParam
```

ou, si vous préférez

```
MOV EBX,OFFSET wParam
```

qui signifie la même chose.

Si vous souhaitez adresser la mémoire dans GoAsm, vous devez donc utiliser la syntaxe

```
MOV EBX,[wParam]
```

1.5 Mnémoniques supportés par GoAsm

1.5.1 Qu'est-ce qu'un “mnémonique”?

Un mnémonique est une instruction sous forme de texte que vous utilisez dans votre script source assembleur. GoAsm assemble ces mnémoniques et les convertit en codes opération (opcodes) que le processeur exécute. Ces codes opération sont parfois appelés *codes machine*. Les mnémoniques sont recommandés par les fabricants de processeurs. Ils sont destinés à transmettre sous forme abrégée et aussi précisément que possible ce que l'instruction fait. Bien qu'il existe maintenant plus de 550 mnémoniques, un programmeur en assembleur n'en utilise seulement que 20 ou 30 régulièrement. Voir une proposition de liste des mnémoniques les plus couramment utilisés dans l'annexe consacrée aux **débutants en assembleur**.

Pour des raisons de portabilité des scripts source et de cohérence en prévision d'éventuelles mises à jour, tous les assembleurs reconnaissent normalement les mnémoniques au niveau où ils se rejoignent dans la fonction d'assemblage. Pour autant, le processeur ignore les mnémoniques et ne fonctionne que dans le code de la machine lui-même. Les programmeurs non familiers de l'assembleur n'utilisent jamais les mnémoniques. Un compilateur travaillant uniquement en "C", par exemple produit encore du code machine, mais il ne fonctionne pas avec les mnémoniques en tant que tels (sauf basculé en mode assembleur en ligne).

1.5.2 Quels mnémoniques sont pris en charge par GoAsm?

GoAsm prend en charge tous les mnémoniques correspondant aux instructions à usage général, y compris les instructions x87 en virgule flottante, les instructions MMX, 3DNow! (avec les extensions), SSE, SSE2, SSE3 et SSSE4, ainsi que AES, ADX, et quelques autres nouvelles instructions. GoAsm prend en charge les instructions pseudo CMP qui peuvent être utilisés avec les registres XMM.
GoAsm ne supporte pas certains mnémoniques qui sont utilisés uniquement pour la programmation 16 bits. C'est le cas de IBTS, IRETW, JCXZ, RETF et XBTS.

Enfin, GoAsm ne supporte pas les mnémoniques qui nécessitent des opérandes supplémentaires, et les cas où il existe des mnémoniques plus faciles à utiliser. Entrent dans cette catégorie :

```
CMPS - utiliser CMPSB ou CMPSD
INS  - utiliser INSB ou INSD
LODS - utiliser LODSB ou LODSD
MOVS - utiliser MOVSB ou MOVSD
OUTS - utiliser OUTSB ou OUTSD
SCAS - utiliser SCASB ou SCASD
STOS - utiliser STOSB ou STOSD
XLAT - utiliser XLATB
```

Chapitre 2
Débuter sur GoAsm

2.1 Construction d'un fichier ASM

Le fichier ASM est un fichier que vous créez et éditez en utilisant un éditeur de texte ordinaire, comme Paws que vous pouvez télécharger à partir de mon site web, **www.GoDevTool.com**, ou de programmes courants comme Notepad (Bloc-Notes) ou Wordpad qui sont livrés avec Windows. Si vous utilisez ce dernier, vous devez vous assurer que vous enregistrez le fichier dans un format qui n'ajoute pas de caractères de contrôle ou de formatage autres que l'habituelle fin de ligne (retour chariot et saut de ligne). Ceci, parce GoAsm ne s'intéresse qu'au texte brut. Vous pouvez vous prémunir contre ces caractères non désirés en sauvegardant le fichier comme document «texte». Si vous n'adjoignez pas une extension au nom de fichier (l'extension désigne les caractères après le "point"), alors l'éditeur peut lui attribuer automatiquement une extension ".txt". Cependant, rien ne vous empêche de la changer en renommant le fichier (vous pouvez exécuter cette opération sur l'Explorateur Windows en pratiquant un clic-droit sur le nom et en sélectionnant la fonction “Renommer”).

Il se peut que vous ne puissiez visualiser l'extension du fichier sur votre ordinateur. Il s'agit, en ce cas, d'une question de paramétrage de l'Explorateur Windows. Pour ce faire, sélectionnez l'élément de menu "Affichage", "Options", "modifiez les options des dossiers et de recherche" puis sur l'onglet "Affichage" et ,enfin, veillez à ce que la case "masquer les extensions des fichiers dont le type est connus" soit décochée. La procédure peut différer légèrement selon la version de Windows.

Il est de tradition chez les programmeurs d'attribuer à leurs scripts source une extension qui correspond au langage dans lequel il est écrit. Par exemple, vous pourriez avoir un fichier assembleur appelé “myprog.asm”. De la même manière, vous trouverez généralement le code source écrit en langage “C” avec l'extension “.c” ou “.cpp” (pour “C ++”), “.pas” pour Pascal et ainsi de suite. Cependant, ces extensions sont totalement neutres d'un point de vue strictement informatique. GoAsm accepte ainsi les fichiers de toute extension de même que ceux qui en sont dépourvus.

Le fichier *.asm* contient vos instructions pour le processeur en mots et nombres. Celles-ci sont converties en code exécutable successivement par l'assembleur puis par l'éditeur de liens. C'est ce code qui sera reconnu et exécuté par le processeur. On dit donc que le fichier *.asm* contient votre "code source" ou votre "script source".

2.2 Insérer du code et des données

A titre d'exemple, examinons le code et les données d'un simple programme Windows 32 bits qui écrit "Hello World (from GoAsm)" dans la fenêtre MS-DOS de l'invite de commande. Voici comment s'écrit le fichier asm :

```
DATA SECTION
;
KEEP DD 0                          ; variable temporaire
;
CODE SECTION
;
START:
PUSH -11                           ; STD_OUTPUT_HANDLE
CALL GetStdHandle                  ; récupère, en EAX, le handle du buffer de l'écran actif
PUSH 0,ADDR KEEP                   ; KEEP reçoit la sortie de l'API WriteFile
PUSH 24,'Hello World (from GoAsm)' ; 24 = longueur de la chaîne
PUSH EAX                           ; handle correspondant au buffer de l'écran actif
CALL WriteFile
XOR EAX,EAX                        ; retourne EAX = 0 comme recommandé par Windows
RET
```

Notez que tout ce qui est après un point-virgule est ignoré jusqu'à la fin de la ligne, de sorte que vous pouvez insérer des commentaires à partir de ce signe. Voir la rubrique **opérateurs** pour d'autres formes de

commentaire. Lire le paragraphe **Qualité des descriptions et commentaires** dans l'Annexe K sur l'importance des commentaires en programmation.

La première ligne de ce fichier ouvre la section de données par `DATA SECTION`. On consultera la rubrique "**sections - déclaration et utilisation**" en ce qui concerne l'importance des sections et comment les utiliser.

Dans cette section, nous déclarons une zone de données de 4 octets (DD signifie un "DWORD" ou "double-mot" qui est de 4 octets) et, comme cette zone va être sollicitée dans le programme, nous l'identifions avec le nom «KEEP» et l'initialisons à zéro. En d'autres termes, nous avons créé la variable 32 bits "KEEP". Lire à cet égard la section **déclaration de données** pour des explications détaillées sur ce point.

Nous ouvrons ensuite la section de code avec le label "START" qui indique au processeur où commencer l'exécution des instructions du programme. C'est ce qu'on appelle habituellement le "point d'entrée". Voir la rubrique **code et point d'entrée** pour de plus amples explications et notamment sur les variantes admises en matière de point d'entrée.

L'instruction suivante "PUSH -11" met la valeur décimale –11 sur la pile en préalable à l'appel de l'API Windows *GetStdHandle* sur la ligne suivante. Il s'agit là du seul paramètre exigé par cette API. La valeur –11 précise ici que la recherche du handle du buffer d'écran est requise. Ce handle est fourni dans le registre EAX en sortie d'API. Lire l'annexe "**comprendre la pile**" pour une explication du fonctionnement de la pile et de l'instruction PUSH. La rubrique **comprendre les nombres finis, négatifs, signés et en complément à deux** explique, en détail, ce que l'on entend précisément par "valeur décimale –11". Enfin, les débutants en Windows liront avec intérêt l'annexe "**pour les débutants en Windows**" qui introduit le fonctionnement des API.

L'instruction d'exécution qui suit PUSH –11 transfère l'exécution à l'API *GetStdHandle* et, à son retour, le registre EAX se retrouve chargé avec la valeur de handle recherchée. L'exécution se poursuit sur la ligne suivante. Lire, sur ce point, la rubrique **transfert de l'exécution à une procédure**.

Après ce premier appel d'API, on trouve 5 PUSHs successifs. Notez que les 2 premiers utilisent une syntaxe spéciale où un seul PUSH permet d'en réaliser plusieurs, les opérandes étant mis à la suite les uns des autres et séparés par une virgule. Ici, il s'agit d'une commodité d'écriture permise par GoAsm et n'ayant rien à voir avec les instructions processeur. Lire à ce sujet la section **instructions répétées** pour en savoir plus. Ces PUSH constituent les paramètres à passer à l'API *WriteFile*. Ces paramètres sont, dans l'ordre : zéro, puis l'adresse de la variable KEEP, puis le nombre 24 décimal qui est la longueur de la chaîne (les mots entre guillemets), puis un pointeur vers de début de cette même chaîne et, enfin, le contenu du registre EAX chargé avec la valeur du handle donnée en retour du précédent appel d'API.

En retour de l'appel d'API `CALL WriteFile`, la valeur zéro est mise dans le registre EAX utilisant l'instruction `XOR EAX,EAX`. C'est la même chose que `MOV EAX,0` mais produit moins d'octets de code. Voir à ce sujet l'**Annexe K "Quelques conseils et astuces de programmation**".

Enfin RET termine le programme en retournant à l'appelant (dans ce cas, Windows lui-même). Voir l'annexe "**pour les débutants en Windows**".

2.3 Assemblage du fichier avec GoAsm

Après avoir écrit comme il convient le code et les données de votre fichier ASM, vous êtes maintenant prêt à finaliser votre programme. Cela se fait en deux temps. Vous devez tout d'abord assembler votre fichier puis le lier. Pour ce faire, vous devez ouvrir une fenêtre MS-DOS[1] (invite de commande). Dans ce cas, vous utilisez la ligne de commande :

```
GoAsm /fo HelloWorld.obj filename
```

où *filename* est le nom de votre fichier asm. Voir la section "**Démarrage de GoAsm**" pour savoir comment utiliser la ligne de commande de GoAsm.

GoAsm produit un fichier «objet» contenant votre code et les données; Ce fichier reçoit l'extension ".obj" et se présente dans un format adapté à l'éditeur de liens. Voir plus d'informations sur **le fichier objet**.

[1] Le lancement de la fenêtre MS-DOS s'obtient en double-cliquant sur CMD.exe dans le répertoire \Windows\System32\

2.4 Lien du fichier objet pour créer le programme EXE

L'étape finale est de "lier" votre programme pour créer l'exécutable final. Vous pouvez utiliser le programme **GoLink** complémentaire à GoAsm pour ce faire. Dès lors, la ligne de commande se présente ainsi :

```
GoLink /console helloworld.obj kernel32.dll
```

(ajoutez le commutateur "-debug coff" si vous envisagez d'examiner le programme dans le débogueur).

Notez que les appels *GetStdHandle* et *WriteFile* s'adressent à KERNEL32.DLL ce qui explique que le nom de cette DLL apparaisse dans la ligne de commande de GoLink. Voir pour **plus d'informations à propos des DLL**. Voir la rubrique “utilisation de GoAsm avec divers linkers” si vous souhaitez procéder à l'édition des liens autrement qu'avec GoLink. Consulter l'aide de GoLink pour connaître les autres options de cet éditeur.

Dans la ligne qui précède, GoLink crée le fichier `HelloWorld.exe`. Vous pouvez ensuite exécuter ce programme à partir de la fenêtre MS-DOS (invite de commande). Tapez HelloWorld et appuyez sur Entrée. Vous verrez la chaîne que vous avez envoyée à l'API *WriteFile* s'écrire dans la console.

Revenons maintenant sur les lignes de votre script source.

Dans un premier temps, vous avez demandé à Windows le handle de la fenêtre de la console, lequel a été renvoyé par l'API *GetStdHandle* qui l'a recherché puis stocké dans le registre EAX. Dans un second temps, ce handle et la chaîne à écrire ont été passés à *WriteFile*. Exprimé autrement, vous avez invité Windows à écrire la chaîne spécifiée dans la console. L'information quant à la manière exacte d'utiliser les API et leur passer les paramètres appropriés est disponible auprès de Microsoft depuis le site MSDN (chercher "Platform SDK»). Enfin il est utile de lire la section consacrée aux suggestions sur la façon d'organiser votre travail de programmation.

Chapitre 3
Éléments de base de GoAsm

3.1 Démarrage de GoAsm

La syntaxe de la ligne de commande est :

GoAsm [command line switches] *filename*[.*ext*]

Où,

filename est le nom du fichier source
[*command line switches*] donne l'emplacement des éventuels commutateurs de la ligne de commande de GoAsm décrits ci-après.

Commutateur de ligne de commande

/b	beep sur erreur
/c	place systématiquement le fichier de sortie dans le répertoire courant
/d	définit un mot (par exemple /d WINVER=0x400)
/e	fichier de sortie vide autorisé
/fo	spécifie le fichier sortie avec son chemin d'accès. Par exemple /fo asm\myprog.obj
/gl	conserve le soulignement d'entête dans les appels externes "C" de la bibliothèque
/h ou /?	aide (affiche les possibilités présentes de la ligne de commande)
/l	crée un fichier contenant le listing d'assemblage
/ms	enrichissement pour mslinker
/ne	pas de messages d'erreur
/ni	pas de messages d'information
/nw	aucun message d'avertissement
/no	aucun message de sortie quel qu'il soit
/sh	partage des fichiers d'en-tête (les fichiers d'en-tête peuvent être ouverts par d'autres programmes lors de l'assemblage)
/x64	assemblage pour processeurs AMD64 ou IA-64
/x86	source assembleur 64 bits en mode de compatibilité 32 bits

Si la spécification du nom du fichier d'entrée ne comporte pas d'extension, GoAsm cherche le fichier sans aucune extension. Si ce fichier est introuvable en tant que tel, GoAsm recherche à nouveau le même fichier agrémenté d'une extension *.asm*.

Si aucun chemin d'accès ne précède le nom du fichier d'entrée, ce dernier est supposé être localisé dans le répertoire courant.

Si aucun nom de fichier n'est précisé pour la création du fichier objet, celui-ci est créé avec le même nom que le fichier d'entrée et affecté de la terminaison *.obj*. Par exemple MyAsm.asm va créer un fichier appelé MyAsm.obj.

Le répertoire qui reçoit le fichier de sortie est :

- le chemin d'accès spécifié si /fo est utilisé, ou *si on ne le mentionne pas*:
- le répertoire courant si /c est spécifié, ou *si on n'utilise pas ce commutateur*:
- le chemin d'accès associé au fichier d'entrée, ou *si aucun répertoire n'est donné*:
- le répertoire courant

Si aucune extension n'est précisée pour le fichier de sortie, *.obj* est créée par défaut. Le fichier de listing d'assemblage emploie le même nom que le fichier de sortie mais avec l'extension *.lst* et il est créé, par ailleurs, dans le même répertoire que celui-ci.

3.2 Sections - déclaration et utilisation

3.2.1 Pourquoi les sections sont nécessaires

Vous devez déclarer une section avant que vous ne commenciez à coder. La raison en est que le processeur a besoin de connaître les attributs des instructions qui lui sont adressées. Notez également que le système Windows se repose sur ces attributs pour identifier les parties de votre code. Les attributs les plus courants sont la lecture seule (ne peut pas recevoir d'écriture), la lecture-écriture (peut recevoir une écriture) et l'exécution (instructions de code). En interne les processeurs traitent l'instruction de la manière la plus appropriée et la plus rapide en rapport avec l'attribut. Par exemple, les instructions de code utilisent le code cache du processeur, le matériau non-constitutif de code est assimilé à des données et peut être pris en charge par le cache de données.

Lorsque vous déclarez une section dans votre script source, GoAsm définit automatiquement l'attribut de la section. Une fois ceci fait, vous pouvez commencer à écrire le code ou les données dans votre programme.

3.2.2 Comment déclarer une section

En programmation Windows, nous sommes intéressés par seulement quatre types de section : le code, les données, les constantes, et les **données non-initialisées**. Vous déclarez le code, les données ou les sections de constantes comme suit :

CODE SECTION

DATA SECTION

CONST SECTION ou CONSTANT SECTION

Les mots "code", "data", "const" et "constant" sont réservés à la déclaration des sections et une erreur sera signalée si ces mots sont utilisés ailleurs dans votre source.

GoAsm permet également de raccourcir les formes de déclaration de section comme suit :

CODE
DATA
CONST

Vous pouvez également utiliser .CODE, .DATA et .CONST si vous le souhaitez.

GoAsm ajoute automatiquement les attributs en fonction du processeur et de Windows. Une section de code reçoit les attributs *lecture*, *exécution*, *code*. Une section de données est dotée des attributs *lecture*, *écriture*, *données initialisées*. Une section const reçoit les attributs *lecture*, *données initialisées* (vous ne pourriez pas écrire dans une section const). Les *données non-initialisées* possèdent les attributs *lecture*, *écriture*, *données non-initialisées*.

A défaut d'ajouter l'attribut SHARED, vous ne pouvez faire fi de ces attributs de votre propre initiative. Ceci est inutile car Windows s'arroge le contrôle total sur les attributs de la section lorsqu'elle est chargée et exécutée. Par exemple, même si vous donnez à une section de code l'attribut d'*écriture*, Windows ne vous permettra pas d'écrire dedans. De la même manière, Windows ne vous permettra pas d'exécuter du code dans une section de données. Vous pouvez néanmoins déroger à ce comportement en appelant l'API *VirtualProtect* au moment de l'exécution.

Dans GoAsm vous pouvez inclure des données en lecture seule (read-only) dans une section de code, même s'il peut en résulter une réduction des performances.

Déclarer une section positionne implicitement certains commutateurs dans GoAsm qui affectent la syntaxe et le codage. Les règles sont les suivantes :

- Tous les labels d'une section de code doivent impérativement se terminer par deux points. Cela permet à GoAsm de distinguer un label de ce qui n'en est pas un, de veiller à ce que les mnémoniques et les directives mal orthographiés soient toujours signalés comme une erreur.
- Les labels réutilisables ne sont autorisés que dans une section de code. Si vous les utilisez dans une section de données, ils seront considérés comme des labels uniques et intégrés à ce titre dans la table des symboles.

- GoAsm signalera une erreur si une instruction tente d'écrire dans une section `const`. La section `const`. est destinée aux données et chaînes initialisées qui ne sont pas appelées à recevoir une écriture.

3.2.3 Section de données non-initialisées

Si vous déclarez des données non-initialisées, GoAsm constitue une section spécifique de données non-initialisées dans le fichier-objet. Elle sera nommée «.bss» en considération d'autres outils. GoAsm lui attribue d'office ce nom que vous ne pouvez pas modifier parce que certains linkers s'attendent précisément à le trouver. Avec la plupart des linkers, y compris GoLink, la section .bss ne trouve pas sa place dans l'exe final. Au lieu de cela elle est fusionnée avec une section d'attribut lecture/écriture dans le fichier exe. Les attributs de la section de données non-initialisées sont la *lecture*, l'*écriture*, les *données non initialisées*.
L'avantage de déclarer des données non initialisées, plutôt que des données initialisées, est que l'exécutable est plus petit. Ceci, parce que l'exécutable se borne à spécifier la quantité de données non-initialisées à réserver sans leur attribuer la moindre valeur. Les buffers de toute sorte sont souvent constitués ainsi. Voir la section **déclaration des données non-initialisées ordinaires**.

3.2.4 Morcellement des sections

Rien n'oblige le programmeur à écrire de grandes sections indivisibles. Par exemple, il est tout-à-fait envisageable d'avoir une portion de données suivie d'une portion de code, elle-même suivie par une nouvelle portion de données et ainsi de suite, sous réserve de prendre soin d'en déclarer la nature à chaque fois avec un des mot-clé suivants :

CODE SECTION ou
DATA SECTION
CONST SECTION

ou leurs formes abrégées, le cas échéant. Vous pouvez le faire aussi souvent que vous le souhaitez au travers de votre script source. GoAsm et l'éditeur de liens se chargent concaténer toutes les instructions destinées à chaque section.

Voir aussi **Sections - Gestion avancée** sur les dénominations de sections, les sections partagées, les section de commande, et les considérations liées à l'alignement de la section.

3.3 Déclaration des données

3.3.1 Qu'est-ce qu'une "donnée"?

D'une certaine manière toutes les instructions transmises à un processeur sont des «données». Mais les programmeurs en assembleur utilisent ce mot pour désigner une information qui est, soit fixe, soit susceptible d'être modifiée au moment de l'exécution et qui ne peut pas être exécutée en tant qu'instruction de processeur. Les données peuvent être classées en 4 catégories :

1. Les données en lecture seule – read-only – spécifiées en tant que telles à la phase d'assemblage (lorsque le programme est compilé) et qui sont conservés dans la section const qui a un attribut de lecture seule. On parle alors de "données initialisées" parce que leur contenu est fixé dans le script source. Au moment de l'exécution, ces données peuvent être lues, mais on ne peut écrire dessus et modifier ainsi leur contenu. Dans votre script source, vous devez leur attribuer des labels de sorte qu'elles puissent être référencées facilement.
2. Les données fixées au moment de l'assemblage et localisées dans la section de données du fichier exécutable. Encore une fois, le contenu des données sera fixé dans votre script source mais, au moment de l'exécution, elles pourront être lues ou modifiées en utilisant les labels de données correspondants.
3. Les données non fixées au moment de l'assemblage, mais qui sont localisées dans une zone qui leur est réservée. Il s'agit des “données non-initialisées” et seule leur taille est recensée dans le fichier exécutable. Dans votre script source assembleur vous spécifiez la quantité de données devant être réservée. Vous pouvez leur attribuer des labels, mais vous ne pouvez pas initialiser leur contenu. L'avantage de ce type de données est qu'elle ne prennent pas de place dans l'exécutable. Au moment du chargement les données sont seulement localisées et ne reçoivent pas de contenu donné à ce stade. Au moment de l'exécution, les données de ce type peuvent être lues ou écrites de la même manière que leurs homologues de la section de données.

4. Les données établies au moment de l'exécution, soit par le programme lui-même, soit par le système. Ces données ne sont pas établies au moment de la phase d'assemblage de votre script source. Elles le sont, en réalité, par le système d'exploitation lorsque votre code est exécuté.

3.3.2 Déclaration des données numériques initialisées

GoAsm se conforme à la syntaxe assembleur traditionnelle pour déclarer des données dans votre script source.
Dans une section data ou cons, un label ne doit pas être terminé par deux points. Dans une section de code cela est nécessaire, pour favoriser l'identification des erreurs de syntaxe. Quelques exemples (utilisant une section de données) :

```
HELLO1   DB 0                ;  1 octet avec le label "HELLO1" fixé à zéro
         DB 0                ;  le second octet fixé à zéro
HELLO2   DW 34h              ;  2 octets (soit un mot) fixé à la valeur 34h
HELLO3   DD 12345678h        ;  4 octets (un dword) fixés à la valeur 12345678h
HELLO4   DD 12345678D        ;  4 octets (un dword) fixés à la valeur décimale 12345678
HELLO5   DD 1.1              ;  4 octets (un dword) fixés à la valeur du nombre réel 1.1
HELLO6   DQ 0.0              ;  8 octets (un qword) fixés à la valeur du nombre réel 0.0
HELLO7   DQ 123456789ABCDEFh ;  8 octets (un qword) fixés à la valeur 123456789ABCDEFh
HELLO8   DQ 1234567890123456 ;  8 octets (un qword) fixés à la valeur décimale 1234567890123456
HELLO9   DT 1.1E0            ; 10 octets (un tword) fixés à la valeur du nombre réel 1.1
HELLOA   DT 123456789ABCDEFh ; 10 octets (un tword) fixés à la valeur 123456789ABCDEFh
```

Notez que DB, DW, DD et DQ acceptent les nombres aussi bien dans le format décimal qu'hexadécimal ; DD, DQ et DT acceptent également les nombres réels.

Voir les sections **déclaration des nombres réels**, **chargement direct de l'exposant et de la mantisse** et **chargement d'un fichier avec INCBIN**.

3.3.3 Déclaration de plusieurs données sur une même ligne

Une virgule après une initialiseur signifie qu'un autre initialiseur est attendu afin de déclarer d'autres données. La syntaxe est la suivante :

```
Label   DB 0,0,0,0           ; 4 octets fixés à zéro
        DW 33h,44h,55h,66h   ; 4 mots initialisés
        DD 33h,44h,55h,66h   ; 4 dwords initialisés
        DD 1.1,2.2           ; 2 DD de nombres réels
        DQ 1.1,2.2           ; 2 DQ de nombres réels
        DQ 3333h,4444h       ; 2 DQ de nombres hexa
        DT 1.1,2.2           ; 2 DT de nombres réels
        DT 5555h,6666h       ; 2 DT de nombres hexa
```

3.3.4 Déclaration de données non-initialisées ordinaires

GoAsm rejoint ici la syntaxe traditionnelle des assembleurs mais, à l'instar de A386, il ne nécessite pas de section non-initialisée (la section «.bss») pour déclarer ce type de variable. Au lieu de cela, un simple point d'interrogation garantit que la donnée est considérée comme non initialisée. Quelques exemples (dans les sections texte data ou const):

```
HELLO1   DB ?     ;  1 octet avec le label "HELLO1" enregistré comme non-initialisé
HELLO2   DW ?     ;  2 octets (word)
HELLO3   DD ?     ;  4 octets (dword)
HELLO4   DQ ?     ;  8 octets (qword)
HELLO5   DT ?     ; 10 octets (tword)
```

Les données non-initialisées orphelines ne sont pas permises : vous ne pouvez pas mélanger les données initialisés et non initialisées à défaut de quoi vous provoquez une erreur :

```
DATA6   DD 5 DUP 0
        DB ?        ; déclaration non autorisée
        DB 0
```

En revanche, l'écriture qui suit est parfaitement correcte :

```
DATA6   DD 5 DUP ?  ; 5 dwords pour le client
        DB ?        ; un octet pour avoir le plat principal
        DB ?        ; et un octet pour avoir les sauces
```

Ceci vous permet de séparer les zones de données non initialisées de sorte que chaque zone séparée puisse avoir son propre commentaire.

Les données non initialisées ne peuvent pas être déclarés tant qu'une section n'a pas été ouverte. Vous pouvez déclarer des données non initialisées au sein de la section de code, mais les labels doivent se terminer par deux points comme il est de règle pour la section de code, par exemple :

```
HELLO1:     DB ?    ; 1 octet avec le label "HELLO1" enregistré comme non-initialisé
HELLO2:     DW ?    ; 2 octets (un word)
```

3.3.5 Déclaration de données dupliquées (DUP)

GoAsm utilise la syntaxe DUP bien connue, mais ne nécessite pas d'initialiseur entre parenthèses. Quelques exemples (dans la section de données):

```
HELLO1    DB 2 DUP 0       ; 2 octets avec le label "HELLO1" tous les deux initialisés à zéro
HELLO1A   DB 800h DUP ?    ; 2K de buffer de données non-initialisées
HELLO2    DW 2 DUP 0       ; 4 octets tous fixés à zéros
HELLO3    DD 2 DUP ?       ; 8 octets dans la section non-initialisée
HELLO4    DD 2 DUP 1.1     ; nombre réel 1.1 dans dword répété 1 fois
HELLO5    DQ 2 DUP 1.1     ; nombre réel 1.1 dans qword répété 1 fois
HELLO6    DQ 2 DUP 333h    ; qword répété 1 fois
HELLO7    DT 2 DUP 1.1     ; nombre réel 1.1 dans tword répété 1 fois
HELLO8    DT 2 DUP 444h    ; tword répété 1 fois
```

Vous pouvez utiliser DUP pour déclarer une donnée globalement, puis en initialiser individuellement chaque élément :

```
HELLO300 DB 3 DUP <23,24,25>  ; déclare 3 octets et les initialise respectivement à 23,24,25
```

qui fait la même chose que :

```
HELLO300 DB 23,24,25          ; déclare 3 octets et les initialise respectivement à 23,24,25
```

Bien qu'il puisse sembler inutile d'y recourir, la syntaxe rend plus facile l'initialisation d'un membre d'une structure si celui-ci contient l'opérateur DUP Voir la section **Initialisation de membres de structure avec déclarations de données DUP**.

3.3.6 Initialisation utilisant des caractères en lieu et place de leurs codes ASCII

Au lieu de devoir initialiser des caractères par le biais de leur code ASCII, vous pouvez obtenir plus directement ce résultat en vous bornant à déclarer les caractères entre guillemets. Par exemple :

```
Letters   DB 'a'          ; au lieu de DB 61h
          DW 'xy'         ; au lieu de DW 7978h
Sample    DD 'form'       ; au lieu de DD 6D726F66h
ZooDay    DQ 'Saturday'   ; au lieu de DQ 7961647275746153h
```

Exception faite du cas de l'**insertion de chaînes Unicode**, GoAsm ne réalise pas de conversion du caractère, de sorte que la valeur réelle insérée dans le fichier objet dépendra du jeu de caractères courant au moment de l'assemblage.

GoAsm *ne mémorise pas* le mot et les déclarations de chaîne Word et Dword ci-dessus en utilisant le stockage inverse. Il rejoint en cela la pratique de NASM qui a pris le contrepied de MASM en la matière. Cela signifie que l'octet de poids faible dans `DW 'xy'` est 'x' et que 'y' est l'octet de poids fort. De même dans `DD 'form'` correspondant au label *Sample* , l'octet de plus faible poids est 'f' et ainsi de suite jusqu'à l'octet de plus fort poids qui est 'm'. L'avantage de cette configuration est de vous permettre, par exemple, de transférer très simplement cette chaîne au moyen du codage ci-dessous :

```
MOV EDI, ADDR BUFFER
MOV EAX,[Sample]
STOSD
```

qui insère dans le buffer la chaîne 'form'.

Les octets non initialisés reçoivent la valeur de zéro. Par exemple :

```
DW 'a'      ; le premier octet est 'a', le second est nul
DD 'ab'     ;'a' puis 'b' puis 2 octets à zéro
```

On peut répéter les initialisations de valeur de caractère comme, par exemple :

```
DD 3 DUP "Hi"
```

Cela insère 'H' puis 'i' puis deux zéros, cette opération étant répétée à trois reprises.

3.3.7 Déclaration de chaînes

Les chaînes peuvent être entre guillemets simples ou doubles. En voici quelques exemples d'utilisation :

```
String1   DB 'Ceci est une chaîne'
          DB 'Ceci est une chaîne avec des guillemets "internes"'
String2   DB "Une chaîne entre guillemets"
          DB "J'apprécie le contenu de la chaîne"
String3   DB '"Une chaîne elle-même entre guillemets"'
          DB "'Une chaîne elle-même entre guillemets simples'"
          DB "'Une chaîne avec ses propres guillemets simples'"
String4   DB """Une chaîne avec ses propres guillemets simples et doubles"""
          DB '''Une chaîne elle-même avec ses guillemets "internes"'''
```

Dans *String4* chaque couple de 2 guillemets consécutifs constitue un guillemet qui est partie intégrante de la chaîne en tant que caractère affichable. Cette convention vaut uniquement pour les guillemets des 2 extrémités de la chaîne qu'ils encadrent (contrairement à GoRC, qui agit également à l'intérieur de la chaîne).

3.3.8 Déclaration d'une chaîne sur plusieurs lignes

Une virgule après une chaîne signifie qu'un autre initialiseur est attendu lequel peut déclarer, soit des données complémentaires, soit une autre chaîne ainsi qu'on peut le voir dans les exemples suivants :

```
String1   DB 'Ceci est une chaîne avec terminateur null',0
          DB 'Première chaîne',0,'Et une autre chaîne',0
String2   DB 22h,"Une chaîne avec ses propres guillemets doubles",22h
```

Les valeurs ASCII que vous pouvez utiliser ici, si le vous souhaitez, sont : 22h pour les guillemets doubles et 27h pour les apostrophes.

3.3.9 Chaînes plus longues

Dans le cas d'une chaîne plus longue, il est possible de la scinder par faute de place sur la ligne et d'en reporter le contenu résiduel à la ligne suivante tout en faisant précéder ce contenu de l'opérateur DB. Par exemple :

```
LongString1   DB 'Son premier programme semblait très prometteur'
              DB 'jusqu'à ce qu'il ne fonctionne pour la première fois',0
LongString2   DB 'Son erreur fondamentale:',0Dh,0Ah
              DB 'il ne l'a pas testé en cours de développement', 0
```

Les valeurs ASCII 0Dh et 0Ah sont respectivement le retour chariot et le saut de ligne. Ils sont utilisés pour commencer une nouvelle ligne lorsque l'ensemble de la chaîne est affichée sur l'écran.

3.3.10 Chaînes Unicode

En programmation Windows, vous avez parfois besoin de déclarer des chaînes Unicode dans les sections data ou const, par exemple dans un modèle de dialogue. Il y a plusieurs façons de procéder dans GoAsm qui sont décrites en détail dans le chapitre "Écriture de programmes Unicode" du volume 2. En bref, vous pouvez utiliser l'une ou l'autre des méthodes suivantes :

- Reposez-vous sur le format Unicode de base du script source (GoAsm peut lire les fichiers Unicode UTF-16 et UTF-8).
- Utilisez le symbole L suivi d'une *apostrophe* utilisé en programmation "C", comme par exemple :
  `DB L'Bonjour comment vas-tu?'`

- Déclarer la séquence Unicode en utilisant DUS :

```
DUS 'Je suis une chaîne Unicode avec une nouvelle ligne et le terminateur null',0Dh,0Ah,0
```

Voir aussi la section "Prépositionnement utilisant la directive STRINGS" dans le volume 2.

3.3.11 Insertion de blocs de données par DATABLOCK

Pour les blocs de données volumineux risquant d'encombrer inutilement le fichier source, il existe une alternative consistant à utiliser **INCBIN** pour charger le contenu ou une partie du contenu du fichier contenant ces données. Sinon, vous pouvez utiliser DATABLOCK_BEGIN et DATABLOCK_END s'il s'avère plus approprié de faire figurer explicitement le bloc de données dans le fichier source lui-même.

La syntaxe d'un DATABLOCK est la suivante :

```
MyBlockData DATABLOCK_BEGIN      ;comment
    .
    . les données sont insérées ici
    .
DATABLOCK_END
```

Ici tout le matériau positionné entre DATABLOCK_BEGIN et DATABLOCK_END est inséré dans le fichier de sortie de l'assembleur, et vous pouvez ensuite adresser les données en utilisant le label MyBlockData.
GoAsm considère que ces données commencent immédiatement après la fin de la ligne contenant DATABLOCK_BEGIN et s'achèvent à la fin de la ligne précédant immédiatement celle contenant DATABLOCK_END.
Les données sont insérées à l'état brut, c'est-à-dire qu'aucune conversion n'est effectuée. Cela signifie que les caractères qui ne peuvent pas être affichés dans un éditeur ordinaire tel que les espaces ou les tabulations, par exemple, seront également chargés. Cela signifie aussi que le format des données et des caractères qui peuvent être utilisées dans les données ne sont limités que par l'éditeur que vous utilisez pour écrire votre code source.

3.3.12 Initialisation utilisant les adresses de labels

Il est fréquent que vous ayez besoin de charger un DWORD avec l'adresse d'un label, de telle sorte qu'après traitement par l'assembleur puis le linker, ledit DWORD contienne un pointeur vers ce label. Le label peut être, soit un label de donnée, soit un label de code. Par exemple :

```
MS1       DB    'Première chaîne à utiliser', 0
MS2       DB    'Deuxième chaîne à utiliser', 0
Strings   DD    MS1, MS2                ; Strings contient l'adresse de MS1 et MS2
```

Alors, si vous souhaitez utiliser la chaîne MS2, il vous est possible d'écrire `MOV ESI,[Strings+4]` au lieu de `MOV ESI,ADDR MS2`
En généralisant, tous les tableaux peuvent être créés en utilisant cette méthode et adressés au moyen du multiplicateur de registre d'index * (scale) comme, par exemple :

```
MOV ESI,[Strings + EAX * 4]
```

Ici, le registre EAX reçoit l'index de la chaîne à utiliser. Lorsque EAX est nul, ESI reçoit l'adresse du label de la première chaîne ; lorsque EAX = 1, ESI reçoit l'adresse du label de la première chaîne et ainsi de suite s'il y a plus de chaînes.

Voici un exemple en utilisant des labels de code :

```
PROCEDURE_TO_CALL DD FIRSTPROC,SECONDPROC
MOV ESI,ADDR PROCEDURE_TO_CALL      ; adresse de la liste de procédures dans ESI
MOV ESI,[ESI+EAX*4]                 ; adresse du label de la procédure recherchée dans ESI
CALL [ESI]                          ; appel de la procédure
```

3.4 Code et point d'entrée

3.4.1 Qu'est-ce que le "code" ?

Le code est constitué des instructions contenues dans une section nommée "Code", qui a les attributs *code* et *execute*. Concrètement, vous indiquez au processeur laquelle des instructions de code doit être exécutée.

Le processeur lit les instructions octet par octet et les exécute. Chaque octet de code exécutable est appelé un *opcode*.

3.4.2 Que fait le "point d'entrée" ?

Dans un exécutable ordinaire (fichier .exe) le point d'entrée caractérise l'adresse où l'exécution commence immédiatement après le chargement. Dans une DLL (fichier .dll), cela désigne l'adresse où l'exécution prend place pendant le processus de chargement.

3.4.3 Comment est contrôlée l'exécution ?

Une fois l'exécution commencée et le point d'entrée atteint, votre programme prend le contrôle de l'exécution et va se poursuivre à partir de cette adresse. Assez souvent, bien que cela ne soit pas une obligation, la première instruction au point d'entrée consiste en un CALL, un saut conditionnel ou inconditionnel à destination de procédures écrites plus avant ou d'API.

3.4.4 Comment établir un point d'entrée ?

De ce qui précède on peut voir que, sauf si votre script source est constitué uniquement de données, il est essentiel de fournir un point d'entrée à votre programme. Dans GoAsm ceci est réalisé très simplement en attribuant un label au point d'entrée puis en indiquant au linker que ledit label est le point d'entrée du programme. On peut également utiliser le label START – suivi de deux points – mot réservé de GoAsm définissant explicitement le point d'entrée et reconnu comme tel par l'éditeur de liens.
Les exemples qui suivent proposent deux syntaxes possibles du commutateur à mettre sur la ligne de commande de GoLink si l'on décide de ne pas utiliser START et de spécifier un label de point d'entrée distinct, par exemple, *entry* :

```
-entry STARTINGADDRESS
/entry STARTINGADDRESS
```

Si vous utilisez ALINK seule la première méthode fonctionne.

L'intérêt du label réservé START est d'éviter de devoir donner au linker une directive spécifique désignant le point d'entrée. GoLink suppose en effet que celui-ci est constitué par label réservé START sauf avis contraire et lorsqu'il est présent. Voici comment spécifier START dans votre script source pour désigner le point d'entrée du programme :

```
START:
```

Cela peut être en majuscules, en minuscules ou en une combinaison des deux.

Nous venons de voir ce qu'il en est en ce qui concerne GoLink. Les linkers concurrents abordent cette question de différentes manières.

Si vous utilisez le MS linker vous devez faire précéder votre label par un caractère de soulignement. Votre label du point d'entrée devient donc _START: dans votre script source. Ensuite, vous devez positionner l'une ou l'autre de ces deux instructions sur la ligne de commande de l'éditeur de liens (sans le caractère de soulignement):

```
-ENTRY START
/ENTRY START
```

On constate ici que le MS linker est conçu pour fonctionner avec un compilateur "C" qui fera précéder les labels globaux d'un caractère de soulignement. Donc, l'éditeur de liens cherche l'étiquette _START, plutôt que START. Les programmeurs en assembleur ont dû s'accommoder de ces bizarreries dans les outils Windows pendant de nombreuses années, mais maintenant nous avons notre indépendance !

Voir aussi la section “**utilisation de GoAsm avec différents Linker**”.

3.5 Labels uniques, réutilisables et à portée paramétrable

3.5.1 Qu'est-ce qu'un label?

Un label est un nom que vous attribuez à un emplacement particulier dans les données ou dans le code dans la perspective de pouvoir y accéder simplement. Il a la même fonction qu'un signet. Cela vous permet de vous référer à cet emplacement et d'y accéder en utilisant un nom. Un *label de données* se réfère aux données ; un *label de code* fait référence à un code exécutable. Un *symbole* est un label qui apparaît dans la

table des symboles du fichier objet et qui peut donc être vu par le débogueur si une version de débogage de l'exécutable est constituée.

3.5.2 Labels uniques

Un label unique correspond au cas général d'un label qui ne peut être utilisé qu'une seule fois dans votre script source et dans les fichiers objets liés. Il est dit de portée «globale», c'est-à-dire, qu'au moment de l'édition des liens, il peut être accessible à d'autres fichiers objets. Généralement, il est d'usage de choisir un nom qui distingue la fonction de donnée de la fonction de code, par exemple NAME_LIST ou CALCULATE_RESULT. Si vous avez paramétré votre linker pour fournir une sortie de débogage, tous les labels uniques seront mis dans la liste des symboles et transmis au débogueur. Dans GoAsm vous établissez un label unique comme suit :

```
NAMEOFLABEL:
```

Cela ne produit aucun code, mais fixe un signet appelé NAMEOFLABEL au point des données ou du code où il apparaît. Si vous êtes dans une section de données, les deux points ne sont pas obligatoires. Il en va de même si un label donne le nom d'une trame de pile automatisée. Par conséquent, les lignes suivantes créent toutes des labels uniques :

```
(dans la section de données)
HELLO   DB 0    ; label HELLO
BYE:    DB 0    ; label BYE
MEAGAIN ;label MEAGAIN

(dans la section de code)
RICE:           ; label RICE
PEAS: FRAME     ; label PEAS
BEANS FRAME     ; label BEANS
```

Vous pouvez voir à partir de cela que tout mot qui n'est pas réputé être une directive, un mnémonique, une déclaration ou initialisation de données, ou un mot réservé de GoAsm sera considéré comme un label. GoAsm attend deux points après un label de section de code. Ceci parce qu'il y a de nombreux mots qui doivent être utilisés dans une section de code et que, s'ils sont mal orthographiés, il est important qu'une erreur soit déclarée plutôt que le mot soit interprété à tort comme un label.

3.5.3 Labels réutilisables

Parfois, vous avez besoin d'apposer des labels sur des parties de votre script source avec des noms que vous avez déjà utilisés auparavant. GoAsm offre deux niveaux de labels réutilisables qui peuvent être employés dans une section de code :

- **labels réutilisables de portée locale** commençant par un point, et
- **labels réutilisables de portée non limitée** composées de chiffres ou d'un caractère suivi de chiffres

La *portée* d'un label définit d'où il peut être consulté en utilisant son propre nom non modifié. Regardons de plus près ces deux types labels réutilisables.

3.5.4 Labels réutilisables de portée locale

Ces types de labels, d'une syntaxe particulière et donc reconnaissables à ce titre, sont créés en utilisant un point suivi d'un label comme, par exemple :

```
.looptop      ;label looptop
.fin          ;label fin
```

La limite de la portée de ces labels spéciaux est balisée par les labels de code uniques présents dans le script source. En d'autres termes, le label peut être sauté à condition qu'il n'y ait pas de label unique sur le chemin. Ainsi, par exemple :

```
JZ >.fin
CALCULATE:
.fin
RET
```

Ici l'instruction de saut JZ ne trouvera pas .fin parce que le label CALCULATE est un label de code unique placé sur le chemin.

Si vous voulez sauter par-dessus un label de code unique pour atteindre un label réutilisable de portée locale, vous pouvez utiliser un autre label de code unique ou un label réutilisable non délimité comme destination du saut. Il vous est également possible, quoique de manière plus marginale, d'utiliser le label de portée locale dans une trame de pile automatisée. Voir, à ce sujet, **labels réutilisables à portée définie dans les trames de pile automatisées**.

Les labels réutilisables de portée locale sont envoyés au débogueur comme des symboles avec leur «propriétaire». Par conséquent le symbole envoyé au débogueur dans l'exemple ci-dessus est CALCULATE.fin, et une autre façon de sauter par-dessus le label unique serait d'écrire JZ> CALCULATE.fin.

3.5.5 Labels réutilisables de portée non-limitée

Vous rencontrerez souvent dans votre code des sauts ou des boucles d'amplitude faible pour lesquels le choix d'un nom de label mûrement réfléchi n'apporte aucune plus-value à la compréhension du listing. Pour ceux-ci vous pouvez utiliser un label dont le nom ne sera pas transmis au débogueur en tant que symbole. Il est utile, par ailleurs, lors du débogage de limiter la table de symboles aux noms les plus importants dans votre code. Ces labels sont constitués soit uniquement de chiffres, soit d'un caractère suivi d'un ou plusieurs chiffres. Vous pouvez également utiliser une variante avec un point décimal qui facilite l'ajout de nouveaux labels locaux au code existant. Le label lui-même doit toujours se terminer par deux points. Voici des exemples de syntaxe de labels réutilisables non délimités :

```
L1:
24:
24.6:
```

Vous pouvez même utiliser deux points tous seuls pour ces destinations de saut de très faible portée dans votre code.

3.6 Sauts vers des labels : sauts de code courts et longs

Il existe plusieurs instructions de saut. Certaines ne vont agir que si les flags sont dans un état particulier. On les appelle « instructions de saut conditionnel ». D'autres, telle qu l'instruction JMP, sauteront toujours à la destination spécifiée indépendamment de l'état des flags. On trouve également les instructions de boucle et leur variante conditionnelle qui s'interrompt si ECX = 0. Enfin, il y a l'instruction CALL qui effectue un saut puis un retour au terme de la procédure appelée. Tous ces instructions ont besoin d'un *label* précisant leur destination.

3.6.1 Les indicateurs de direction

Afin de rendre votre script source plus lisible, GoAsm propose des indicateurs de direction pour préciser la direction du saut. L'indicateur de direction "retour" est facultative. Par exemple, en utilisant des labels réutilisables à portée locale :

```
JZ > .fin          ; sauter en avant à .fin
JMP > .exit        ; sauter en avant à .exit
LOOP .looptop      ; boucle arrière vers .looptop
LOOP <.looptop     ; boucle arrière vers .looptop (forme alternative)
```

Voici un exemple en utilisant des labels non délimités :

```
JZ > L10           ; saut en avant à L10
JNC L3             ; saut en arrière à L3
JNC < L3           ; saut en arrière à L3 (forme alternative)
JMP 100            ; saut en l'arrière à 100
```

3.6.2 Sauts vers des labels uniques

Ceux-ci sont traités différemment, selon que le saut est effectué ou non en utilisant un mnémonique de saut conditionnel.

3.6.3 Sauts conditionnels à des labels uniques

Vous pouvez coder des sauts *conditionnels* à des labels uniques de la même manière que vous le feriez pour des sauts à des labels de portée locale ou non-délimités. En d'autres termes, utilisez l'indicateur vers l'avant ">" si le saut est plus avant dans le script source. En option, vous pouvez utiliser l'indicateur vers l'arrière "<" pour signifier que le saut est à un lieu en amont dans le script source, ou vous pouvez l'omettre. Fondamentalement, GoAsm vous permettra de ne pas sauter d'un fichier en utilisant un saut conditionnel. Ainsi, au lieu de coder :

```
JZ EXTERNALLABEL
```

vous pourriez écrire

```
JNZ >
JMP EXTERNALLABEL
:
```

Ceci, pour faciliter la vérification des erreurs. GoAsm suppose qu'un saut conditionnel est censé aboutir à un endroit à l'intérieur du script source existant.

3.6.4 Sauts inconditionnels à destination de labels uniques

Vous pouvez utiliser un indicateur de direction pour ces sauts si vous le souhaitez, mais vous n'y êtes pas contraint. L'indicateur de direction ne fera que dire à GoAsm de rechercher le label dans le script source. GoAsm ne dira pas au linker de chercher le label dans d'autres scripts source. Si vous n'utilisez pas d'indicateur de direction, GoAsm va néanmoins trouver le label s'il existe dans le script source, mais, si tel n'est pas le cas, il va dire au linker de le rechercher dans d'autres scripts source. Par exemple :

```
JMP LABEL                ; cherche le label dans tous les scripts source
JMP < INTERNALLABEL1     ; ne cherche le label qu'en amont dans le script source
JMP> INTERNALLABEL2      ; ne cherche le label qu'en aval dans le script source
```

3.6.5 Sauts vers les 2 points

La ponctuation consistant en deux points isolés est traitée comme un label non délimité et peut être utilisée pour vos sauts les moins significatifs, par exemple :

```
:
CALL PROCESS
LOOPZ <
```

ou

```
CMP EAX,EDX
JZ >
CALL PROCESS
:
RET
```

3.6.6 L'intérêt de sauts longs ou courts

Un saut court utilise un mécanisme de déplacement relatif qui tient seulement sur 2 octets. Il invite le processeur à revenir en arrière ou à aller vers l'avant avec une amplitude de +127 octets ou –128 octets par rapport à la position courante. L'amplitude du saut est contenue dans le deuxième octet de l'*opcode*, ce qui en explique la limitation au valeurs précédemment indiquées.
Pour surmonter cette contrainte, il existe une variante de cette instruction contenant 6 octets. Il s'agit de la forme longue de l'instruction de saut relatif.
L'utilisation de sauts courts non seulement resserre votre code mais en augmente également la vitesse d'exécution parce que le processeur doit lire et exécuter moins d'octets. Cette considération pourra être déterminante dans les structures d'instructions en boucle qui sont exécutés plusieurs fois.

3.6.7 Comment forcer GoAsm à coder un saut long

Utilisez soit l'opérateur LONG, soit << ou >>. Par exemple :

```
JZ >>.fin          ; long saut avant vers .fin
JZ LONG >.fin      ; long saut avant vers .fin (variante)
JC <<A1            ; long saut arrière vers A1
```

```
JC LONG A1          ; long saut arrière vers A1 (variante)
JC LONG <A1         ; long saut arrière vers A1 (variante)
```

Notez qu'il n'y a aucune forme longue de l'instruction LOOP et de ses variantes, ni de JECXZ. Si vous avez besoin d'un saut long de ces instructions utiliser à la place :

```
DEC ECX
JNZ LONG L2         ; saut long remplaçant LOOP
OR ECX,ECX          ; test de ECX = 0
JZ LONG >L44        ; saut long remplaçant JECXZ
```

3.6.8 Principes de codage des sauts longs ou courts

GoAsm essaie toujours de générer le plus petit code possible, en cohérence avec le fait qu'il s'agit d'un assembleur en une passe. Voici les règles observées :

- GoAsm codera toujours un saut long si c'est spécifié (pour les instructions qui admettent cette possibilité).
- Pour les sauts en amont vers des labels uniques et portée locale, GoAsm codera automatiquement un saut court si c'est possible, sinon, à défaut, un saut long.
- Pour les sauts en amont vers des labels non délimités, GoAsm codera un saut court.
- Pour les sauts en aval vers des labels non délimités et aussi pour des labels de portée locale, GoAsm codera un saut court.
- Pour les sauts en aval vers des labels uniques, GoAsm va coder un saut long.

GoAsm affichera une erreur si un court saut est spécifié, mais ne peut être atteint. Ceci, pour vous assurer que vous n'avez pas commis d'erreur dans votre script source. Par exemple, vous pourriez avoir codé un saut court tout en oubliant d'ajouter la destination du saut à votre script source.

3.7 Accès aux labels

3.7.1 Obtention de l'adresse d'un label (ADDR et OFFSET)

Les opérateurs ADDR et OFFSET permettent d'obtenir l'adresse d'un label. Dans l'exécutable final sous Windows, ils fournissent la distance du label par rapport au début de la section, ainsi que la position de la section dans la mémoire virtuelle. En d'autres termes, c'est l'adresse du label en mémoire lorsque l'exécutable est chargé et s'exécute.

Voici des exemples d'utilisation de **labels uniques** :

```
MOV ESI,ADDR Process_dabs     ; ESI = adresse de code du label Process_dabs
MOV ESI,ADDR Hello2           ; ESI = adresse de la chaîne avec le label Hello2
MOV ESI,ADDR HelloX+10h       ; ESI = adresse 16 octets au-delà du label HelloX
```

Exemple utilisant un **label réutilisable de portée locale** :

```
MOV ESI,ADDR CALCULATE.fin    ; ESI = adresse de code du label .fin
                              ; dans la procédure CALCULATE
```

Exemple utilisant une **structure formelle** :

```
MOV ESI,ADDR Lv1.pszText      ; ESI = adresse du membre psztext dans
                              ; la structure formelle Lv1
```

Pour le code 64 bits, noter qu'un PUSH, ARG, ou MOV vers la mémoire d'un ADDR ou d'un OFFSET concernant un label non-local (les labels locaux sont gérées différemment) ***fera usage du registre R11*** *et profitera de l'adressage relatif RIP plus court de l'instruction LEA de la manière suivante :*

```
LEA R11,ADDR Non_Local_Label
PUSH R11

LEA R11,ADDR Non_Local_Label
MOV [MEMORY64],R11
```

Ce sera également le cas avec INVOKE en passant les arguments avec ADDR, ***qui comprend également l'utilisation de pointeurs vers une chaîne ou une donnée brute*** *(ex. 'Bonjour' ou <'H', 'i', 0>).*

3.7.2 Lecture de données à partir de l'emplacement pointé par un label

La lecture des données à partir de l'endroit pointé par un label est tout à fait différente de l'action consistant à obtenir l'adresse du même label. Ici vous lisez la valeur de données dans le domaine de la mémoire concernée. Cela doit être fait à l'aide des crochets. Exemples :

```
MOV ESI,ADDR Hello1      ; ESI = adresse du label Hello1
MOV EAX,[ESI]            ; EAX = valeur mémoire à l'emplacement du label Hello1
```

ou, ce qui revient au même :

```
MOV EAX,[Hello1]         ; EAX = valeur mémoire à l'emplacement du label Hello1
```

3.7.3 Écriture à l'emplacement pointé par un label

Ici, vous provoquez une écriture de donnée comme suit :

```
MOV ESI,ADDR Hello1      ; ESI = adresse du label Hello1
MOV [ESI],EAX            ; contenu de EAX dans la mémoire dword correspondant au label Hello1
```

ou ce qui revient au même :

```
MOV [Hello1],EAX         ; contenu de EAX dans la mémoire dword correspondant au label Hello1
```

3.7.4 Lecture et écriture sur des labels utilisant un déplacement

Supposons que vous ayez une structure simple de données déclarée comme suit :

```
PARAM_DATA  DD 0      ; +0h
            DD 0      ; +4h
            DD 55h    ; +8h
            DD 0      ; +0Ch
            DD 0      ; +10h
```

Vous pouvez utiliser le label pour lire et écrire dans une partie particulière de la structure en utilisant une valeur de déplacement conformément à l'exemple suivant :

```
MOV ESI,ADDR PARAM_DATA      ; ESI = offset de PARAM_DATA
MOV EAX,[ESI+8h]             ; EAX = lecture du troisième DWORD de PARAM_DATA
MOV [ESI+8h],EDX             ; on remplace ce 3ème DWORD par le contenu de EDX
```

ou ce qui fait la même chose :

```
MOV EAX,[PARAM_DATA+8h]      ; EAX = lecture du troisième DWORD de PARAM_DATA
MOV [PARAM_DATA+8h],EDX      ; on remplace ce 3ème DWORD par le contenu de EDX
```

La valeur de déplacement peut prendre toute valeur jusqu'à 0FFFFFFFFh. Elle peut être positive ou négative. Les éléments non numériques doivent être séparés par le signe plus.
Voir la section "**Structures – différents types et utilisation**".

3.7.5 Lecture et écriture sur des labels utilisant l'indexation

Supposons que vous ayez 16 dword de données déclarés comme suit :

```
PARAM_DATA DD 10h DUP 0
```

Vous pouvez utiliser l'indexation (scaling) en complément du registre d'index comme suit :

```
MOV ESI,ADDR PARAM_DATA
MOV EAX,[ESI+ECX*4]              ; EAX = lecture du dword pointé par ESI et indexé par ECX
MOV [ESI+ECX*4],EDX              ; insertion de EDX en remplacement du dword lu précédemment
```

ou, ce qui revient au même :

```
MOV EAX,[PARAM_DATA+ECX*4]       ; EAX = lecture du dword pointé par ESI et indexé par ECX
MOV [PARAM_DATA+ECX*4],EDX       ; insertion de EDX en remplacement du dword lu précédemment
```

Vous pouvez utiliser une indexation de 0, 2, 4 ou 8. Les instructions qui suivent sont toutes valides :

```
MOVZX EAX,B[PARAM_DATA+ECX]      ; octet pointé par [PARAM_DATA+ECX] avec ext. à 0 sur EAX
MOVZX EAX,W[PARAM_DATA+ECX*2]    ; mot pointé par [PARAM_DATA+ECX*2] avec ext. à 0 sur EAX
MOV Q[PARAM_DATA+ECX*8],EDX      ; insertion EDX dans Qword pointé par [PARAM_DATA+ECX*8]
```

Les lettres B, W et Q précédant le crochet ouvrant sont des abréviations spécifiques à GoAsm décrivant respectivement les modificateurs de type *Byte Ptr, Word Ptr* et *Qword Ptr*.
Les éléments non numériques doivent être séparés par le signe plus.

Dans le codage 32 bits, seuls registres 32 bits à usage général peuvent être utilisés comme registre d'index – EAX, EBX, ECX, EDI, EDX, ESI ou EBP. Vous ne pouvez pas utiliser ESP en tant que registre d'index.
En codage 64 bits, vous pouvez utiliser les registres 32 bits à usage général ou les nouveaux registres en mode d'adressage 32 bits (R8D à R15D). Vous pouvez également utiliser les extensions 64 bits des registres à usage général – RAX, RBX, RCX, RDI, RDX, RSI, ou RBP –, et les nouveaux registres 64 bits R8 à R15. Vous ne pouvez pas utiliser RSP en tant que registre d'index. *Notez que les instructions ci-dessus qui utilisent PARAM_DATA et l'indexation n'utilisent pas l'adressage relatif RIP, de sorte que la base de l'image doit être bien en dessous 7FFFFFFFh.*

3.7.6 Lecture et écriture sur des labels utilisant indexation et déplacement

Supposons que vous ayez 24 dwords de données déclarés comme suit où le dernier dword dans chaque cas détient le résultat requis :

```
PARAM_DATA  DD 19h,0,0,22222h
            DD 1Ah,0,0,44444h
            DD 1Bh,0,0,66666h
            DD 1Ch,0,0,88888h
            DD 1Dh,0,0,0AAAAAh
            DD 1Eh,0,0,0CCCCCh
```

Alors, vous pouvez utiliser une indexation (scaling) et un déplacement de la manière suivante :

```
MOV ESI,ADDR PARAM_DATA         ; adresse de départ de la table de Dwords
CMP EAX,[ESI+ECX*4]             ; voir si EAX = dword pointé par cette table
JNZ >L2                         ; non
MOV EDX,[ESI+ECX*4+0Ch]         ; oui, alors on récupère le résultat dans EDX
```

ou, plus directement, et ce qui revient au même :

```
CMP EAX,[PARAM_DATA+ECX*4]      ; voir si EAX = dword pointé par cette table
JNZ >L2                         ; non
MOV EDX,[PARAM_DATA+ECX*4+0Ch] ; oui, alors on récupère le résultat dans EDX
```

Vous devez utiliser l'indexation au moyen des seules valeurs 0, 2, 4 ou 8. La valeur de déplacement peut être toute valeur allant jusqu'à 0FFFFFFFFh. Dans votre script source, cette valeur peut être positive ou négative. Les éléments non numériques doivent être séparés par le signe plus.

Dans le codage 32 bits, seuls les registres 32 bits à usage général peuvent être utilisés comme registres d'index – EAX, EBX, ECX, EDI, EDX, ESI ou EBP. Vous ne pouvez pas utiliser ESP en tant que registre d'index.

En codage 64 bits, vous pouvez utiliser les registres 32 bits à usage général ou les nouveaux registres en mode d'adressage 32 bits (R8D à R15D). Vous pouvez également utiliser les extensions 64 bits des registres à usage général – RAX, RBX, RCX, RDI, RDX, RSI, ou RBP – ainsi que les nouveaux registres 64 bits R8 à R15. Vous ne pouvez pas utiliser RSP en tant que registre d'index. *Notez que les instructions ci-dessus qui utilisent PARAM_DATA et l'indexation n'utilisent pas l'adressage relatif RIP, de sorte que la base de l'image doit être bien en dessous 7FFFFFFFh.*

3.8 Appel (ou Saut) à des procédures

3.8.1 Qu'est-ce qu'une "procédure" ?

Une procédure est une série d'instructions de code avec un label auquel l'exécution peut être transférée. Les procédures peuvent également être nommées "fonction", "routine" ou "sous-programme". Voici un exemple de procédure courte :

```
PROCESS_HASH:    ; label permettant d'atteindre la procédure
XOR EAX,EAX
MOV EDX,ESI
CALL PH23
MOV EDX,866h     ;retour de la procédure avec EDX = 866h
RET
```

3.8.2 Transfert de l'exécution à une procédure

Habituellement, l'exécution est transférée à la procédure par l'utilisation de l'instruction CALL. Celle-ci impose tout d'abord au processeur de pousser sur la pile (PUSH) la position dans le code juste après l'instruction CALL, puis de poursuivre l'exécution dans la procédure appelée. A la fin de la procédure, on trouve une instruction RET (abrégé de RETURN) qui invite le processeur à retirer de la pile (POP) la position dans le code immédiatement après le CALL mémorisée précédemment, à placer cette valeur dans le pointeur d'instructions EIP, puis à reprendre l'exécution à partir de ce point.
Exceptionnellement l'exécution peut être transférée à la procédure par l'utilisation de l'instruction JMP. A la fin de la procédure, on peut également rencontrer une autre instruction JMP, comme dans l'exemple qui suit :

```
PROCESS_HASH:
XOR EAX,EAX
MOV EDX,ESI
CALL PH23            ;transfère l'exécution à la procédure PH23 et retourne après celle-ci
MOV EDX,866h         ;retour de la procédure avec EDX = 866h
JMP >SOMEWHERE_ELSE
;
START:               ;point d'entrée de l'exécution
JMP PROCESS_HASH
;
```

3.8.3 Syntaxe de CALL et JMP en direction d'une procédure

La manière habituelle de faire un CALL ou un JMP en direction d'une procédure est d'utiliser le label de code marquant le début de la procédure. Par exemple :

```
CALL PROCESS_HASH
JMP PROCESS_HASH
```

Parfois, l'adresse de la procédure destination peut être conservée en mémoire, pointée par un label ou un registre ou même localisée à un endroit connu dans la mémoire ainsi que l'illustrent les différents exemples qui suivent :

```
CALL [PROCADDRESS]
CALL [PROCTABLE+20h]
CALL [ESI]
CALL [ESI+EDX]
JMP [4000000h]
```

Il peut arriver enfin que l'adresse de la procédure destination soit détenue par un registre auquel cas la syntaxe du CALL ou du JMP peuvent prendre la forme suivante :

```
CALL EAX
JMP EDI
```

3.8.4 Syntaxes plus complexes du CALL et du JMP

Nous espérons que vous n'aurez jamais à utiliser l'une des formes qui suivent, mais GoAsm les autorise néanmoins (en utilisant soit CALL, soit JMP) :

```
#define Hello PROCESS_HASH
CALL Hello            ; traité comme un CALL à PROCESS_HASH
CALL 100h             ; traité comme un CALL à une adresse relative
CALL [HELLO3+ECX+EDX*4]
CALL [HELLO3+ECX+EDX*4+9000h]
CALL $$               ; CALL au départ du début de la section courante
CALL $+20h            ; CALL 20h octets plus loin que la position courante
```

3.8.5 CALL et JMP vers des procédures en dehors du fichier objet ou de la section

Certains assembleurs vous obligent à signaler dans votre script source au moyen de la directive EXTRN que la destination d'un appel est quelque part en dehors du fichier objet. Ils imposent également que la même destination soit marquée comme GLOBAL ou PUBLIC. GoAsm vous dispense de ces subtilités de

syntaxe en considérant que, si la destination d'un l'appel se révèle introuvable lors de l'assemblage, elle est supposée relever d'un appel externe. Aussi, tous les labels qui ne sont pas locaux ou qui ont des noms réutilisables sont supposés être "globaux". GoAsm fonctionne de la même manière lorsqu'un appel ou un saut doit être effectué à une section de code avec un autre nom.

Donc, si vous voulez appeler une procédure dans un autre script source (lequel produira un autre fichier objet) appelez-la simplement de la manière habituelle. De même, si vous avez une procédure dans un autre exécutable (généralement une DLL), vous pouvez procéder de même.

Par exemple, supposons que vous ayez écrit My.Dll incluant un algorithme de calcul que vous souhaitez utiliser avec le label CALCULATE. On pourrait l'appeler comme suit :

```
CALL CALCULATE
```

Dans votre liste des DLL que vous donnerez à GoLink, vous mentionnerez My.Dll. GoLink cherchera d'abord le label de code CALCULATE dans les fichiers objets, mais regardera ensuite dans les DLL spécifiées. La plupart des autres linkers regardent dans les fichiers de bibliothèque (fichiers .lib) pour les fonctions qu'ils contiennent, ce qui signifie que vous avez à constituer un fichier lib. De toute façon, si l'on s'en tient à la syntaxe GoAsm, vous n'avez plus rien à faire dans votre script source. Si l'éditeur de liens ne trouve pas la destination de l'appel, une erreur sera affichée.

Cette forme de l'appel est un appel relatif utilisant l'*opcode* E8.

Vous pouvez également utiliser cette forme :

```
CALL [CALCULATE]
```

Pour ce type d'appel, GoAsm utilise les *opcodes* FF15. Il s'agit d'un appel à une adresse absolue. Dans l'assembleur 32 bits, c'est un appel à une adresse de 32 bits, mais dans l'assembleur 64 bits, c'est un appel à une adresse de 64 bits.

Voir aussi :

bibliothèques de code statique
import direct par ordinal ou par Dll spécifique
utilisation de la librairie C Runtime

3.9 Appel des APIs Windows 32 et 64 bits

L'appel des API Windows (qui résident dans les Dlls système de Windows) est très simple dans le cas où aucun paramètre n'est requis. Par exemple dans Windows 32 bits, vous pouvez écrire :

```
CALL GetModuleHandle
```

ou sa variante plus évoluée qui peut être utilisée soit pour Windows 32 bits ou 64 bits :

```
INVOKE GetModuleHandle
```

Il n'y a rien d'autre à mettre dans le script source. Dans la mesure où la fonction appelée réside en dehors de l'exécutable que vous élaborez, il revient à l'éditeur de liens de trouver la DLL qui contient la procédure *GetModuleHandle* et il va y enregistrer le nom de la DLL à cet effet. GoLink effectuera les recherches nécessaires au moyen de la liste des DLL que vous fournissez.

La plupart des API Windows, cependant, attendent des paramètres (souvent désignés également comme «arguments») lorsqu'elles sont appelées. Il incombe au programmeur de s'assurer que ces paramètres sont envoyés à l'API correctement. Ils contiennent les informations, ou des pointeurs vers des informations, qui indiquent à l'API ce qu'elle doit faire. Parfois, ils contiennent des adresses de zone mémoire où l'API doit insérer des informations.

La manière de communiquer les paramètres à l'API varie selon que vous assembliez en Windows 32 ou 64 bits. Chaque système d'exploitation utilise en effet des conventions d'appels spécifiques qui affectent la façon dont les paramètres sont envoyés et utilisés. Windows 32 bits utilise la convention d'appel standard (STDCALL) et Windows 64 bits utilise la convention d'appel qualifiée de "rapide" (FASTCALL).

GoAsm propose, dans ce but, les opérateurs ARG et INVOKE qui peuvent être utilisés indifféremment sur les plates-formes 32 ou 64 bits. L'assembleur génère le code approprié selon la convention d'appel utilisée. Si vous écrivez pour 32 bits avec aucune velléité de transposition en 64 bits, vous pouvez utiliser PUSH et CALL pour la transmission des paramètres, mais si vous voulez garantir la portabilité de votre code vers Windows 64 bits ultérieurement, vous devrez les remplacer par ARG et INVOKE. Dans les code source 32

et 64 bits vous êtes libre d'utiliser CALL pour appeler des procédures dans vos propres exécutables, à moins que vous ne leur envoyiez les paramètres selon l'une des conventions d'appel suivantes :

Dans la convention d'appel STDCALL utilisée dans Windows 32 bits, tous les paramètres sont mis sur la pile par l'appelant, et le pointeur de pile (ESP) est déplacé vers le haut des paramètres sur la pile. Ensuite, l'API est appelée. Celle-ci utilise les paramètres sur la pile et avant de revenir, restaure cette dernière à l'équilibre en déplaçant le pointeur de la pile à la position qu'il avait avant la mise sur la pile du premier paramètre.

Dans la convention d'appel FASTCALL utilisée dans Windows 64 bits, les quatre premiers paramètres sont chargés successivement dans les registres RCX, RDX, R8 et R9 au lieu d'être mis sur la pile. Cependant, les éventuels paramètres suivants sont mis sur la pile. L'appelant doit faire en sorte que le pointeur de la pile (dans ce cas RSP) soit déplacé vers le haut des paramètres comme d'habitude, incluant notamment les quatre premiers paramètres qui sont détenus dans des registres (ce qui revient à autoriser l'API de les récupérer sur la pile comme s'ils avaient été mis là en premier lieu). Une autre différence est que l'API ne restaure pas la pile à l'équilibre avant de revenir de l'appel. Cette particularité facilite les choses pour les quelques API qui ne disposent pas d'un nombre fixe de paramètres.

Si vous désirez que le même script source puisse indifféremment être assemblé en 32 ou 64 bits, il est indispensable que vous envoyiez les paramètres à l'aide de ARG puis que vous appeliez l'API en utilisant INVOKE. Un exemple simple en montre le principe :

```
ARG 40h,RDX,RAX,[hwnd]
INVOKE MessageBoxA
```

Lors de l'assemblage 32 bits, ARG agit de manière identique à PUSH, et INVOKE produit le même effet que CALL. GoAsm accepte une instruction PUSH d'un registre à usage général 64 bits, et donc PUSH RDX est traité de la même manière que PUSH EDX au nombre de bits près, bien évidemment. Par conséquent, l'appel ci-dessus fonctionne sur les deux plateformes mais reçoit une traduction différenciée selon le cas :

Plateforme 32 bits

```
PUSH    40h
PUSH    EDX
PUSH    EAX
PUSH    [hwnd]
CALL    MessageBoxA
```

Plateforme 64 bits

```
MOV    R9,40h
MOV    R8,RDX
MOV    RDX,RAX
MOV    RCX,[hwnd]
SUB    RSP,20h
CALL   MessageBoxA
ADD    RSP,20h
```

L'assemblage 64 bits, produit, comme on peut le voir, un résultat radicalement différent à partir du même code.
Voir le chapitre **Programmation en 64 bits** pour plus de détails.

3.9.1 Appel des APIs Windows – Utilisation de INVOKE

Il est important évidemment d'envoyer les paramètres à l'API dans le bon ordre. INVOKE vous aide à le faire en vous permettant de mettre les paramètres après le nom de l'API comme en C. Cela est appréciable aussi lorsque vous travaillez avec la documentation de Windows qui décrit toujours les paramètres des API en utilisant la syntaxe du C. Par exemple, voici comment l'API *MessageBox* y est décrite :

```
int   MessageBox(
      HWND hwnd,            // handle of owner window
      LPCTSTR lpText,       // address of text in message box
      LPCTSTR lpCaption,    // address of title of message box
      UINT uType            // style of message box
      );
```

Avec INVOKE vous pouvez respecter le même ordre, par exemple :

```
INVOKE MessageBoxA, [hwnd],EAX,EDX,40h
```

qui équivaut à :

```
ARG 40h,RDX,RAX,[hwnd]
INVOKE MessageBoxA
```

Notez que ARG (comme PUSH) lit les paramètres d'une façon, tandis que les paramètres après INVOKE sont lus dans l'autre sens.

INVOKE et ses nombreux paramètres peuvent être écrits sur plusieurs lignes en utilisant le caractère de continuation :

```
INVOKE CreateWindowExA,   WS_EX_OVERLAPPEDWINDOW, ADDR szClassName, \
                          ADDR szWindowName,\
                          WS_OVERLAPPEDWINDOW+THING,\
                          100,16,400,0,0,0,[hInstance],0
```

Puisque GoAsm considère les paramètres de INVOKE à partir de la fin, les erreurs vers la fin seront trouvées en premier.

Lors de l'utilisation de INVOKE, si vous souhaitez ranger vos paramètres dans un mot défini, alors GoAsm les récupèrera toujours dans le bon ordre, par exemple :

```
z_function_params=3,2,1
INVOKE z_function, z_function_params
```

produit le même code que :

```
ARG 1,2,3
INVOKE z_function
```

3.9.2 Appel des APIs Windows – Versions ANSI et Unicode

Les API Windows qui gèrent des entrées ou sorties de caractères (généralement sous forme de chaînes de caractères) ont généralement deux versions différentes : une version ANSI et une version Unicode. La version ANSI gère les chaînes en ANSI, standard selon lequel un seul octet de valeur 0 à 255 représente un caractère unique basé sur le jeu de caractères courant. Ces caractères sont parfois appelés caractères "multi-octets". De manière on ne peut plus simple, le nom de la version ANSI de l'API se termine par un "A" comme dans l'exemple de l'API *CreateWindowEx* utilisée ci-dessus. La version Unicode gère les chaînes en format Unicode et utilise à cet effet deux octets par caractère sur la base de la table standard des caractères Unicode. Ceux-ci sont parfois qualifiés de caractères "larges" (wide). La version Unicode de l'API se terminera logiquement par un "W" à la fin de son nom.

Dans votre script source, vous devez spécifier les API que vous souhaitez appeler en ajoutant un A ou un W à la fin du nom de l'API, selon le cas. Lorsque vous soumettez votre fichier objet au linker et que ce dernier aura été incapable de trouver l'API dans un autre exécutable (ou dans les fichiers .lib si vous n'utilisez pas GoLink) c'est probablement parce que vous aurez oublié d'ajouter le A ou le W. Vous pourriez également avoir omis de fournir à GoLink le nom de la DLL détenant l'API (ou les fichiers .lib appropriés si vous n'utilisez pas GoLink).

Si vous voulez automatiser le bon appel d'API « A » ou « W », les règles suivantes doivent être observées :

- le programme est en version ANSI sauf spécification contraire,
- la version Unicode peut être obtenue, soit en mettant le commutateur /d sur la ligne de commande de GoAsm, soit en écrivant la ligne `#define UNICODE` au début du script source.

Voir le chapitre relatif à l'écriture de programmes Unicode dans le volume 2 pour de plus amples informations sur le sujet.

Il n'y a aucune différence entre l'assemblage 32 et 64 bits à cet égard pour la simple raison que Windows 64 bits a des versions ANSI et Unicode des API tout comme Windows 32 bits.

3.10 Les pointeurs de chaînes et de données avec PUSH et ARG

3.10.1 Pointeurs sur des chaînes terminées par un zéro

GoAsm permet une extension de PUSH ou ARG qui est très utile en programmation sous Windows. Souvent, dans ce cas, vous êtes dans la situation d'envoyer à une API un paramètre qui est un pointeur vers une chaîne se terminant par zéro. Supposons, par exemple, l'appel d'API suivant (en 32 bits) :

```
MBTITLE     DB 'Hello',0
MBMESSAGE   DB 'Click OK',0
PUSH 40h,ADDR MBTITLE,ADDR MBMESSAGE,[hwnd]
CALL MessageBoxA
```

Pour simplifier cette syntaxe, GoAsm permet d'utiliser PUSH ou ARG comme suit tout en garantissant un résultat identique :

```
PUSH 40h,'Hello','Click OK',[hwnd]
CALL MessageBoxA
```

ou, si vous écrivez un script source permettant la compatibilité entre les plates-formes 32 et 64 bits :

```
ARG 40h,'Hello','Click OK',[hwnd]
INVOKE MessageBoxA
```

et, si vous préférez envoyer les paramètres par INVOKE :

```
INVOKE MessageBoxA, [hwnd],'Click OK','Hello',40h
```

Vous pouvez enfin utiliser cette même facilité avec des chaînes Unicode comme suit :

```
ARG 40h,L'Hello',L'Click OK',[hwnd]
INVOKE MessageBoxW
```

ou

```
INVOKE MessageBoxW, [hwnd],L'Click OK',L'Hello',40h
```

Lorsque vous utilisez l'une de ces formes, la chaîne sera toujours terminée par zéro. Il se trouve en effet que GoAsm place la chaîne dans la section const. si il y en a une (la section data s'il y en a une, sinon, à défaut, dans la section de code) et y ajoute un terminateur null. Puis, GoAsm crée l'instruction correcte et lui donne un pointeur vers la chaîne. Aucun symbole n'est produit à des fins de débogage.

En assemblage 64 bits, GoAsm garantit que les chaînes Unicode sont alignées sur la limite de mot requise par le système. *Notez que ceci est similaire à PUSH ADDR et* ***fera usage du registre R11*** *tout en tirant avantage de la taille réduite de l'adressage relatif RIP de l'instruction LEA.*

3.10.2 Mise en pile de pointeurs dans des données brutes

Vous pouvez faire la même chose que précédemment avec les données brutes ordinaires (en octets) en utilisant les opérateurs < et >. Par exemple :

```
PUSH <23,24,25>                 ; mise en pile d'un pointeur des octets 23,24,25
```

ou

```
PUSH <23,6 DUP 20h,23>          ; mise en pile d'un pointeur des octets 23,6 espaces, puis 23
```

ou

```
PUSH <'Hi',0Dh,0Ah,'There',0>   ; mise en pile d'un pointeur sur la chaîne terminée
                                ; par un zéro sur 2 lignes (RC+LF au milieu)
```

On peut également utiliser les opérateurs < et > de cette manière avec ARG et après INVOKE. Dans ce cas de figure, GoAsm place la déclaration des données entre les opérateurs < et > dans la section const. s'il y en a une (ou la section data s'il y en a une, sinon à défaut, dans la section de code). Puis GoAsm crée l'instruction appropriée et calcule un pointeur vers les données. Aucun symbole n'est constitué à des fins de débogage.
Notez que lorsque vous utilisez les opérateurs < et > de cette manière, aucun terminateur Null n'est ajouté aux chaînes.

Lors de l'assemblage 64 bits, GoAsm garantit que les données sont alignées sur une limite de mot rendue nécessaire par le système si les données contiennent des chaînes Unicode.

3.11 Mémorisation de pointeurs de chaîne et données dans des registres

Vous pouvez également déclarer des chaines terminées par un zéro et des données tout en établissant simultanément leurs pointeurs dans des registres à l'aide de la syntaxe suivante :

```
MOV EAX,ADDR 'This is a string'
MOV EAX,ADDR <'String',0Dh,0Ah>
```

Lorsque GoAsm traite ce type de code, il constitue une chaîne terminée par un zéro – sans que ce dernier ne soit spécifié – ou les données entre les opérateurs < et > dans la section *const.* s'il y en a une (ou la section *data* si elle existe, sinon, dans la section *code*). Puis, GoAsm fournit le pointeur des données ainsi créées à l'instruction. Aucun symbole n'est produit à des fins de débogage.

Notez que lorsque vous utilisez les opérateurs < et >, aucun terminateur Null n'est rajouté aux chaînes.

Notez également comment ceci diffère de la syntaxe relative au chargement de caractères immédiats dans un registre. Cette différence réside dans l'utilisation de l'opérateur ADDR.

Tout ceci fonctionne de la même façon en programmation 64 bits, sauf que GoAsm garantit qu'une chaîne ou une donnée Unicode sont alignées WORD dans la mémoire ainsi que le requiert le système. *Notez que ceci est similaire à PUSH ADDR et* ***fera usage du registre R11*** *tout en tirant avantage de la taille réduite de l'adressage relatif RIP de l'instruction LEA.*

3.12 Utilisation de caractères immédiats dans le code

GoAsm *ne* renverse *pas* l'ordre des word et dword mémorisés sous forme de caractères immédiats comme MASM le pratique. Ainsi, par exemple, GoAsm utilise-t-il le format suivant :

```
MOV AL,'1'
MOV AX,'12'          ; considéré comme des octets - 1 en AL, 2 en AH
MOV EAX,'ABCD'       ; considéré comme des octets - A en premier, puis B puis C puis D (AL)
```

Cela facilite l'ajout des chaînes courtes à la mémoire. Par exemple, pour ajouter l'extension *.fil* à un nom de fichier stocké en mémoire, vous pouvez coder :

```
MOV [EDI],'.fil'     ;ou
MOV EAX,'.fil'
MOV [EDI],EAX
```

et non pas :

```
MOV [EDI],'lif.'     ;ou
MOV EAX,'lif.'
MOV [EDI],EAX
```

L'instruction CMP (comparaison) travaille de la même manière :

```
CMP AL,'1'
CMP EAX,'ABCD'
CMP [EDI],'.fil'
```

Cela ne change pas l'ordre inverse habituel dans les opérandes qui ne seraient pas entre guillemets. Ainsi, par exemple lorsque vous souhaitez ajouter un retour chariot, puis un saut de ligne au texte que vous souhaitez réutiliser :

```
MOV AX,0A0Dh
STOSW
```

Ici, le retour chariot (0Dh) qui est en AL, est chargé le premier en mémoire, puis le saut de ligne (0Ah) dans AH est chargé en mémoire d'adresse immédiatement supérieure.

Si la chaîne est plus courte que le type du registre ou de la mémoire, des zéros absolus de comblement sont ajoutés automatiquement :

```
MOV EAX,'ABC'        ; code A, puis B, puis C, puis zéro
```

Lors de l'écriture du code source pour les programmes Unicode vous pouvez garantir que les caractères immédiats sont Unicode ou, si nécessaire, basculés d'ANSI en Unicode. Voir, à ce sujet, les sections “Utilisation de la chaîne correcte dans les valeurs immédiates sous guillemets” et “commutation des chaînes sous guillemets et immédiates” dans le chapitre du volume 2 consacré à l'Unicode.

En programmation 64 bits, vous pouvez utiliser les registres 64 bits pour y stocker des chaînes immédiates de 8 caractères de long, par exemple :

```
MOV RAX,'Saturday'
```

Cependant, l'instruction CMP n'admet pas les valeurs immédiates supérieures à 32 bits, de sorte que par exemple

```
CMP RAX,'Saturday'
```

affiche une erreur. En mode 64 bits, seules sont autorisées les comparaisons entre registre et mémoire et entre deux registres sur cette variante de l'instruction CMP.

3.13 Indicateurs de Type

3.13.1 Intérêt et syntaxe

Penchons-nous sur l'instruction

```
MOV [ESI],20h
```

Elle consiste à stocker le nombre 20h à l'adresse mémoire spécifiée par le contenu du registre ESI. Mais une question importante apparaît immédiatement. Le nombre doit-il être chargé comme un octet, un mot, un dword ou tout autre type de donnée ? De la réponse dépend le nombre d'octets de mémoire – à partir de l'adresse fournie par ESI – qui doivent être modifiés. Tous les assembleurs exigent donc à cet effet un indicateur de type pour lever l'ambiguïté mais avec une syntaxe sensiblement différente ainsi que le montrent les exemples suivants (en utilisant dword comme exemple):

```
MOV DWORD PTR [ESI],20h     ; MASM
MOV DWORD [ESI],20h         ; NASM
MOV D[ESI],20h              ; A386
```

GoAsm utilise la syntaxe A386 qui nécessite beaucoup moins de frappe de sorte que les indicateurs de type rencontrés sont :

B	signifie byte	(1 octet)
W	signifie word	(2 octets)
D	signifie dword	(4 octets)
Q	signifie qword	(8 octets)
T	signifie tword	(10 octets)

Vous pouvez également utiliser ces deux indicateurs de type commutable :

- **S** est un indicateur dont la taille varie selon les caractères utilisés : 1 (octet) en ANSI et 2 (octets) en Unicode.

  ```
  MOV S [EDI],0 ; insère un simple zéro si ANSI, un double zéro si Unicode.
  ```

  Voir la section “Utilisation de l'indicateur de type commutable pour Unicode/ANSI” dans le chapitre du volume 2 consacré à l'Unicode.

- **P** est un indicateur dont la taille varie selon le format 32/64 bits (par défaut, 4 pour 32 bits, 8 pour 64 bits)

  ```
  MOV P[RDI],0  ; 0 sur qword à RDI si 64 bits, 0 sur dword à EDI si 32 bits
  ```

  Voir la section **Utilisation de l'indicateur de type commutable pour 32/64 bits**

3.13.2 L'indicateur de type est également requis pour les références mémoire nominatives

Comme NASM, GoAsm ne vérifie pas les types, de sorte qu'il ne peut connaître la taille d'une opération telle que celle-ci :

```
INC [COUNT]
```

Ici, GoAsm ignore tout de la taille de COUNT qui peut être indifféremment un octet, un mot (word), un double-mot (dword) ou un quadruple mot (qword) et ceci, bien que la variable ait été préalablement déclarée. Par conséquent, vous devez impérativement préciser le type de l'opération, comme par exemple :

```
INC B[COUNT]
```

Bien que cette obligation impose un peu plus de travail au programmeur, force est de reconnaitre que le script source s'en trouve plus facile à lire et à comprendre, puisque vous pouvez immédiatement voir la taille de l'opération de l'instruction, plutôt que de devoir explorer l'ensemble du programme pour vous enquérir de la taille avec laquelle COUNT a été déclarée. Du reste, l'utilisation d'indications de type tenant en une seule lettre compense le côté fastidieux induit par la répétition de ces indications.

3.13.3 Quelles instructions requièrent un indicateur de type ?

Généralement toutes les instructions où la taille de l'opération n'est pas implicite sont concernées. Certains des exemples qui suivent utilisent des références de mémoire nominatives alors que d'autres préfèrent des références de mémoire soutenues par des registres :

```
AND B[MAINFLAG],0FEh
ADC W[EAX],66h
ADD D[MEM_AREA],66h
BT D[EBX],31D
CMP D[HELLOWORD],0Dh
DEC D[ECX]
DIV B[HELLO]
INC D[EDX]
MOV B[MEM_AREA],23h
MOVSX EDX,B[EDI]
MUL B[HELLO]
NEG W[ESI]
NOT D[HELLO3]
OR B[MAINFLAG],1h
SETZ B[BYTETEST]
SHL W[IAMAWORD],23h
SHL D[IAMADWORD],CL
SUB D[EBP+10h],20D
TEST B[ESP+4h],1h
XOR D[IMAWORD],11111111h
```

Et, en programmation 64 bits, vous pouvez également voir, par exemple

```
ADC W[RAX],66h
BT D[R12],31D
INC Q[RDX]
NEG W[R15D]
```

3.13.4 Les instructions qui ne requièrent pas d'indicateur de type

Entrent dans cette catégorie les opérations dont la taille de l'opération est évidente, du fait de l'utilisation d'un registre, par exemple :

```
AND [MAINFLAG],CL
CMP [HELLOWORD],EDI
MOV [IAMABYTE],AL
MOV [IAMADWORD],ESI
OR [MAINFLAG],BH
XCHG CL,[ESI]
```

De fait, aucune des instructions MMX, XMM ou 3DNow! ne nécessite un indicateur de type. Il en va de même pour plusieurs des instructions en virgule flottante x87. Les autres peuvent prendre plus d'une taille d'opérande. Il y a aussi plusieurs instructions qui n'acceptent qu'une taille d'opérande, de sorte qu'avec celles-ci, l'indicateur de type n'est pas requis. Il en va ainsi, par exemple, des instructions CALL, JMP, PUSH et POP qui ne supportent qu'un dword (en 32 bits). Voir cependant la section relative aux **demi-opérations de pile** concernant l'utilisation de PUSHW et POPW. Enfin, certaines des instructions les moins courantes n'ont pas besoin d'un indicateur de type, par exemple ARPL, BOUND, BSF, BSR, CMOV (sous toutes ses formes), CMPXCHG et CMPXCHG8B.

3.14 Instructions répétées

Les instructions répétées sont possibles pour PUSH, POP, INC, DEC, et bien sûr lors de la déclaration des données, par exemple :

```
PUSH 0,23h,[hwnd],ADDR lParam,EAX
POP EAX,[EBP+2Ch],[hwnd]
DEC ECX,EDX,[COUNT]
INC [EBP+10h],EDI
DB 23h,24h,25h
```

Les instructions ici sont toujours assemblés de gauche à droite.

3.15 Nombres et arithmétique

3.15.1 Nombres

La plupart des assembleurs utilisent la syntaxe suivante pour les nombres :

```
66ABCDEh    ; nombre hexadécimal
34567789    ; nombre décimal
1100011B    ; nombre binaire
1.0         ; nombre réel
```

```
1.0E0       ; nombre réel (autre forme)
```

GoAsm accepte ces nombres, mais prend également en charge les formats suivants :

```
9999999D    ; nombre décimal
0x456789    ; nombre hexadécimal
```

Un nombre hexadécimal qui commence par une lettre (A à F caractérisant, par convention, les valeurs décimales de 10 à 15) doit impérativement être précédé d'un zéro, par exemple :

```
0A789ABCDh
```

ou

```
0xA789ABCD
```

3.15.2 Arithmétique

GoAsm peut effectuer des opérations arithmétiques limitées à l'intérieur des déclarations de données, du quantitatif de DUP, des définitions, lors de la déclaration de définitions, lors de l'utilisation des définitions, et dans des opérandes des instructions de code. Vous n'êtes pas autorisé à utiliser le signe de multiplication (astérisque) à l'intérieur de crochets autrement que lors de l'utilisation d'un registre d'index.

Soyez prudent en utilisant les opérateurs logiques OR, AND et NOT puisque ceux-ci sont à la fois des mnémoniques processeur et des directives de l'assembleur. Bien que GoAsm sache les distinguer dans leurs deux acceptions, il vous est possible de recourir à une écriture distinctive en utilisant, pour les directives, les symboles | pour OR, & pour AND, et ! pour NOT.

Les opérations arithmétiques entre parenthèses sont effectuées en premier lieu, sinon les calculs sont effectués strictement de gauche à droite. Voici quelques exemples :

```
DB 2*3
DB (2+30h)/(2+1)
DD (2000h+40h-20h)/2
DD SIZEOF HELLO/2
DD 444444h & 226222h
DB 20h/2 DUP 44h
DB 6+2 DUP 0
#define globule (2*3)/2
DB globule
DD globule|100h
DD 2D00h>>8
DQ 2D00h<<48
```

```
MOV EAX,globule|100h
MOV EAX,SIZEOF HELLO*2
MOV EAX,ADDR HELLO+10h
MOV EAX,0x68+0x69-0x70
MOV EAX,[MemName+0x68+0x69-0x70]
MOV EAX,[ESI*4+45000h]
MOV EAX,[ESI*4+SIZEOF HELLO/2]
MOV EAX,8+8*2       ; résultat = 32
MOV EAX,8+(8*2)     ; résultat = 24
```

Le quotient des divisions, lorsque cette opération est utilisée, est arrondi selon la règle de l'entier le plus proche ainsi que le montrent les exemples suivants :

```
MOV EAX,32/3        ; met 11 dans EAX
MOV EAX,31/3        ; met 10 dans EAX
MOV EAX,10/4        ; met 3 dans EAX
```

GoAsm considère que les multiplications et divisions effectuées dans ce cadre le sont en utilisant des nombres *non signés*. MUL et DIV sont utilisées lors de la compilation et non leur homologues signés IMUL et IDIV.

3.15.3 Déclaration des nombres réels

Les nombres réels sont des nombres qui ont la capacité de représenter une valeur inférieure à 1. GoAsm attend des nombres réels présents dans le script source qu'ils prennent la forme d'un nombre à virgule flottante, techniquement composé de plusieurs chiffres et d'un point parmi ceux-ci conformément à la notation anglo-saxonne qui prévaut ici. Le point doit être représenté par le caractère correspondant au code ASCII 2Eh et peut se situer n'importe où pourvu qu'il soit unique et positionné entre 2 chiffres consécutifs. Le nombre réel peut, si nécessaire, être assorti d'un exposant décimal signé à la fin du nombre (en utilisant l'indicateur d'exposant "e" ou "E" conformément à la norme IEEE relative aux nombres en virgule flottante). Les registres x87 en virgule flottante du processeur peuvent accepter des nombres réels dans les résolutions 32, 64 ou 80 bits. Le 3DNow! et les instructions SSE travaillent avec des nombres réels 32 bits et les instructions SSE2 utilisent des nombres réels 64 bits.

Parfois, ces types sont nommés :

32 bits simple-précision
64 bits double-précision
80 bits double-précision étendue

Donc, les nombres réels peuvent être déclarées comme dwords (32 bits), qwords (64 bits) ou twords (80 bits). Voici quelques exemples de déclarations de données en nombres réels :

```
DD 1.6789E3
DQ 1.6789E3
DT 1.6789E3
DD 3 DUP 7.6789E-2
DQ 678.27896435E3
DT 1.2
```

Vous pouvez également déclarer directement le nombre PI soit en tant que tword, qword ou dword comme suit :

```
DD PI       ; pi comme un dword
DQ PI       ; pi comme un qword
DT PI       ; pi comme un tword
```

GoAsm tente d'obtenir une précision maximale dans la fourniture de PI en écrivant une valeur connue directement dans la mantisse.

Vous pouvez également déclarer un nombre réel comme suit :

```
PUSH 1.1
MOV EAX,1.1
```

Ces deux formes utilisent un format 32 bits pour le nombre réel. La première place ce nombre sur la pile et la seconde le place dans le registre spécifié.

3.15.4 Précision de conversion de GoAsm

GoAsm utilise des algorithmes spéciaux pour garantir une précision optimale lors du chargement de la déclaration de nombre réel en tant que donnée. Dans le cas de la déclaration d'un Tword (80 bits) le calcul est effectué si nécessaire sur un maximum de 92 bits, avant d'être arrondi puis inséré dans la mantisse 64 bits. La conversion en un qword (64 bits) est effectuée en utilisant la précision maximale disponible (53 bits de mantisse) avec un arrondi à la valeur la plus proche. La conversion en un dword (32 bits) est effectuée en utilisant la précision maximale disponible (24 bits de mantisse) avec un arrondi à la valeur la plus proche.

3.15.5 Chargement direct de l'exposant et de la mantisse

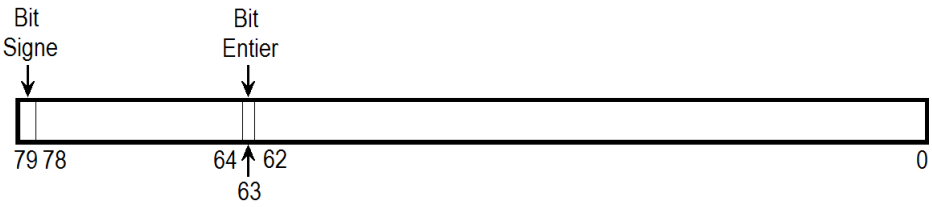
Au lieu d'utiliser des nombres réels pour charger les registres en virgule flottante, vous pouvez déclarer un Tword et charger la mantisse et l'exposant directement en utilisant l'instruction FLD. Pour ce faire, vous aurez besoin de connaître les valeurs de mantisse et d'exposant à charger (ceux-ci peuvent être soit calculés, soit trouvés et vérifiés en utilisant l'une des fonctions de simulation FPU de GoBug). Supposons, par exemple, que vous vouliez une représentation du nombre π aussi précise que possible et vous sachiez, par ailleurs, que cette représentation consiste en un exposant de +0002 et une mantisse de +C90FDAA22168C235h. Vous pouvez donc déclarer ce nombre par :

```
DIRECT_PI DT 4000C90FDAA22168C235h
```

et le charger en utilisant :

```
FLD T[DIRECT_PI]
```

Considérons maintenant la figure ci-après qui montre la structure d'un registre de données x87.



Le bit le plus significatif – bit 79 – dans la déclaration de Tword qui précède est le bit de signe indiquant que le nombre réel est positif ou négatif. Ici, le nombre est positif puisque ce bit est à zéro. Le reste des

quatre premiers chiffres hexadécimaux contient l'exposant qui est donc représenté sur 15 bits. Pour mieux faire tenir les valeurs d'exposant positives et négatives sur ce format, la valeur nulle est fixée à 3FFEh, ce qui autorise dès lors les exposants à évoluer entre –3FEEh et +4001h sans risquer d'atteindre le bit le plus significatif – le bit 79 évoqué plus haut. Le reste des chiffres hexadécimaux – 0 à 62 inclus – contient la mantisse.

Il est beaucoup plus difficile de charger la mantisse et l'exposant directement à partir de données déclarées en Dword et Qword. Ceci, parce que la répartition entre mantisse et exposant dans ce format numérique s'affranchit partiellement de la modularité 4 bits, base de la notation hexadécimale. Il résulte de cette structure particulière une certaine difficulté à chiffrer les nombres hexadécimaux à déclarer, outre le fait que l'exposant, pour ne rien simplifier, subit un codage qui accentue cette opacité.

3.16 Caractères dans GoAsm

3.16.1 Chaînes de caractères

Dans votre script source, vous serez souvent amené à représenter des caractères sous différentes formes. Par exemple :

```
Mess    DB 'I am a string of characters',0
PUSH    'This is supposed to be a carat ^'
MOV     EAX,'£$|@'
```

Il faut se demander quelles valeurs réelles sont chargées par GoAsm à l'issue du traitement de ces instructions ? Au moment de l'assemblage GoAsm consulte votre script de source en utilisant les tables de caractères Windows, puis le lit caractère par caractère. En d'autres termes, GoAsm se voit attribuer la valeur des caractères dans le script source par Windows. Lorsque GoAsm charge dans le fichier objet les chaînes de caractères décrites ci-dessus, il charge la même valeur de caractère qui lui est donnée par Windows. Dans le cas de conversion de chaînes ANSI en chaînes Unicode, ces derniers sont passés d'abord par l'API *MultiByteToWideChar*. Cela signifie que la valeur donnée à GoAsm par Windows correspondra au jeu de caractères courant (page de code). En conséquence, vous devez vous assurer que le jeu de caractères utilisé dans l'ordinateur qui supporte GoAsm est le jeu de caractères pour lesquels votre programme est conçu pour fonctionner.

Si vous utilisez un script source qui est dans un format Unicode (UTF-8 ou UTF-16), alors la question de la page de code disparaît. Les caractères corrects sont donnés par leur valeur Unicode.

3.16.2 Caractères spécifiés directement

Parfois, vous aurez à spécifier des caractères par leur valeur réelle dans la table courante pour pouvoir faire face à d'éventuelles variations de jeu de caractères. C'est par exemple le cas pour le symbole représentant l'opérateur logique OR par une barre verticale, lequel peut prendre 2 valeurs de code ASCII :

```
CMP AL,124D         ; est-ce que ce code est un OR dans ce jeu de caractères ?
JZ >L4              ; oui
CMP AL,221D         ; est-ce que ce code est un OR dans cet autre jeu de caractères ?
JZ >L4              ; oui
```

Cette formulation vous permet d'autoriser une possible variation du propre jeu de caractères de l'utilisateur. Si nécessaire, vous pouvez organiser votre code de telle sorte qu'il teste le jeu de caractères de l'utilisateur au moment de l'exécution, et qu'il teste les caractères ou utilise les chaînes corrigés en conséquence. Vous pouvez également tester la langue de la machine de l'utilisateur et fournir des chaînes dans la langue correcte. Les API de ressources offrent une solution à cet égard et cela peut être fait automatiquement – voir le manuel du compilateur de ressources GoRC dans le volume 2.

3.17 Opérateurs

Voici certains opérateurs qui peuvent être utilisés dans le script source et qui ont une signification particulière pour GoAsm.

```
,               - l'instruction n'est pas terminée, continuer
; ou //         - ligne de commentaire - ignorer tout jusqu'à la fin de la ligne
/*.........*/   - commentaire continu - ignore ce qui est entre les astérisques
```

```
\               - l'instruction ou la donnée se poursuivent sur la ligne suivante
- nombre        - le nombre est négatif
! nombre        - inverse le nombre (comme NOT)
NOT             - inverse le nombre
~ nombre        - idem
+               - signe plus
-               - signe moins
*               - symbole de multiplication
/               - symbole de division
|               - OR bit à bit
OR              - OR bit à bit
&               - AND bit à bit
AND             - AND bit à bit
<< nombre       - décalage à gauche d'un bit sur le nombre spécifié
>> nombre       - décalage à droite d'un bit sur le nombre spécifié
(....)          - exécute le calcul prioritairement entre les parenthèses
```

dans une définition a une signification spéciale. Voir la section **Utilisation du double-dièse dans les définitions**.

Chapitre 4
Fonctionnalités avancées

4.1 Structures – différents types et utilisation

4.1.1 Qu'est-ce qu'une structure ?

Les structures sont des zones de données de taille fixe qui contiennent des données réparties dans divers composants (éléments de structure). Elles peuvent aller de dispositions très souples à d'autres, très formalisées, avec même des structures au sein d'autres structures (structures imbriquées). Elles peuvent être des zones de données résultant d'une déclaration de données ordinaire ou de modèles STRUCT. Les structures sont très importantes en programmation Windows et GoAsm en supporte tous les types.
Voir aussi le paragraphe suivant concernant les **Unions**.

4.1.2 Utilisation de structures simples en programmation Windows

Considérons la structure LV_COLUMN qui est utilisée pour organiser les colonnes dans un contrôle ListView. Le code suivant envoie le message LVM_INSERTCOLUMN (valeur 101Bh) au contrôle ListView pour constituer une nouvelle colonne avec le numéro d'index correspondant dans EAX. Les détails des colonnes sont contenus dans la structure LV_COLUMN. Voici comment ils pourraient être utilisé en code 32 bits :

```
PUSH ADDR LV_COLUMN,EAX,101Bh,hListView
CALL SendMessageA   ;insère la colonne indexée par EAX
```

Regardons maintenant de plus près la structure LV_COLUMN. Dans le fichier d'en-tête de Windows Commctrl.h (version pre- Win_IE 300) qui contient des informations sur la structure, elle est décrite comme une structure de six dwords. En un sens, donc, la structure peut être considérée comme une succession de 6 dwords qui peuvent être déclarés très simplement comme suit :

```
LV_COLUMN DD 6 DUP 0
```

Cependant, dans l'information de Windows, il apparaît que chacun des six double-mots a un nom qui donne une idée de ce pourquoi il est utilisé, ce qui est utile. De plus, le premier dword s'avère être un masque qui identifie lesquels des éléments suivants de la structure sont valides. Ce masque est important car une version ultérieure de la structure révèle, par exemple, la présence de deux autres membres, ce qui impose *de facto* un masque différent. Donc, il pourrait être préférable de déclarer une structure de données de ce genre de telle sorte que le masque puisse être initialisé avec une valeur, et que vous puissiez voir les noms dans votre script source :

```
LV_COLUMN
  DD 0Fh     ;+0h mask
  DD 2h      ;+4h fmt=LVCFMT_CENTER=2
  DD 0       ;+8h cx
  DD 0       ;+0Ch pszText
  DD 0       ;+10h cchTextMax
  DD 0       ;+14h iSubItem
```

Ici, remarquons que, tandis que nous déclarions la structure de données, nous en avons profité pour initialiser deux des membres avec des valeurs qui ne changeront pas et que nous avons inclus dans les commentaires l'offset des différents membres, leurs noms ainsi que d'autres informations.

4.1.3 Lecture et écriture dans une structure simple

Il est très facile de lire et d'écrire sur la structure simple décrite précédemment :

```
MOV EDI,ADDR LV_COLUMN
MOV ESI,ADDR ColumnText     ; récupère en ESI la colonne de texte à utiliser
MOV [EDI+0Ch],ESI           ; et la communique au membre approprié de la structure
MOV D[EDI+8h],50D           ; et établit la largeur à 50 pixels
```

ou, vous pouvez utiliser :

```
MOV ESI,ADDR ColumnText     ; récupère en ESI la colonne de texte à utiliser
```

```
MOV [LV_COLUMN+0Ch],ESI     ; et la communique au membre approprié de la structure
MOV D[LV_COLUMN+8h],50D     ; et établit la largeur à 50 pixels
```

4.1.4 Structures plus formelles utilisant STRUCT

Certains programmeurs préfèrent être plus formels dans l'utilisation des structures en utilisant notamment un *modèle de structure*. Cela se fait en deux étapes. La première consiste à élaborer un modèle en utilisant STRUCT et en lui donnant un nom. A ce stade, les données ne sont pas encore déclarées.

Voici un exemple de modèle de structure portant le nom LV_COLUMN :

```
LV_COLUMN STRUCT
   mask          DD 0Fh    ; mask
   fmt           DD 2h     ; LVCFMT_CENTER=2
   cx            DD 0
   pszText       DD 0
   cchTextMax    DD 0
   iSubItem      DD 0
ENDS
```

Deux commentaires ont été ajoutés ici pour faciliter la compréhension de l'initialisation de deux membres de la structure. Noter que ENDS (littéralement END STRUCT) marque la fin du modèle. Vous pouvez également marquer cette fin en lui donnant le nom de la structure suivi de ENDS comme ci-dessous :

```
LV_COLUMN ENDS
```

La seconde étape consiste ensuite à *utiliser* le modèle. Vous y parvenez en le nommant et en faisant précéder ce nom d'un label, par exemple :

```
Lv1 LV_COLUMN
```

A ce stade, vous venez de déclarer six double-mots en utilisant le modèle de la structure LV_COLUMN et vous avez donné, à la déclaration de la structure, le label Lv1.

4.1.5 Les symboles créés par les structures formelles

Dans GoAsm, des symboles sont créés pour le label de la structure elle-même et également pour chaque membre nommé de la structure. Ceux-ci peuvent ensuite être référencés directement et transmis aussi comme tels au débogueur.

Ainsi, par exemple :

```
RECT STRUCT
   left     DD
   top      DD
   right    DD
   bottom   DD
ENDS
rc RECT
```

crée les symboles :

```
rc
rc.left
rc.top
rc.right
rc.bottom
```

4.1.6 Lecture et écriture sur la structure formelle

L'utilisation de la structure formelle vous permet d'être plus explicite dans votre script source lors de la lecture et l'écriture dans la structure. Par exemple :

```
MOV ESI,ADDR ColumnText     ; récupère en ESI la colonne de texte à utiliser,
MOV [Lv1.pszText],ESI       ; ... la communique au membre approprié de la structure
MOV D[Lv1.cx],50D           ; ... et établit la largeur à 50 pixels
```

ou même

```
MOV ESI,ADDR ColumnText     ; récupère en ESI la colonne de texte à utiliser
```

```
MOV EDX,ADDR Lv1.pszText    ; récupère en EDX l'adresse du membre psztext
MOV [EDX],ESI               ; et charge le texte à utiliser
MOV EDX,ADDR Lv1.cx         ; récupère en EDX l'adresse du membre cx
MOV D[EDX],50D              ; et établit la largeur à 50 pixels
```

Mais, rien ne vous empêche de recourir à une écriture plus classique qui produit, pour autant, le même résultat :

```
MOV ESI,ADDR ColumnText     ; récupère en ESI la colonne de texte à utiliser,
MOV [Lv1+0Ch],ESI           ; ... la communique au membre approprié de la structure
MOV D[Lv1+8h],50D           ; ... et établit la largeur à 50 pixels
```

Bien qu'elle se révèle plus complexe à mettre en œuvre, la première méthode vous permet de suivre votre code dans le débogueur symbolique. Les symboles de la structure y apparaîtront en entier, c'est-à-dire en associant label de structure et nom du membre, ce qui procure un gain de lisibilité évident. En effet, GoAsm crée des symboles pour tous les membres de la structure et transmet ceux-ci au linker. Il me semble que cette possibilité est spécifique à GoAsm et qu'elle est absente des autres assembleurs.

4.1.7 Récupération de l'offset des membres d'une même structure

Il peut arriver que vous ayez à connaître l'offset d'un membre au sein d'une structure. Il vous suffit pour cela de vous référer au nom de la structure auquel vous ajoutez un point puis le nom du membre concerné. Par exemple, si :

```
POINT STRUCT
    left   DD 0
    right  DD 0
ENDS
```

alors, l'instruction `MOV EBX,POINT.right` charge la valeur 4 dans EBX, qui est la distance de l'élément par rapport au début de la structure.

Cette façon de récupérer un offset est parfois utile pour obtenir des informations envoyées par Windows dans une structure. A titre d'exemple, la procédure de callback *OFNHookProc* reçoit de Windows de l'information dans un message de WM_NOTIFY. Le paramètre *lParam* contient un pointeur vers une structure OFNOTIFY. Il s'agit d'une structure imbriquée de la forme suivante :

```
OFNOTIFY STRUCT
   hdr        NMHDR
   lpOFN      DD
   pszFile    DD
ENDS
```

dans laquelle la structure NMHDR est :

```
NMHDR STRUCT
   hwndFrom   DD
   idFrom     DD
   code       DD
ENDS
```

Donc, au sein de votre procédure de fenêtre, vous pouvez obtenir la valeur du membre idFrom dans le NMHDR (identifiant du contrôle d'envoi du message) comme suit :

```
MOV ESI,[EBP+14h]                ; récupère en ESI le pointeur de la structure OFNOTIFY
MOV EAX,[ESI+OFNOTIFY.hdr.idFrom]
MOV EDX,[ESI+OFNOTIFY.pszFile]
```

En fait, il advient ici que OFNOTIFY.hdr.idFrom contient la valeur 4 ; NOTIFY.pszFile contient la valeur 10h. Ce sont leurs offsets corrects par rapport au début de la structure OFNOTIFY. Bien sûr, les structures concernées doivent être connues de GoAsm. Cela se fait en incluant les modèles de structure dans le script source assembleur, quelque part en amont dans le fichier.

4.1.8 Redéfinition de l'initialisation de la structure

Supposons que vous ayez une structure appelée RECT comme suit :

```
RECT STRUCT
   left     DD 10
   top      DD 10
   right    DD 120
   bottom   DD 90
ENDS
```

Vous pouvez remplacer l'initialisation de la structure en utilisant les opérateurs < et >, { et }. Par exemple

```
rc1 RECT <0,20,120,300>
```

réinitialise les double-mots dans la structure de données à 0, 20, 120 et 300 respectivement.

Vous pouvez utiliser le point d'interrogation et la virgule, ou tout simplement la virgule pour ignorer certains membres, par exemple :

```
rc1 RECT <0,?,?,300>
rc1 RECT <0,,,300>
```

Ici vous remplacez seulement les premier et quatrième membres de la structure.

En utilisant des accolades, vous pouvez nommer explicitement les membres que vous souhaitez réinitialiser :

```
rc1 RECT {left=2,top=5}
```

Vous pouvez même mélanger les deux méthodes :

```
rc1 RECT <{left=2,top=5},300h>
```

Lors de l'utilisation des accolades, il n'est pas nécessaire de spécifier les noms complets de symbole (dans l'exemple ci-dessus, ces noms complets seraient "rc1.left" et "rc1.top"). Au lieu de cela vous pouvez vous limiter au seul nom du membre ("left" et "top" dans notre cas). La réinitialisation est également effectuée dans les structures imbriquées. Aussi, si vous utilisez les mêmes noms pour les membres au sein d'une structure imbriquée, il est possible d'initialiser plusieurs membres à la fois en utilisant une accolade de réinitialisation.

4.1.9 Initialisation de membres de structure avec déclarations de données DUP

Si les membres d'une structure sont établis en utilisant l'opérateur DUP, vous pouvez procéder simplement à leur initialisation en utilisant une chaîne ou en spécifiant chaque élément au sein des délimiteurs < et > :

```
UP STRUCT
   DB 27 DUP 0
   DB 2  DUP 0
ENDS
Pent UP <'My cat was born on 23 April',<23h,4h>>
```

L'exemple qui suit décrit la structure GUID et une initialisation typique pour une COM :

```
GUID STRUCT
     Data1 dd ?
     Data2 dw ?
     Data3 dw ?
     Data4 db 8 dup ?
GUID ENDS
IID_IShellLink GUID <0000214eeh, 00000h, 00000h, <0c0h, 00h, 00h, 00h, 00h, 00h, 00h, 46h>>
```

4.1.10 Quelques règles de syntaxe concernant STRUCT

Une règle importante veut que, dans la mesure où GoAsm est un assembleur à une seule passe, les modèles de structure soient impérativement écrits dans le script source avant qu'ils ne soient utilisés. En effet, l'assembleur ne peut pas connaître à l'avance la taille de la structure. GoAsm est plutôt plus tolérant avec la syntaxe de STRUCT que d'autres assembleurs. Par exemple, STRUC a la même signification que STRUCT. De même, il n'y a pas besoin de fournir des valeurs initiales du tout, et il n'y a pas d'importance à ce que les membres ne soient pas nommés, ainsi qu'en attestent les exemples suivants qui sont parfaitement valides en dépit des apparences :

```
RECT STRUCT
    left     DD
    top      DD
    right    DD
    bottom   DD
ENDS
```

et

```
RECT STRUCT
    left     DD 0
             DD 2 DUP 0
    bottom   DD 0
ENDS
```

mais aussi

```
RECT STRUCT DD 4 DUP 0 ENDS
```

sont des déclarations de structure tout aussi valides les unes que les autres. Toutefois, lorsque les membres sont nommés, ils doivent l'être sur une nouvelle ligne.
Vous pouvez réutiliser le nom de membres de la structure, pourvu que le nom de la structure soit différent, par exemple :

```
RECT STRUCT
    left     DD 0
    top      DD 0
    right    DD 0
    bottom   DD 0
ENDS
RECT2 STRUCT
    left     DD 0
    top      DD 0
    right    DD 0
    bottom   DD 0
ENDS
```

Si vous utilisez ? dans l'initialisation des membres de la structure, vous obtenez le même effet qu'avec la valeur zéro. Cela ne se traduit pas, en tout cas, par des données enregistrées comme non-initialisées, comme ce serait le cas avec une déclaration de données ordinaire. De la sorte,

```
RECT STRUCT
    left     DD ?
    top      DD ?
    right    DD ?
    bottom   DD ?
ENDS
rc1 RECT
```

est parfaitement valable, mais les données vont dans la section de données avec initialisation automatique à zéro, comme si des zéros avaient été utilisés.

Dans un modèle de structure, vous pouvez mettre des données supplémentaires sur une même ligne de la manière habituelle. Voici un modèle de structure de quatre Dword utilisant cette facilité :

```
RECT STRUCT
   lefttop        DD 0,0
   rightbottom    DD 0,0
ENDS
```

4.1.11 Déclarations de structure répétées

Ce procédé peut être utile pour créer des tableaux et des tables utilisant des modèles de structure. Par exemple :

```
RECT <>, <>, <>, <>
```

crée quatre structures RECT (quatre Dword dans chaque). Dans la mesure où aucun label n'a été utilisé devant RECT, aucun symbole ne sera créé et transmis au débogueur. Dans cet exemple :

```
Buffer RECT <0,0,10,10>,<5,5,20,20>,<8,8,30,30>
```

un tableau est constitué de trois structures RECT (quatre mots chacune) initialisées selon les valeurs indiquées. Des symboles ne seront élaborés que pour la toute première structure. Ceci afin d'éviter la duplication des noms de symboles.

Si vous voulez que les membres de la matrice aient des noms de symboles uniques, il vous est possible de procéder, par exemple, comme suit :

```
Buffer1 RECT <0,0,10,10>
Buffer2 RECT <5,5,20,20>
Buffer3 RECT <8,8,30,30>
```

ou

```
Buffer RECT3 <0,0,10,10,   5,5,20,20,   8,8,30,30>
```

où RECT3 est une structure de 3 RECTs.

Si vous n'avez pas besoin d'initialiser les structures, vous pouvez les répéter en utilisant, soit :

```
Buffer RECT <>,<>,<>
```

ce qui crée trois structures RECT, soit :

```
Buffer RECT,RECT,RECT
```

qui fait la même chose.

Vous pouvez également utiliser des DUP pour répéter des structures, comme, par exemple :

```
ThreeRects    RECT 3 DUP <>
FiveRects     RECT 5 DUP <23,24,25,26>
```

Dans le deuxième exemple, chaque RECT est initialisé à la même valeur. L'initialisation de structures dupliquées, de cette manière, ne peut seulement être faite qu'au plus haut niveau et non dans des structures imbriquées.

4.1.12 Structures imbriquées utilisant STRUCT

Les structures peuvent être imbriqués en utilisant une structure à l'intérieur d'une autre, de sorte que

```
RECT STRUCT
    left   DD 0
    top    DD 0
    right  DD 0
    bottom DD 0
ENDS
StructTest STRUCT
    a DD   6
    b RECT
    c DD   7
    d DD   8
ENDS
```

Dans ce cas, `Hello StructTest` crée sept Dword. Les symboles créés (et passés au débogueur) sont :

```
Hello
Hello.a
Hello.b
Hello.b.left
Hello.b.top
Hello.b.right
Hello.b.bottom
Hello.c
Hello.d
```

et peuvent être lus ou écrits de la manière habituelle, par exemple

```
MOV D[Hello.b.left],100h      ; construction d'un rectangle commençant à 256 pixels
```

De même que les membres d'une structure, les structures imbriquées n'ont pas nécessairement besoin d'être nommées, de sorte que ce qui suit est parfaitement valable :

```
StructTest STRUCT
      DD     6
      RECT
     c DD     7
     d DD     8
ENDS
```

Structures imbriquées internes

Les structures peuvent être imbriquées en déclarant une structure au sein d'une structure, de sorte que

```
StructTest STRUCT
     a        DD 6
     b STRUCT
      left    DD 0
      top     DD 0
      right   DD 0
      bottom  DD 0
      ENDS
     c        DD 7
     d        DD 8
ENDS
```

Puis

```
Hello StructTest
```

produisent le même résultat que StructTest dans l'exemple précédent. La seule différence réside dans le fait que la structure *b* n'est pas disponible pour une utilisation ailleurs.

4.1.13 Outrepasser l'initialisation dans les structures imbriquées

Vous devez utiliser avec prudence les délimiteurs < et > pour initialiser les membres corrects de la structure imbriquée. Chaque délimiteur < permettra d'aller plus loin dans l'imbrication tandis que chaque délimiteur > permettra d'en sortir un peu plus. Au sortir d'une imbrication, une virgule est attendue après le délimiteur >. Donc, dans la structure imbriquée StructTest évoquée précédemment :

```
rc1 StructTest <23,<10,20,120,300>,44,55>
```

va non seulement initialiser la structure principale mais également son membre *b* imbriqué. En revanche :

```
rc1 StructTest <,<10,20,120,?>,44,55>
```

outrepassera seulement l'initialisation de certains membres, de même que la formulation suivante qui est rigoureusement équivalente :

```
rc1 StructTest <,<10,20,120,>,44,55>
```

Enfin, l'écriture qui suit n'apportera aucune modification au membre *b* imbriqué :

```
rc1 StructTest <,,44,55>
```

Une bonne façon de suivre la cohérence des délimiteurs est de visualiser mentalement les membres que vous ne voulez pas modifier par un point d'interrogation. Vous pouvez même les insérer dans l'expression pour en faciliter la lecture. Par exemple le dernier exemple peut être écrit :

```
rc1 StructTest <?,?,44,55>
```

4.1.14 Priorité d'outrepassement

Plus grand est le niveau d'outrepassement, plus grande sera sa priorité. Supposons, par exemple, que vous ayez ces modèles de structure :

```
RECT STRUCT
     left    DD 1
     top     DD 2
     right   DD 3
     bottom  DD 4
```

```
ENDS
StructTest STRUCT
    a  DD    6
    b  RECT  <3333h,4444h,5555h,>
    c  DD    7
    d  DD    8
ENDS
```

Puis

```
Hello StructTest <,<,0Bh,0Ch,>,,>
```

Alors, RECT sera initialisé à 3333h,0Bh,0Ch,4.
Outrepasser le nom des membres en utilisant les accolades {} bénéficie d'une priorité plus élevée que de le faire en utilisant les délimiteurs < et >.

4.1.15 Utilisation de chaînes dans les structures

Vous pouvez utiliser des chaînes dans les structures de la même manière que vous le feriez dans une déclaration de données ordinaire, par exemple :

```
StringStruct STRUCT
   DB 'I am a lonely string in a struct',0
   DB 'I will keep you company',0
ENDS
```

4.1.16 Structures avec des chaînes : initialisation et outrepassement

Si une structure est déclarée avec ?, alors elle peut être de n'importe quelle taille lorsqu'elle est utilisée avec des chaînes. Par exemple :

```
Rect STRUCT
    a  DB  ?
       DB  0
    b  DB  ?
       DB  0
ENDS
RC1 Rect <'Hello',,'Goodbye'>
```

fixera la structure Rect en des chaînes terminées par un zéro de 5 et 7 octets respectivement.
Lorsque la taille du membre d'une structure est déjà définie, par exemple,

```
Rect STRUCT
    a  DB  'Hello'
       DB  0
    b  DB  'Goodbye'
       DB  0
ENDS
```

alors, l'outrepassement de l'initialisation ne changera pas la taille des membres, de sorte que par exemple :

```
RC1 Rect <'Goodbye',,'Hello'>
```

se traduira par la chaîne tronquée 'Goodb' au label Rect.a et par la chaîne 'Hello' au label Rect.b, le reste de cette dernière étant comblée avec 2 zéros.

L'initialisation des membres des structures établie à l'aide DUP peut également être surpassée par des chaînes. Par exemple :

```
UP STRUCT
   DB 20 DUP 0
ENDS
Pent UP <"Hello">
```

se traduit par la chaîne 'Hello' suivie par 15 zéros.

Voir aussi la section “Utilisation des chaînes Unicode dans les structures” du volume 2 dans le chapitre conscré à l'Unicode.

4.1.17 Assemblage conditionnel dans les structures

Vous pouvez utiliser l'assemblage conditionnel directement dans une structure comme, par exemple

```
NMHDR STRUCT
   hwndFrom  DD
   idFrom    DD
   code      DD
ENDS
;
NMTTDISPINFO STRUCT
   hdr       NMHDR
   lpszText  DD
   #if STRINGS UNICODE
     szText  DW 80 DUP ?
   #else
     szText  DB 80 DUP ?
   #endif
   hinst     DD
   uFlags    DD
   lParam    DD
ENDS

DATA
Use1 NMTTDISPINFO
Use2 NMTTDISPINFO <<>,,"Hello",,,,>
```

Vous pouvez utiliser l'assemblage conditionnel directement à l'intérieur d'une structure, par exemple :

La deuxième utilisation de la structure va assembler la chaîne 'Hello', soit en Unicode, soit en ANSI selon le paramétrage, en conséquence, de la directive STRINGS.

Voir la section relative à l'**assemblage conditionnel**.

4.2 Unions

4.2.1 Définition

Les unions, comme les structures, sont des zones de données de taille fixe qui contiennent des données dans divers composants (les membres d'union). Comme pour les structures, aucune zone de données n'est effectivement créée lorsque vous *déclarez* un modèle d'union. Cette déclaration se fait lorsque vous utilisez le modèle. Les unions diffèrent des structures en ce que chaque membre de l'union commence à la même adresse dans la mémoire. Les unions se révèlent utiles par exemple si vous souhaitez utiliser des labels différents pour traiter la même zone de données. Le label que vous adressez au moment de l'exécution pourrait alors compter sur des éventualités telles que la version du système d'exploitation sur lequel le programme est en cours d'exécution. La taille d'une union est toujours réglée sur celle de la plus grande déclaration de données en son sein. Vous pouvez mélanger les unions avec des structures pour former des modèles complexes. Vous pouvez déclarer des unions dans les données locales et les répéter de la même manière que vous le pouvez pour les structures.

Considérons, par exemple, le modèle d'union déclaré comme suit :

```
Thing UNION
      Cat   DD 0
      Dog   DW 0
      Rat   DB 0
ENDS
```

Vous pouvez utiliser ce modèle en écrivant :

```
Hungry Thing
```

Ceci définit alors à part une zone de données de 4 octets (Dword). Mais, pourquoi seulement 4 octets ? Tout simplement parce que chaque membre commence à la même place et que l'on prend seulement en

considération la taille du plus grand d'entre eux. La fin du modèle de l'union est marquée, au choix, par ENDS ou ENDUNION si vous préférez.

Les symboles créés par cette union sont :

```
Hungry
Hungry.Cat
Hungry.Dog
Hungry.Rat
```

Et vous pouvez adresser ces labels de la manière habituelle, par exemple :

```
MOV [Hungry.Cat],EAX
MOV AL,[Hungry.Dog]
MOV ESI,[Hungry.Rat]
```

Ce qui, bien sûr, puisque chaque membre commence à la même place, est identique à :

```
MOV [Hungry.Cat],EAX
MOV AL,[Hungry.Cat]
MOV ESI,[Hungry.Cat]
```

4.2.2 Unions imbriquées

Vous pouvez imbriquer des unions dans des structures, des structures dans des unions ou des unions dans des unions. Par exemple

```
Laugh STRUCT
    Balm       DW 0
    Ointment   DB 0
ENDS
;
Zebra UNION
    Tiger      DD 0
    Hyaena     Laugh
ENDS
;
Lion STRUCT
    BagPuss    DB 3 DUP 0
    Striped    Zebra
ENDS
;
Fierce Lion
```

qui produit les symboles et offsets correspondants suivants dans Fierce :

```
Fierce                          +0
Fierce.BagPuss                  +0
Fierce.Striped                  +3
Fierce.Striped.Tiger            +3
Fierce.Striped.Hyaena           +3
Fierce.Striped.Hyaena.Balm      +3
Fierce.Striped.Hyaena.Ointment  +5
```

4.2.3 Unions imbriquées en interne

Les unions peuvent être imbriquées en les déclarant dans une structure ou une union, telles que le montre l'exemple suivant :

```
Lion STRUCT
    BagPuss  DB 3 DUP 0
    Striped UNION
             Tiger DD 0
             Hyaena STRUCT
                 Balm       DW 0
                 Ointment   DB 0
```

```
            ENDS
    ENDS
ENDS
;
Fierce Lion
```

produit le même résultat que l'exemple précédent. La seule différence est que Laugh et Zebra ne sont pas disponibles pour être utilisés ailleurs.

4.2.4 Initialisation des membres d'union

De la même manière que dans les structures, vous pouvez utiliser les opérateurs < et > pour initialiser les unions avec des chaînes ou des valeurs numériques, comme, par exemple :

```
Cat UNION
   Ginger   DB
   Tortie   DW
   Grey     DD
   Tabby    DQ
ENDS

Hungry      Cat <"a string for Ginger">
Anxious     Cat <,4444h>                ;initialise le mot
Sleepy      Cat <,,55555555h>           ;initialise le double-mot
Insistent   Cat <,,,6666666666666666h>  ;initialise le quadruple-mot
```

Il est encore plus difficile, lorsque vous utilisez les unions, de suivre les opérateurs < et >. En lieu et place et à votre convenance, vous pouvez spécifier le nom du membre inclus entre les opérateurs { et }, ou vous pouvez les initialiser lors du lancement de l'exécution, par exemple :

```
Scaredy  Cat  {Ginger="a string for Ginger"}
GString  DB   "a string for the Grey cat"
MOV [Scaredy.Grey],ADDR GString       ;charge un pointeur vers GString
```

Rappelez-vous que, dans la mesure où les membres d'unions sont au même endroit, une initialisation ultérieure peut en écraser une précédente.
Le point d'interrogation en option (?) est utile pour montrer que vous ne voulez pas effacer un remplacement précédent.

Exemple :

```
Laugh STRUCT
    Balm       DW 6666h
    Ointment   DB
ENDS
;
Zebra UNION
    Tiger      DD 88888888h
    Hyaena     Laugh
ENDS
;
Lion STRUCT
    BagPuss    DB
               DB
               DB
    Striped Zebra
ENDS
;
Fierce Lion <{Ointment=0AAh}22h,33h,44h,<?,<?,55h>>>
```

qui initialise la zone de données de la manière suivante :

```
At Fierce.BagPuss              (at offset +0) 22h,33h,44h
Then at Fierce.Striped.Tiger   (at offset +3) 66h,66h,0AAh,88h
```

Il apparaît ici que le dword Tiger à +3 dans l'union Zebra a été initialisé à 88888888h mais qu'il a ensuite été remplacé par les valeurs de Balm et Ointment (qui étaient dans la même union). Seul le dernier octet a survécu.

4.3 Définitions : Equates, macros et #define

4.3.1 Quand faire quelque chose signifie quelque chose d'autre

Equates, macros et #defines donnent un sens à un mot. En d'autres termes, le mot est *défini*. Une fois défini, et à partir de ce point dans le script source, la définition est utilisée en lieu et place du mot d'origine si le contexte le permet. Si le mot tel que défini désigne :

- **un nombre**, alors les programmeurs en assembleur appellent la définition une "equate" parce que, traditionnellement, le mot est défini en utilisant les opérateurs EQU ou =.
- **une chaîne**, alors traditionnellement, on parle d'une "equate" texte.
- **quelque chose de plus élaboré qu'un nombre ou une chaîne** – une série d'instructions, par exemple – alors les programmeurs appellent cela une "macro". Ceci parce que, selon un usage répandu, le mot est défini en utilisant l'opérateur MACRO.

Lors de l'utilisation GoAsm, pour les définitions qui peuvent être construites sur une seule ligne, vous aimeriez utiliser EQU, =, ou #define comme vous le feriez en langage C. Il suffit d'utiliser celui des trois que vous aimez le mieux. Vous pouvez utiliser le caractère de continuation ("\") pour permettre aux définitions de s'étendre sur plus d'une ligne, mais il est préférable d'utiliser MACRO ... ENDM à la place de manière à éviter d'éventuels problèmes de syntaxe.

Dans la mesure où GoAsm est un assembleur qui ne travaille qu'en une seule passe, vous devez vous assurer que vos définitions ne sont pas utilisées avant qu'elles ne soient déclarés dans le script source. Une fois qu'un mot a été défini, vous pouvez modifier sa définition mais GoAsm vous en avertira car une telle intervention peut ne pas être intentionnelle.

Voici quelques exemples de mise en œuvre de définitions :

4.3.2 Définitions de mots représentatifs de nombres ou chaînes (exemples de données)

Voici trois exemples définissant un mot comme une valeur constante. Le premier utilise =, le second utilise EQU et le troisième, #define. Ils ont tous les trois le même effet.

```
WS_CHILD=40000000h
WS_CHILD EQU 40000000h
#define WS_CHILD 40000000h
```

Outre une constante numérique, vous pouvez utiliser une expression arithmétique, des chaînes ou même d'autres définitions lorsque vous définissez un mot. En voici quelques exemples :

```
SKIP_VALUE EQU 20h|40h
#define SKIP_VALUE 20h|40h
HelloText='Hello world'
#define HelloText "Hello world"
MANIA=SKIP_VALUE+WS_CHILD
```

Si vous ne donnez pas de valeur à l'*equate*, celle-ci prend la valeur 1 par défaut, qui correspond à TRUE dans la philosophie Windows. Par exemple :

```
NT_VERSION=
NT_VERSION EQU
#define NT_VERSION
```

Une fois un mot défini, vous pouvez l'utiliser dans presque toute situation où la définition est valide, par exemple :

```
DB HelloText
PUSH WS_CHILD|WS_VISIBLE|SS_OWNERDRAW
MOV EAX,WS_CHILD
MOV EAX,[ESI+SKIP_VALUE]
MOV EAX,MANIA+800h
```

4.3.3 Définition de mots en substitution d'instructions de code

Voici un exemple de la façon dont vous pouvez attribuer à un mot tout ou partie de la fonctionnalité d'une instruction de code :

```
#define lParam [EBP+14h]
```

Vous pouvez utiliser alors la définition ainsi créée de la manière suivante :

```
MOV lParam,EAX      ;identique à MOV [EBP+14h],EAX
```

Ici, on peut dire, d'une certaine manière, que le texte '[EBP+14h]' a tout simplement été remplacé par lparam dans le script source.

4.3.4 Utilisation d'arguments dans la définition de mots

Les arguments sont des valeurs qui sont données lorsque les définitions sont utilisées. Ces valeurs sont ensuite utilisées dans la définition ou la macro elles-mêmes. Ainsi, par exemple :

```
RECTB(%a,%b,%c,%d) = DD %a,%b,%c,%d
```

Ensuite, vous pouvez déclarer quatre Dwords initialisés comme spécifié dans les arguments :

```
rc1 RECTB (10,10,100,200)  ; identique à DD 10,10,100,200
```

Voici maintenant un autre exemple utilisant #define :

```
#define DBDATA(%a,%b) DB %a DUP %b
DBDATA(3,'x')              ; identique à DB 3 DUP 'x'
```

Il existe une règle de syntaxe importante lorsque vous utilisez des arguments dans les définitions : *il ne doit pas y avoir d'espace entre le nom de la définition et l'ouverture de parenthèse qui précède l'énoncé des arguments*. Ainsi :

```
RECTB(%a,%b,%c,%d)
```

est correct, mais

```
RECTB (%a,%b,%c,%d)
```

est erroné. Cette contrainte syntaxique, déroutante à certains égards, garantit que GoAsm reconnaît les paramètres entre parenthèses en tant qu'arguments et seulement en tant que tels.
Vous devez également vous assurer que les noms des arguments sont inhabituels et ne risquent donc pas de se retrouver dans tout autre matériau utilisé dans la définition ou la macro. Cela évite que d'autres choses soient remplacées par inadvertance. D'où le de signe pourcentage utilisé dans les exemples ci-dessus.

4.3.5 Définitions réparties sur plusieurs lignes

La définition peut occuper plusieurs lignes pour en accroitre la lisibilité et, notamment, pour vous permettre d'ajouter des commentaires. Par exemple :

```
#define WS_POPUP            0x80000000L
#define WS_BORDER           0x00800000L
#define WS_SYSMENU          0x00080000L
#define WS_POPUPWINDOW      (WS_POPUP          | \
                             WS_BORDER         | \
                             WS_SYSMENU)
```

Ce dernier exemple a été pris directement sur le fichier d'en-tête de Windows Winuser.h et vous pouvez voir qu'il épouse la syntaxe typique du "C". GoAsm ne peut que s'en réjouir et tolère, de surcroît, l'absence éventuelle des parenthèses :

```
#define WS_POPUPWINDOW      WS_POPUP          | \
                            WS_BORDER         | \
                            WS_SYSMENU
```

Vous pouvez préférer utiliser MACRO ... ENDM à la place du caractère de continuation auquel cas l'exemple ci-dessus peut être re-écrit sous la forme :

```
WS_POPUPWINDOW  MACRO  WS_POPUP    |
                       WS_BORDER   |
                       WS_SYSMENU
                ENDM
```

4.3.6 Définitions multi-ligne : exemple de trame de pile (callback windows)

Ceci s'applique uniquement à la programmation 32 bits.

Vous pouvez utiliser la méthode de définition multi-ligne pour constituer un mot signifiant plusieurs lignes d'instructions de code. Par exemple :

```
OPEN_STACKFRAME(a) =  PUSH EBP \
                      MOV EBP,ESP \
                      SUB ESP,a*4 \
                      PUSH EBX,EDI,ESI
CLOSE_STACKFRAME   =  POP ESI,EDI,EBX \
                      MOV ESP,EBP \
                      POP EBP
```

En utilisant MACRO...ENDM, cela donne :

```
OPEN_STACKFRAME(a) MACRO  PUSH EBP
                          MOV EBP,ESP
                          SUB ESP,a*4
                          PUSH EBX,EDI,ESI
                   ENDM
CLOSE_STACKFRAME   MACRO  POP ESI,EDI,EBX
                          MOV ESP,EBP
                          POP EBP
                   ENDM
```

Dans cet exemple, le mot OPEN_STACKFRAME est défini pour élaborer une trame de pile qui pourrait typiquement être utilisée dans une procédure de fenêtres appelée par le système Windows. Il possède un argument qui détient le nombre de mots dans la trame de pile permettant d'accepter des données locales (le pointeur de pile est déplacé par ce paramètre de sorte que la pile puisse être utilisée pour stocker les données locales). Le mot CLOSE_STACKFRAME ferme, quant à lui, la trame de pile. Voici maintenant comment utiliser ces définitions. Dans la section de code :

```
WndProc:              ; nom de cette procédure
OPEN_STACKFRAME (6)   ; création d'un espace pour 6 dwords de données locales
                      ;----------------- insertion du code de la procédure de fenêtre ici
CLOSE_STACKFRAME
RET 10h               ; retrait de la pile des 4 paramètres envoyés par Windows
```

Ajoutons maintenant un peu de raffinement de sorte que la pile puisse être accessible de manière plus compréhensible :

```
lParam =[EBP+14h]  ;
wParam =[EBP+10h]  ; on se tient prêt à accéder aux paramètres qui
uMsg   =[EBP+0Ch]  ; sont envoyés par Windows à la procédure de fenêtre
hwnd   =[EBP+8h]   ;
;
hDC    =[EBP-4h]   ; certains noms d'éléments de données
hBrush =[EBP-8h]   ; sont souvent utilisées dans des
hPen   =[EBP-0Ch]  ; procédures de fenêtre différentes
DATA1  =[EBP-10h]  ;
DATA2  =[EBP-14h]  ; espace pour plus de données locales
DATA3  =[EBP-18h]  ;
```

A l'intérieur de la trame de pile constituée ici, les paramètres envoyés par Windows (hwnd, uMsg, wParam et lParam) seront toujours sur la pile de EBP+14h à EBP+8h. À EBP+4h, nous trouvons l'adresse de retour du call. En EBP nous avons la valeur précédente de EBP qui nous avons poussée au moment de la constitution de la trame de pile. Puis, de EBP–4h à EBP–18h, nous avons l'espace dévolu à nos données locales qui, dans cet exemple, peuvent être accessibles en utilisant les définitions hDC, hBrush, hPen, DATA1, DATA2 et DATA3 (ou tout nom que vous voudrez bien leur attribuer). À EBP–1Ch nous avons la valeur de EBX quand ce registre a été poussé en pile lorsque la trame de cette dernière a été élaborée. De même EDI est en EBP–20h et ESI en EBP–24h. Toutes ces valeurs sont protégées tandis que la trame de pile reste

ouverte (elles ne seront pas écrasées par d'autres fonctions avant la fin du callback). Pour accéder aux données dans la trame de pile, vous devez vous assurer que vous ne modifiez EBP en aucune manière (ou si vous le faites, vous le restaurez à sa valeur initiale). Vous ne devez pas accéder aux données par leur nom. Dans cet exemple `MOV EAX,[hBrush]` est la même chose que `MOV EAX,[EBP-8h]`. Ceci est une question de style et de choix personnel. En utilisant ces méthodes, vous pouvez établir autant de données locales que vous le souhaitez, et si vous vous en tenez à une méthode fixe comme celle-ci, vous saurez toujours où sont vos données locale. Dans cet exemple, le premier Dword des données locales est toujours à EBP–4h. Soustraire 4 de cette valeur pour accéder à chaque Dword supplémentaire de données locales.

Il y a beaucoup d'autres façons de travailler avec la pile dans les callbacks. Voir, à ce sujet, les sections **Trames de pile pour Callback en 32 et 64 bits**, et **Trames de pile automatisées utilisant FRAME ... ENDF, LOCALS et USEDATA**.

Voir aussi l'Annexe I Comprendre la pile (parties 1 et 2) dans les annexes.

4.3.7 Assemblage conditionnel dans les macros

Si vous utilisez l'assemblage conditionnel dans vos définitions il est recommandé d'utiliser MACRO ..ENDM au lieu de la méthode des caractères de continuation mentionnée plus haut.

Voici un exemple :

```
STRINGS UNICODE
CODE
;
#define REPORT
;
MBMACRO(%lpTextW) MACRO
#ifdef REPORT
INVOKE MessageBoxW,0,addr %lpTextW,"Report",40h
#endif
ENDM
;
MBMACRO("Ce code a été assemblé !")
```

Dans le code ci-dessus, la boîte de message (Message Box) est affichée de manière classique via l'API *MessageBox*. On a vu précédemment que la ligne `#define REPORT` sans autre paramètre donne à `REPORT` la valeur 1 (TRUE). Cette situation autorise l'affichage. Mais si `REPORT` était suivi d'une valeur, alors il n'y aurait pas de message affiché. La chaîne "Ce code a été assemblé!" est envoyée comme un argument à la macro.
Voir la section Assemblage conditionnel.

4.3.8 Comptage des arguments avec ARGCOUNT

Ceci s'applique uniquement à la programmation 32 bits.

ARGCOUNT renvoie le nombre d'arguments donnés lorsque la définition est utilisée, ce qui peut être utilisé avec l'assemblage conditionnel dans les définitions. Par exemple, considérons la macro nommée macro26 ainsi définie :

```
macro26(%a,%b,%c,%d,%e,%f) =  #if ARGCOUNT=6      \
                               PUSH %f            \
                              #endif              \
                              #if ARGCOUNT >=5    \
                               PUSH %e            \
                              #endif              \
                              #if ARGCOUNT >=4    \
                               PUSH %d            \
                              #endif              \
                              #if ARGCOUNT >=3    \
                               PUSH %c            \
                              #endif              \
                              #if ARGCOUNT >=2    \
                               PUSH %b            \
                              #endif              \
```

```
                                #if ARGCOUNT >=1    \
                                 PUSH %a            \
                                #endif
```

Imaginons maintenant que nous utilisions macro26 comme suit :

```
macro26(4,3,2,1)
```

Dans ce cas, ARGCOUNT prendrait la valeur 4 et le code résultant correspondrait exactement à l'instruction multiple suivante :

```
PUSH 1,2,3,4
```

Dans l'exemple ci-dessus, les deux premiers PUSH sont ignorés parce que ARGCOUNT est ni 6 (dans le premier test) ni supérieur ou égal à 5 (dans le second test).

Cela peut être amélioré pour établir une macro d'appel de fonction "C" où la pile est effacée après l'appel (le nombre correct d'octets est ajouté au pointeur de pile ESP après l'appel) :

```
macro26(%x,%a,%b,%c,%d,%e,%f) =  #if ARGCOUNT=7      \
                                   PUSH %f           \
                                  #endif             \
                                  #if ARGCOUNT >=6   \
                                   PUSH %e           \
                                  #endif             \
                                  #if ARGCOUNT >=5   \
                                   PUSH %d           \
                                  #endif             \
                                  #if ARGCOUNT >=4   \
                                   PUSH %c           \
                                  #endif             \
                                  #if ARGCOUNT >=3   \
                                   PUSH %b           \
                                  #endif             \
                                  #if ARGCOUNT >=2   \
                                   PUSH %a           \
                                  #endif             \
                                  CALL %x            \
                                  ADD ESP,ARGCOUNT-1*4
```

et ici, on peut voir une autre manière d'y arriver :

```
cinvoke(funcname,%1,%2,%3,%4,%5) = \
                                  #if ARGCOUNT=1       \
                                  invoke funcname       \
                                  #elif ARGCOUNT=2       \
                                  invoke funcname,%1      \
                                  #elif ARGCOUNT=3         \
                                  invoke funcname,%1,%2     \
                                  #elif ARGCOUNT=4           \
                                  invoke funcname,%1,%2,%3    \
                                  #elif ARGCOUNT=5             \
                                  invoke funcname,%1,%2,%3,%4  \
                                  #elif ARGCOUNT=6               \
                                  invoke funcname,%1,%2,%3,%4,%5 \
                                  #endif                          \
                                  #if ARGCOUNT>1                   \
                                  ADD ESP,ARGCOUNT-1*4              \
                                  #endif
```

Lesquels pourraient alors être utilisés comme suit :

```
cinvoke(_cprintf,23,24,25,26,27)
macro26(_cprintf,23,24,25,26,27)
```

Si vous ne souhaitez utiliser le caractère de continuation, vous pouvez utiliser MACRO ... ENDM en lieu et place.

4.3.9 Utilisation des double-dièses dans les définitions

Un double-dièse dans une définition a pour fonction de joindre deux éléments en supprimant tous les espaces entre les deux. Cela vous permet de créer un seul mot à partir d'un ou plusieurs composants, par exemple :

```
LVERS=0030
MVERS=0044h
VERSION=LVERS##MVERS
;
MOV EAX,VERSION
```

Ici, VERSION est défini par le nombre 00300044h.

4.3.10 L'utilisation des définitions et ses limites

Certains programmeurs utilisent des définitions aussi souvent que possible. À mon avis, cela rend le script source plus difficile à lire. Aussi convient-il de déterminer dans quelle mesure quelqu'un d'autre que le concepteur pourrait être amené à parcourir le code source et à en comprendre les tenants et les aboutissants à supposer, bien évidemment, que cela soit souhaitable... Si cette personne doit souvent se référer à d'autres fichiers ou à des listes de définitions pour ce faire, alors le codage manquera, à l'évidence, de clarté. Utilisées cependant avec modération en programmation Windows, les définitions peuvent se révéler utiles pour expliciter le script source. Par exemple :

```
PUSH WS_CHILD|WS_VISIBLE|SS_OWNERDRAW
```

est plus parlant que :

```
PUSH 5000000Dh
```

bien qu'un commentaire approprié puisse, en l'espèce, éclairer utilement cette instruction sans qu'il soit nécessaire d'utiliser une définition :

```
PUSH 5000000Dh        ; WS_CHILD, WS_VISIBLE, SS_OWNERDRAW
```

En revanche, la formulation qui suit nuit à la clarté du code et doit donc être évitée :

```
#define wParam EBP+10h
MOV EAX,[wParam]      ; identique à MOV EAX,[EBP+10h]
```

On notera en effet que la référence [wParam] fait apparaître wParam comme un label, ce qui n'est justement pas le cas ici. Mieux vaut écrire en l'occurrence :

```
#define wParam [EBP+10h]
MOV EAX,wParam
```

Ceci est plus clair ainsi parce que, dans GoAsm, la seule chose que vous pouvez aborder de cette manière sans l'aide des crochets est une définition.

Ceci qui suit est également contestable et doit être évité à tout prix :

```
#define GET_LPARAM MOV EAX,[EBP+14h]
GET_LPARAM
```

Une meilleure pratique de programmation suggère d'écrire :

```
CALL GET_LPARAM
```

et de réaliser correctement cet appel sous forme de fonction. Cependant lors de la manipulation de la pile, il est très difficile d'utiliser une procédure dans la mesure où CALL et RET modifient eux-mêmes la pile. Donc, dans ce cas, il peut être pratique d'utiliser une définition si la clarté du script source n'en souffre pas. Voir, à ce titre, les exemples OPEN_STACKFRAME et CLOSE_STACKFRAME proposés précédemment.

Enfin, il semble qu'il y ait peu d'intérêt à écrire :

```
THOUSAND=1000D
MOV EAX,THOUSAND
```

alors qu'une formulation plus directe serait tout aussi claire :

```
MOV EAX,1000D
```

A titre d'illustration et en guise de conclusion, imaginez que vous souhaitiez que votre script source soit compréhensible par des non-anglophones. Dans ce cas, il vous serait parfaitement possible de traduire tous

les mnémoniques du processeur en utilisant des *equates* pour parvenir à vos fins. Voir à ce sujet la section "Utilisation de mots définis dans les fichiers Unicode" dans le chapitre du volume 2 consacré à l'Unicode.

4.4 Importation : utilisation des librairies run-time

Avec GoAsm il est facile d'utiliser la bibliothèque run-time du C. Elle consiste en un certain nombre de fonctions contenues dans Crtdll.dll ou Msvcrt.dll (ou leurs variantes) qui se trouvent habituellement sur un ordinateur Windows dans le dossier système. L'information sur cette librairie est disponible sur le site **Microsoft's developer site (MSDN)**. La principale chose à retenir lors de l'utilisation de ces fonctions est que même si vous envoyez des paramètres à la fonction au moyen de la pile, la fonction ne restaure pas celle-ci. Par conséquent, vous aurez besoin de le faire vous-même en utilisant l'instruction `ADD ESP, x` après l'appel, *x* étant le nombre d'octets utilisés pour les paramètres.

4.5 Import : données, par ordinal, par dll spécifiques

4.5.1 Import de données

Au moment de l'exécution, les données ne peuvent être importés qu'*indirectement*. Cela veut dire que vous ne pouvez importer qu'un *pointeur* vers des données. Cependant, en utilisant ce pointeur, vous pouvez obtenir les données elles-mêmes.
Par exemple, en supposant que DATA_VALUE est un export de données dans un autre programme vous obtenez, avec GoAsm, le pointeur et la donnée comme suit :

```
MOV EBX,[DATA_VALUE]    ; pointeur vers DATA_VALUE dans EBX
MOV EAX,[EBX]           ; on récupère la donnée dans EAX à partir de ce pointeur
```

De la même manière que pour l'importation des procédures d'autres programmes au moment de l'édition de liens, vous donnez au linker (GoLink, en l'occurrence) le nom de l'exécutable contenant l'import.

4.5.2 Import direct par ordinal

Le type d'import auquel nous sommes confronté en utilisant la *procédure* CALL importera la procédure par son *nom*. De manière plus détaillée lorsque le chargeur Windows démarre le programme, il parcourt les DLL pour les importations requises par le programme. Il le fait en comparant les noms des exportations dll aux noms des imports du programme. Pour accélérer ce processus avec les Dlls *privées* l'exportation et l'importation par ordinale sont parfois utilisées. Par ce biais, le chargeur peut trouver la bonne importation en utilisant un index dans une table de la DLL. Notez qu'il est imprudent de le faire dans le cas des DLL du système Windows puisque l'uniformité des nombres ordinaux des exportations n'est pas garantie selon les différentes versions de DLL.

En utilisant GoAsm et son complément GoLink, vous pouvez importer par ordinale par cette syntaxe simple :

```
CALL MyDll:6
```

Cette procédure va appeler le numéro 6 dans MyDll.dll. Notez que l'extension "dll" est implicite si aucune extension n'est mentionnée. Supposons que vous vouliez une DLL pour appeler une fonction dans l'exécutable principal par ordinal. Dans ce cas, vous pourriez utiliser :

```
CALL Main.exe:15
```

qui appellera la 15ème fonction de Main.exe.

Les appels ordinaux utilisant la forme absolue (c'est-à-dire utilisant les opcodes FF15) emploient, quant à eux, la syntaxe qui suit :

```
CALL [Main.exe:15]
```

Vous n'avez pas à inclure le chemin d'accès au fichier dans le CALL. GoLink réalise une recherche étendue des fichiers spécifiés, mais s'il était toutefois nécessaire de fournir un chemin ce serait dans le cadre de l'éditeur de liens plutôt qu'au niveau du CALL du script de l'assembleur.

Evidemment pour utiliser cette méthode d'appel d'une fonction par ordinal vous devez vous assurer que le nombre ordinal de la fonction est fixé. Voir la section **export par ordinal**.

Note : ce qui précède s'applique uniquement à GoLink

Il y a une autre façon d'utiliser les ordinaux en utilisant *LoadLibrary* pour charger la DLL (ou retourner un handle si elle est déjà chargée), puis en appelant *GetProcAddress* en passant la valeur ordinale pour obtenir la valeur de la procédure à appeler. Enfin, vous appelez la procédure telle que fournie par *GetProcAddress*.

4.5.3 Import par des dll spécifiques

Occasionnellement, vous voudrez peut-être forcer le linker à utiliser une importation en provenance d'une DLL particulière. Vous pourriez avoir besoin de procéder de la sorte si deux DLL voire plus (désignées au linker au moment de la phase d'édition de liens) comportent des fonctions portant le même nom. Il est donc possible, dans ce cas, de forcer GoLink à établie des liens avec une DLL particulière en utilisant la syntaxe :

```
CALL NameOfDll:NameOfAPI
```

Note : ceci s'applique uniquement à GoLink

4.6 Export de procédures et de données

Vous pouvez faire en sorte que vos procédures et données soient disponibles pour d'autres exécutables en les *exportant*. Dans Windows, il est habituel que les DLL soient utilisées pour les exportations, mais parfois une DLL a besoin d'appeler une procédure ou d'utiliser les données d'un fichier Exe et, dans ce cas, ce dernier exporte également. L'export peut être fait soit au moment de l'édition de liens (vous indiquez au linker les symboles à exporter), soit au moment de l'assemblage avec GoAsm. Dans ce dernier cas, GoAsm donne alors au linker les informations d'export via la section .drectve dans le fichier objet (à noter que tous les linkers ne supportent pas cette particularité).
Il y a deux façons de déclarer les exports dans GoAsm. Vous pouvez les déclarer tous au début de votre fichier (avant qu'une quelconque section soit déclarée) ou les déclarer dans votre code source au fur et à mesure que vous avancez dans son écriture. Vous pouvez utiliser l'une ou l'autre de ces méthodes selon votre propre préférence.

Voici un exemple de déclaration de toutes les exportations avant que les sections ne soient déclarées :

```
EXPORTS CALCULATE, ADJUST_DATA, DATA_VALUE
```

Voici un exemple de déclaration d'export en cours de développement :

```
EXPORT CALCULATE:
CMP EAX,EDX          ; code pour la
JZ >4                ; procédure
```

Ce code exporte le label à la procédure CALCULATE.
Si vous préférez, vous pouvez avoir les deux mots sur des lignes séparées comme, par exemple :

```
EXPORT
CALCULATE:
CMP EAX,EDX          ; code pour la
JZ >4                ; procédure
```

4.6.1 Export de données

Les données ne peuvent être exportées qu'*indirectement*. Seul, un pointeur vers ces données peut être exporté. Par l'utilisation de ce pointeur, le programme d'importation peut toutefois obtenir lui-même les données.
Vous pouvez utiliser exactement la même méthode pour exporter un label de données que celle utilisée pour un label de code. Il n'y a pas besoin de déclarer le label comme étant de donnée ou de code. Il en est ainsi parce que la tâche de vérification de l'appartenance du label à une section de données ou à une section de code incombe à l'éditeur de liens. Voici un exemple d'export du label de données :

```
EXPORT DATA_VALUE DD 0
```

Cette instruction exporte l'étiquette de données DATA_VALUE. Le destinataire peut obtenir la valeur de DATA_VALUE de la manière suivante :

```
MOV EBX,[DATA_VALUE]      ; EBX = pointeur vers DATA_VALUE
MOV EAX,[EBX]             ; EAX = valeur de DATA_VALUE
```

4.6.2 Export par ordinal

Normalement, les exports sont *nominatifs*. Lorsque le chargeur Windows démarre le programme, il parcourt les DLL pour y chercher les importations requises par le programme. Cela se fait en confrontant les noms des exportations des DLL aux noms des importations du programme. Pour accélérer ce processus avec DLLs *private*, l'exportation et l'importation par ordinal sont parfois utilisées. Dans ce cas, le chargeur peut trouver la bonne importation en utilisant simplement un index destiné à une table dans la DLL. Notez qu'il est imprudent de le faire dans le cas des DLL du système Windows puisque les nombres ordinaux des exportations peuvent varier selon les versions de DLL.
Pour utiliser cette méthode, il est clairement impératif que le programme d'exportation spécifie une valeur ordinale pour une exportation particulière et que l'éditeur de liens ne puisse y apporter aucune modification. Encore une fois, en utilisant GoAsm vous pouvez spécifier la valeur ordinale correcte et la passer au linker via la section .drectve (cependant, les linkers ne supportent pas tous cette pratique).
Voici comment spécifier une exportation par ordinal si les exportations sont listées avant que les sections ne soient déclarées :

```
EXPORTS CALCULATE:2, DATA_VALUE:6
```

Ici, le linker sera chargé d'utiliser les ordinaux 2 et 6 pour les exportations. Si vous utilisez la méthode alternative de déclarer les exportations (au sein d'une section) vous pouvez utiliser par exemple :

```
EXPORT:2 CALCULATE:
```

ou dans le cas de données :

```
EXPORT:6 DATA_VALUE DD 0
```

4.6.3 Export anonyme par ordinal

L'export par ordinal n'empêche pas le nom de l'export d'apparaître dans l'exécutable final. Il en est ainsi parce que c'est le programme d'importation qui décide d'importer par ordinal ou par nom. Tout ce que le programme d'exportation peut faire se limite à fixer la valeur ordinale. Cependant, il peut arriver que le programmeur ne souhaite pas qu'un nom pour l'exportation apparaisse dans l'exécutable final. On peut observer de telles exportations « sans nom » dans les DLL système par exemple, probablement dans le but de masquer le travail effectué par des fonctions particulières. Il est possible de recourir à ce procédé dans GoAsm en ajoutant la déclaration NONAME à la fin de l'exportation. Par exemple :

```
EXPORT:2:NONAME
CALCULATE:
```

Ici, la valeur du label de code CALCULATE sera exportée comme nombre ordinal 2, mais le nom de l'exportation n'apparaîtra pas dans l'exécutable final. Cela signifie que si un autre programme essayait d'appeler la fonction CALCULATE, il enregistrerait un échec. La fonction peut seulement être appelée par ordinal.

4.7 Sauvegarde et restauration des flags et des registres avec USES...ENDU

L'instruction USES suivie d'une liste de registres provoque la mise en pile (PUSH) de ces derniers selon l'ordre dans lequel ils apparaissent dans la liste. Puis, jusqu'à ce que ENDU soit rencontré dans le script source (ou ENDUSES si vous préférez), tout RET rencontré provoquera le dépilage (POP) des mêmes registres dans l'ordre inverse. Par exemple :

```
ProcX:
USES EAX,EBX,ECX    ; prêt à mettre les registres en pile (PUSH)
CMP EAX,ESI         ; le premier mnémonique rencontré rend les PUSHes effectifs
;
; code de la procédure
;
.finnc
CLC                 ; met à zéro le flag de carry
RET                 ; on retire tous les registres de la pile en ordre inverse du USES (POP)
                    ; avant d'exécuter le RET
```

```
.finc
STC                 ; met à 1 le flag de carry
RET                 ; on retire tous les registres de la pile en ordre inverse du USES (POP)
                    ; avant d'exécuter le RET
ENDU                ; on désactive cette action de POP automatique lors d'un RET
```

Vous pouvez également empiler (PUSH) et dépiler (POP) les flags de manière automatique en utilisant l'instruction FLAGS qui est un mot réservé dans GoAsm :

```
USES FLAGS
```

Vous ne pouvez modifier ou compléter la liste des registres à préserver à l'intérieur-même de la procédure. Pour ce faire, vous devez recourir à une instruction ENDU suivie d'une liste USES actualisée. Si vous avez besoin d'un RET ne déclenchant pas une restitution automatisée des registres vous devez utiliser RETN qui a la fonctionnalité d'un RET "normal". Notez que RETN est une directive de l'assembleur et non une instruction processeur.

En programmation 64 bits, vous pouvez utiliser non seulement la version étendue des registres généraux (de RAX à RSP) mais aussi les nouveaux registres 64 bits (R8 à R15). Vous pouvez également utiliser les versions 32 bits des registres généraux (EAX à ESP). Ceci, parce que l'opcode du PUSH est identique, qu'il s'agisse de registres 32 ou 64 bits. Donc, si vous assemblez en 32 ou 64 bits,

```
USES RAX, RBX, RCX
```

produira le même code que

```
USES EAX, EBX, ECX
```

Cela contribue à la transportabilité de votre code entre les deux plates-formes.

4.8 Trames de pile pour Callback en 32 et 64 bits

Voir également l'annexe I “Comprendre la pile” **partie 1** et **partie 2**.

4.8.1 Introduction

En programmation Windows, les trames de pile sont nécessaires pour les procédures de fenêtre, les procédures d'hameçonnage (hooking), le super-classement et le sous-classement, les procédures de chargement et de déchargement de DLL et autres callbacks. Les procédures callback sont toutes appelées par Windows, en utilisant le propre thread de votre programme.

Dans GoAsm la création et l'utilisation de trames de pile est entièrement automatisée lorsque vous utilisez FRAME ... ENDF.

Voir la section **trames de pile automatisées** pour savoir comment l'utiliser en pratique.

Entre Windows 32 et 64 bits les trames de pile sont différentes et doivent donc être traitées de manière spécifique. Mais la syntaxe pour utiliser FRAME ... ENDF ainsi que leurs instructions d'accompagnement telles que LOCALS et USEDATA..ENDU est la même sur les deux plateformes. Pour cette raison, lorsque vous utilisez FRAME ... ENDF il est possible d'utiliser le même script source pour les deux. Voir le chapitre **écriture de programmes 64 bits** pour plus d'informations sur la programmation 64 bits en général.

4.8.2 Trames de pile en Windows 32 bits

Le travail dévolu à la trame de pile est dicté par la convention d'appel utilisée. Windows 32 bits utilise la convention d'appel standard (STDCALL). Dans ce contexte, la trame de pile dans une procédure de fenêtre doit accomplir quatre actions :

- accéder aux paramètres envoyés par Windows (qui sont justement envoyés sur la pile)
- rétablissement de l'équilibre de la pile avant de retourner à l'appelant
- allocation d'un espace sur la pile pour les données locales
- préservation pour Windows du contenu des registres EBX, ESI, EDI et EBP si ces derniers sont appelés à être modifiés.

Ces 4 actions sont aussi importantes les unes que les autres.

Accès aux paramètres à partir de la pile

Windows pousse les paramètres sur la pile avant d'appeler votre procédure de fenêtre de la même manière que vous le faites avant d'appeler une API dans vos programmes. Lorsque Windows appelle une procédure de fenêtre, il envoie les paramètres suivants sous forme de Dwords placés sur la pile (les mots utilisés ici sont ceux couramment utilisés pour les nommer) :

```
hwnd        handle de la fenêtre
uMsg        identificateur de message
wParam      donnée
lParam      donnée
```

Votre procédure de fenêtre a besoin d'accéder à ces paramètres. Une façon de le faire est de les extraire de la pile par des POP successifs en tant que données statiques, mais il est plus judicieux (et plus sûr) de conserver ces paramètres sur la pile et de les adresser directement. Cela vaut d'autant mieux que les procédures de fenêtres *s'appellent elles-mêmes* parfois. Cela peut sembler étrange, mais un exemple suffira à s'en convaincre. Supposons que votre procédure ait besoin de remplir la fenêtre avec un matériau approprié au bon moment. Ceci s'appelle "peindre" la fenêtre. Cela se fait en réponse au message Windows WM_PAINT (numéro de message 0Fh). Maintenant, la bonne façon de répondre à ce message consiste tout d'abord à appeler l'API *BeginPaint*, puis à peindre la fenêtre, puis appeler enfin l'API *EndPoint*. Une des choses que *BeginPaint* fait est de préparer la fenêtre pour la peinture. Ce faisant, elle envoie un autre message à votre procédure de fenêtre, cette fois WM_ERASEBKGND (numéro de message 14h). Ainsi, alors que votre procédure de fenêtre est aux prises avec ce deuxième message, elle n'est pas encore revenue de l'API *BeginPaint*. Après que le deuxième message a été traité, *BeginPaint* sera de retour. Ainsi, la procédure de fenêtre est-elle *récursive*, ce qui revient à dire qu'elle peut revenir sur elle-même. Pour chaque message (sauf pour hwnd) les paramètres seront différents. Si ces derniers sont maintenus sur la pile, cela signifie que chaque fois que la procédure de fenêtre sera appelée les paramètres seront conservés sur une partie différente de la pile et ne pourront pas être écrasés.

Voici, à titre d'exemple, une trame de pile typique sur 32 bits :

```
TypicalStackFrame:
PUSH EBP          ; sauvegarde de EBP qui va être altéré                  ┐   appelé
MOV EBP,ESP       ; EBP mémorise la valeur courante du pointeur de pile   ┘ "prologue"
;                 ; POINT "X"
;                 ; code pour isoler le message WM_PAINT
PUSH ADDR PAINTSTRUCT
PUSH [hwnd]
CALL BeginPaint   ; API prêt à peindre la fenêtre
;                 ; peinture de la fenêtre et appel de l'API EndPaint
;
MOV ESP,EBP       ; rétablissement ancienne valeur du pointeur de pile    ┐   appelé
POP EBP           ; restauration de EBP                                   │ "épilogue"
RET 10h           ; retour à l'appelant en ajustant le pointeur de pile   ┘
```

Pendant toute récursion, ESP sera changé dans la mesure où une utilisation ultérieure de la pile interviendra. En revanche, EBP sera toujours sauvegardé et restauré par cette procédure de sorte qu'il pourra toujours être invoqué pour accéder à la partie correcte de la pile afin que le message soit traité.

Cette particularité sera probablement mieux illustrée en approfondissant le fonctionnement de la pile dans l'exemple qui précède. Dans une procédure de fenêtre typique répondant au message WM_ERASEBKGND, prenons un instantané de la pile lorsque l'exécution est au point "X". La pile va alors ressembler à ceci (j'ai passé sous silence une bonne partie de l'utilisation de la pile par souci de clarté) :

ebp-4h	le prochain PUSH viendra ici
ebp	sauvegarde valeur de EBP
ebp+4h	adresse de retour de l'appelant
ebp+8h	hwnd
ebp+0Ch	uMsg
ebp+10h	wParam

ebp+14h	lParam
ebp+18h	autre usage de la pile (BeginPaint, etc.)
ebp+1Ch	
ebp+20h	
ebp+24h	
ebp+28h	sauvegarde valeur de EBP
ebp+2Ch	adresse de retour de l'appelant
ebp+30h	hwnd
ebp+34h	uMsg
ebp+38h	wParam
ebp+3Ch	lParam

Ici, les paramètres lParam, wParam, uMsg et hwnd qui sont sur la pile de EBP+3Ch à EBP+30h sont ceux envoyés à la procédure de fenêtre sur le message WM_PAINT. Puis, à EBP+2Ch nous trouvons l'adresse de l'appelant qui envoie le message WM_PAINT (ce sera un appel en provenance d'une DLL Windows). A EBP+28h se situe la valeur de EBP enregistrée sur la première instruction d'entrée dans la procédure de fenêtre sur le message WM_PAINT. Entre EBP+24h et EBP+1Ch on trouve l'utilisation de la pile de votre procédure de fenêtre avant que vous n'appeliez l'API *BeginPaint*. Concrètement, ces emplacements seraient occupés par deux PUSHes, puis l'adresse de retour sur l'appel de l'API (dans votre procédure de fenêtre) mise sur la pile. Le quatrième PUSH ici pourrait être quelque chose poussé sur la pile par *BeginPaint* lui-même avant d'envoyer le message WM_ERASEBKGND. En pratique, il pourrait y avoir beaucoup plus d'utilisation de la pile à l'intérieur *BeginPaint* ici. La prochaine chose que vous voyez à EBP+14h est à nouveau lParam. Mais celui-ci est différent de son homologue situé à EBP+3Ch. Il s'agit du lParam envoyé avec WM_ERASEBKGND. Le reste des entrées à la valeur actuelle de EBP sont en rapport avec le message WM_ERASEBKGRND.

Voyons maintenant ce qui se produit au retour de *BeginPaint*. Puisque nous savons que *BeginPaint* restaurera toujours EBP à la valeur qui était la sienne à l'entrée de l'API, nous savons que EBP au retour pointera vers la pile à EBP+28h dans la trame de pile ci-dessus. A partir de ce point, vous pouvez voir que les paramètres antérieurs (ceux envoyés avec WM_PAINT) peuvent être consultés à l'aide de EBP+8h, EBP+0Ch, EBP+10h et EBP+14h comme avant. Ils ont été préservés et n'ont pas été écrasés par l'appel à *BeginPaint* et par la récursion dans la procédure de fenêtre.

Restauration de la pile à l'équilibre avant de retourner à l'appelant

C'est également le travail de la procédure de Windows que de restaurer la pile à l'équilibre avant qu'elle ne retourne à l'appelant. Cela implique de déplacer le pointeur de pile à une valeur supérieure et multiple de 4 octets (soit un double-mot) pour chaque argument envoyé (chaque PUSH de l'appelant a réduit la valeur de ESP de quatre octets afin qu'il pointe sur un emplacement plus élevé sur la pile). C'est exactement ce que Windows fait lui-même lorsque vous appelez une API. Par exemple, dans

```
PUSH ADDR PAINTSTRUCT
PUSH [hwnd]
CALL BeginPaint     ; fait en sorte qu'on puisse être prêt à peindre la fenêtre
```

le pointeur de pile (ESP) a été réduit de 8 octets en raison des 2 PUSHes avant l'appel de l'API *BeginPaint*. Mais après le retour de *BeginPaint*, le pointeur de la pile récupère sa valeur d'origine d'avant les 2 PUSHes. Ce rétablissement est du au fait que la pile a été incrémentée de 8 octets en sortie de *BeginPaint* avant de retourner à votre code.

La plupart des procédures Windows nécessitent 4 paramètres de sorte que le pointeur de pile doit être incrémenté de 16 octets pour la restaurer. D'autres types de callback ont des nombres différents de paramètres. Le SDK de Windows donne les informations appropriées sur ceux-ci.

L'appelant peut utiliser `ADD ESP, 10h` pour ajouter les 16 octets nécessaires en retour d'appel, mais le plus simple est de confier ce réajustement à la procédure elle-même qui déplacera donc le pointeur de la pile avant de retourner à l'appelant en utilisant l'instruction RET suivie d'un nombre, par exemple RET 10h. Cette instruction obtient par un POP l'adresse de retour de l'appelant, incrémente le pointeur de pile (re-

gistre ESP) du nombre d'octets nécessaires – dans ce cas 16 –, puis renvoie finalement l'exécution à l'adresse de retour de l'appelant.

Création d'un espace sur la pile pour les données locales

Une autre tâche importante dévolue à la procédure callback est d'allouer, en cas de besoin, un espace aux données locales. Il s'agit de données qui sont conservées dans la pile pendant que l'exécution se poursuit dans la procédure callback. Elles sont perdues lorsque l'exécution quitte la procédure. Nous avons déjà vu comment les données sur la pile (sous la forme de paramètres) sont conservées en constituant une trame de pile. L'espace pour les données locales utilise le même principe. Supposons que nous ayons besoin d'espace pour 3 Dwords sur la pile parce que nous voulons préserver ces données cependant que la procédure de fenêtre est en récursion. Le code à utiliser pourrait être le suivant :

```
TypicalStackFrame:
PUSH EBP        ; sauvegarde de EBP qui va être altéré                  ┐
MOV EBP,ESP     ; EBP mémorise la valeur courante du pointeur de pile    │ "prologue"
SUB ESP,0Ch     ; on constitue un espace pour les données locales        ┘
;               ; POINT "X"
;
;               ; code de la procédure de fenêtre
;
MOV ESP,EBP     ; rétablissement ancienne valeur du pointeur de pile     ┐
POP EBP         ; restauration de EBP                                    │ "épilogue"
RET 10h         ; retour à l'appelant en ajustant le pointeur de pile    ┘
```

Ici, nous avons déplacé le pointeur de pile de 12 octets, ce qui est exactement la même chose que si nous avions procédé à 3 PUSH successifs. Ceci permet d'obtenir une zone de la pile qui ne peut être utilisée à d'autres fins.

En utilisant FRAME ... ENDF vous pouvez créer automatiquement la trame de pile.

Voici une utilisation typique de FRAME ... ENDF qui fait la même chose que `TypicalStackFrame` ci-dessus et fournit des noms pour les paramètres qui sont envoyés à la procédure de fenêtre et des noms pour chaque DWORD de donnée locale :

```
WndProc FRAME hwnd,uMsg,wParam,lParam
LOCALS hDC,hInst,KEEP
;           ; POINT "X"
;
;           ; le code vient ici
;
RET
ENDF
```

La pile courante apparaît comme ceci au point "X" :

ebp–10h	le prochain PUSH viendra ici
ebp–0Ch	espace pour donnée locale KEEP ← **ESP est actuellement ici (sommet des données locales)**
ebp–8h	espace pour donnée locale hInst
ebp–4h	espace pour donnée locale hDC
ebp	sauvegarde valeur de EBP ← **EBP valeur donnée de ESP lorsqu'il était ici**
ebp+4h	adresse de retour de l'appelant
ebp+8h	hwnd
ebp+0Ch	uMsg
ebp+10h	wParam
ebp+14h	lParam
ebp+18h ebp+1Ch	} autre usage de la pile

Bien sûr, si le pointeur de pile est soumis à ces déplacements, il va de soi qu'il doit être restauré en sortie. Mais cette fois, l'opération est automatique, car EBP est tout simplement restauré à sa valeur avant de retourner à l'appelant, et EBP avait du reste été enregistré *avant* le déplacement du pointeur de pile pour faire de la place à des données locales.

Préservation pour Windows des registres EBX, ESI, EDI et EBP

Enfin, il est important de rappeler que votre procédure de fenêtre se doit de préserver les registres EBX, ESI, EDI et EBP. EBP est déjà mémorisé puis restauré respectivement par les codes de prologue et d'épilogue. Quant à EBX, ESI et EDI, il est prudent de les sauvegarder puis les restaurer systématiquement même s'ils ne sont pas modifiés par votre procédure de fenêtre afin de prévenir une éventuelle modification ultérieure de votre code impactant un ou plusieurs de ces registres. Au reste, vous pouvez compter sur le fait quasi-certain qu'une API Windows préservera ces registres. Cette caractéristique est particulièrement utile dans la programmation assembleur parce que vous pouvez conserver les handles utiles ainsi que d'autres valeurs dans ces registres tout en appelant des API. Vous pouvez facilement préserver EBX, ESI et EDI en recourant à l'instruction USES, ainsi que le montre l'exemple suivant :

```
WndProc FRAME hwnd,uMsg,wParam,lParam
USES EBX,ESI,EDI
LOCALS hDC,hInst,KEEP
;
;                ; le code est placé ici
;
RET
ENDF
```

4.8.3 Trames de piles avec Windows 64 bits

Le travail qui incombe à la trame de pile est dicté par la convention d'appel utilisée. Windows 64 bits utilise la convention d'appel dite rapide (FASTCALL). Au lieu que l'appelant mette les paramètres sur la pile comme dans la convention STDCALL, les quatre premiers paramètres sont mis dans les registres RCX, RDX, R8 et R9. Les éventuels paramètres complémentaires sont mis sur la pile. Ainsi, la procédure de fenêtre doit-elle exécuter les tâches qui suivent :

- enregistrer sur la pile les paramètres envoyés dans les registres et accéder à tous les paramètres supplémentaires envoyés par Windows sur la pile
- fournir un espace sur la pile pour les données locales
- préserver pour Windows les registres dits "non volatiles" dès qu'ils sont modifiés. Il s'agit de RBP, RBX, RDI, RSI, R12 à R15 et XMM6 à XMM15.

Il n'est pas nécessaire que la procédure de fenêtre restaure l'équilibre de la pile, avant de retourner à l'appelant. Ce travail est fait par l'appelant.

Enregistrement et accès aux paramètres

Les registres RCX, RDX, R8 et R9 sont qualifiés de "volatiles" dans le sens où Windows ne garantit pas le maintien de leur intégrité au travers d'un appel d'API. Pour autant, ils accueillent les quatre premiers paramètres d'un appel d'API. Cela signifie que, dès que votre procédure de fenêtre fait un appel d'API, il est possible que les paramètres soient écrasées. Pour cette raison, il est indispensable d'en conserver une copie quelque part. L'utilisation des registres non volatiles pourrait être une solution, mais la documentation de Windows recommande qu'ils soient maintenus sur la pile. Apparemment, cela se fait dans les API elles-mêmes. Dans la convention d'appel FASTCALL telle que documentée et mise en œuvre dans Windows 64 bits, l'appelant est tenu de déplacer RSP négativement de 32 octets avant de faire l'appel afin de fournir un espace sur la pile pour que les paramètres soient sauvegardés. L'espace de pile ainsi réservé est appelé – qui l'eût cru ? – "espace réservé". Chaque paramètre a son propre espace réservé connu. Evidemment, on pourrait se demander pourquoi FASTCALL a été choisi puisque les paramètres se retrouvent, de toute façon, sur la pile. Il se dit que cette situation traduirait une intention non aboutie de la part des programmeurs !

Pour faire face à l'obligation d'enregistrer les paramètres envoyés dans les registres, lorsque vous utilisez FRAME ... ENDF sur une plateforme 64 bits, il faut savoir que GoAsm génère automatiquement des instructions qui prennent l'allure suivante au début de la trame de la pile :

```
MOV [RSP+8h],RCX
MOV [RSP+10h],RDX
MOV [RSP+18h],R8
MOV [RSP+20h],R9
PUSH RBP
MOV RBP,RSP
```

Ce code met les paramètres dans leurs espace réservé sur la pile. S'il y a moins de quatre paramètres, ces instructions ne sont pas toutes émises. Notez que le cinquième paramètre, s'il existe, est déjà sur la pile à [RSP+28h], le sixième paramètre à [RSP+30h], etc. Les deux dernières instructions rétablissent RBP dans sa fonction de pointeur vers les données après l'avoir préalablement sauvegardé de telle sorte qu'il puisse être restauré ultérieurement.

Dans l'épilogue vous vous attendez à voir quelque chose comme :

```
LEA RSP,[RBP]
POP RBP
RET
```

L'instruction LEA est utilisée ici à la place d'un simple `MOV RSP,RBP` de manière à aider le gestionnaire d'exception Windows à identifier l'épilogue.

Construire un espace sur la pile pour les données locales

Cela fonctionne exactement de la même manière que pour une trame de pile 32 bits sauf que chaque élément de donnée locale doit être au moins de la taille d'un Qword. Ainsi, par exemple, pour trois Qwords de données locales, l'instruction destinée à créer la place nécessaire devrait s'écrire `SUB RSP,18h`.

Préservation des registres non volatiles pour Windows

RBP est déjà sauvegardé par le code du prologue puis restauré par celui de l'épilogue. Comme pour les registres à usage général, si les registres RBX, RDI, RSI et R12 à R15 sont modifiés par la procédure de fenêtre, ils auront besoin de voir leur valeur initiale restaurée. La meilleure façon de le faire est d'utiliser l'instruction USES qui les copie sur la pile. Les registres XMM6 à XMM15 peuvent être sauvegardés et restaurés en bloc, en utilisant les instructions du processeur FXSAVE et FXRSTOR qui n'agissent toutefois qu'au niveau de l'Unité de Calcul en virgule flottante (FPU).

Considérons maintenant l'utilisation assez classique de FRAME ... ENDF qui suit :

```
WndProc FRAME hwnd,uMsg,wParam,lParam
USES RBX,RSI,RDI
LOCALS hDC,BUFFER[256]:B
;                   ; POINT 'X'
;                   ; le code s'écrit ici
;
RET
ENDF
```

Voici maintenant comment la trame de pile se présente en assemblage 64 bits, en utilisant les valeurs de RSP et RBP au début du code au POINT 'X' (notez que RBP affiche 32 octets de moins que RSP affichait en entrant dans la procédure : ce décalage est dû aux instructions PUSH portant successivement sur les registres RBP, RBX, RSI et RDI):

RBP–118h	le PUSH suivant viendra ici
RBP–110h	} buffer de 256 octets ← **RSP est actuellement ici (sommet des données locales)**
RBP–8h	hDC
RBP	sauvegarde valeur de RDI ← **RBP valeur donnée de RSP lorsqu'il était ici**
RBP+8h	sauvegarde valeur de RSI
RBP+10h	sauvegarde valeur de RBX
RBP+18h	sauvegarde valeur de RBP
RBP+20h	Adresse de retour à l'appelant
RBP+28h	emplacement du paramètre hwnd (à l'origine dans RCX)
RBP+30h	emplacement du paramètre uMsg (à l'origine dans RDX)
RBP+38h	emplacement du paramètre wParam (à l'origine dans R8)
RBP+40h	emplacement du paramètre lParam (à l'origine dans R9)
RBP+48h	paramètre n° 5 (si présent)
RBP+50h	paramètre n° 6 (si présent)

4.9 Trames de pile automatisées utilisant FRAME...ENDF, LOCALS et USEDATA

4.9.1 Introduction

Les bases

FRAME ... ENDF, implémenté dans GoAsm, est similaire au PROC ... ENDP utilisé dans MASM tout en offrant beaucoup plus de possibilités. Les sous-programmes peuvent également utiliser les données sur la pile en utilisant **USEDATA ... ENDU**. Et vous pouvez déclarer des données locales *dynamiquement*. Cela vous permet, à l'intérieur d'une procédure de fenêtre, de déclarer uniquement les données locales qui sont effectivement nécessaires à un message particulier. Vous pouvez utiliser *des mots définis localement* qui opèreront uniquement à l'intérieur de l'espace délimité par FRAME..ENDF ou des zones de USEDATA..ENDU qui leur sont associées.

La syntaxe est plus claire et le script source est beaucoup plus facile à comprendre, car il n'y a pas de contrôle de type ou de paramètre.

Voici une manière de procéder si vous utilisez FRAME ... ENDF pour constituer une trame de pile automatisée :

```
WndProc FRAME hwnd,uMsg,wParam,lParam
USES EBX,ESI,EDI
LOCALS hDC,BUFFER[256]:B
;
;           ; le code s'écrit ici
;
RET
ENDF
```

Lorsque vous utilisez FRAME..ENDF de cette manière, GoAsm crée une trame de pile à votre insu. Pour cette raison, il convient d'être un peu méfiant. Les programmeurs en assembleur aiment savoir tout ce qui se passe, et c'est d'ailleurs la raison majeure pour laquelle ils utilisent ce langage ! Nous allons donc décrire cette question en détail bien qu'il ne soit pas nécessaire d'en connaître les tenants et les aboutissants avec précision.
Si vous l'estimez nécessaire, vous pouvez passer sur ces détails et vous intéresser plutôt au fonctionnement de FRAME ... ENDF dans l'exemple de programme **Hello World 3** proposé en annexe.

Dans le code ci-dessus, **FRAME** prescrit à GoAsm de constituer une trame de pile automatisée dont **ENDF** marquerait la fin. Les mots après "FRAME" sont les **paramètres**. Dans notre cas, il y a quatre paramètres qui ont pour nom ceux qui sont indiqués. Il n'y a pas besoin d'ajouter quoi que ce soit d'autre puisque GoAsm connaît la taille des paramètres. En codage 32 bits, ils sont toujours en format Dword ; en codage 64 bits, ils sont toujours en format Qword.

USES désigne à GoAsm les registres qui ont besoin d'être préservés dans la trame. Ici, nous utilisons des registres 32 bits, mais en assemblage 64 bits, l'instruction se lit comme USES RBX, RSI, RDI sans avoir à modifier le code source (dans le cas de PUSH *registre* c'est le même opcode qui est généré pour chacune des deux plate-formes).

LOCALS vous permet de déclarer un label pour les données locales dans la trame. GoAsm ajoute la taille de ces données locales et réserve l'espace à cet effet sur la pile. Lors de la déclaration des données locales, Dword est la valeur par défaut en assemblage 32 bits, Qword est la valeur par défaut en assemblage 64 bits. La valeur par défaut est utilisée si vous ne spécifiez pas une taille pour les données. Ainsi dans l'exemple, hDC est une valeur Dword. Il y a aussi une zone sur la pile appelée BUFFER (tampon). Celle-ci est de 256 octets en raison de la notation [256]:B. Au lieu de B, vous pourriez utiliser W, D, Q ou T pour, respectivement, Word, Dword, Qword ou Ten-Word. Vous pouvez également utiliser le nom d'une structure ainsi que le décrit la section **Utilisation des structures comme données locales dans une trame de pile**.

GoAsm crée automatiquement le **code de prologue** comme décrit dans les sections 32 bits ou 64 bits ci-dessus. GoAsm va ajouter le code d'épilogue à chaque fois qu'il rencontre un RET dans la trame délimitée par FRAME ... ENDF.

Accès aux paramètres et aux données locales à l'intérieur d'une trame automatisée

Dans une trame constituée de cette façon vous pouvez accéder aux paramètres envoyés à la procédure de fenêtre. Dans l'exemple suivant écrit pour Windows 32 bits, les offsets accolés à EBP qui sont générés par GoAsm sont donnés sur l'hypothèse qu'il n'y a pas de déclaration USES (le codage 64 bits est très similaire mais utilise RBP et chaque poste de pile occupe 8 octets au lieu de 4) :

```
PUSH [hwnd]             ; équivaut à PUSH [EBP+8h]
MOV EAX,[uMsg]          ; équivaut à MOV EAX,[EBP+0Ch]
MOV EBX,ADDR wParam     ; équivaut à LEA EBX,[EBP+10h]
PUSH ADDR lParam        ; équivaut à PUSH EBP suivi de ADD D[ESP],14h
MOV EBX,[hDC]           ; équivaut à MOV EBX,[EBP-10h]
MOV EBX,ADDR BUFFER     ; équivaut à LEA EBX,[EBP-110h]
```

Dans cette portion de code, on voit que GoAsm se charge de trouver sur la pile la bonne position de la variable à laquelle vous souhaitez accéder afin d'en lire ou modifier le contenu. Dès lors, votre seul souci se réduit à connaître le nom de la variable tout en faisant l'impasse sur sa position au sein de la pile. Si vous utilisez l'option /l sur la ligne de commande, vous pouvez vous rendre compte par vous-même de la réalité de ce codage en consultant le fichier-listing. Sinon, vous pouvez également l'observer en utilisant le débogueur.

Notez que l'adresse du buffer est donnée *à son point le plus négatif*. Il est donc correct de coder :

```
MOV D[BUFFER+10h],44h
```

si vous insérez la valeur 44h à un Dword dont l'octet le moins significatif est à 16 octets du début du buffer.

Notez que GoAsm définit la valeur de EBP *après* avoir poussé en pile (PUSHing) les registres mentionnés dans la déclaration USES. Cette disposition permet à l'ensemble des données locales d'être ajusté dynamiquement sur une base spécifique à un message. Mais cela signifie aussi que, si vous avez une déclaration USES dans un FRAME, l'offset des paramètres par rapport à EBP sera plus grand que le contraire. Ainsi, par exemple, si vous mettez en pile (PUSH) trois registres dans une FRAME avec une déclaration USES comme suit :

```
USES EBX,EDI,ESI
```

alors, EBP sera mis en pile plus loin et au-delà des paramètres avec un décalage de 12 octets. Donc, dans cet exemple, *hwnd* serait à [EBP+14h], *Msg* à [EBP+18h], *wParam* à [EBP+1Ch] et *lParam* à [EBP+20h]. Lors du codage vous ne devez pas vous soucier de la position exacte des paramètres relatifs à EBP, mais vous devez être au courant de cette particularité si vous examinez votre code dans le débogueur. Voir aussi la section **ce que vous pouvez voir dans le débogueur**.

Utilisation de structures comme données locales dans une trame de pile

Dans l'exemple qui suit, les données locales de taille adaptée à la structure RECT, préalablement déclarée dans votre script source, sont établies sur la pile.

```
RECT STRUCT
   left     DD 0
   top      DD 0
   right    DD 0
   bottom   DD 0
ENDS
;
WndProc FRAME hwnd,uMsg,wParam,lParam
LOCALS hDC,rc1:RECT
;
;                ; le code s'écrit ici
;
RET
ENDF
```

Chaque élément de la structure RECT est accessible de la même façon que si elle était en données statiques. Par exemple (en utilisant encore des exemples 32 bits) :

```
MOV EAX,[rc1.right]        ; équivaut à MOV EAX,[EBP-0Ch]
MOV EAX,[ESI+RECT.right]   ; équivaut à MOV EAX,[ESI+8h]
MOV EAX,SIZEOF RECT        ; équivaut à MOV EAX,10h
```

```
MOV EAX,ADDR rc1.right      ; équivaut à LEA EAX,[EBP-0Ch]
PUSH [rc1.right]            ; équivaut à PUSH [EBP-0Ch]
PUSH ADDR rc1.right         ; équivaut à PUSH EBP suivi de ADD D[ESP],-0Ch
PUSH ADDR rc1               ; équivaut à PUSH EBP suivi de ADD D[ESP],-14h
```

4.9.2 Pratique des trames de pile automatisées

Quelques considérations pratiques

La manière par laquelle vous voudrez utiliser les facilités offertes par FRAME ... ENDF sera une question de choix personnel comme nous allons le voir :

- Vous pouvez englober tout votre code de procédure de fenêtre dans une trame FRAME... ENDF. Dans ce cas, vous devrez vous assurer que les sous-routines utilisent un RET "normal" en utilisant **RETN**. Vous pouvez souhaiter maintenir la trame FRAME...ENDF aussi compacte que possible, mais accéder encore aux paramètres et données locales de l'extérieur. Ce sera possible en spécifiant une zone **USE-DATA... ENDU**.
- Vous pouvez déclarer des données locales sous forme de message spécifique plutôt que pour la trame de pile dans son ensemble. Vous pouvez le faire en positionnant la déclaration **LOCAL**.
- Vous pouvez souhaiter combiner ces méthodes avec une table de message pour créer une procédure de fenêtre réduite.
- Enfin, vous pouvez vouloir libérer les zones de données locales et construire ensuite de nouvelles zones de données locales avec l'instruction **LOCALFREE**.

Appel de procédures à l'intérieur d'une trame de pile – Usage de RETN

Vous pouvez avoir autant de points de retour de la procédure FRAME utilisant RET que vous le souhaitez. Chacun va produire un code d'épilogue qui sera exécuté au moment de quitter la trame de pile. Cependant, cela signifie également que, si vous avez des procédures additionnelles à l'intérieur de la trame de pile, vous devez veiller à les conclure par RETN (RET "normal") pour éviter la création de code d'épilogue pour celles-ci. Il en résulterait en effet des dépilages inappropriés, source irrémédiable de plantage...
Seule la procédure FRAME principale peut être appelée depuis l'extérieur de celle-ci. L'appel à d'autres procédures peut entraîner des résultats imprévisibles. Ceci, parce que les procédures dans l'enveloppe FRAME...ENDF s'attendent à adresser des paramètres et des données locales en utilisant le pointeur de pile (RBP ou EBP) alors qu'il n'a pas été initialisé en cas d'appel de l'extérieur.

Voici un exemple concret :

```
WndProc FRAME hwnd,uMsg,wParam,lParam
USES EBX,ESI,EDI
LOCAL hDC,BUFFER[256]:B
MOV EAX,[uMsg]          ; récupération du message envoyé par Windows
CMP EAX,0Fh             ; on regarde si c'est WM_PAINT
JNZ >L2                 ; non
CALL WINDOW_PAINT       ; on peint la fenêtre
XOR EAX,EAX             ; renvoie zéro pour montrer que le message est traité
RET                     ; restauration de la pile et retour à Windows
;
L2:
ARG [lParam],[wParam],[uMsg],[hwnd]
INVOKE DefWindowProcA   ; permet à Windows de traiter avec le message
RET                     ; restauration de la pile et retour à Windows
;
WINDOW_PAINT:
;  code pour peindre la fenêtre
RETN                    ; exécute un retour ordinaire de la procédure de peinture
;
ENDF                    ; stoppe toute action FRAME à partir de ce point.
```

Appel de procédures à l'extérieur d'une trame de pile
Utilisation de USEDATA...ENDU

Vous pouvez préférer conserver une trame plus compacte et ne procéder à des CALL qu'en direction de *l'extérieur de celle-ci* pour accroître la compacité de votre code et en améliorer ainsi la lisibilité. Vous pouvez le faire tout en conservant l'accès aux paramètres et aux données locales dans la trame en utilisant la déclaration de USEDATA suivie du nom de la trame concernée. Par exemple, le message WM_PAINT dans la trame ci-dessus pourrait appeler cette procédure :

```
PAINT:
USEDATA WndProc
INVOKE BeginPaint, [hwnd],ADDR lpPaint      ;récupère en EAX le DC à utiliser
MOV [hDC],EAX
INVOKE Ellipse, [hDC], [lpPaint.rcPaint.left],    \
                       [lpPaint.rcPaint.top],     \
                       [lpPaint.rcPaint.right],   \
                       [lpPaint.rcPaint.bottom]
INVOKE EndPaint, [hwnd],ADDR lpPaint
XOR EAX,EAX
RET
ENDU
```

Nous venons de lister ici le code de la procédure PAINT qui utilise les données locales dans la FRAME appelée WndProc. Tout ce code est extérieur à l'enveloppe FRAME...ENDF.
Vous pouvez également utiliser USEDATA pour accéder aux données locales dans d'autres zones USEDATA..ENDU.
Assurez-vous que USEDATA est utilisée *postérieurement dans le script source* à toutes les déclarations de paramètres et de données locales sur lesquels elle repose. Ceci est dû au fait que GoAsm est un assembleur travaillant en une seule passe et qu'il a besoin, pour ce faire, de trouver la position de ces données pleinement définie.
Si une procédure appelée à partir d'une zone FRAME ou USEDATA n'a besoin d'accéder à aucun paramètre, donnée locale, ou mot définis localement, alors elle ne doit pas avoir sa propre déclaration USEDATA.

Points de sortie multiples ou procédure à l'intérieur d'une zone USEDATA... ENDU

Tout comme lors de l'utilisation de FRAME ... ENDF, vous pouvez avoir autant de points de retour de la procédure USEDATA utilisant RET que vous le souhaitez. Chacun va produire le code d'épilogue approprié qui est exécuté au moment de quitter la zone USEDATA. Cependant, cela signifie également que si vous avez des procédures supplémentaires au sein de la zone USEDATA vous devez veiller à utiliser RETN (RET "normal") pour éviter que l'assembleur ne leur crée un code d'épilogue.
Seule la première procédure USEDATA doit être appelée depuis l'extérieur de la zone USEDATA..ENDU. Ceci, parce que le code approprié pour accéder à la pile ne sera mis en place que pour cette première procédure.

4.9.3 Utilisation avancée des trames de pile automatisées

Déclaration de donnée locale à message spécifique
Positionnement de l'instruction LOCAL

Dans les exemples exposés jusqu'ici, la zone des données locales localisée sur la pile avait été déclarée globalement pour le FRAME. Mais vous pouvez préférer mettre en place tout ou partie des données locales sur une base de message spécifique. Voici un exemple de cette façon de faire :

```
WndProc FRAME hwnd,uMsg,wParam,lParam
USES EBX,ESI,EDI
LOCAL hDC               ; déclare hDC pour un usage de trame étendue
MOV EAX,[uMsg]          ; récupération du message envoyé par Windows
CMP EAX,0Fh             ; est-ce WM_PAINT ?
JNZ >L2                 ; non
CALL WINDOW_PAINT       ; c'est WM_PAINT, alors on peint la fenêtre
XOR EAX,EAX             ; on retourne zéro pour signifier que le message a été traité
```

```
RET                     ; restauration de la pile et retour à Windows
;
L2:
ARG [lParam],[wParam],[uMsg],[hwnd]
INVOKE DefWindowProcA   ; permet à Windows de traiter avec le message
RET                     ; restauration de la pile et retour à Windows
;
ENDF                    ; arrête toute action de FRAME à partir de ce point
;
WINDOW_PAINT:
USEDATA WndProc         ; utilise les paramètres et les données locales de WndProc
LOCAL ps:PAINTSTRUCT    ; construit des zones de données locales
LOCAL BUFFER[1024]:B    ; spécifiquement à ce message
;
ARG ADDR ps,[hwnd]
INVOKE BeginPaint       ; prêt à peindre la fenêtre
MOV [hDC],EAX           ; sauvegarde le contexte de périphérique dans la donnée locale hDC
;  code pour peindre la fenêtre
RET                     ; effectue un retour ordinaire de la procédure de peinture
ENDU                    ; met fin à l'utilisation de la trame de données WndProc
```

Création d'une procédure de fenêtre réduite

En utilisant les méthodes décrites, vous pouvez créer une procédure de fenêtre en utilisant une table de messages. La procédure de fenêtre en tant que telle n'a pas besoin doit être plus complexe que ceci :

```
WndProc FRAME hwnd,uMsg,wParam,lParam
MOV EAX,[uMsg]
MOV ECX,SIZEOF MESSAGES/8
MOV EDX,OFFSET MESSAGES
:
DEC ECX
JS >.notfound
CMP [EDX+ECX*8],EAX    ; est-ce le message correct ?
JNZ <                  ; non, on va voir le suivant...
CALL [EDX+ECX*8+4]     ; appel de la procédure correcte pour le message
JNC >.exit
.notfound
INVOKE DefWindowProcA,[hwnd],[uMsg],[wParam],[lParam]
.exit
RET
ENDF
```

Quelque part dans les sections data ou const, on pourrait imaginer le tableau suivant pour les messages (dans la pratique, il y en aurait beaucoup plus que cela) :

```
MESSAGES   DD 1h,CREATE       ; la valeur du message puis l'adresse du code
           DD 2h,DESTROY
           DD 0Fh,PAINT
NextLabel:
```

Puis, dans la section de code (et après WndProc) vous pourriez avoir le code suivant pour le traitement de ces messages :

```
CREATE:
USEDATA WndProc       ; utiliser les données de la pile dans la trame
                      ; de la procédure de fenêtre
USES EBX,EDI,ESI      ; préservation des registres pour Windows
LOCALS LocalData      ; établissement de la zone de données locales requise
;
; code à exécuter sur le message WM_CREATE
;
```

```
XOR EAX,EAX          ; retourne NC et EAX = 0 pour continuer à créer la fenêtre
RET                  ; restauration des registres puis RET
ENDU                 ; arrêt de toute action automatique et accès aux données
```

Dans la procédure de fenêtre réduite, *DefWindowProc* n'est pas appelée à moins que le message ne se trouve pas dans la table de message ou que le code du message retourne le flag de Carry à 1. Certains messages doivent appeler *DefWindowProc* même s'ils sont traités par la procédure de fenêtre – Voir le SDK Windows à ce sujet.

Mots définis localement utilisant #localdef ou LOCALEQU

Dans une zone FRAME..ENDF vous pouvez utiliser des *mots définis localement*. La définition peut être effectuée soit dans la zone FRAME..ENDF elle-même, soit dans une zone USEDATA..ENDU associée.

Leur champ d'application est limité aux zones d'action de FRAME ou USEDATA. Voir la section **héritage et portée d'action avec USEDATA..ENDU** pour plus de détails sur la façon dont cela fonctionne en pratique.

Vous définissez ces mots locaux en utilisant #localdef (ou LOCALEQU si vous préférez – ils font la même chose).

Par exemple :

```
FrameProc1 FRAME Param
#localdef THING1 23h
THING2 LOCALEQU 88h
;
MOV EAX,THING1      ; définition locale 23h
MOV EAX,THING2      ; définition locale 88h
;
RET
ENDF
;
MyFunction44: USEDATA FrameProc1
#localdef THING3 0CCh
;
MOV EAX,THING1      ; la définition locale devrait être 23h
MOV EAX,THING2      ; la définition locale devrait être 88h
;
RET
ENDU
```

Dans l'exemple ci-dessus, si THING1 et THING2 sont définis globalement (en utilisant #define ou EQU), cette définition est ignorée (la définition locale est prioritaire).

#undef a une priorité de portée locale. Si le mot appelé à être indéfini se trouve sur localement, alors #undef s'applique à lui. Sinon, #undef s'appliquera à un label global.

Portée d'un label réutilisable dans les trames de pile automatisées

On peut accéder aux labels réutilisables commençant par un point n'importe où au sein d'une trame de pile automatisée (qui peut être établie à l'aide FRAME...ENDF). D'autres labels uniques au sein de la trame sont ignorés pour cet usage de sorte que, par exemple,

```
ExampleProc FRAME Param
CMP EDX,EAX
JZ >.fin
LABEL1:
XOR EAX,EAX
.fin
RET
ENDF
LABEL2:
```

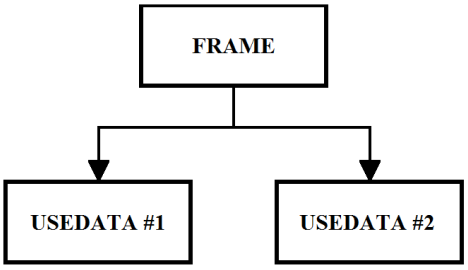
Ici, le saut vers le label `.fin` fonctionnera encore malgré l'existence de LABEL1. En effet, le label `.fin` porte sur toute la trame et pas seulement sur le code entre LABEL1 et LABEL2. En d'autres termes, l'utilisation de FRAME ... ENDF étend le champ d'application d'un label réutilisable à l'ensemble de la trame.

Héritage et portée avec USEDATA..ENDU

Une zone USERDATA..ENDU peut être associée soit directement avec une FRAME, soit avec une autre zone USERDATA..ENDU.

Cela permet aux utilisateurs expérimentés de sélectionner les données locales et les mots définis qu'une zone USEDATA peut utiliser.

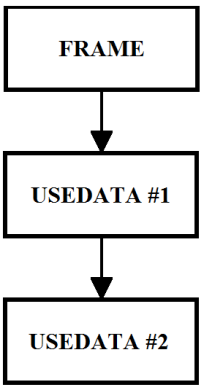
La disposition habituelle est d'avoir, pour chaque zone de USEDATA, un enfant de FRAME :



Ici chaque zone USEDATA peut accéder aux paramètres de FRAME, aux données locales et aux mots définis. Notez cependant que Usedata#2 a ses propres données locales et mots définis. Ceux-là seuls peuvent être référencés dans Usedata#2. Si Usedata#1 avait essayé d'y accéder (ou le code dans FrameExample, d'ailleurs), ils n'auraient pas été trouvés.

```
FrameExample FRAME Param
LOCAL LocalLabel1
#localdef CONSTANT 23h
;
RET
ENDF
;
Usedata#1: USEDATA FrameExample
MOV EAX,[LocalLabel1]
MOV EAX,CONSTANT
MOV EAX,[Param]
RET
ENDU
;
Usedata#2: USEDATA FrameExample
LOCAL Specific
#localdef SPECIFIC_CONSTANT 444444h
MOV EAX,[LocalLabel1]
MOV EAX,CONSTANT
MOV EAX,[Param]
MOV EAX,[Specific]
MOV EAX,SPECIFIC_CONSTANT
RET
ENDU
```

Dans la représentation qui suit, la première zone USEDATA est l'émanation de FRAME et la seconde zone USEDATA est sa sous-émanation.



Ici, bien que chaque zone USERDATA puisse accéder aux paramètres de la trame, aux données locales et aux mots définis, Usedata#2 peut également faire référence à des données

```
FrameExample FRAME Param
LOCAL LocalLabel1
#localdef CONSTANT 23h
;
RET
ENDF
;
Usedata#1: USEDATA FrameExample
LOCAL Specific
#localdef SPECIFIC_CONSTANT 444444h
RET
ENDU
;
Usedata#2: USEDATA Usedata#1
MOV EAX,[LocalLabel1]
MOV EAX,CONSTANT
MOV EAX,[Param]
MOV EAX,[Specific]
```

locales et des mots définis localement dans Usedata#1.

```
MOV EAX,SPECIFIC_CONSTANT
RET
ENDU
```

Diffusion des données locales et constitution de nouvelles données locales
Utilisation de LOCALFREE

LOCALFREE est disponible uniquement pour les programmes 32 bits et n'est pas supporté par les modes x86 ou x64 en raison de l'information d'utilisation de la pile en prologue enregistrée pour la gestion des exceptions.

Vous pouvez utiliser LOCALFREE pour libérer les zones de données locales pour en constituer de nouvelles. Cela peut aider à économiser de la mémoire si vous utilisez beaucoup la pile. LOCALFREE rendra une donnée locale existante déclarée dans une FRAME ou une zone USEDATA...ENDU, dans laquelle elle apparaît, inaccessible à tout le code placé en aval dans votre script source. Il n'affectera pas les données locales dans d'autres zones FRAME ou USEDATA. Lorsque GoAsm rencontre LOCALFREE dans le script source, il provoque la restauration de ESP/RSP à sa valeur dans la zone FRAME ou USEDATA avant que toute donnée locale n'ait été déclarée. Vous pouvez ensuite déclarer de nouvelles données locales à l'aide de LOCAL ou LOCALS.

Utilisez LOCALFREE uniquement lorsque la pile est à l'équilibre. Ne l'utilisez pas s'il y a des PUSHes en suspens qui doivent être POPpés. En effet, le changement d'ESP/RSP effacera en pratique tout PUSH en suspens.

A la fin d'une procédure vous n'avez pas besoin d'utiliser LOCALFREE puisque la pile est restaurée automatiquement sur un RET, de toute façon.

Voici un exemple de la façon d'utiliser LOCALFREE :

```
CREATE:
USEDATA WndProc          ; utilisation des données de la pile dans la trame
                         ; de la procédure de fenêtre
USES RBX,RDI,RSI         ; préservation des registres pour Windows
LOCALS BUFFER[4000]:B    ; établissement d'un grand buffer sur la pile
;
; part de code correspondant à l'exécution sur le message WM_CREATE
;
LOCALFREE                ; effacement du grand buffer
LOCALS BUFFER[256]:B     ; établissement d'un buffer plus petit sur la pile
;
; part de code correspondant à l'exécution sur le message WM_CREATE
;
XOR RAX,RAX              ; retourne NC et RAX=0 pour continuer la création de la fenêtre
RET                      ; restauration des registres puis RET
ENDU                     ; met fin à toute action automatique et accès aux données
```

4.9.4 Considérations syntaxiques

Quelques points de syntaxe lors de l'utilisation de FRAME...ENDF

- La syntaxe de FRAME ... ENDIF est comme suit, avec les éventuelles variations mentionnées ci-dessous :

```
CodeLabel:
    FRAME Parameter List     ; si des paramètres sont nécessaires
    USES Register List       ; s'il est nécessaire de sauvegarder des registres
    LOCAL Local List         ; si des variables locales sont requises
    ;
    ret
    ENDF
```

- Une instruction FRAME doit être précédée par un label, soit immédiatement avant sur la même ligne, soit sur la ligne qui précède. Il s'agit du "nom de trame".

- Une seule déclaration FRAME par trame est autorisée.
- Tous les paramètres doivent être immédiatement après la déclaration FRAME, séparés par des virgules. Pour poursuivre, le cas échéant, sur la ligne suivante, utiliser le caractère de continuation "\".
- Vous pouvez automatiquement sauvegarder et restaurer les registres à l'intérieur d'une trame avec l'instruction USES. ENDF arrête l'action de USES.
- Les sauts ne peuvent être que dans le FRAME lui-même. En effet, un FRAME a son propre code d'épilogue unique qui doit être mis en œuvre.
- Les CALLs peuvent s'adresser à l'extérieur de la trame. Si vous avez besoin d'accéder aux paramètres de la trame comme les données locales ou des mots définis, utilisez USEDATA.
- Si vous appelez une fonction dans la même trame, cette fonction doit utiliser RETN (RET "normal") au lieu de RET. Cela empêche la génération d'un code d'épilogue au moment de quitter la fonction.
- Les données locales doivent être déclarées en utilisant une ou plusieurs instructions LOCAL. Après la première instruction de code suivante, vous ne serez pas en mesure d'utiliser LOCAL à nouveau à moins que vous n'ayez libéré les données locales existantes en utilisant l'instruction LOCALFREE.
- LOCALFREE doit être suivie d'une instruction LOCAL ou LOCALS.
- Vous ne pouvez utiliser l'instruction LOCALFREE que si la pile est à l'équilibre (pas de PUSH en suspens devant être corrigé par un POP).
- LOCALFREE n'est pas autorisé dans les modes x86 ou x64.
- Vous pouvez utiliser LOCALS à la place de LOCAL si vous préférez.
- Vous ne pouvez pas avoir une déclaration USEDATA l'intérieur d'une trame.
- Les labels de portée locale (dont le nom commence par un point) vont travailler n'importe où dans le cadre FRAME...ENDF. Leur limite de portée est la trame et l'instruction ENDF elles-mêmes, et, en tout cas, aucun autre label qui apparaît dans la trame.
- Fermez la trame à l'aide de ENDF (ou ENDFRAME si vous préférez), et éventuellement, faites précéder cette instruction par le nom de la trame concernée.
- Dans la mesure où GoAsm est un assembleur travaillant en une seule passe, les données locales doivent être déclarées dans le script source avant d'envisager leur utilisation.
- Puisque GoAsm s'appuie sur EBP/RBP dans une trame pour accéder aux paramètres et données locales sur la pile, ne pas modifier ces registres dans la trame ou les procédures appelées par la trame à moins que cet accès ne soit pas nécessaire dans la procédure. Si EBP/RBP est changé toujours restaurer sa valeur par la suite.
- Une trame peut en appeler une autre et lui passer ainsi les paramètres sur la pile, mais, dans la mesure où EBP/RBP seront modifiés dans ce processus, les données originales de la pile ne seront pas accessibles dans la trame appelée.

Quelques points de syntaxe lors de l'utilisation de USEDATA...ENDU

- La syntaxe de USEDATA...ENDU est comme suit, avec les éventuelles variations mentionnées ci-dessous :

```
CodeLabel:
    USEDATA SourceData
    USES Register List     ; si les registres ont besoin d'être sauvegardés
    LOCAL Local List       ; si des variables locales sont requises
    ;
    ret
    ENDU
```

- Une instruction USEDATA doit être précédée par un label, soit immédiatement avant, soit sur la ligne qui précède. Il s'agit du "nom USEDATA".
- *SourceData* peut être un nom de trame ou le nom d'une procédure USEDATA.
- Si *SourceData* est un nom de trame, les paramètres et les données locales établies dans la trame seront accessibles dans la zone USEDATA... ENDU.
- Si *SourceData* est le nom d'une procédure USEDATA, alors tous les paramètres de donnée locale et de mots définis qui étaient accessibles au sein de cette procédure peuvent être consultés.
- Vous pouvez automatiquement sauvegarder et restaurer des registres dans une zone USEDATA avec l'instruction USES. ENDU arrête l'action de USES.
- Les sauts ne peuvent être que dans la zone USEDATA elle-même. En effet, une zone USEDATA a son propre code d'épilogue unique qui doit être mis en œuvre.

- Les Calls peuvent s'adresser à l'extérieur de la zone de USEDATA. Si vous avez besoin d'accéder aux paramètres de données locales ou de mots définis de la zone USEDATA, constituez une autre zone USEDATA..ENDU.
- Si vous appelez une fonction dans la même zone USEDATA, cette fonction doit utiliser RETN (RET "normal") au lieu de RET. Cela empêche la génération du code d'épilogue au moment de quitter la fonction.
- Les données locales doivent être déclarées en utilisant une ou plusieurs instructions LOCAL. Après la première instruction de code suivante, vous ne serez pas en mesure d'utiliser LOCAL à nouveau à moins que vous n'ayez libéré les données locales existantes en utilisant l'instruction LOCALFREE.
- LOCALFREE doit être suivie d'une instruction LOCAL ou LOCALS.
- Vous ne pouvez utiliser l'instruction LOCALFREE que si la pile est à l'équilibre (pas de PUSH en suspens devant être corrigé par un POP).
- LOCALFREE n'est pas autorisé dans les modes x86 ou x64.
- Vous pouvez utiliser LOCALS la place de LOCAL si vous préférez.
- Une procédure USEDATA peut en appeler une autre sans perte de données puisque EBP/RBP n'est pas modifié.
- Les labels de portée locale (dont le nom commence par un point) fonctionnent normalement dans les zones USEDATA. Les labels de codes constitue leur frontière de portée.
- Au lieu d'utiliser ENDU pour fermer la zone USEDATA, vous pouvez utiliser ENDUSEDATA en lieu et place. En option, pour des questions de clarté du script, le nom du USEDATA peut précéder cette déclaration.
- Dans la mesure où GoAsm est un assembleur travaillant en une seule passe, les données locales doivent être déclarées dans le script source avant d'envisager leur utilisation.
- Puisque GoAsm s'appuie sur EBP/RBP dans une zone USEDATA pour accéder aux paramètres et aux données locales sur la pile, ne pas modifier ces registres dans la zone USEDATA ou les procédures appelées au sein de la zone USEDATA à moins que cet accès ne soit pas nécessaire dans la procédure. Si EBP/RBP est changé toujours restaurer sa valeur par la suite.

Ce que vous pouvez voir dans le débogueur

Pour établir et utiliser des trames de pile automatisées, GoAsm génère du code supplémentaire. Lorsque vous examinez votre code dans le débogueur ce code supplémentaire peut être source de confusion et gêner l'identification du code que vous recherchez. Une façon de voir ce que GoAsm a inséré est de regarder le fichier listing produit à l'issue de l'assemblage (option /l dans la ligne de commande).

Voici une brève description de quelques lignes de code supplémentaires que vous pourrez observer.

Dans les **trames 32 bits** (FRAME), GoAsm commence par mettre en pile EBP et les registres mentionnés par USES, puis met la valeur du pointeur de pile ESP dans EBP par `MOV EBP,ESP`. Sur un RET, cet ordre est inversé, de sorte que vous verrez `MOV ESP,EBP` suivi par un ou plusieurs POPs de registre. Un espace est constitué pour les données locales par un simple `SUB ESP,x` où *x* représente le chiffrage en octets de l'espace requis pour les données locales.

Dans des **trames 64 bits** (FRAME), GoAsm commence par mémoriser sur la pile les paramètres n°1 à n°4 à l'aide d'une instruction telle que `MOV [RSP+8h],RCX` comme **décrit précédemment**. Puis après avoir mis en pile RBP et registres prévus par USES, `MOV RBP,RSP` est utilisée pour allouer à RBP la capacité d'adresser la trame de pile. En sortant de FRAME, l'instruction `LEA RSP,[RBP]` est utilisée pour restaurer RSP prêt à effectuer un POP des registres et, finalement, un RET.

Le code tel que `MOV EAX,[EBP-34h]` ou `LEA, [EBP-56h]` ou `PUSH EBP, ADD D[ESP],-60h` (ou leurs équivalents utilisant des registres de 64 bits) sera généré **au moment de l'accès aux données locales**. Les valeurs d'offset seront positives lors de l'**accès aux paramètres**.

Dans les **zones USEDATA**, comme GoAsm ne connait pas, au moment de l'assemblage, le nombre d'utilisations de la pile qu'il y a eu avant l'appel à la procédure USEDATA, il constitue une sorte de bouclier de 100h octets (200h octets en assemblage 64 bits) pour garantir que pareille utilisation de la pile est protégée contre les sur-écritures. Pour cette raison, le nombre de décalages utilisés lors de l'accès aux données locales peut être plus grand que prévu.

Les utilisateurs expérimentés pourront procéder comme pour ajuster la taille de l'écran. Cela peut être fait en utilisant cette syntaxe, par exemple :

```
USEDATA WndProc SHIELDSIZE:20h     ; en assemblage 32 bits
USEDATA WndProc SHIELDSIZE:40h     ; en assemblage 64 bits
```

Ceci restreint la protection à seulement 8 valeurs poussées en pile (8 Dwords en assemblage 32 bits, 8 Qwords en assemblage 64 bits), ce qui serait approprié si vous étiez certain qu'au moment de l'exécution il n'y aurait jamais plus de sept PUSHes et un CALL avant la déclaration de données locale dans la procédure USEDATA. Rappelez-vous que vous devez compter tous les PUSHs avant le CALL, le CALL lui-même et tous les sous-appels, ainsi que tout PUSH causé par la déclaration des utilisations dans la procédure de USEDATA. Une fois que SHIELDSIZE est définie, il reste à cette valeur pendant le reste du script source jusqu'à leur modification.

Dans les zones USEDATA, la valeur du pointeur de pile n'est pas conservée dans le registre EBP/RBP car celui-ci héberge déjà le pointeur de la pile à l'entrée dans la trame. Au lieu de cela, GoAsm maintient la valeur du pointeur de pile à un endroit convenable sur la pile. Cela se fait lorsque les premières données locales de la zone USEDATA sont déclarées. De manière à le faire en toute sécurité, GoAsm ajoute plusieurs lignes de code aboutissant à `MOV EAX,[EAX-4h]` (ou `MOV RAX,[RAX-8h]` en assemblage 64 bits). GoAsm utilise EAX/RAX au cours de ce processus, mais restaure sa valeur après coup, de sorte que vous pouvez toujours l'utiliser pour transmettre des informations à la procédure. Sur un RET vous verrez la valeur du ESP/RSP en cours de restauration en utilisant `POP ESP` ou `POP RSP` selon le cas.

LOCALFREE provoquera aussi une restauration du pointeur de pile.

Les instructions USES provoqueront des PUSHs des registres concernés avant que ESP/RSP ne soit sauvegardé et des POPs de registres après sa restauration.

4.10 Assemblage Conditionnel

4.10.1 Qu'est-ce que l'assemblage conditionnel et pourquoi est-il utilisé ?

L'assemblage conditionnel vous permet de sélectionner, lors de la phase d'assemblage, la partie de votre script source que vous souhaitez voir assemblée. Cela peut être utile si, par exemple, vous voulez constituer différentes versions de votre programme à partir d'un même script source.

4.10.2 Les directives conditionnelles

Dans ce domaine, GoAsm utilise simplement la syntaxe du langage C qui est basée sur les directives #if, #ifdef, #else, #elif (ou #elseif) et #endif. La syntaxe de la structure de base d'une directive conditionnelle dans sa forme la plus simple est la suivante :

```
#if condition
  text A
#endif
```

Ici, si la condition est VRAIE, le texte A sera assemblé. A contrario, si la condition est FAUSSE, l'assembleur ignorera le texte A et poursuivra la compilation à partir de #endif.

Vous pouvez ajouter quelque chose à faire si la condition est FAUSSE de cette manière :

```
#if condition
  text A
#else
  text B
#endif
```

Ici, si la *condition* est VRAIE, texte A sera assemblé, mais pas texte B.
A contrario, si la *condition* est FAUSSE, texte A sera ignoré et seul texte B sera assemblé.
Le #endif indique la fin de la trame conditionnelle, de sorte que tout le texte au-delà sera assemblé normalement.
La déclaration #else doit toujours précéder #endif.

Vous pouvez ajouter une condition supplémentaire à la structure :

```
#if condition1
  text A
#elif condition2
```

```
  text B
#endif
```

Ici, si *condition1* est VRAIE, texte A sera assemblé, texte B sera ignoré et l'assemblage se poursuivra à partir du #endif. Toutefois, si *condition1* est FAUSSE, texte A ignoré jusqu'à #elif précédant le test de *condition2*. Si donc *condition2* est VRAIE, texte B sera assemblé. Notez, au passage, que "#elif" est identique à "#elseif".

L'ajout du #else à la trame conditionnelle ci-dessus produit :

```
#if condition1
  text A
#elif condition2
  text B
#else
  text C
#endif
```

Ici, si *condition1* est VRAIE, texte A sera assemblé tandis que texte B et texte C seront ignorés et l'assemblage se poursuivra à partir du #endif. Toutefois, si *condition1* est FAUSSE, texte A sera ignoré jusqu'à #elif testant *condition2*. Si donc *condition2* est VRAIE, texte B sera assemblé et texte C ignoré ; si, toutefois *condition2* est FAUSSE, texte B sera ignoré jusqu'au #else qui fera que texte C sera assemblé.

Vous pouvez avoir autant de #elifs (ou #elseif) que vous le souhaitez dans chaque trame conditionnelle, mais il ne peut y avoir qu'un #else par trame, et chaque #if doit avoir un #endif correspondant. Certains programmeurs imbriquent les trames conditionnelles, mais cela peut devenir très confus et ne peut constituer une bonne pratique de programmation. Si toutefois ce choix est retenu, il est recommandé que vous étiquetiez chaque #endif avec un commentaire de sorte que vous puissiez voir à quel #if il se réfère.

4.10.3 Types d'instructions #if

#ifdef *identifier*

où *identifier* est un mot qui peut être défini dans le script source ou dans un fichier d'inclusion. Cette déclaration renvoie VRAI si *identifier* est défini et FAUX dans le cas contraire. *identifier* doit être un mot et pas un nombre, ni une chaîne entre guillemets.

#ifndef *identifier*

comme ci-dessus, mais cette déclaration retourne FAUX si *identifier* est défini et VRAI s'il n'est pas défini.

#if *expression*

où *expression* peut être un nombre, la déclaration retournant alors FAUX pour 0, et VRAI pour les valeurs non nulles.

où *expression* peut être un *identifier* qui évalue un nombre.

où *expression* peut être l'opérateur défini utilisé comme suit :
defined *identifier*
defined(*identifier*)
qui renvoie VRAI (1) si l'identifiant est défini, et FAUX (0) s'il n'est pas défini, de manière similaire à #ifdef.

L'opérateur ! (point d'exclamation) peut être utilisé devant ces expressions simples pour inverser le résultat de la condition, de sorte que par exemple :
#if !0 retournerait VRAI, et
#if !defined *identifier* retournerait FAUX (0) si *identifier* était défini, et VRAI (1) dans le cas contraire, de manière similaire à #ifndef.

où *expression* peut être plus complexe et revêtir la forme :
identifier opérateur-relationnel value

identifier doit être un mot défini ailleurs dans le fichier, dans un fichier inclure ou dans la ligne de commande. Il ne peut pas être un nombre.

l'*opérateur relationnel* peut être l'un des éléments suivants :

> =	supérieur ou égal à
<=	inférieur ou égal à
==	égal à
=	égal à
!=	différent de
>	supérieur à
<	inférieur à

value peut être un nombre ou un mot qui est défini ailleurs dans le fichier, dans un fichier inclus ou dans la ligne de commande, qui évalue un nombre.

où *expression* peut être plus complexe, combinant plusieurs expressions avec l'opérateur AND conditionnel && ou l'opérateur conditionnel OR ||:

expression1 && expression2
expression1 || *expression2*

où *expression1* et *expression2* sont l'un des types d'expressions précédents.

L'instruction && renvoie VRAI si *expression1* et *expression2* sont VRAIES.
Si *expression1* est fausse, alors *expression2* est pas évaluée.

L'instruction || renvoie VRAI si *expression1* ou *expression2* est VRAIE.
Si *expression1* est VRAIE, alors *expression2* n'est pas évaluée.

Notez que, pour les expressions multiples avec plusieurs opérateurs conditionnels, l'évaluation de la déclaration est effectuée avec un traitement simple de gauche à droite. Normalement && a préséance sur ||. Donc si vous placez ces expressions conditionnelles avec && en premier, vous devez obtenir des résultats similaires.

4.10.4 Exemples d'assemblage conditionnel

```
#define HELLO
;
#ifdef HELLO
BSWAP EAX      ; inverse l'ordre des octets dans EAX si HELLO est défini
#endif
#if HELLO==3
OUTPUT DD 3h   ; si HELLO est défini comme 3, déclarer le label de donnée OUTPUT à 3
#elif WINVER>=400h
OUTPUT DD 4h   ; donnée alternative data si WINVER est égal ou plus grand que 400h
#else
OUTPUT DD 5h   ; donnée alternative si aucun des cas ci-dessus n'est avéré
#endif
```

Vous pouvez définir un mot dans la ligne de commande permettant de déclencher la prise en compte des bonnes partie de votre script source pour l'assemblage. Par exemple

```
GoASM /l /d WINVER=401h MyProg.ASM
```

ou simplement

```
GoASM /l /d VERSIONA MyProg.ASM
```

signifie que le mot VERSIONA sera défini et qu'il pourra être testé par #ifdef.

Voir également :

assemblage conditionnel dans les macros
assemblage conditionnel dans les structures

4.11 Inclusion de fichiers – #include et INCBIN

L'inclusion de fichiers contenant du code assembleur ou simplement des structures et des définitions peut être une alternative intéressante pour opacifier votre script source en le rendant difficile à lire et à suivre. En effet, il faut du temps au lecteur pour se plonger dans le fichier numéro 2 pour comprendre le dossier numéro 1 et réciproquement. Néanmoins les fichiers contenant des structures et des définitions Windows sont populaires et GoAsm offre un support complet pour inclure des fichiers.

GoAsm distingue deux types de fichier à inclure :

Type	Effet
Fichiers avec une extension en "a" ou "A", par exemple MyInclude.asm, ou simplement MyInclude.a	Avec ce type de fichier, au moment où l'inclusion est déclarée dans votre script source, l'assemblage est détourné dans le fichier d'inclusion. Et si vous produisez un fichier listing à l'aide de l'option /l, vous constaterez que le contenu du fichier inclus apparait dans le listing général du programme. Ne pas utiliser ce type de fichier si votre fichier à inclure ne contient que des définitions, structures et autres. Cela va ralentir GoAsm inutilement, car il va chercher des mnémoniques et instructions assembleur dans ledit fichier.
Fichiers sans extension en "a" ou "A", par exemple MyInclude.inc, ou simplement MyInclude	Avec ce type de fichier, aucun assemblage n'est effectué dans le fichier d'inclusion. Seules les définitions et les structures dans ce fichier sont examinées et enregistrées. Si vous produisez un fichier listing à l'aide de l'option /l, vous constaterez que le contenu du fichier inclus n'y apparaitra pas. Utilisez cette extension si votre fichier d'inclusion ne contient que des définitions, des structures et fonctionnalités similaires (communément appelés fichiers d'en-tête). GoAsm fera un enregistrement de tout cela dans le cas où ils seraient appelés plus tard dans le script source principal. Pour cette raison, un grand fichier d'inclusion va ralentir GoAsm. Normalement GoAsm ne permet pas d'ouvrir des fichiers à inclure qui ne seraient pas dotés d'une extension en "a" ou "A". Une telle restriction contribue à faciliter la vérification des erreurs. Mais il vous est possible de contourner cette fonctionnalité si vous voulez permettre, par exemple, aux mêmes fichiers d'entête d'être disponibles dans un environnement de compilation parallèle. Pour ce faire, il vous suffit de spécifier le commutateur /sh (fichiers "share header") dans la ligne de commande.

4.11.1 Syntaxe pour #include

#include *path\filename*

path\filename peut être, soit : une chaîne entre guillemets
une chaîne sans guillemets
une chaîne de caractères encadrée par les symboles '<' et '>'

GoAsm va chercher le fichier en utilisant le chemin d'accès spécifié. Si aucun chemin n'est spécifié, il va le chercher dans le répertoire courant. Si le fichier reste malgré tout introuvable, il va le chercher dans le répertoire fourni par la chaîne d'environnement INCLUDE. Vous pouvez paramétrer la chaîne d'environnement INCLUDE en utilisant la commande DOS SET dans la fenêtre MS-DOS (invite de commande), ou en appelant l'API *SetEnvironmentVariable*. Vous pouvez également utiliser le panneau de configuration (*paramètres système avancés, variables d'environnement*) si votre système d'exploitation le permet, le tout suivi d'un redémarrage.
Remarque : il peut y avoir une chaîne d'environnement différente pour chaque répertoire ou sous-répertoire. Assurez-vous que la chaîne d'environnement que vous souhaitez utiliser est dans le répertoire courant.

Vous pouvez imbriquer des #include *file* mais il est de bonne pratique d'éviter cela autant que possible.

4.11.2 Chargement d'un fichier avec INCBIN

INCBIN vous permet de charger des morceaux de matériau à partir d'un fichier directement dans une section de données ou de code sans autre traitement. Vous pouvez choisir le nombre d'octets à sauter dès le début du fichier et/ou la quantité à charger à partir du fichier. Voici des exemples de la façon d'utiliser INCBIN :

```
DATA SECTION
BULKDATA INCBIN MyFile.txt     ; charge l'ensemble de MyFile.txt dans la section de
                               ; données avec le label BULKDATA
   INCBIN MyFile.txt,100       ; évite les 100 premiers octets mais charge le reste du fichier
   INCBIN MyFile.txt,100,300   ; évite les 100 premiers octets mais charge 300 octets
```

Voir également la **section insertion de blocs de données**.

4.12 Fusion (merging) Utilisation de bibliothèques de code statiques (.lib)

4.12.1 Que sont les bibliothèques de code statiques ?

Les bibliothèques de code statiques sont des fichiers avec l'extension ".lib" contenant un ou plusieurs fichiers objets COFF. Ces fichiers objets contiennent du code et des données relatifs à des fonctions prêtes à l'emploi. Le matériau à l'intérieur du fichier de bibliothèque est au format binaire (code machine), et non du code source. Le fichiers de bibliothèque contiennent un index avec une liste des fonctions et les labels de code et de données qu'ils utilisent. Les bibliothèques de code statiques doivent être distingués des bibliothèques liées dynamiquement (DLL) et de bibliothèques d'importation qui contiennent simplement une liste des fonctions exportées par DLL.

4.12.2 Comment utiliser une bibliothèque de code statique

Dans GoAsm, vous pouvez utiliser du code et des données prêts à l'emploi dans des bibliothèques de code statiques simplement en appelant la fonction requise dans votre script source, en mentionnant le nom de la bibliothèque contenant cette fonction. Par exemple :

```
CALL zlibstat.lib:compress
```

Vous pouvez également utiliser des *equates* pour simplifier le libellé du CALL, par exemple

```
LIB1=c:\prog\libs\zlibstat.lib
CALL LIB1:compress
```

En utilisant INVOKE, ces exemples deviennent :

```
INVOKE zlibstat.lib:compress,[pCompHeap],ADDR ComprSize,[pHeap],[DataSize]
INVOKE LIB1:compress,[pCompHeap],ADDR ComprSize,[pHeap],[DataSize]
```

Si votre chemin d'accès contient des espaces, vous devez mettre le chemin d'accès et le nom de fichier entre guillemets.

4.12.3 Qu'advient-il lorsque vous appelez une fonction issue d'une bibliothèque ?

Le codage qui précède enjoint à GoAsm de charger le code et les données associés et de les fusionner avec le fichier de sortie de GoAsm (fichier objet) lors de l'assemblage. Vous pouvez ensuite envoyer le fichier de sortie au linker de la manière habituelle, mais l'éditeur de liens est pas concerné du tout par le code et les données source (le fichier de bibliothèque) : l'ensemble du code et des données associées à la fonction ont déjà été chargé par GoAsm. Le seul travail supplémentaire que vous pourriez avoir besoin de faire au moment de la phase d'édition de liens est de veiller à ce que le linker soit informé de toutes les DLL supplémentaires nécessitées par les fonctions prêtes-à-l'emploi. Par exemple, une fonction prête-à-l'emploi peut appeler une API Windows dans OLEAUT32.dll. Afin d'éviter une erreur de «symbole introuvable», il est indispensable de mentionner cette DLL dans la ligne de commande de GoLink. Si vous utilisez un autre éditeur de liens, vous devez ajouter la bibliothèque d'importation pour OLEAUT32.dll à la liste des bibliothèques d'importation donnée à l'éditeur de liens.

4.12.4 La méthode Microsoft dans tout ça ?

La méthode de GoAsm concernant l'utilisation de bibliothèques de code statiques diffère très sensiblement de celle utilisée par les outils Microsoft. Le MS Linker ajoute le code et les données *au moment de l'édition des liens*. Si vous préférez, vous pouvez toujours utiliser cette méthode avec GoAsm. Pour ce faire vous devez charger les fichiers de sortie de GoAsm dans MS Link et demander à ce dernier de rechercher les fonctions dans la bibliothèque de code statique appropriée.

Voir la section **Utilisation de GoAsm avec différents Linkers**.

4.12.5 Comment GoAsm trouve le fichier LIB approprié

Si le chemin d'accès du fichier .lib n'est pas donné dans l'appel, GoAsm conduira ses recherches dans le répertoire courant et dans celui donné dans l'environnement LIB. Vous pouvez paramétrer la chaîne d'environnement LIB à l'aide de la commande DOS SET dans la fenêtre MS-DOS (invite de commande), en appelant l'API *SetEnvironmentVariable* ou en utilisant le panneau de configuration (*paramètres système avancés, variables d'environnement*) si votre système d'exploitation le permet, le tout suivi d'un redémarrage.
Remarque : il peut y avoir une chaîne d'environnement différente pour chaque répertoire ou sous-répertoire. Assurez-vous que la chaîne d'environnement que vous souhaitez utiliser est dans le répertoire courant.

Vous avez besoin de spécifier une seule fois le chemin d'accès du fichier lib dans votre script source (dans un appel à un fichier lib) et GoAsm l'utilisera automatiquement pour tous les fichiers lib spécifiés du même nom.

4.12.6 Usage de JMP au lieu de CALL/INVOKE

Vous pouvez effectuer un saut vers une fonction de la manière suivante :

```
JMP MyLib.lib:MainProc
```

Cette méthode peut être employée si, par exemple, le code de la bibliothèque contient un call à l'API *ExitProcess* pour mettre fin au programme.

4.12.7 Visualisation du contenu d'un fichier LIB

Il est très utile de pouvoir identifier les fonctions disponibles dans les bibliothèques de code statiques. Il existe un certain nombre d'outils disponibles à cet effet, mais l'un des plus utiles est :

PEview de Wayne J. Radburn qui vous permet de visualiser le contenu de différents fichiers y compris les fichiers lib. Vous pouvez voir à partir de cet outil que les membres individuels des bibliothèques de code statiques sont toujours des fichiers ".obj", mais que les membres individuels de bibliothèques importées sont toujours des fichiers ".dll".

DUMPBIN est un utilitaire Microsoft qui est généralement associé à MASM et aux compilateurs C. La ligne de commande **LINK -dump** (utilisant donc l'éditeur de liens Microsoft) est fonctionnellement identique et fournit une liste d'options si elle est utilisée sans autres paramètres également. Il existe diverses options, mais par exemple,

```
DUMPBIN /LINKER MEMBER: 1 MYLIB.LIB> MyLib.dmp
```

donnera (dans le fichier MyLib.dmp) des informations sur le premier membre du fichier de bibliothèque et `DUMPBIN /ALL MyLib.lib` donnera des informations sur tous les membres.

PEDUMP, enfin, est un utilitaire écrit par Matt Pietrek qui peut visualiser, octet par octet, le contenu d'un fichier PE (y compris un fichier lib) d'une manière ordonnée (voir également **LIBDUMP** du même auteur).

4.12.8 Vos propres fichiers LIB

L'utilitaire Microsoft LIB.EXE, qui repose sur LINK.EXE et aussi MsPDB50.dll, permet d'élaborer des fichiers de bibliothèque de code statique. Supposons que vous ayez un fichier objet appelé *calculate.obj* qui contient une fonction que vous souhaitez réutiliser. Il vous est possible d'en faire une bibliothèque au moyen de la syntaxe de ligne de commande suivante, par exemple :

```
LIB calculate.obj
```

Vous obtiendrez ainsi *calculate.lib*. Et, pour ajouter un autre fichier objet à cette même bibliothèque, vous pouvez procéder comme suit :

```
LIB calculate.lib added.obj
```

Cela va ajouter *added.obj* à la bibliothèque *calculate.lib*. Ceci est utile si vous voulez garder vos fonctions dans les bibliothèques, afin qu'elles puissent être réutilisées sans avoir à écrire à nouveau le code dans vos scripts source. Ces bibliothèques seront également utiles pour distribuer vos fonctions tout en conservant votre code source à votre seule discrétion. LIB.EXE et ses composants font partie des outils MSDN qui peuvent être téléchargés gratuitement à partir du site Microsoft MSDN (partie du SDK). Le téléchargement exact ne cesse de changer de sorte que des tâtonnements seront peut-être nécessaires pour obtenir ces fichiers. Il est fort probable également que LIB soit rangé du côté d'outils de compilation tels que VC++ ou MASM.

4.12.9 Augmentation de taille de la bibliothèque de code statique

L'appel d'une fonction dans une bibliothèque de code statique agrandit le fichier de sortie GoAsm au prorata du code et des données associés de la fonction. Souvent, du reste, la fonction dépend elle-même d'autres fonctions au sein du même fichier de la bibliothèque, ce qui entraîne encore plus de code et de données à charger. Vous pouvez visualiser ce qui est chargé en utilisant le commutateur /l (édition d'un fichier listing) dans la ligne de commande de GoAsm et en examinant le fichier listing résultant. Malheureusement, certains code et données peuvent être chargés tout en n'étant pas effectivement utilisés. Vous pouvez néanmoins visualiser les labels de code et de données inutilisés à l'aide du commutateur /unused de GoLink.

4.12.10 Callbacks et dépendance à l'égard des données

Certaines fonctions de fichiers de bibliothèque s'attendent à trouver des labels spécifiques de code et de données à l'exécutable qui doivent donc impérativement faire partie de votre script source. Par exemple la procédure de callback *RegisterDialogClasses* et ses variables de données associées doivent être établies dans votre script source afin d'utiliser SCRNSAVE.LIB pour faire un économiseur d'écran.

4.12.11 Cas des données sans code

Les bibliothèques sont destinées à donner accès à des *fonctions* au moment de la compilation plutôt que simplement des *données*. Toutefois, si vous voulez charger une bibliothèque particulière au moment de la compilation de sorte que vous puissiez accéder à ses données, vous pouvez appeler un label de code approprié dans un code *inutilisé* figurant dans votre script source (le label de code n'est jamais appelé). Par exemple :

```
CALCULATE1:
                    ; lignes de code ici
RET
CALL Lib1:DUMMY     ; garantit que Lib1 est chargé au moment de la compilation
CALCULATE2:
                    ; lignes de code ici
RET
```

4.12.12 Intégration du code et des données, priorités attribuées aux labels de même nom

Si vous constituez vos propres fichiers lib, il vous sera utile de comprendre comment le code et les données de bibliothèque sont intégrés au code et aux données résultant du script source principal. Lors de l'utilisation des bibliothèques de code, il est courant de rencontrer des noms de labels identiques, et si cela se produit lors du chargement d'une bibliothèque, GoAsm utilise seulement l'information associée au tout premier label et ignore celle relative aux autres labels de même nom.

Supposons, par exemple, que vous ayez une zone de données appelé « BUFFER » déclarée soit dans le script source principal, soit dans la bibliothèque. Il peut y avoir plusieurs fonctions dans le script source principal ou dans les différents composants de bibliothèque ou même dans d'autres bibliothèques qui pourraient utiliser BUFFER.

Alors, où BUFFER devrait-il être déclaré et ce que se passerait-il s'il l'était plus d'une fois ? Une question similaire se pose avec les fonctions. Le code pour celles-ci peut être dans le script source principal ou dans une bibliothèque. Il peut être dupliqué en plusieurs endroits. La réponse à ces questions se trouve dans les règles de priorité.

Ces règles sont les suivantes :

1. Le script source principal GoAsm et tous fichiers inclus avec extension en "a" ont toujours la priorité. En d'autres termes, tout label de code ou déclaration des données dans les scripts trouveront toujours leur chemin dans le fichier de sortie GoAsm avec le code et les données auxquels ils apposent des labels.

2. Sous réserve du 1 ci-dessus, les appels formels de la bibliothèque (utilisant le format *library:functionname*) ont priorité dans l'ordre dans lequel ils sont appelés.

Ces règles signifient que les bibliothèques de code sont en mesure d'appeler des fonctions et d'utiliser les données dans les scripts source directement (sans aucune aide de l'éditeur de liens). Elles signifient également que tous les labels dans une bibliothèque qui a déjà été utilisée dans le script source ou dans une bibliothèque qui a déjà été appelée seront ignorés. Supposons par exemple que BUFFER soit déclaré dans le script source avec une taille de 256 octets. Si library1 le déclare à son tour à 128 octets, ce label est ignoré et BUFFER sera de 256 octets dans le fichier de sortie. Et de plus, bien que la zone de données réservée à BUFFER dans library1 soit chargée dans le fichier de sortie, le label BUFFER ne la pointera pas et s'en tiendra à la zone déclarée dans le script source. Maintenant, si un appel plus tard, Library2 déclare BUFFER à 1024 octets, encore une fois cette zone de données ira dans le fichier de sortie, mais BUFFER continuera à pointer la zone de données d'origine. La raison pour laquelle GoAsm traite ainsi les labels de même nom est double. Tout d'abord, il serait impossible pour un assembleur (ou un linker d'ailleurs) de déterminer quel label est prioritaire au vu de sa *taille*. En effet, dans GoAsm au moment de l'assemblage et, plus certainement, au moment de l'édition de liens, la taille d'une zone particulière pointée par un label n'est pas connue avec certitude. Ceci, parce que lesdites zones sont parfois agrandies par des zones de code et de données dépourvues de label, ou parfois d'autres labels sont utilisés comme pointeurs vers une position intermédiaire dans la zone.

Les règles ont aussi une signification vis-à-vis de l'*ordre dans lequel sont disposées les données*. Supposons que votre bibliothèque repose sur des données détenues dans un tampon et également sur des données débordant parfois dans une zone élargie nommée BUFFER_EXT et déclarée immédiatement après BUFFER dans la bibliothèque. Maintenant, dans l'exemple ci-dessus BUFFER ne pointerait pas en réalité vers l'endroit attendu (juste avant BUFFER_EXT). Au contraire, il pointerait vers la première déclaration de BUFFER ailleurs dans la section de données.

Au terme de cette réflexion, je suggère donc que les règles suivantes soient respectées lors de la création de fichiers lib :

1. Utilisez uniquement des labels de même nom dans le script source et dans la bibliothèque si ces labels sont appelés à pointer vers la même chose au même endroit, à savoir, à la première déclarée en tant que label.

2. Si des labels de données de même nom doivent être utilisés dans différentes fonctions dans le fichier lib, il faut savoir que la taille de la zone de données qu'ils identifient sera fixée par la première déclarée en tant que label. Aussi, ne vous attendez pas à ce que la zone de données que le label identifie soit placée dans une position particulière dans l'exécutable.

4.12.13 Utilisation de fichiers-objet uniques

Actuellement GoAsm ne supporte pas d'appeler les mêmes fonctions de la bibliothèque à partir de plus d'un module (script source). Si tel est le cas, vous verrez apparaître des erreurs "Duplication de symbole" en provenance de l'éditeur de liens. Pour contourner ce problème, vous devez concentrer tous les appels à la bibliothèque sur la même fonction dans une bibliothèque en un seul script source. Dans des versions ultérieures de GoAsm et de GoLink un support pourrait être ajouté pour les appels vers la même fonction de bibliothèque de plus d'un script source en cas d'utilité avérée pour les utilisateurs.

4.13 Unicode

GoAsm et son programme complémentaire GoRC (compilateur de ressources) lisent tous les deux des fichiers Unicode UTF-16 et UTF-8, peuvent prendre leurs commandes en Unicode et produire leur sortie en Unicode. Cela signifie que si vous utilisez un éditeur Unicode, il est possible d'utiliser des noms de fichiers, des commentaires, des labels de code et de données, des mots définis (equates, macros et structs), des exportations et aussi des chaînes de données, tous en Unicode. GoAsm dispose d'un certain nombre de fonctionnalités pour vous aider à écrire des programmes en Unicode ou même créer une version Unicode et ANSI de votre programme source à partir d'un seul script. Voir le chapitre “écriture de programmes Unicode” du volume 2 pour plus d'informations sur tous ces sujets.

4.14 Assemblage 64 bits

L'utilisation du commutateur /x64 dans la ligne de commande de GoAsm bascule l'assemblage en mode 64 bits, et permet de produire un fichier objet COFF 64 bits au format PE+. GoRC (le compilateur de ressources) et GoLink (l'éditeur de liens) peuvent également travailler en 64 bits et produire des exécutables destinés à fonctionner sous Windows 64 bits pour les processeurs AMD64 et EM64T. Bien que le code exécutable 32 bits diffère sensiblement de son homologue 64 bits et dans la mesure où les principes de base utilisés dans l'écriture du code source restent les mêmes, il est possible d'utiliser le même code source pour les deux plates-formes. Ce qui fait que le code existant source 32 bits peut être porté à 64 bits. AdaptAsm.exe peut même aider à effectuer cette conversion, le cas échéant.

Voir
Appel des API Windows en 32 bits et 64 bits
Trames de pile pour Callback en 32 et 64 bits
Programmation en 64 bits.

4.15 Mode compatible x86 (assemblage 32 bits utilisant un source 64 bits)

L'utilisation du commutateur /x86 dans la ligne de commande de GoAsm bascule l'assemblage en mode de compatibilité x86 et vous permet de traiter un code source utilisant les registres à usage général étendus RAX, RBX, RCX, RDX, RDI, RSI, RBP et RER. Dans ce mode, ces registres sont lus par GoAsm comme s'ils étaient EAX, EBX, ECX, EDX, EDI, ESI, EBP et ESP. Cela vous permet d'utiliser des instructions comme :

```
MOV RSI,ADDR String
MOV [RDI],AL
MOV RAX,[hInst]
```

Ces instructions et d'autres semblables vont travailler à la fois dans les modes de compatibilité x64 et x86.

En plus de cela, dans le mode de compatibilité x86,

- "x86" devient un mot défini et identifiable pour l'assemblage conditionnel à l'aide de #if (non sensible à la casse)
- ARG RAX devient PUSH EAX
- ARG ADDR WNDCLASS devient PUSH ADDR WNDCLASS
- ARG 800h devient PUSH 800h
- ARG [hInst] devient PUSH [hInst]
- INVOKE fonctionne comme STDCALL
- JCXZ instruction fonctionne comme JECXZ

Notez que le commutateur /x86 ne doit pas être utilisé dans la ligne de commande pour assembler du code source Win32 (ne l'utiliser que pour du code source commutable 32/64 bits).

Même en mode de compatibilité x86, vous ne pouvez pas utiliser les nouveaux registres AMD64/EM64RT, R8 à R15, XMM 8 à XMM 15, ni les formats d'adressage des nouveaux registres SIL, DIL, BPL, SPL, R8W à R15W ou R8D à R15D. En effet, ils ne sont pas disponibles pour une utilisation par un exécutable 32 bits.

Tout code source incompatible avec un exécutable 32 bits doit être éliminé au moment de la phase d'assemblage au moyen des techniques d'**assemblage conditionnel**.

Voir **Appel des API Windows en 32 bits et 64 bits** dans le manuel GoAsm pour plus d'informations à propos de ARG et INVOKE.

Voir le fichier **Hello64World3** comme exemple de code source qui peut produire, soit une fenêtre "Hello World" à partir d'un programme simple Win32, soit son équivalent en Win64.

Voir aussi le chapitre **Programmation en 64 bits** pour s'informer des différences entre les programmes 32 et 64 bits.

4.16 Sections – Gestion avancée

4.16.1 Attribuer un nom aux sections

Le fichier objet attend un *nom* pour chaque section et GoAsm le fournit par défaut. Exprimé autrement, il n'est pas nécessaire du tout de nommer les sections. Les noms par défaut utilisés par GoAsm sont "code" pour une section de code, "data" pour une section de données et "const" pour une section const (ou constante).

Vous pouvez, si vous le souhaitez, baptiser une section en faisant suivre la déclaration de la section par un nom choisi par vous, comme, par exemple :

```
DATA SECTION MySect
```

ou

```
DATA SECTION "Hello are you well?"
```

Chaque section dotée d'un nom sera unique. Par conséquent, vous pouvez créer autant de sections que vous le désirez en attribuant à chacune un nom différent. En temps normal, cependant, vous aurez besoin d'une seule section de données, d'une section de code et d'une section de const. Dans les programmes plus importants vous pouvez souhaiter avoir plus d'une section. Il est possible que cela puisse rendre le débogage un peu plus facile. Bien que GoAsm vous permette d'utiliser un nom de section de plus de huit caractères, l'éditeur de liens le limitera à huit caractères lorsqu'il constituera l'exécutable.

4.16.2 Ajout de l'attribut “shared”

L'attribut shared (flag 10000000h) lorsqu'il est utilisé dans la section de données d'une Dll invite le chargeur de Windows à fournir une seule copie des données contenues dans cette section à chaque exécutable utilisant la DLL. Sans ce flag, chaque exécutable obtiendrait une copie distincte des données. Cela pourrait être le moyen pour un exécutable d'envoyer des données à un autre, sans avoir à utiliser le mappage de fichiers. Pour définir cet attribut ajouter le mot “shared” juste après la déclaration du nom de section, par exemple :

```
DATA SECTION "MyData" SHARED
```

Noter qu'une section partagée (shared) doit avoir un nom unique attribué de la manière indiquée ci-dessus. Vous ne voudrez probablement pas utiliser l'un des noms de section par défaut, à savoir "data", "code" ou "const", puisqu'ils sont normalement réservés à des sections non partagées.

GoAsm traite toute donnée non-initialisée comme initialisée à zéro dans une section partagée. Cela évite les problèmes qui pourraient se poser si une donnée non-initialisée *non partagée* est également requise dans un module (il ne peut y avoir qu'une section de données non-initialisée dans un fichier objet) ; ou encore des problèmes qui se poseraient au moment de l'édition de liens en l'absence de section appropriée à laquelle attacher ces données non initialisées partagées.

4.16.3 Ordre des sections

GoAsm insère les sections dans le fichier objet dans l'ordre où elles apparaissent dans le script source. Il est normalement de la responsabilité de l'éditeur de liens d'ordonner les sections dans l'exécutable final. Si l'éditeur de liens est paramétré pour suivre le même ordre que celui des fichiers objets, vous pouvez modifier l'ordre des sections dans l'exécutable final en changeant l'ordre de leur déclaration dans le script source. *Dans les sections elles-mêmes*, vous pouvez demander à l'éditeur de liens d'ordonner les éléments de données brutes individuelles d'une certaine manière en ajoutant le suffixe $ au nom de la section. Par exemple :

```
CODE SECTION 'Asm$b'
;
CODE SECTION 'Asm$a'
;
```

Ici, le linker veillera à ce que le code de la section intitulée Asm$a apparaisse dans l'exécutable avant celui de la section intitulée Asm$b. En fait, l'éditeur de liens regroupera le code dans une même section (appelée Asm) et dans l'ordre attendu. Le caractère situé immédiatement après le signe dollar est utilisé uniquement pour préciser l'ordre souhaité et, lorsque l'on compare les noms de section, l'éditeur de liens ne considèrera seulement que les caractères précédant le symbole dollar. Enfin, puisque, dans l'exécutable, les sections ne peuvent pas avoir des noms de plus de huit caractères vous devez limiter, dans la pratique, le nombre de caractères devant le signe dollar à huit.

4.16.4 Alignement d'une section

Les sections ont un alignement par défaut de 16 octets. En d'autres termes, elles commencent sur une adresse multiple de 16. Cette valeur peut être modifiée pour un fichier objet pour contrôler comment l'éditeur de liens aligne le contenu de telle section avec le contenu de telle autre section en provenance d'autres fichiers objets. Pour spécifier un alignement de section différente, ajouter la mention ALIGN *value* juste après la déclaration de section, où *value* est la taille de l'alignement requis en octets et prendre les valeurs 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 , 1024, 2048, 4096, ou 8192 (ce qui correspond à la forme 2^n). Cela peut être utile pour les projets avec plusieurs fichiers sources qui ont besoin de construire différents tableaux d'informations, par exemple les tables de SEH dans des projets 64 bits :

```
CONST SECTION '.xdata' ALIGN 8
;
;UNWIND_INFO
;
CONST SECTION '.pdata' ALIGN 4
;
;RUNTIME_FUNCTION
;
```

4.17 Adaptation de fichiers-source existants à GoAsm

A l'origine, **AdaptAsm.exe** a été écrit dans le but d'alléger les tâches de modification de syntaxe rendues nécessaires pour adapter des fichiers source existants issus d'autres assembleurs à la syntaxe 32 bits de GoAsm. AdaptAsm.exe peut maintenant aider à l'adaptation de la syntaxe 32 bits GoAsm (ou peut-être un autre assembleur) en syntaxe 64 bits. Pour plus de détails sur ce sujet, voir la section **utilisation de AdaptAsm pour aider à convertir en format 64 bits**.

AdaptAsm s'utilise à partir de la simple ligne de commande suivante :

```
AdaptAsm [command line switches] inputfile[.ext]
```

Si aucune extension du nom du fichier d'entrée n'est spécifiée, l'extension .asm est présumée.
Si aucune extension du nom du fichier de sortie n'est spécifié, l'extension .adt est présumée.
Les commutateurs de ligne de commande sont :

```
/h = affiche l'aide
/a = adapte un fichier A386
/m = adapte un fichier MASM
/n = adapte un fichier NASM
/fo = spécifie le chemin d'accès du fichier de sortie. Ex : AdaptAsm /fo GoAsm\adapted.asm
/l = crée un fichier de sortie avec l'extension .log
/o = supprimer l'alerte précédant toute écriture sur le fichier d'entrée
/x64 = adapte le fichier à la plateforme 64 bits
```

Vous pouvez assimiler un fichier TASM à un fichier MASM s'il est écrit dans le mode masm. Je n'ai pas inclus de version concernant le *mode idéal* TASM.

AdaptAsm crée le fichier de sortie en utilisant le même nom et le même chemin d'accès que ceux du fichier d'entrée, mais avec l'extension.adt (sauf si un nom et un chemin différent sont spécifiés). Le fichier de sortie rend compte, en son début, du nombre de modifications rendues nécessaires et exécutées par la fonction d'adaptation.

AdaptAsm ne peut pas écrire dans le fichier source, sauf si le fichier de sortie porte le même nom, extension comprise. Même dans ce cas, il vous sera demandé de confirmer votre souhait d'écraser le fichier, sauf bien évidemment si le commutateur /o a été mentionné dans la ligne de commande. Je vous suggère de n'opter pour le remplacement du fichier d'origine que si vous en détenez une copie quelque part. Je n'ai pas écrit d'antidote !

Si le commutateur /l est spécifié, **AdaptAsm** produit un fichier de sortie de même nom et de même répertoire que le fichier de sortie mais avec l'extension .log. Cela montre les changements qui ont été faits. Les numéros de ligne fournis font référence aux numéros de lignes du fichier d'entrée.

4.17.1 Ce que fait AdaptAsm lorsqu'il convertit différents fichiers-source

Voir tableau des pages 91 et 92

4.17.2 Ce que AdaptAsm ne fait pas...

Malgré le traitement de votre script de source par AdaptAsm vous devrez :

- Vérifier la déclaration des sections. La syntaxe primaire de GoAsm est DATA SECTION, CODE SECTION, CONST SECTION ou CONSTANT SECTION, mais CODE (ou .CODE), DATA (ou .DATA), et CONST (ou .CONST) sont également acceptés.
- Ajouter des indicateurs de type aux instructions où ce type ne va pas de soi pour GoAsm.
- Convertir les macros à la syntaxe GoAsm (si vous avez encore besoin d'elles avec GoAsm).
- Si vous utilisez MASM, vos initialisations de structure peuvent utiliser, soit les délimiteurs <...>, soit les délimiteurs {...}. Pour GoAsm vous devez vous assurer que seuls les délimiteurs <...> sont utilisés à cet effet (les délimiteurs{....} étant réservés à l'initialisation des membres individuels comme dans TASM – voir **plus**).
- Si vous utilisez NASM, convertir vos modèles de structure à la syntaxe GoAsm.
- Si vous utilisez NASM, vérifier tous les sauts de labels locaux définis à l'intérieur de labels non-locaux (par ex. JZ MyProc.34) et renommer si nécessaire.
- Vérifier tous les labels de la section de code qui ont deux points (AdaptAsm ne le fait pas, de peur de l'ajout inopiné de 2 points à un non-label).
- Vérifiez que AdaptAsm n'a pas ajouté des crochets pour encadrer des equates, des noms de macro et de STRUC. AdaptAsm tente néanmoins d'obtenir une liste complète de ceux-ci et d'éviter de leur donner ainsi des crochets dans la mesure du possible.
- Vérifiez si tous les CALLs ou JMPs qui comporteraient des références de mémoire entre crochets (ce qui serait inhabituel). AdaptAsm n'ajouter pas de crochets pour ces instructions.
- Convertir toutes les instructions Masm REPEAT, .REPEAT, REPT, .UNTIL, .UNTILCXZ WHILE, .WHILE, .ENDW, .IF/.ELSE/.ELSEIF/.ENDIF par les instructions traditionnelles de l'assembleur CMP, TEST, LOOP et des instructions de saut.
- Vérifiez que toutes les lignes utilisant des remplacements de segment (segment overrides) sont correctement codées.
- Remplacer toute utilisation de SIZESTR par le procédé de codage utilisant l'opérateur $ (position du pointeur d'instructions courant).
- Si vous avez utilisé TYPEDEF, sa suppression et son remplacement par un script approprié sont indispensables.
- Si vous utilisez modèles de structures MASM, veiller à ce que les symboles *supérieur à* et *inférieur à* sont utilisés à la place des accolades pour une initialisation séquentielle.
- Si vous utilisez modèles de structures MASM, veiller à ce que les symboles *supérieur à* et *inférieur à* sont utilisés à la place des accolades pour une initialisation séquentielle.
- Comme il n'y a pas de priorité des opérateurs dans GoAsm vérifier toutes les opérations arithmétiques s'appuyant sur une priorité particulière. Par exemple, `MOV EAX,8+8*2` équivaut à 32 avec GoAsm.
- Lors de la conversion PROC ... ENDP en trames de pile automatisées FRAME...ENDF, GoAsm ignore l'attribut STDCALL, qui est la norme dans Windows 32 bits. Il devrait de toute façon être retiré pour Windows 64 bits qui utilise FASTCALL (GoAsm commute automatiquement entre les deux). Si vous avez utilisé d'autres attributs après PROC vous pouvez probablement les supprimer entièrement – GoAsm les laisse seuls – en les vérifiant à la main.

Bien que AdaptAsm explore les fichiers “include” afin d'obtenir des renseignements sur les equates, les macros et des modèles de structure, il n'adapte pas ces fichiers. S'il y a du code ou des données dans ces fichiers, vous devez les soumettre individuellement à AdaptAsm de la même manière que pour le script principal.

Pour l'effet du commutateur /x64 voir **Utilisation de AdaptAsm pour faciliter la conversion en programmes 64 bits**			
Action	fichiers **A386** utilisant /a	fichiers **Masm** utilisant /m	fichiers **Nasm** utilisant /n
Sauf si le mot est défini (par ex. au moyen d'un equate), les crochets sont ajoutés à toutes les références de mémoire où ils ne sont pas déjà présents, par exemple : `MOV EBX,MEM_REFERENCE` devient `MOV EBX,[MEM_REFERENCE]` mais `MOV EBX,OFFSET MEM_REFERENCE` est laissé seul	oui	oui	non
Met les références mémoire agrémentées de crochets dans la forme correcte, par exemple : `MOV DX,sHEXw[ECX*2]` devient `MOV DX,[sHEXw+ECX*2]`	oui	oui	non
Ajoute ADDR à des références mémoire NASM qui ne disposent pas des crochets.	non	non	oui
Les indicateurs de type et les remplacements BYTE, BYTE PTR, WORD, WORD PTR, DWORD, DWORD PTR, QWORD, QWORD PTR, et TWORD, TWORD PTR sont remplacés respectivement par les raccourcis équivalents B, W, D, Q et T.	oui	oui	oui

Renverse l'ordre de tous les messages immédiats entre guillemets afin qu'ils soient lus dans le bon sens, par exemple :.
`MOV [ESI],'exe.'` devient `MOV [ESI],'.exe'`
`MOV EAX,'morf'` devient `MOV EAX,'from'`
`DW 'GJ'` devient `DW 'JG'`
`DD 'dcba'` devient `DD 'abcd'`
	oui	oui	non

Change HELLO LABEL en HELLO:	oui	oui	non
Le label local @@: de MASM et les sauts correspondants @F et @B sont convertis au format GoAsm. Ils reçoivent des numéros de séquence dans le fichier et, le cas échéant, l'indicateur de sens '>' est ajouté dans les instructions de saut.	non	oui	non
Les labels locaux de NASM précédées d'un point – par exemple ".23" – sont convertis au format GoAsm. Le nombre reste inchangé, mais le cas échéant, l'indicateur de sens '>' est ajouté dans l'instruction de saut.	non	non	oui
Dans les instructions de saut, les indicateurs de proximité NEAR et SHORT sont enlevés car plus du tout utilisés.	oui	oui	oui
Remplace les registres FPU désignés seulement par un chiffre (0 à 7) par la syntaxe ST0 à ST7. Par exemple : FDIV 0,1 devient FDIV ST0,ST1	oui	non	non
Change les registres FPU exprimés sous la forme ST(0) à ST(7) en ST0 to ST7, comme par exemple : FDIV ST(0),ST(1) devient FDIV ST0,ST1	non	oui	non
Les déclarations de données effectuées par BYTE, ACHAR, SBYTE, sont changées en DB. Les déclarations de données effectuées par WORD, SWORD, SHORTINT sont modifiées en DW.	non	oui	non

Suite et fin du tableau page suivante

Suite et fin du tableau

Action	fichiers **A386** utilisant /a	fichiers **Masm** utilisant /m	fichiers **Nasm** utilisant /n
Les déclarations de données effectuées par BYTE, ACHAR, SBYTE, sont changées en DB. Les déclarations de données effectuées par WORD, SWORD, SHORTINT sont modifiées en DW. Les déclarations de données effectuées par DWORD, HDC, ATOM, BOOL, HDWP, HPEN, HRGN, HSTR, HWND, LONG, LPFN, UINT, HFILE, HFONT, HICON, HHOOK, HMENU, HRSRC, HTASK, LPINT, LPSTR, LPVOID, WCHAR, HACCEL, HANDLE, HBRUSH, HLOCAL, LPARAM, LPBOOL, LPCSTR, LPLONG, LPTSTR, LPVOID, LPWORD, SDWORD, WPARAM, HBITMAP, HCURSOR, HGDIOBJ, HGLOBAL, INTEGER, LONGINT, LPBYTE, LPCTSTR, LPCVOID, LPDWORD, LRESULT, POINTER, WNDPROC, COLORREF, HPALETTE, HINSTANCE, HINTERNET, HMETAFILE, HTREEITEM, HCOLORSPACE, LOCALHANDLE, GLOBALHANDLE, HENHMETAFILE sont toutes changées en DD. Les déclarations de données effectuées par QWORD et DWORDLONG sont changées en DQ. Les déclarations de données effectuées par TWORD sont changées en DT.	non	oui	non
La duplication de syntaxe de données TIMES utilisée dans NASM est changée par la méthode DUP de déclaration les données multiple. De même, RESB / RESW / RESD utilisés dans NASM pour réserver les données non-initialisées sont remplacés par la méthode DUP? de déclaration des données non-initialisées.	non	non	oui
TEXTEQU est changé dans sa version de type "C" #define. Les equates EQU et = ne sont pas changées car supportées parGoAsm.	non	oui	non
La directive INCLUDE est remplacée par #INCLUDE	oui	oui	oui
La directive %INCLUDE est remplacée par #INCLUDE	non	non	oui
La série de directives IF/ELSE/ELSEIF/ENDIF/IFDEF (assemblage conditionnel) est remplacée par #IF/#ELSE/#ELSEIF/#ENDIF/#IFDEF. Les directives .IF/.ELSE/.ELSEIF/.ENDIF/.IFDEF et .WHILE et .BREAK ne sont pas traitées. Elles doivent être éliminées et le process réécrit manuellementen conséquence.	non	oui	non
La série de directives %IF/%ELSE/%ELSEIF/%ENDIF/%IFDEF (assemblage conditionnel) est remplacée par #IF/#ELSE/#ELSEIF/#ENDIF/ #IFDEF La directive %DEFINE est remplacée par #DEFINE	non	non	oui
Commentaires sur toutes les lignes commençant par EXTERNE ou EXTERN, GLOBAL, ou PUBLIC.	oui	oui	oui
PROC est remplacé par FRAME et ENDP par ENDF. Dans le code de MASM, les paramètres et la déclaration USES sont équivalents à la syntaxe GoAsm.	oui	oui	non
La taille des données LOCAL dans une trame de pile automatisée est remplacée par des versions plus abrégées dans GoAsm (B, W, Q, T) et D est supprimé complètement car correspondant à la valeur par défaut dans GoAsm.	non	oui	non
Change EVEN en ALIGN	oui	oui	oui
Plusieurs lignes qui ne supportent pas de commentaire, par exemple . NAME, TITLE, SUBTITLE, SUBTTL, PROTO lines etc.	oui	oui	oui
Plusieurs lignes que GoAsm ne prend pas en charge sont tout simplement supprimées. Par exemple .ERR, .EXIT, .LIST, .286 etc.	oui	oui	oui
Le mot COMMENT est remplacé par un point-virgule	oui	oui	oui

Chapitre 5
Divers

5.1 Instructions PUSH spéciales

5.1.1 Demi-opérations de pile

Ne s'applique qu'à la programmation 32 bits.

Dans Win32, les données sur la pile sont gérées en Dwords, et la valeur du pointeur ESP est toujours sur une limite DWORD après une opération PUSH ou POP. GoAsm prend cependant en charge des demi-opérations de pile, qui consistent à effectuer des PUSH et des POP sur deux octets seulement au lieu de quatre. Lorsque vous utilisez ces instructions, vous devez effectuer un PUSH ou un POP une seconde fois pour maintenir ESP à un multiple de DWORD. Pour rendre la syntaxe explicite, GoAsm requiert l'utilisation de PUSHW et POPW pour ces opérations de demi-pile. PUSH et POP ne peuvent pas être utilisés car ils effectuent systématiquement une opération de pile DWORD. A titre d'exemple, les instructions de demi-pile peuvent être utilisées en réponse au message de WM_LBUTTONDOWN :

```
MOUSEX_POS DD 0
MOUSEY_POS DD 0
PUSH [EBP+14h]          ; met lParam sur la pile
POPW [MOUSEX_POS]       ; extrait de la pile le mot le moins significatif
POPW [MOUSEY_POS]       ; puis le mot le moins significatif
```

ou lorsque vous utilisez certaines API qui reçoivent certaines données de la pile respectivement dans le mot de poids fort et celui de poids faible :

```
PUSH ADDR lpFileTime
PUSHW [wFatTime]
PUSHW [wFatDate]
CALL DosDateTimeToFileTime
```

GoAsm supporte également les instructions PUSHAW, PUSHFW et POPFW, POPAW, bien que vous ne devriez normalement pas y recourir dans la mesure où GoAsm est utilisé uniquement en programmation 32 bits.

5.1.2 Sauvegarde en pile des flags et restauration

Au lieu d'utiliser PUSHF et POPF pour, respectivement, sauvegarder en pile et restituer les flags (mot d'état), vous pouvez recourir, si vous préférez, à :

```
PUSH FLAGS
```

et

```
POP FLAGS
```

Cette fonctionnalité peut également être utilisée avec **INVOKE** et **USES**.
FLAGS est donc un mot réservé de GoAsm et ne peuvent être utilisé comme un label.

Voir également :

Pointeurs de chaînes et de données avec PUSH et ARG
Trames de pile pour Callback en 32 et 64 bits

5.2 Changements de segment

En programmation Windows les remplacements de segments sont très rarement utilisés et généralement limités au registre de segment FS.

Dans GoAsm, le remplacement de segment peut se situer avant ou après le mnémonique, et revêtir les formes suivantes :

```
FS OR D[24h],100h
OR FS D[24h],100h
FS MOV [ESI],EAX
MOV FS[ESI],EAX
```

Pour autant, les remplacements de segments ne peuvent pas être dans une position où ils risqueraient d'être confondus avec un registre de segment, pas plus qu'ils ne peuvent d'ailleurs se situer à l'intérieur de crochets, de sorte que les configurations suivantes sont à proscrire impérativement :

```
PUSH FS[0]        ; utiliser plutôt FS PUSH [0]
POP FS[0]         ; utiliser plutôt FS POP [0]
MOV [FS:0],EAX    ; utiliser plutôt FS MOV [0],EAX
```

5.3 Utilisation de l'information du script source

L'information du script source suivante est disponible au moment de la compilation :

```
@line             ligne courante en cours d'assemblage
@filename         nom de fichier du script source principal en cours d'assemblage
@filecur          fichier courant en cours d'assemblage
```

Ces mots sont insensibles à la casse des caractères utilisés. **@filecur** montre le fichier suivant courant dans un **include de fichier en "a"**, alors que **@filename** montre le nom du tout premier script source soumis à GoAsm au démarrage.

@line fournit un entier de 32 bits qui peut être utilisé comme suit :

```
MOV EAX,@line     ; EAX récupère le numéro de ligne
PUSH @line        ; le numéro de ligne est poussé en pile
DD @line          ; le numéro de ligne est déclaré en mémoire
```

@filename et **@filecur** fournissent des pointeurs vers une chaîne contenant le nom et peuvent être utilisés comme suit :

```
PUSH @filename    ; pointeur vers chaîne terminée par un zéro
DB @filename      ; chaîne non-terminée par un zéro
DB @filename,0    ; chaîne terminée par un zéro
```

5.4 Utilisation des compteurs d'emplacement $ et $$

5.4.1 Signification des compteurs d'emplacement

$ position au point d'utilisation en mémoire dans l'exécutable tel que chargé par Windows

$$ position au début de la section en cours dans la mémoire dans l'exécutable tel que chargé par Windows

Parce que ces deux opérateurs donnent des positions en mémoire dans l'exécutable tel que chargé par Windows, leur valeur n'est pas connue pendant la phase d'assemblage par GoAsm, ni par le linker pendant la phase d'édition des liens. Ils agissent à cet égard comme un label de code ou de donnée. Lorsque vous les utilisez, ils ne possèdent pas de valeur, mais ils peuvent être soustraits les uns aux autres ou à des références de mémoire au sein de la même section pour produire une valeur. En effet, leurs valeurs relatives sont connues.

5.4.2 Utilisation des compteurs d'emplacement

Le compteur d'emplacement $ est utile, par exemple, pour obtenir la taille d'une chaîne :

```
HELLO DB 'He finally got the courage to talk to her',0
LENGTHOF_HELLO DB $-HELLO
```

Notez que le compteur d'emplacement $ marque à la fois la position du label LENGTHOF_HELLO et la position de la fin de la chaîne HELLO, de sorte que la longueur de la chaîne sera obtenue par simple soustraction au compteur $ de l'emplacement du label HELLO.

Voici maintenant un exemple où l'on obtient la taille d'un tableau de Dwords en utilisant une méthode de calcul légèrement différente aboutissant à ce que la taille soit contenue dans le premier DWORD de la table elle-même :

```
MESSAGES   DD ENDOF_MESSAGES-$
           DD MESS1,MESS2,MESS3,MESS4,MESS5
ENDOF_MESSAGES:
```

qui est la même chose que :

```
MESSAGES   DD ENDOF_MESSAGES-MESSAGES
           DD MESS1,MESS2,MESS3,MESS4,MESS5
ENDOF_MESSAGES:
```

On voit ici que le premier Dword de la table de valeurs contient la valeur 24 puisque le premier mot est compté également. Avec un peu d'arithmétique, vous pouvez en déduire le nombre de valeurs contenues dans le tableau (ici, au nombre de 5) :

```
MESSAGES   DD (ENDOF_MESSAGES-$-4)/4
           DD MESS1,MESS2,MESS3,MESS4,MESS5
ENDOF_MESSAGES:
```

Dans cette instruction :

```
LABEL400:
JMP $+20h   ;poursuit l'exécution 20h octets plus loin
```

Le compteur d'emplacement $ reflète la position de LABEL400 qui est la même que le début de l'instruction JMP. Par conséquent, les cinq octets constitutifs du codage de l'instruction JMP elle-même (saut relatif utilisant l'opcode E9) doivent être pris en compte dans le calcul.

Voici quelques autres exemples d'utilisation dans la section de code :

```
CALL $$          ; call vers le début de la section courante
MOV EAX,$-$$     ; EAX = distance de l'emplacement actuel par rapport
                 ; au début de la section courante
```

Voici quelques autres exemples d'utilisation de $ et $$ dans une section de données :

```
HELLOZ DD $$             ; HELLOZ contient la position du début de la section
DB 100-($-$$) DUP 0      ; bloc de zéros de la variable HELLOZ jusqu'à l'offset 100
```

Lorsqu'il est utilisé dans une définition, le compteur d'emplacement se réfère à l'emplacement où la définition est *utilisée* plutôt qu'à celui où elle est *déclarée*. Par exemple, dans ce qui suit, $$ se réfère au début de la section de code dans la mesure où la définition de *globule* est utilisée dans la section de code :

```
#define globule $$+2+3
CODE SECTION
MOV EAX,globule
```

5.5 Alignement et utilisation de ALIGN

5.5.1 Qu'est-ce qu'un alignement ?

Toutes les zones de la mémoire ont certaines limites et des points d'alignement, même si l'on considère les adresses virtuelles utilisées par Windows. Prenez les adresses virtuelles 400000h et 410000h où, en général, vous trouverez peut-être les sections de code et de données de votre programme chargées par Windows. Ces deux adresses peuvent être considérées comme pouvant partir à la frontière d'un mot, d'un dword, d'un paragraphe (16 octets) ou même à la frontière d'entités de 400h (qui valent 1K) ou de 10000h (qui correspondent à 64 Ko). En revanche, les adresses 400001h et 410001h ne peuvent s'aligner sur aucun des formats précédents. les adresses 400002h et 410002h pourront être calées sur une limite de mot. Elles seront donc qualifiées de « word aligned ». 400004h et 410004h correspondront à des limites word et dword ; 400010h et 410010h, à des limites word, dword et paragraphe. Ces dernières adresses sont donc correctement décrites comme étant alignées word, dword et paragraphe.

Une approche plus simple consiste à s'intéresser à la divisibilité des adresses pour en déterminer l'alignement :

Adresse	Alignement	Caractéristiques adresse hexa
divisible par 2	Word	le quartet de plus faible poids est pair
divisible par 4	Dword	le quartet de plus faible poids est divisible par 4
divisible par 16	Para	le quartet de plus faible poids est nul

En assembleur, vous pouvez imposer l'alignement des données et du code. Nous allons examiner d'abord l'alignement des données puis nous nous intéresserons brièvement à l'alignement du code.

5.5.2 Nécessité d'un alignement des données

Il y a deux raisons qui militent en faveur d'un alignement de vos données :

- La première consiste à satisfaire Windows.
- La deuxième est d'essayer d'obtenir un peu de vitesse supplémentaire dans l'exécution de vos programmes. Certaines instructions du processeur voient en effet leurs performances accrues par un judicieux alignement des données.

Alignements imposés par Windows

Dans **Windows 32 bits** (NT/2000/XP et Vista fonctionnant comme Win32) de nombreux pointeurs de données à destination des API nécessitent un alignement Dword, et souvent, cette exigence n'est pas documentée.

Même sous Windows 9x, plusieurs pointeurs doivent être alignés Dword comme, par exemple, les structures DLGITEMTEMPLATES et DLGITEMTEMPLATESEX. De plus, le Menu, la classe, le titre et les données de police d'un DLGTEMPLATE doivent être alignés Word et les structures utilisées dans les API de gestion de réseau doivent être alignées Dword.

Certains membres de structures bitmap doivent être alignés en interne. XP impose que la hauteur et la largeur des bitmaps compatibles soit toujours divisible par quatre afin de garantir que chaque ligne soit correctement alignée.

Certaines instructions SSE et SSE2 exigent un alignement de 16 octets de la zone de mémoire à laquelle ils ont affaire. C'est le cas, par exemple, des instructions FXSAVE, FXRSTOR, MOVAPD, MOVAPS et MOVDQA.

Dans **Windows 64 bits** les exigences d'alignement sont encore plus strictes. Il est essentiel de veiller à ce que les membres de structure soient alignés sur leur «frontière naturelle». Ainsi, un mot doit-il être sur une limite Word, une valeur Dword sur une limite Dword, une valeur Qword sur une limite Qword, etc. Cela ne fonctionne que si la structure elle-même est bien alignée sur la limite correcte. Fondamentalement, la structure doit être alignée sur la limite naturelle de son membre le plus important. Il est également important que la même structure se termine sur la *fin* de la limite naturelle de son membre le plus important en ajoutant, le cas échéant, des octets de bourrage. Toujours dans Windows 64 bits, le pointeur de pile RSP doit aussi toujours être aligné selon une modularité de 16 octets lors d'un appel d'API. Voir la section **Exigences en matière d'alignement** dans le chapitre consacré à la programmation 64 bits.

Quelle que soit la plateforme (32 bits ou 64 bits), les résultats sont imprévisibles si l'alignement est incorrect, allant de la simple non-apparition de contrôles, à la sortie pure et simple du programme.

Alignements améliorant la vitesse d'exécution

L'alignement qui permet d'obtenir une vitesse d'exécution optimale varie d'un processeur à l'autre mais généralement, il est de bon usage que les données soient alignées dans la mémoire en fonction de la taille dans laquelle elles opèrent. Par exemple un alignement dword est plausible pour une table de dwords. En théorie, les Qwords et Twords doit être alignés Qword pour une performance optimale.

5.5.3 Réalisation d'un alignement de données correct

En **Win32**, GoAsm aligne automatiquement les structures sur une limite Dword, à la fois quand elles sont déclarées comme données locales et dans la section de données. Ainsi, vous pouvez être certain que tous vos structures fonctionneront. Cependant, vous pouvez ajouter un alignement supplémentaire à ces structures qui sont déclarées dans la section de données en utilisant l'opérateur ALIGN. Vous pouvez également choisir d'utiliser ALIGN sur les données ordinaires (hors structures) pour obtenir la meilleure performance possible.

Un bon alignement peut généralement être réalisé automatiquement en déclarant les données par séquence de taille dans la section de données. Vous pourriez ainsi déclarer tous les Qwords d'abord, puis les dwords, les mots, les octets et, enfin, les chaînes. Les Twords contenant 10 octets, peuvent bouleverser cet ordonnancement. Ils est donc souhaitable de les déclarer en premier puis de corriger l'alignement en utilisant ALIGN.

En **Win64**, GoAsm aligne automatiquement les structures et leurs membres en fonction de la limite naturelle de ces structures et de leurs membres. GoAsm bourre aussi la taille de la structure en fonction des besoins. Enfin, GoAsm aligne automatiquement le pointeur de pile en préalable à un appel d'API.

Voir le chapitre **Programmation en 64 bits** pour plus d'information et voir comment cela fonctionne en pratique.

5.5.4 Alignement du code

Le bon alignement du code relève plus de l'impondérable et dépend véritablement du processeur exécutant le code. Rick Booth aborde le sujet dans son livre "Inner Loops" ("boucles internes") publié par Addison Wesley.
J'ai inclus quelques tests de vitesse dans TestBug qui montrent la différence qui peut résulter d'un bon alignement lors des opérations de lecture, d'écriture ou de comparaison du contenu de la mémoire à une valeur.

5.5.5 Utilisation de ALIGN

GoAsm reconnaît l'opérateur ALIGN *value*, où *value* est la taille de l'alignement requis en octets. Dès lors, GoAsm alignera les données ou le code qui suivent à la limite correcte pour assurer l'alignement. Par exemple :

```
ALIGN 4       ; la donnée suivante sera alignée sur un dword
ALIGN 16      ; (ou ALIGN 10h) aligne ce qui suit sur un paragraphe de 16 octets
```

Afin de réaliser l'alignement dans une section de code, GoAsm effectue un "bourrage" d'octets au moyen d'une succession d'instructions NOP (opcode 90h) qui ont la particularité de n'exécuter aucune opération. Pour une section de données ou const, GoAsm se livre également à un "bourrage" mais utilisant ici des zéros disposés à la bonne place.

Voir aussi **Sections - Gestion avancée** s'agissant de l'alignement de la section.

5.6 Utilisation de SIZEOF

L'opérateur SIZEOF peut être utilisé comme ADDR ou OFFSET, mais au lieu de donner l'*adresse* d'un label de code ou de donnée, il donne sa taille en octets. Comme GoAsm est un assembleur travaillant en une seule passe, SIZEOF ne renvoie la valeur appropriée que pour un label qui commence et se termine avant cet opérateur dans le script source. SIZEOF peut être utilisé soit pour des données ordinaires et du code, soit pour des données locales.

5.6.1 Utilisation de SIZEOF sur les labels de données

Voici quelques exemples illustrant la manière dont SIZEOF peut être utilisé (où Hello est un label de donnée) :

```
MOV EAX,SIZEOF Hello
MOV EAX,SIZEOF(Hello)
MOV EAX,[ESI+SIZEOF Hello]
SUB ESP,SIZEOF Hello
DD SIZEOF Hello
Label DB SIZEOF Hello DUP 0
MOV EAX,SIZEOF Hello+4
MOV EAX,SIZEOF Hello-4
MOV EAX,SIZEOF Hello/2
MOV EAX,SIZEOF Hello*2
```

Lorsqu'il agit sur un label de donnée dans un ensemble de données brutes, l'opérateur SIZEOF calcule la distance entre le label mentionné et le label suivant ou la fin de la section si celle-ci s'avère plus proche, comme le montre l'exemple suivant :

```
Hello    DB 0
         DD 0
Hello2   DB 0
```

alors,

```
MOV EAX,SIZEOF Hello
```

charge dans EAX la valeur 5.

5.6.2 Utilisation de SIZEOF avec des chaînes

Vous pouvez utiliser SIZEOF avec des chaînes (en remplacement du LENGTHOF de Masm). Par exemple :

```
WrongB DB 'You pressed the wrong button!',0
```

alors,

```
MOV EAX,SIZEOF WrongB
```

renvoie la longueur de la chaîne, terminateur null compris.

5.6.3 Utilisation de SIZEOF sur les labels de code

Dans cet exemple :

```
START:
XOR EAX,EAX
XOR EAX,EAX
XOR EAX,EAX
XOR EAX,EAX
LABEL:
MOV EAX,SIZEOF START
```

La valeur 8 sera chargée dans EAX. C'est la taille des 4 instructions `XOR EAX,EAX`.

5.6.4 Utilisation de SIZEOF avec les structures

Vous pouvez utiliser SIZEOF avec des structures pour en retourner la taille. Par exemple, si vous avez

```
Rect STRUCT
   left   DD
   top    DD
   right  DD
   bottom DD
ENDS
rc Rect
```

alors, les instructions

```
MOV EAX,SIZEOF Rect
```

et

```
MOV EAX,SIZEOF rc
```

chargent toutes les deux la valeur 16 dans EAX.

Utilisation de SIZEOF avec des membres de structures

Vous pouvez utiliser SIZEOF pour retourner la taille de membres d'une structure qui est calculée dans ce cas par rapport au label suivant. Par exemple, si vous avez

```
Rect STRUCT
   left   DD
          DD
   right  DD
   bottom DD
ENDS
rc Rect
```

alors, les instructions

```
MOV EAX,SIZEOF rc.left
```

et

```
MOV EAX,SIZEOF Rect.left
```

retournent toutes les deux la valeur 8.

5.6.5 Utilisation de SIZEOF avec des unions et des membres d'union

Vous pouvez utiliser SIZEOF pour retourner la taille d'une union ou de l'un de ses membres, mais gardez à l'esprit que chaque membre retournera la même taille (c'est-à-dire la plus grande taille de membre). Ainsi, par exemple :

```
Sleep STRUCT
     DW 2222h
     DB 0h
ENDS
Ness UNION
     Possums DB L'Balance'
     Koalas Sleep
     Devils  DB 'Roar'
ENDS
Happy Ness
SizeLabel   DD SIZEOF Happy
            DD SIZEOF Ness
            DD SIZEOF Happy.Possums
            DD SIZEOF Happy.Koalas
            DD SIZEOF Happy.Devils
```

Chaque Dword dans SizeLabel contient 14, qui est la taille du plus grand membre de l'union, à savoir Happy.Possums, qui contient une chaîne Unicode de 14 octets de long.

5.6.6 Influence de l'initialisation d'une structure sur sa taille

Lors du processus d'obtention de la taille, les arguments qui pourraient être utilisés pour modifier la taille de la structure sont ignorées. Considérons, par exemple, la structure suivante :

```
StringStruct STRUCT
   DB ?
ENDS
```

Dans cette situation, l'instruction

```
MOV EAX,SIZEOF StringStruct
```

retourne une taille de 1 octet dans EAX, même si la structure avait préalablement été mise en œuvre à l'aide de :

```
LongString StringStruct <'Je hais les structures'>
```

qui agrandit la structure de 22 octets dans cette mise en œuvre. De même, si la longueur de la chaîne repose sur la résolution d'une définition, par exemple

```
StringStruct STRUCT
   DB LONGSTRING
ENDS
```

alors

```
MOV EAX,SIZEOF StringStruct
```

retournerait également une taille de 1 octet dans EAX, quelle que soit la valeur de LONGSTRING.

Les tailles de structure susceptibles de varier avec les définitions *sont*, par exemple, de la forme :

```
Rect STRUCT
   DB TWELVE DUP 0
ENDS
```

Et, dans ce cas, la taille de la structure sera effectivement modifiable.

5.6.7 Initialisation d'une structure avec SIZEOF

Vous pouvez initialiser une structure avec sa propre taille ou la taille d'autre chose, par exemple :

```
PARAM_STRUCT STRUCT
     DD 0
     DD 0
     DD 0
```

```
ENDS
ps1 PARAM_STRUCT <SIZEOF PARAM_STRUCT,,>
```

5.6.8 Utilisation de SIZEOF avec des données locales

La taille des données locales est retournée, par exemple :

```
MyWndProc FRAME
LOCALS hDC,DemonFlag:B,Buffer[256]:B,MyRect:RECT
MOV EAX,SIZEOF hDC          ; 4 en 32 bits, 8 en 64 bits (taille par défaut)
MOV EAX,SIZEOF DemonFlag    ; 1
MOV EAX,SIZEOF Buffer       ; 256
MOV EAX,SIZEOF MyRect       ; 16
RET
ENDF
```

La taille retournée ignore tout comblement opéré par GoAsm pour aligner les données locales correctement sur la pile (toutes les données locales sont alignées DWORD en assemblage 32 bits et alignées QWORD en assemblage 64 bits).
Voir aussi la section **Utilisation de l'instruction LOCALS**.

5.7 Utilisation des branchements prédictifs

5.7.1 Qu'est-ce qu'une prédiction de branchement ?

Les processeurs modernes (par exemple les Pentium du haut de gamme) sont capables d'améliorer sensiblement leur vitesse d'exécution en prédisant à l'avance la prochaine instruction qui sera exécutée après une instruction de saut conditionnel. Par exemple :

```
CMP EDX,EAX      ; compare EDX et EAX
JZ >L1           ; saut au label L1 si EDX = EAX
                 ; autres instructions
L1:
```

Le processeur ne peut être certain à 100%, à l'avance, que EDX sera égal à EAX au moment où l'instruction de comparaison est exécutée. Il pourrait prédire que cela est peu probable, auquel cas, il déciderait que les instructions situées immédiatement après le saut conditionnel doivent être exécutées à la suite. Si cette prédiction s'avère exacte, ces instructions seront donc exécutées immédiatement et sans aucune perte de temps. Dans le cas contraire, il y aurait une certaine perte de temps à passer à l'instruction correcte (juste après L1) en lieu et place. Différents algorithmes de prédiction de branchement sont utilisés par le processeur, y compris certains qui sont capables de tirer parti de leurs erreurs. Une des prédictions de base utilisées comme point de départ postule que :

> *Tous les sauts conditionnels avant ne pourront avoir lieu,*
> *Tous les sauts conditionnels arrière (boucles arrière) auront lieu.*

Il est établi que, dans un code normal, les sauts conditionnels arrière seront effectifs avec une probabilité de 80%, alors que les sauts conditionnels avant seront plus rares. Il est dit que, globalement, la prédiction par défaut est correcte dans 65% du temps. Donc prédire si oui ou non le saut conditionnel aura lieu peut accélérer le code, en particulier sur une série de boucles de retour. En tant que programmeur en assembleur, vous pouvez produire du code rapide en tenant compte des mécanismes de prédiction par défaut du processeur que vous programmez d'autant plus finement que vous êtes en mesure de contrôler complètement votre code.

5.7.2 Les prédicteurs de branchement 2Eh et 3Eh

Dans le P4 (et peut-être dans certains processeurs antérieurs) vous pouvez donner au processeur un "conseil" quant à savoir si oui ou non la branche est susceptible de se produire.
On utilise à cet effet les octets de prédiction de branchement 2EH et 3Eh qui sont insérés immédiatement avant l'instruction de saut conditionnel. Respectivement, ils signifient ce qui suit :

2Eh – prédit que le branchement *ne s'exécutera pas* la plupart du temps
3Eh – prédit que le branchement *s'exécutera* la plupart du temps

L'utilisation du **prédicteur de branchement 2Eh** serait utile (par exemple), si vous aviez un branchement arrière correspondant à un saut conditionnel qui ne se produit que dans le cas d'une erreur présumée rare. Nous avons vu que le processeur, dans son mode de fonctionnement de base, prévoit normalement que le branchement arrière est considéré comme étant susceptible de se produire, ce qui ralentit le code puisque nous attendons précisément le contraire. C'est là que l'insertion du prédicteur 2Eh intervient pour neutraliser, à point nommé, cet automatisme.

Le **prédicteur de branchement 3Eh** serait utile (exceptionnellement) si vous souhaitez créer une boucle où le saut conditionnel serait au début d'un fragment de code et la destination du saut plus avant dans le code. Cette boucle devrait normalement fonctionner plus lentement qu'une boucle de type normal en raison de la prévision par défaut, par le processeur, que le branchement avant est peu probable, à moins que le processeur ne soit capable de tirer parti de ses erreurs de prévision initiales.

5.7.3 Insertion de prédicteurs de branchement

Intel ne propose aucun mnémonique pour insérer les octets de prédiction de branchement. Vous pouvez procéder comme suit :

```
DB 2Eh      ;ou
DB 3Eh
```

Si vous préférez, vous pouvez insérer les octets prédiction de branchement automatiquement. A cet effet, GoAsm utilise la syntaxe qui suit pour insérer l'octet de prédiction de branchement approprié pour le processeur P4 :

```
hint.nobranch     ;insère 2Eh
hint.branch       ;insère 3Eh
```

Pour en revenir au premier exemple, si EDX est normalement égal à EAX, alors ce code sera plus rapide sur le P4 en ajoutant la prédiction de branchement suivante :

```
CMP EDX,EAX           ; compare EDX et EAX
hint.branch JZ >L1    ; effectue normalement le branchement à L1
                      ; autres instructions
L1:
```

J'ai inclus un test de vitesse dans TestBug qui démontre l'amélioration de la vitesse obtenue en utilisant ce mécanisme de prédiction de branchement. Il apparaît par exemple que le code s'exécute environ 1,5 fois plus rapidement sur un processeur par la mise en œuvre de ce procédé.

5.8 Syntaxe des registres FPU, MMX et XMM

Registres	**Syntaxe GoAsm**
FPU x87 (virgule flottante)	ST0, ST1, ST2, ST3, ST4, ST5, ST6 et ST7
Registres MMX et 3DNow!	MM0, MM1, MM2, MM3, MM4, MM5, MM6 et MM7
Registres XMM	XMM0, XMM1, XMM2, XMM3, XMM4, XMM5, XMM6 et XMM7
Les 8 nouveaux registres XMM des processeurs 64 bits	XMM8, XMM9, XMM10, XMM11, XMM12, XMM13, XMM14 et XMM15

Lire **Ecriture des programmes 64 bits** pour connaitre les autres nouveaux registres et leurs méthodes d'adressage.

L'opcode “Wait” (9Bh)

Puisque GoAsm ne fonctionne que sur les processeurs modernes intégrant l'unité de calcul en virgule flottante, il émet l'opcode Wait seulement pour les instructions suivantes :

```
FCLEX
FINIT
FWAIT
FSAVE
FSETPM
FSTCW
FSTENV
FSTSW
```

5.9 Information sur la durée d'assemblage (GOASM_REPORTTIME)

GoAsm peut vous indiquer le temps qu'il a fallu pour assembler les différentes parties de votre script source si vous portez le mot GOASM_REPORTTIME une ou plusieurs fois dans le texte. Le temps restitué par GOASM_REPORTTIME est :

- celui écoulé depuis la spécification GOASM_REPORTTIME précédente (ou le début de l'assemblage),
- celui qui est nécessaire pour atteindre la prochaine spécification GOASM_REPORTTIME (ou la fin de l'assemblage).

Ces durées excluent les phases de mise en place et de nettoyage.

GOASM_REPORTTIME est utile si vous voulez examiner différentes configurations du script source et en déduire celles qui permettront d'accélérer l'assemblage, ou pour identifier une éventuelle partie du script qui ralentirait à elle seule l'assemblage.

5.10 Autres interruptions GoAsm

Vous pouvez envoyer un message à la console à l'occurrence d'un événement lors de l'assemblage (qui peut éventuellement être activé par assemblage conditionnel) en utilisant, par exemple :

```
GOASM_ECHO L'Assembleur a atteint de nouveaux sommets
```

Le matériau à écrire dans la console n'a pas à être entre guillemets, bien qu'il puisse l'être également.

Enfin, vous pouvez forcer GoAsm à arrêter l'assemblage et à en sortir par :

```
GOASM_EXIT
```

5.11 Fichier-listing d'assemblage GoAsm

GoAsm produit un fichier-listing si on lui demande de le faire avec le commutateur /l dans la ligne de commande. Le fichier utilise les mêmes nom et répertoire que le fichier objet de sortie, mais il est pourvu de l'extension .lst. Il est dans le même format que celui de votre fichier source (un fichier source Unicode se traduira par un fichier-listing Unicode). Le volume 2 décrit, de manière plus précise, la programmation en Unicode.

Le listing montre, d'une part les opcodes qui ont été générés à la suite de chaque instruction sur le côté gauche de la page, d'autre part l'instruction et les commentaires sur le côté droit de la page. Si une ligne est trop longue pour être représentée, elle est purement et simplement tronquée. Dans le cas des opcodes un symbole en forme d'ellipse avertit qu'une troncature est intervenue. Les références de mémoire qui requièrent une relocalisation (c'est-à-dire une valeur sera insérée soit par GoAsm, soit par le linker) sont spécifiées entre crochets. Cependant, la relocation insérée ne sera pas affichée.

Les définitions sont présentées telles que définies ou étendues.
Les fichiers d'inclusion pourvus d'un "a" ou "A" dans l'extension sont traités en tant que partie du script source et leur contenu sera inclus dans le listing. Le contenu des fichiers inclus pourvus d'autres extensions ne seront pas inclus dans le listing.
Faites-moi savoir s'il y a quelque chose d'autre que vous aimeriez que le fichier-listing affiche ou si vous souhaitez en modifier le format.

5.12 Messages d'erreur et d'avertissement de GoAsm

GoAsm donne toute l'information relative aux erreurs et alertes sur la ligne de commande. Si GoAsm constate une erreur dans le script source il cessera l'assemblage et interrompra le traitement avec le code de retour TRUE (EAX = 1). GoAsm affiche alors dans la fenêtre MS-DOS (invite de commande) la ligne et l'origine (le script source) de l'erreur survenue, une description de l'erreur, et peut également visualiser le mot ou la ligne de texte incriminés. Si le mot ou la ligne sont définis, il peut montrer l'origine de la définition. Certaines erreurs dans les fichiers inclus «non-a» (ceux qui ne sont pas considérés comme faisant partie du script source) sont tout simplement ignorées, d'autres sont présentées comme des alertes, en fonction de la nature de l'erreur.

Si vous utilisez le commutateurs /b, vous entendrez également un bip sur une erreur.

Je n'ai pas opté pour la faculté offerte à l'assembleur de continuer en dépit d'une erreur, considérant que celle-ci se traduit souvent par d'autres erreurs induisant un phénomène cumulatif, le tout tendant à obscurcir

en définitive la cause originelle. J'ai trouvé que cette réponse à une simple erreur est tout à fait adéquate à la lumière de la syntaxe simplifiée de GoAsm dont on espère réduire les erreurs dans votre script source de toute façon. En plus de cela, si vous écrivez votre code de manière *incrémentale*, j'ai bon espoir que les erreurs seront peu nombreuses.

Je me suis également prononcé contre l'idée consistant à recenser les erreurs dans un fichier séparé, car il serait légitimement fastidieux d'avoir à ouvrir un autre fichier afin d'y lire les erreurs plutôt que de lire directement cette information sur la ligne de commande. Du reste, vous pouvez parfaitement rediriger toutes les sorties GoAsm dans un fichier que vous pourrez lire plus tard en utilisant la commande DOS de redirection, comme par exemple :

```
GoAsm Test.asm >output.fil
```

Une autre façon de contrôler la sortie d'erreurs et d'avertissements GoAsm est d'utiliser les commutateurs suivants sur la ligne de commande :

/b bip sur erreur
/ne pas de messages d'erreur
/ni pas de messages l'information
/nw pas de messages d'avertissement
/no aucun message de sortie du tout

Un simple *avertissement* sera délivré si un mot a été défini plus d'une fois dans la ligne de commande ou dans le script source, mais l'assemblage sera autorisé à continuer pour autant. En effet, il serait inhabituel de définir un mot plus d'une fois et il se peut également que ce soit une erreur de programmation. Il est parfaitement possible d'annuler une définition précédente en utilisant #undef pour que le mot puisse être défini. Dans ce cas, aucun avertissement n'est donné.

Vous obtiendrez également un avertissement si :

(a) vous essayez d'inclure le même fichier deux fois à moins que ce ne soit le type de fichier d'inclusion qui contienne lui-même script source, dans ce cas, il y aura une erreur à la place ;

(b) vous essayez de déclarer plus de 1 Mo de données en double ;

(c) (dans certains cas) si vous utilisez un indicateur de type alors que ce n'est pas nécessaire.

Dans un fichier batch, vous pouvez capter le retour d'erreur avec ERRORLEVEL et en déduire une action. Par exemple, dans ce qui suit, le processus est interrompu s'il y a un retour d'erreur :

```
GoAsm MyFile.asm
IF ERRORLEVEL 1 PAUSE
```

5.13 Utilisation de GoAsm avec différents linkers

5.13.1 Utilisation de GoLink

GoLink est un éditeur de liens libre écrit par moi et disponible à partir de mon site web www.GoDevTool.com. Il accepte les fichiers objet au format PE produits par GoAsm (ou d'autres assembleurs et compilateurs) et les fichiers RES ou OBJ issus de GoRC pour produire un fichier exécutable.
GoLink présente certains avantages par rapport aux autres linkers. En particulier il est conçu pour collaborer étroitement avec GoAsm sur les importations et les exportations, et peut rendre compte des données et labels de code redondants lors de l'édition de liens des fichiers produits par GoAsm. Mais ses principaux avantages sont qu'il fonctionne rapidement en réduisant les «bagages». Les fichiers sont réduits au minimum. Les fichiers LIB ont été supprimés. Au lieu de cela, GoLink regarde les exécutables actuellement stockés sur l'ordinateur au moment de l'édition de liens pour obtenir ses informations sur les importations requises par l'exécutable qu'il constitue. Comme la plupart d'entre eux sont dans la mémoire de toute façon s'il s'agit de DLLs système, cela peut être fait rapidement.

Voir le chapitre consacré à GoLink pour une information complète sur cet éditeur et la manière de l'utiliser (Volume 2). Notez que l'action de GoAsm puis celles de GoLink peuvent utilement être automatisées à l'aide d'un fichier batch (avec l'extension .bat) visant à créer un exécutable. En voici un exemple simple :

```
GoAsm MyProg.asm
GoLink MyProg.obj Kernel32.dll User32.dll
```

Le lancement de ce fichier batch va aboutir à la création du fichier Windows PE *myprog.exe* avec les importations en provenance des DLLs mentionnées. L'adresse d'entrée START est implicite mais cela peut être spécifié.

Vous pouvez utiliser un fichier de commandes avec GoLink par exemple :

```
GoLink @command.fil
```

Au lieu (ou en plus) de spécifier les DLL dans la ligne de commande de GoLink ou les fichiers, vous pouvez utiliser **#DYNAMICLINKFILE** dans le code source GoAsm. La syntaxe est :

```
#dynamiclinkfile path/filename, path/filename
```

La virgule est facultative. L'ensemble *répertoire/nom de fichier* peut être entre guillemets. Un ou plusieurs ensembles *répertoire/nom de fichier* peuvent être spécifiés. Vous n'êtes pas obligé de fournir le chemin d'accès lors de la spécification de fichiers système car GoLink consulte automatiquement le contenu des répertoires système. Le nom du fichier doit être stipulé avec son extension qui peut être .dll, .ocx, .exe ou .drv.

Le nom de fichier est envoyé à GoLink dans la section .drectve du fichier objet créé par GoAsm. Il est utilisé pour donner des informations et des directives à l'éditeur de liens, mais n'a aucune incidence sur l'exécutable final.
Vous devez utiliser GoLink à partir de la version 26.5 pour que cela fonctionne.

Il y a plusieurs autres commutateurs et options da la ligne de commande lors de l'utilisation GoLink. Pour plus de détails consulter le fichier d'aide Golink, GoLink.htm.

5.13.2 Utilisation de ALINK

ALINK est un éditeur de liens libre de Anthony Williams. La sortie de ALINK est plutôt fournie, ce qui fait qu'il s'avère préférable de la rediriger vers un fichier que vous pourrez regarder plus tard. Je vous suggère de constituer à cet effet un fichier batch avec une extension .bat contenant les lignes suivantes :

```
GoAsm MyProg.asm
ALINK @Respons.fil >link.opt
```

Vous exécutez ce fichier batch en saisissant son nom sur la ligne de commande dans une fenêtre MS-DOS (invite de commande) et en appuyant sur Entrée.

Respons.fil est un fichier avec les instructions à l'éditeur de liens qui pourraient être les suivantes (à titre d'exemple) :

```
-m              ; produit un fichier map
-oPE            ; produit un fichier au format PE
-o MyProg.Exe   ; impose le nom du fichier de sortie
-entry START    ; précise le point d'entrée
-debug          ; signifie que des symboles debug doivent être constitués
kernel32.lib    ;
COMCTL32.lib    ;  un fichier lib pour chaque API
COMDLG32.lib    ;  qui est appelé dans votre programme.
user32.lib      ;  fait usage de ALIB qui est
gdi32.lib       ;  un composant du package ALINK.
shell32.lib     ;
MyProg.obj      ; le fichier d'entrée
```

Vous pouvez organiser votre travail en laissant les différents fichiers nécessaires entreposés dans des dossiers différents. Dans ce cas, vous devez inclure les chemins d'accès correspondants dans les instructions données à GoAsm et à ALINK.

5.13.3 Utilisation de l'éditeur de liens Microsoft

L'éditeur de liens Microsoft est disponible gratuitement dans le SDK Microsoft ou en téléchargeant le package MASM. Voir **www.GoDevTool.com** pour plus d'informations ou essayez différents liens vers d'autres sites. L'éditeur de liens Microsoft est plus difficile à utiliser car il prévoit que le label d'adresse de démarrage de programme et les appels externes soient «enrichis» de la manière dont ils sont émis par les compilateurs C et par MASM. L'enrichissement attendu prévoit, en l'occurrence, un caractère de soulignement précédant le label et (dans le cas d'un appel de fonction) le symbole @ suivi du nombre d'octets poussés sur la pile lorsque la fonction est appelée. Normalement GoAsm n'utilise pas d'enrichissement parce que mon objectif a été de tout simplifier autant que possible.

Enrichissement automatique avec le commutateur /ms

Vous pouvez demander à GoAsm de fournir automatiquement l'enrichissement nécessaire en utilisant le commutateur /ms dans la ligne de commande (en 32 bits seulement mais cette fonction est inactivée avec l'utilisation du commutateur /x64). Cette configuration invitera GoAsm à enrichir tous les labels de code, des CALL et INVOKE.

Dans chacun de ces cas, l'enrichissement est sous cette forme :

```
_CodeLabel@x
```

où le label est déclaré comme CodeLabel et où *x* est le nombre d'octets utilisés par les paramètres de CodeLabel. Enrichi de cette manière, CodeLabel est disponible pour d'autres fichiers objets devant être liés par le MS linker (et peuvent donc être appelées à partir de ces autres fichiers objets). Et si CodeLabel réside dans une DLL, le MS Linker va le reconnaître en tant que tel à partir d'un fichier .lib fabriqué à partir de la DLL et qui lui est donné au moment de la phase d'édition de liens.

Au moment de l'édition des liens, le MS linker attend que la valeur de "*@x*" corresponde exactement entre l'appelant et l'appelé. Ceci est donc une forme limitée de contrôle de paramètre. Lorsque le commutateur /ms est utilisé, GoAsm doit donc compter le nombre de paramètres utilisés par le label de code afin d'obtenir la valeur de "*@x*" correcte. Pour ce faire, dans le cas des labels de FRAMEs, GoAsm compte le nombre de paramètres déclarés dans la trame et ajuste l'enrichissement en conséquence. Dans le cas d'un appel utilisant INVOKE, GoAsm compte à nouveau le nombre de paramètres utilisés.

Cependant, GoAsm ne peut pas compter le nombre de paramètres pour un appel utilisant CALL et suppose qu'il n'y en a pas. Pour cette raison, s'il y a des paramètres, *vous devez impérativement utiliser INVOKE pour que l'enrichissement puisse fonctionner correctement* (et non pas un PUSH *xxx* ordinaire suivi d'un CALL). Notez également que si vous utilisez ARG avant INVOKE, chaque argument doit posséder sa propre ligne (ce qui élimine toute représentation telle que ARG 1,2,3).

Ainsi, par exemple, si vous utilisez le code suivant avec le commutateur /ms sur la ligne de commande de GoAsm :

```
HelloProc FRAME hwnd,arg1,arg2
INVOKE MessageBoxA, [hwnd],'Click OK','Hello',40h
```

alors, GoAsm va insérer le symbole HelloProc dans le fichier objet sous la forme

```
_HelloProc@12
```

et mettre la fonction appelée dans le fichier objet sous la forme

```
_MessageBoxA@16
```

En effet, GoAsm sait que 12 octets sont sur la pile dans le cas de HelloProc et que 16 octets sont poussés sur la pile avant que MessageBoxA ne soit appelée.

A partir de la version 0.49 de GoAsm, les labels de codes ordinaires sans paramètres sont également enrichis. Ceci doit permettre que de tels labels de code puissent être reconnus à l'extérieur au moment de l'édition des liens de sorte qu'ils puissent être appelés par d'autres fichiers objets créés par des outils MS. On leur donne un nombre d'octets de paramètre nul. Cela comprend l'étiquette donnant elle-même l'adresse de départ. Donc, supposons que votre adresse de départ dans votre script source soit START: (non précédée par un caractère de soulignement). Elle se trouve maintenant enrichie sous la forme :

```
_START@0
```

et, pour procéder à une édition de liens correcte avec MS Linker, vous devrez inclure ce qui suit dans sa ligne de commande ou le fichier :

```
-ENTRY START@0
```

Enrichissement manuel

Vous pouvez également utiliser le MS linker en effectuant les modifications manuelles suivantes sur votre script source :

- Assurez-vous que le label donnant l'adresse de démarrage du programme commence bien par un caractère de soulignement. Mais *omettez* ce caractère de soulignement lorsque vous le déclarez au MS Linker.
- Si vous avez plus d'un fichier objet à envoyer au MS Linker, et qu'il y a des fonctions dans un fichier objet appelées à partir d'un autre fichier objet, enrichir les labels correspondant à ces fonctions. Les labels de code ordinaires sans paramètres auront besoin d'un caractère de soulignement avant et de @0 à la fin du nom. Les labels de code avec des paramètres ont besoin d'un caractère de soulignement en tête

et de *@x* après le nom où *x* est le nombre d'octets de paramètres (chaque paramètre comprenant quatre octets).

- Enrichissez tous les appels de l'API avec un caractère de soulignement en tête et *@x* après le nom, *x* étant le nombre d'octets poussés sur la pile lorsque l'API est appelée (chaque PUSH occupant quatre octets).

Par exemple :

```
PUSH 12h
CALL _GetKeyState@4        ; test si la touche Alt est pressée
```

L'API *GetKeyState* ne requiert qu'un seul paramètres ce qui fait que quatre octets sont mis sur la pile.

Si vous faites une DLL, vous aurez besoin d'utiliser un label pour l'adresse de départ enrichie de la même manière, par exemple

```
_DLLENTRY@12:
```

Ceci indique que le label DLLENTRY est appelé avec 12 octets poussés sur la pile, soit 3 dwords.

Après avoir effectué ces changements dans votre script source, vous êtes prêt à constituer un fichier batch avec l'extension .bat pour l'assembler, puis traiter le fichier résultant dans l'éditeur de liens. En voici une rédaction possible :

```
GoAsm MyProg.asm
LINK @Respons.fil
```

Vous lancez le fichier batch en saisissant son nom en regard de l'invite de commande MS-DOS puis en validant sur Entrée. Le fichier Respons.fil pourrait contenir les lignes suivantes :

```
/OUT:MyProg.Exe          ; donne le nom du fichier de sortie
/MAP                     ; produit un fichier map
/SUBSYSTEM:WINDOWS       ; élabore un exécutable Windows GDI
/ENTRY:START             ; correspond au _START du script source!
/DEBUG:FULL              ; faire une sortie debug
/DEBUGTYPE:COFF          ; faire des symboles COFF embarqués
MyProg.obj               ; le fichier d'entrée
comctl32.lib             ;
user32.lib               ; fichiers lib pour chaque API
gdi32.lib                ; qui appelle votre programme, ces
kernel32.lib             ; fichiers sont livrés avec le linker
```

Définitions facilitant l'interfaçage avec l'éditeur de liens Microsoft

Pour donner à votre code source une meilleure apparence en utilisant l'éditeur de liens Microsoft vous pouvez faire en sorte que les appels d'API soient enrichis de manière appropriée en définissant le nom de l'API pour une utilisation tout au long de votre code, par exemple :

```
GetKeyState=_GetKeyState@4
CALL GetKeyState
```

ou, pour appeler l'API plus directement, vous pouvez utiliser :

```
GetKeyState=__imp__GetKeyState@4
CALL [GetKeyState]
```

5.13.4 Conservation de symboles de soulignement dans les appels de bibliothèque avec le commutateur /gl

Certains éditeurs de liens, tels que MinGW Linker, attendent au moins un symbole de soulignement pour les appels vers les fonctions C. Si dans GoAsm vous appelez une fonction dans une bibliothèque de code statique, cette bibliothèque peut faire des appels à des fonctions C. Mais GoAsm élimine normalement le symbole de soulignement dans la perspective que vous allez normalement utiliser GoLink. Mais si vous utilisez un éditeur de liens de ce type, il faut que le soulignement soit conservé. C'est précisément la fonction du commutateur /gl dans la ligne de commande.

Chapitre 6
Programmation en 64 bits

Ce chapitre est destiné à ceux qui souhaitent écrire des programmes 64 bits pour les processeurs AMD64 et Intel IA 32/64 fonctionnant en x64 (Windows 64 bits), en utilisant GoAsm (assembleur), GoRC (compilateur de ressources) et GoLink (linker). Il peut également intéresser ceux qui écrivent des programmes en assembleur 64 bits pour Windows avec d'autres outils.

6.1 Introduction à la programmation 64 bits

6.1.1 Programmer en 64 bits, c'est simple !

Malgré les différences entre les processeurs 64 bits et leurs homologues 32 bits et entre les systèmes d'exploitation x64 (Win64) et Win32, l'écriture des programmes Windows 64 bits est aussi facile qu'elle l'était dans Win32 avec l'assembleur GoAsm.

En fait, vous pouvez utiliser sans hésitation le même code source pour créer un exécutable pour les deux plates-formes si vous vous conformez à l'ensemble de règles décrites plus avant dans ce chapitre.

Vous pouvez également convertir un code source 32 bits existant en 64 bits sachant qu'une partie du travail nécessaire à cet effet peut être fait automatiquement en utilisant le convertisseur AdaptAsm.

6.1.2 Différences entre les exécutables 32 bits et 64 bits

Bien que les exécutables 32 bits et 64 bits s'appuient sur le même format PE (Portable Executable), il existe un certain nombre de différences importantes. Leur ampleur a pour conséquence que le code 32 bits ne pourra fonctionner sur Win64 qu'en utilisant le Windows du sous-système Windows (WOW64). Cela fonctionne par l'interception des appels d'API de l'exécutable et la conversion des paramètres en accord avec les exigences de Win64. En revanche, le code 64 bits ne fonctionnera pas du tout sur les plates-formes 32 bits.

L'exécutable contient un flag qui indique au système au moment du chargement s'il est en 32 ou 64 bits. Si le chargeur x64 voit un exécutable 32 bits, WOW64 se lance-automatiquement. Cela signifie qu'il est impossible de faire cohabiter dans le même exécutable du code 32 et 64 bits.

Il résulte de ce qui précède que le programmeur doit choisir entre :

- Faire une version de l'application (Win32) qui fonctionnera sur les deux plates-formes.
- Faire deux versions de l'application, une pour Win32, l'autre pour Win64.

Pour ceux qui sont intéressés par le format de fichier PE, voici un résumé des principales différences entre les exécutables 32 bits et 64 bits :

- Le format de fichier PE pour les fichiers Win64 se nomme «PE +».
- La taille du champ d'en-tête optionnel dans l'en-tête COFF est 0F0h dans un fichier PE+ et 0E0h dans un fichier PE.
- Le "type de machine" dans l'en-tête COFF n'est pas 14CH (comme pour les processeurs x86), mais 8664h (pour le processeur AMD64).
- Le «nombre magique» au début de l'en-tête optionnel est 20Bh au lieu de 10Bh.
- La "majorsubsystemversion" dans un fichier PE+ est de 5 au lieu de 4 dans un fichier PE.
- L'"image" exécutable (le code/data tel que chargé en mémoire) d'un fichier Win64 est limitée en taille à 2 Go. En effet, les processeurs AMD64 / EM64T utilisent l'adressage relatif pour la plupart des instructions, et l'adresse relative est conservée dans un dword. Un dword suffit seulement à gérer une valeur relative de ± 2 Go.
- La table d'adresses d'importation (où le chargeur remplace les adresses des appels externes telles que les adresses des API dans les DLLs système) est élargie à 64 bits, tout comme la table d'importation. En effet, l'adresse d'appels externes pourrait être n'importe où dans la mémoire.
- La base d'image préférée, les champs SizeofStackReserve, SizeofStackCommit, SizeofHeapReserve et SizeofHeapCommit dans l'en-tête optionnel sont élargis de 4 à 8 octets.
- L'adresse de base par défaut dans Win64 est 400000h comme dans les fichiers Win32.
- Les exécutables 64 bits qui assurent correctement en Win64 intégral une gestion d'exception complète contiennent une section .pdata gérant les tables nécessaires à ce effet.

Vous pouvez explorer les arcanes d'un fichier PE en utilisant l'utilitaire **PEview de Wayne J. Radburn**.

6.1.3 Différences entre Win32 et Win64 (pour AMD64/EM64T)

Voici les principales différences entre Win32 et Win64 susceptibles d'intéresser le programmeur en assembleur ou Windows :

- **Convention d'appel**. Win32 utilise la convention STDCALL alors que Win64 utilise la convention FASTCALL. Dans STDCALL tous les paramètres envoyés à une API sont poussés successivement sur la pile (avec un PUSH). Dans Win32 le pointeur de pile (ESP) est réduit de 4 octets à chaque PUSH. En STDCALL il est de la responsabilité de l'API de restaurer l'équilibre de la pile en sortie.
  En FASTCALL, les quatre premiers paramètres sont envoyés à l'API par l'intermédiaire de registres (dans l'ordre : RCX, RDX, R8 puis R9), mais le cinquième paramètre et les suivants sont classiquement mis en pile par des PUSH. Dans Win64, le pointeur de pile (RSP) est réduit de 8 octets à chaque PUSH. Contrairement à STDCALL, il n'est de la responsabilité de l'API de rétablir l'équilibre de la pile en fin d'exécution. Cette tâche incombe à l'appelant de l'API. Ce dernier doit également veiller à ce qu'il y ait un espace suffisant sur la pile pour que l'API puisse y stocker les paramètres passés dans les registres. En pratique, cela est obtenu en réduisant le pointeur de pile de 32 octets juste avant l'appel.
  Notez que dans GoAsm tous le travail requis par la convention d'appel FASTCALL se fait automatiquement si vous utilisez INVOKE ou ARG suivie de INVOKE. Voir la section concernant le **codage pour se conformer à la convention d'appel FASTCALL**. L'utilisation de ARG et INVOKE est décrite en détail dans le corps du manuel GoAsm.
  Notez que GoAsm ne fait pas encore ceci pour les paramètres qui doivent être envoyés dans les registres XMM (ie. dans instructions en virgule flottante).
- Windows utilise la convention FASTCALL pour appeler les **procédures de fenêtre** et autres procédures callback dans votre application. Cela signifie que vos procédures de fenêtre vont récupérer les paramètres d'une manière différente sous Win64. En outre, les procédures de fenêtre n'ont plus à restaurer la pile à l'équilibre en sortie.
  Notez que GoAsm mettra en œuvre ces choses automatiquement si vous utilisez FRAME ... ENDF qui est décrite en détail dans le corpus du manuel GoAsm.
- Toutes les fonctions utilisant une trame de pile (y compris les procédures de fenêtres) doivent respecter certaines règles si elles souhaitent faire usage de la **gestion d'exceptions**. Les outils doivent également ajouter des enregistrements de trame d'exception à l'exécutable. Ces aspects seront également gérés automatiquement par les outils "Go". *Notez cependant qu'ils ne sont pas encore disponibles à ce jour.*
- **Volatilité des registres**. En Win32, les procédures de fenêtres et autres procédures callback doivent restaurer le contenu des registres EBP, EBX, EDI et ESI avant de retourner à l'appelant (si la valeur de ces registres est modifiée). Cette fonction est assurée par les API Windows (ces registres ne changent pas lorsque vous appelez une API). C'est pour cette raison qu'ils reçoivent le qualificatif de registres "non volatiles". Dans Win64, cette catégorie de registres est étendue à RBP, RAX, RDI, RSI, R12 à R15 et XMM6 à XMM15.
  Les registres dits «volatiles» sont ceux qui peuvent être modifiés par les API, et dont vous n'avez pas besoin pour sauvegarder et restaurer dans vos procédures de fenêtre et autres procédures callback. Dans Win32, les registres volatils à usage général étaient EAX, ECX et EDX. Ceux-ci sont maintenant étendus à RAX, RCX, RDX, et R8 à R11.
- Vous pourriez ne pas avoir prévu cela, mais dans l'assemblage 64-bit pour AMD64, les **pointeurs de code et de donnée**s dont les adresses sont dans l'exécutable ne sont encore qu'à 32 bits. Cela rejoint le fait que l'**adressage relatif RIP** limite la taille de l'exécutable à 2 Go. Les pointeurs vers des adresses externes, telles que les fonctions de DLLs, sont à 64 bits afin que la fonction puisse se situer n'importe où dans la mémoire. Voir, à ce sujet la section **tailles des adresses de call**.
- En Win64 la **taille de donnée** de tous les handles et pointeurs est maintenant de 64 bits au lieu de 32 bits. Voir à ce sujet la section **modifications des types de données Windows**.
- En Win64 il y a des exigences plus strictes pour l'alignement de la pile, des données, et pour les structures (voir la section **Alignement et comblement automatiques des structures et de leurs membres**).

- Les **API Windows** ont été modifiées pour fonctionner en 64 bits. Il y a, cependant, un petit nombre de nouvelles APIs pour gérer les exigences supplémentaires de fonctionnement en 64 bits. En voici la liste :

  GetClassLongPtr
  GetWindowLongPtr
  SetClassLongPtr
  SetWindowLongPtr

  Notez que, comme dans Win32, vous pouvez écrire vos applications soit avec la version ANSI (cas le plus courant), soit avec la version Unicode des APIs. Voir le chapitre "Écriture des programmes Unicode" du volume 2.

6.1.4 Différences entre les processeurs x86 et x64

Les principales différences résident dans l'extension de la gamme registres disponibles, des modifications apportées à quelques instructions et l'utilisation de l'adressage relatif RIP. Les notes ci-dessous font référence au processeur AMD64 en mode 64 bits. Dans ce mode ce processeurs peut également faire fonctionner des exécutables 32 bits.

6.1.5 Registres

Le AMD64 ajoute plusieurs nouveaux registres à ceux disponibles dans la série de processeurs x86, et propose également de nouvelles façons d'aborder les registres existants.

- Les registres à "usage général" EAX, EBX, ECX, EDX, ESI, EDI, EBP et ESP sont tous élargis à 64 bits et prennent respectivement pour noms RAX, RBX, RCX, RDX, RSI, RDI, RBP et RSP
- Vous pouvez toujours accéder au Dword de poids faible de ces registres (ie. les 32 bits moins significatifs) en utilisant les noms originels EAX, EBX, ECX, EDX, ESI, EDI, EBP et ESP.
- Vous pouvez toujours accéder au mot de plus faible poids de ces registres (ie. les 16 bits moins signficatifs) en utilisant les noms originels AX, BX, CX, DX, SI, DI, BP et SP.
- Vous pouvez toujours accéder au premier octet de RAX, RBX, RCX et RDX (ie. les 8 bits les moins significatifs) en utilisant les noms existants AL, BL, CL, DL comme dans le processeur x86. De plus, il vous est désormais possible de communiquer avec l'octet de plus faible poids des registres "d'index" en utilisant SIL, DIL, BPL et SPL. SIL correspond, par exemple, aux 8 bits les moins significatifs du registre d'index RSI.
- De même, vous pouvez toujours accéder aux deuxième octet (bits 8 à 15)de RAX, RBX, RCX et RDX en utilisant les noms existants AH, BH, CH, DH comme dans le processeur x86. Cependant, les opcodes pour ce faire ont été modifiés dans le processeur AMD64. Ils entrent en conflit maintenant avec les opcodes désignant la version 8 bits des registres étendus R8 à R15. Vous ne pouvez donc pas utiliser AH, BH, CH, DH et R8B à R15B dans la même instruction.
- Il y a huit nouveaux registres 64 bits (les «registres étendus») nommés R8 et R15.
- Le dword de plus faible poids de ces registres (ie. les 32 bits les moins significatifs) est accessible sous la forme R8D à R15D.
- Le mot de plus faible poids de ces registres (ie. les 16 bits les moins significatifs) est accessible sous la forme R8W à R15W.
- L'octet de plus faible poids de ces registres (ie. les 8 bits les moins significatifs) est accessible sous la forme R8B à R15B.
- Il y a 8 nouveaux registres XMM (128-bit) nommés XMM 8 à XMM 15.
- Les registres MMX 64 bits (MM0 à MM7) sont toujours disponibles. Comme dans le processeur x86 ils sont également utilisés en tant que registres à virgule flottante (ST0 à ST7) pour les instructions x87 en virgule flottante.
- Le pointeur d'instruction est maintenant dans le registre 64 bits RIP.

6.1.6 Instructions

- Certaines instructions ne sont pas disponibles dans le mode 64 bits. Les opcodes sont maintenant utilisés à d'autres fins. La liste complète est fournie dans les manuels AMD et Intel et comprend AAA, AAD, AAM, AAS, DAA, DAS, PUSH CS, ainsi que les opérations PUSH et POP sur les registres de segment DS, ES et SS.
- La portée des instructions est élargie pour accueillir les nouveaux registres et les nouvelles formes d'adresse, par exemple :

```
MOV RAX,immediate  ; mémorise un nombre 64 bits dans un registre 64 bits
JRCXZ >L1          ; si RCX = 0 saut vers l'avant au label L1
```

- Les instructions de chaîne sont maintenant agrandies pour permettre l'adressage 64 bits pour, par exemple :

```
LODSB   ; équivaut maintenant à MOV AL,[RSI] puis INC RSI
LODSW   ; équivaut maintenant à MOV AX,[RSI] puis ADD RSI,2
LODSD   ; équivaut maintenant à MOV EAX,[RSI] puis ADD RSI,4
LODSQ   ; NOUVEAU ! équivaut à MOV RAX,[RSI] puis ADD RSI,8
CMPSB   ; équivaut maintenant à CMP B[RSI],B[RDI] puis INC RSI,RDI
CMPSQ   ; NOUVEAU ! équivaut à CMP Q[RSI],Q[RDI] puis ADD RSI,8 et ADD RDI,8
MOVSW   ; équivaut maintenant à MOV W[RDI],W[RSI] puis ADD RSI,2 et ADD RDI,2
MOVSQ   ; NOUVEAU ! équivaut à MOV Q[RDI],Q[RSI] puis ADD RSI,8 et ADD RDI,8
SCASD   ; équivaut maintenant à CMP [RDI],EAX puis ADD RDI,4
SCASQ   ; NOUVEAU ! équivaut à CMP [RDI],RAX puis ADD RDI,8
STOSQ   ; NOUVEAU ! équivaut à MOV [RDI],RAX puis ADD RDI,8
```

  Les préfixes de répétition REP, REPZ et REPZ utilisent RCX en lieu et place de ECX. Les instructions de boucle LOOP, LOOPZ et LOOPNZ font de même. L'instruction de consultation de table XLAT utilise RBX plutôt que EBX.
- En dehors de ce qui précède, la seule nouvelle instruction notable utilisable par les programmeurs est MOVSXD qui peut déplacer de 32 bits de données à partir d'un registre ou de la mémoire vers un registre 64 bits, avec extension du bit de signe 31 dans tous les bits supérieurs. On note aussi un ensemble de nouvelles instructions système.
- Dans l'AMD64, chaque instruction PUSH et POP déplace le pointeur de pile de 8 octets au lieu de 4 octets comme dans le processeur x86. Cela signifie que l'instruction PUSH *registre 32 bits* n'est plus admise sur le processeur AMD64. Pour aider à la compatibilité du code source, GoAsm traite (par exemple) PUSH EAX comme équivalent à PUSH RAX. En mode /x86, GoAsm traite PUSH RAX comme équivalent à PUSH EAX. Ces interprétations automatiques de l'écriture initiale exigent une grande attention de la part du programmeur.
- PUSH *immédiate* sur l'AMD64 prend une valeur immédiate limitée à 32 bits (nombre), en étend le signe du bit 31 en copiant ce dernier du bit 32 au bit 63 inclus, puis pousse en pile l'ensemble 64 bits ainsi constitué. Il n'y a en effet aucune instruction capable de pousser directement une valeur immédiate 64 bits sur la pile. Pour cette raison, l'instruction PUSH ADDR CHOSE n'est pas reconnue sur l'AMD64 (la valeur de décalage est traitée comme étant *immédiate*). Le problème ici est que la valeur immédiate courante de tout décalage est inconnue jusqu'au moment de l'édition de liens et qu'au moment de l'assemblage il est impossible pour l'assembleur de savoir si ce décalage est au-dessus 7FFFFFFFh et qu'il serait ainsi touché par l'extension de signe.

  Par conséquent, dans GoAsm, PUSH ADDR CHOSE ***fait usage du registre R11*** *et profite de l'adressage relatif RIP plus court de LEA avec le codage suivant :*

```
LEA R11,[CHOSE]
PUSH R11
```

- Les instructions 3DNow! sont toujours disponibles dans le AMD64. On ne sait pas si elles sont maintenant disponibles sur les processeurs compatibles avec la technologie Intel EM64T.

6.1.7 L'adressage relatif RIP

Certaines instructions du processeur AMD64 qui adressent le code ou les données, utilisent l'adressage relatif RIP pour ce faire. L'adresse relative est contenue alors dans un dword qui fait partie de l'instruction. Lorsque vous utilisez ce type d'adressage, le processeur ajoute trois valeurs : (a) le contenu du dword contenant l'adresse relative (b) la longueur de l'instruction et (c) la valeur de RIP (le pointeur d'instruction cou-

rant) au début de l'instruction. La valeur résultante est alors considérée comme l'adresse absolue de donnée ou de code devant être traitée par l'instruction. Dans la mesure où l'adresse relative peut être négative, il est possible d'adresser la donnée ou le code plus tôt, ou plus tard, dans la représentation de RIP. La gamme est d'environ ± 2 Go, en fonction de la taille de l'instruction. Si l'on considère que l'adressage relatif ne peut s'exercer au-delà de cette plage, cela constitue la limite pratique de taille des représentations 64 bits.

L'adressage relatif RIP agit à l'insu de l'utilisateur. Le processeur utilise ce mode si les opcodes contiennent certaines valeurs (dans l'octet ModRM, le champ Mod est égal à 00 binaire, et le champ r/m est égal à 101 binaire). Vous ne pouvez pas en prendre le contrôle, sauf en changeant le type d'instructions que vous utilisez. En général, voici les règles qui déterminent si oui ou non une instruction utilise l'adressage relative RIP :

- Les adresses dans les données ne peuvent pas utiliser l'adressage relatif RIP puisque la valeur de ce registre ne peut être connue au moment où ces adresses sont définies. Au lieu de cela, une adresse absolue pour l'insertion est calculée au moment de l'édition des liens. Ainsi, par exemple, les instructions suivantes n'utilisent pas l'adressage relatif RIP, mais lui préfèrent l'adressage absolu :

```
MyDataLabel1 DQ MyDataLabel3      ; adresse du label de donnée
MyDataLabel2 DQ MyCodeLabel1      ; adresse du label de code
MyDataLabel3 DQ $                 ; utilisation du pointeur courant de donnée
MyDataLabel4 DD MyDataLabel3      ; adresse du label de donnée
MyDataLabel5 DT MyCodeLabel1      ; adresse du label de code
MyDataLabel6 DD $                 ; utilisation du pointeur courant de donnée
```

  Notez que, dans la pratique, l'adresse absolue est contenue dans un DWORD et non dans un QWORD. Voilà pourquoi, dans les exemples qui précèdent, les adresses de donnée et de code peuvent être contenues dans une déclaration de données DWORD. Cette restriction est possible parce que la taille pratique de l'image est limitée à 2 Go de toute façon en raison des restrictions imposées par l'adressage relatif RIP.

- Les offsets convertis en valeurs immédiates soit au moment de l'assemblage, soit au moment de l'édition des liens utilisent l'adressage absolu plutôt que l'adressage relatif. Par exemple, les instructions suivantes n'utilisent pas l'adressage relatif RIP, mais lui préfèrent l'adressage absolu :

```
MOV RAX,ADDR MyDataLabel3         ; adresse du label de donnée copiée dans un registre
MOV MM0,ADDR MyCodeLabel1         ; adresse du label de code copiée dans un registre
MOV Q[RSP],ADDR MyDataLabel3      ; adresse du label de donnée copiée dans une adresse
                                  ; mémoire
MOV Q[RSP],ADDR MyCodeLabel1      ; adresse du label de code copiée dans une adresse
                                  ; mémoire
```

  Cependant, GoAsm code en fait `MOV RAX, ADDR MyDataLabel3` *et les instructions similaires en utilisant l'instruction LEA plus courte, qui utilise l'adressage relatif RIP.*

  A noter également que pour un MOV à destination de la mémoire d'un ADDR, GoAsm utilise le registre R11 et tire avantage de l'adressage relatif RIP plus court de LEA avec le codage suivant :

```
LEA R11,ADDR Non_Local_Label
MOV [Memory64],R11
```

- Voici des exemples d'autres instructions qui utilisent l'adressage relatif RIP :

```
MOV RAX,[MyDataLabel3+55h]        ; adresse du label de donnée
RCL Q[MyDataLabel3],1             ; adresse du label de donnée
MOV Q[MyDataLabel3],20h           ; adresse du label de donnée
PAVGUSB MM3,[MyDataLabel3]        ; instruction 3DNow!
CALL ExitProcess                  ; adresse du label de code (API système)
JMP InternalCodeLabel1            ; adresse du label de code à l'intérieur du module
CALL InternalCodeLabel1           ; adresse du label de code à l'intérieur du module
CALL ExternalCodeLabel1           ; adresse du label de code à l'extérieur du module
PUSH [MyData]                     ; sauvegarde du contenu d'un label de donnée
POP [MyData]                      ; restauration du contenu d'un label de donnée
```

  Notez que, dans le cas d'un appel externe, l'adresse relative pointe sur la table d'adresses d'importation (Import Address Table). Dans la mesure où cette table est maintenant étendue à 64 bits, il est possible d'appeler un label de code partout dans la mémoire.

- LEA utilise l'adressage relatif RIP. Par exemple :
```
LEA RBX,MyDataLabel3  ; charge dans RBX l'adresse du label de donnée
```
- L'adressage relatif RIP n'est pas utilisé lorsque le label de donnée ou de code est complété par un registre d'index. Bien que cela puisse sembler étrange, la raison semble être que l'ajout d'informations sur le registre sur l'opcode signifie que le processeur ne peut plus reconnaître l'instruction en tant qu'utilisatrice de l'adressage relatif RIP (dans l'octet ModRM, le champ Mod n'est plus égal à 00 binaire, et le champ r/m n'est plus égal à 101 binaire). Cela signifie que les instructions suivantes utilisent des adresses absolues plutôt que relatives :
```
MOV RAX,[ESI+MyData]
RCL Q[EBX+MyData],1
MOV Q[RSI*2+MyData],44444444h
PAVGUSB MM3,[R12+MyData]
LEA RBX,MyData+RSI
CALL [MyCall+RDI]
JMP [MyJump2+RDI]
PUSH [MyCall+RSI]
POP [MyCall+R12]
```
Dans la mesure où l'adressage relatif RIP n'est pas utilisé ici, la Base Image doit être inférieure à 7FFFFFFF pour que ces types d'instructions fonctionnent correctement. Ces instructions doivent être réaménagées si vous utilisez une plus grande Base Image, sauf à pratiquer l'édition de liens avec l'option /LARGEADDRESSAWARE.

Ayant à l'esprit que la taille de l'image est limitée à 2 Go par les dispositions ci-dessus, on pourrait penser que les avantages de l'adressage relatif RIP sont quelque peu limités. Le seul intérêt de cette pratique semble résider dans le fait qu'elle réduit le nombre de relocations qui devraient être effectuées par le chargeur si une DLL venait à être chargée à une adresse inattendue. Le chargeur aurait alors besoin d'ajuster toutes les adresses absolues en fonction de la base de l'image réelle, alors que des adresses relatives n'auraient pas à être modifiées car elles se réfèrent à d'autres parties de l'image virtuelle de l'exécutable. Cependant, il est de bonne pratique pour le programmeur de choisir une base d'image appropriée au moment de l'édition de liens pour éviter la nécessité de relocations dans une DLL en premier lieu. Un bon exemple de ceci est celui des DLLs système. Elles ont toutes une base d'image différente qui permet d'éviter efficacement des conflits potentiels de l'image dans la mémoire qui nécessiterait la réinstallation au moment du chargement.

6.1.8 Taille des adresses de Call

Lors de l'assemblage 64 bits, un simple CALL à un label de code, comme par exemple

```
CALL CALCULATE
```

sera codé E8 en tant qu'appel relatif RIP utilisant un dword pour fournir le décalage de RIP. La destination de cet appel pourrait être un label de code interne (c'est-à-dire une procédure ou une fonction au sein de l'exécutable lui-même). Ou elle pourrait être un label de code *externe*, comme une API dans une DLL système ou un label de code exporté par un autre EXE ou une DLL. La première destination de l'appel d'un label de code externe est la Table d'Adresses d'Importation (Import Address Table) qui fait partie de l'exécutable lui-même. Cette table est écrite par le chargeur lorsque l'exécutable commence. Par conséquent, pendant l'exécution, la table contient les adresses absolues dans la mémoire virtuelle de la destination éventuelle de l'appel. Dans un exécutable 64 bits, la table contient des valeurs 64 bits, de sorte que l'appel relatif RIP codé E8 est capable d'appeler une procédure ou une fonction partout dans la mémoire.

L'appel à une adresse mémoire identifiée par un label de donnée, contenue dans un registre ou dans un emplacement mémoire pointé par un registre est traité, en revanche, de manière différente. Il n'est pas acheminé par l'intermédiaire de la Table d'Adresses d'Importation. Cet appel doit également permettre d'autoriser la destination de l'appel partout dans la mémoire. Pour ce faire, il doit utiliser une adresse absolue 64 bits. Voici quelques exemples de ces types d'appels :

```
CALL RAX
CALL EAX          ; produit le même code que CALL RAX
CALL [Table+8h]
CALL [RSI]
CALL [ESI]        ; produit le même code que CALL [RSI]
```

Ici, vous devez tenir compte du fait que l'appel est dirigé vers un QWORD et non vers un DWORD.

Voir la section **quelques pièges à éviter lors de la conversion du code source existant**.

6.2 Modifications apportées aux types de données Windows

Voici une liste des changements apportés aux types de données entre 32 et 64 bits :

6.2.1 Tous les handles passent de DWORD à QWORD

Par exemple

```
HACCEL, HINSTANCE, HBRUSH, HBITMAP
HCOLORSPACE, HCURSOR, HDC, HFONT
HICON, HINSTANCE, HKEY, HLOCAL
HMENU, HMODULE, HPEN, HPALETTE, HWND
(+ les autres commençant par H)
```

Exceptions : HRESULT, HFILE demeurent des DWORDs, et HALF_PTR (voir ci-dessous)

6.2.2 Tous les pointeurs passent de DWORD à QWORD

Par exemple

```
LPCSTR, LPCTSTR, LPLONG, LPSTR
(+ les autres commençant par LP)
PBOOL, PHANDLE, PHKEY, PVOID
(+ les autres commençant par P)
DWORD_PTR, ULONG_PTR, UINT_PTR
(+ les autres s'achevant par _PTR)
et LRESULT
```

Exceptions : HALF_PTR, et UHALF_PTR qui sont maintenant des dwords au lieu de words
POINTER_32 which remains a 32-bit pointer

6.2.3 WPARAM et LPARAM deviennent des QWORDs au lieu de DWORDs

Voici une liste des types de données qui demeurent inchangés :

ATOM	demeure un word	LONG64	demeure un qword
BOOL	demeure un dword	LONGLONG	demeure un qword
CHAR	demeure un byte	POINT	demeure 2 dwords
DWORDLONG	demeure un qword	RECT	demeure 4 dwords
COLORREF	demeure un dword	SHORT	demeure un word
INT	demeure un dword	UINT	demeure un dword
INT32	demeure un dword	UINT32	demeure un dword
INT64	demeure un qword	UINT64	demeure un qword
LANGID	demeure un word	ULONG	demeure un dword
LCTYPE	demeure un dword	ULONG32	demeure un dword
LCID	demeure un dword	ULONG64	demeure un qword
LGRPID	demeure un dword	ULONGLONG	demeure un qword
LONG	demeure un dword	USHORT	demeure un word
LONG32	demeure un dword		

6.2.4 Utilisation de l'indicateur de type commutable

La modification ci-dessus du type d'une donnée peut nécessiter la modification correspondante d'un indicateur de type. La lettre P est réservée en tant qu'indicateur de type paramétrable dans toutes les situations où GoAsm pourrait s'attendre à en trouver un. Les choses peuvent se présenter de la manière suivante :

```
#if x64
   P = 8
#else
   P = 4
#endif
```

P peut prendre une valeur correspondant à l'un quelconque des indicateurs de type prédéfinis tels que B, W, D, Q ou T. Concrètement, P contient le nombre d'octets du type. Dans l'exemple ci-dessus, P représente le type Q (valeur 8) ou D (valeur 4). Par conséquent, vous pouvez contrôler la taille de l'instruction avec elle, par exemple :

```
MOV P[RDI],0            ; RAZ du qword à RDI si 64-bit, du dword à EDI si 32-bit
LOCAL POINTERS[10]:P    ; fait un pointeur de buffer local de 80 octets si 64 bits,
                        ; de 40 bits si 32 bits
```

6.3 Exigences en matière d'alignement

Les exigences du système dans Win64 en matière d'alignement du pointeur de pile, des données et des membres de structure sont beaucoup plus strictes que dans Win32. Un mauvais alignement peut causer, au mieux une perte de performance, au pire, une exception ou une sortie inopinée du programme.

6.3.1 Alignement de la Pile

Le pointeur de pile (RSP) doit être aligné sur une modularité de 16 octets lors d'un appel d'API. Cependant, cet alignement est géré automatiquement par GoAsm si vous utilisez INVOKE. Voir sur ce point la section **alignement automatique de la pile**.

6.3.2 Alignement des Données

Toutes les données doivent être alignées sur une «frontière naturelle». Donc, un octet peut être aligné sur 1 octet, un mot sur 2 octets, un DWord sur 4 octets et un QWord sur 8 octets. Un TWord devrait également être aligné sur un QWord. GoAsm procède à un alignement automatique lorsque vous déclarez des données locales (dans une zone FRAME ou USEDATA). Mais il sera nécessaire que vous organisiez vos propres déclarations de données pour garantir que les données seront correctement alignées. La meilleure façon d'y parvenir est de déclarer tout d'abord tous les QWords, puis tous les DWords, puis tous les Words et, enfin, tous les octets. Les TWords (10 octets) éventuels pourraient cependant mettre à mal l'alignement des déclarations suivantes, de sorte qu'il est recommandé de les déclarer en tout premier lieu, puis de déclarer les types inférieurs dans l'ordre décroissant (QWord, puis, DWord, etc.) en ayant soin de les faire précéder par la directive d'alignement ALIGN 8.

S'agissant des chaînes et en conformité avec les règles ci-dessus, les chaînes Unicode doivent être alignées sur 2 octets, alors que les chaînes ANSI peuvent l'être sur 1 octet.

Lorsque des structures sont utilisées, elles doivent être alignées sur la *« frontière naturelle »* du plus grand membre. Tous les membres de la structure doivent également être alignés correctement, et la structure elle-même doit être comblée, le cas échéant, par des octets de bourrage de manière à ce qu'elle puisse se terminer sur une limite naturelle (le système peut être amené à écrire dans cette zone). En raison de l'importance fondamentale de cette contrainte, GoAsm aligne automatiquement les structures depuis la version 0.56 (beta). Voir la section **Alignement et comblement automatiques des structures et de leurs membres**.

6.4 Structures Windows en programmation 64 bits

Windows utilise souvent les structures pour envoyer et recevoir des informations lorsque l'on utilise des API. En 64 bits, ces structures sont susceptibles d'être considérablement différentes de leurs homologues 32 bits en raison de l'élargissement de nombreux types de données à 64 bits. Voir la section **Modifications apportées aux types de données Windows**.

6.4.1 Structure WNDCLASS

Prenons par exemple la structure WNDCLASS qui est utilisée lorsque vous souhaitez enregistrer une classe de fenêtre :

WNDCLASS en 64 bits

```
WNDCLASS STRUCT
         style DD 0     ;+0 style de classe de fenêtre
               DD 0     ;+4 octets de comblement
   lpfnWndProc DQ 0     ;+8 pointeur vers la procédure de fenêtre
               DD 0     ;+10 nb d'octets supplémentaires à allouer après la structure
               DD 0     ;+14 nb d'octets supplémentaires à allouer après l'instance de fenêtre
     hInstance DQ 0     ;+18 handle de l'instance contenant la procédure de fenêtre
         hIcon DQ 0     ;+20 handle de l'icône de classe
       hCursor DQ 0     ;+28 handle du curseur de classe
 hbrBackground DQ 0     ;+30 identifie la brosse de l'arrière-plan de la classe
  lpszMenuName DQ 0     ;+38 pointeur vers le nom de ressource pour le menu de classe
 lpszClassName DQ 0     ;+40 pointeur vers la chaîne contenant le nom de la classe de fenêtre
ENDS
```

Un certain nombre de membres sont maintenant des QWords, alors qu'auparavant il s'agissait de DWords comme vous pouvez le voir dans la version 32 bits ci-dessous. Le style de classe à l'offset + 0h demeure un DWord, mais dans la version 64-bit, un bourrage de quatre octets est nécessaire parce que le membre suivant est un QWord. Ceci est conforme à l'exigence selon laquelle les membres d'une structure doivent être alignés sur leur frontière naturelle. Suivent, jusqu'à la fin de la structure, une série de QWords incluant 3 pointeurs. Le premier (en +8h) est utilisé pour pointer la procédure Window elle-même, le second, en + 38h pointe le nom de la ressource de la classe Menu et le troisième, en +40h, pointe le nom de classe de fenêtre. Ceci en dépit du fait que les programmes 64 bits tels qu'implémentés par Win64 pour le processeur AMD64 utilisent uniquement des pointeurs 32 bits où ces pointeurs donnent les adresses des données internes. On peut en attribuer la raison au fait que les structures utilisées ici sont les mêmes que celles qui sont mises en œuvre pour la famille de processeurs Intel IA-64 (lesquels emploient, pour leur part, des pointeurs 64 bits en direction des données internes). Enfin, les handles de la structure sont également étendus à 64 bits.

WNDCLASS en 32 bits

```
WNDCLASS STRUCT
         style DD 0     ;+0 style de la classe de fenêtre
   lpfnWndProc DD 0     ;+4 pointeur vers la procédure de fenêtre (callback)
               DD 0     ;+8 nb d'octets supplémentaires à allouer après la structure
               DD 0     ;+C nb d'octets supplémentaires à allouer après l'instance de fenêtre
     hInstance DD 0     ;+10 handle de l'instance contenant la procédure de fenêtre
         hIcon DD 0     ;+14 handle de l'icône de classe
       hCursor DD 0     ;+18 handle du curseur de classe
 hbrBackground DD 0     ;+1C identifie la brosse de l'arrère-plan de la classe
  lpszMenuName DD 0     ;+20 pointeur vers le nom de la ressource pour le menu de classe
 lpszClassName DD 0     ;+24 pointeur de la chaîne contenant le nom de classe de fenêtre
ENDS
```

Voici un autre exemple, cette fois celui de la structure DRAWITEMSTRUCT sous ses formes 32 et 64 bits :

```
UINT CtlType          ;+0
UINT CtlID            ;+4
UINT itemID           ;+8
UINT itemAction       ;+C
UINT itemState        ;+10
HWND hwndItem         ;+14
HDC hDC               ;+18
RECT rcItem           ;+1C
ULONG_PTR itemData    ;+2C

(taille totale structure = 30h octets)
```

DRAWITEMSTRUCT en 32 bits

```
UINT CtlType          ;+0
UINT CtlID            ;+4
UINT itemID           ;+8
UINT itemAction       ;+C
UINT itemState        ;+10
  dwords de bourrage
HWND hwndItem         ;+18
HDC hDC               ;+20
RECT rcItem           ;+28
ULONG_PTR itemData    ;+38
(taille totale structure = 40h octets)
```

DRAWITEMSTRUCT en 64 bits

En 64 bits, on retrouve l'exigence selon laquelle la structure est agrandie de telle sorte qu'elle se termine sur la limite naturelle de son plus grand membre. Cette contrainte est satisfaite en ajoutant le bourrage nécessaire à l'extrémité de la structure.

Dans ces mêmes conditions, PAINTSTRUCT devient, en 64 bits :

```
PAINTSTRUCT STRUCT
          DQ 0    ;+0 hDC
          DD 0    ;+8 fErase
    left  DD 0    ;+C  left   ┐
     top  DD 0    ;+10 top    │ RECT
   right  DD 0    ;+14 right  │
  bottom  DD 0    ;+18 bottom ┘
          DD 0    ;+1C fRestore
          DD 0    ;+20 fIncUpdate
   DB 32 DUP 0    ;+24 rgbReserved
          DD 0    ;+44 octets de bourrage pour obtenir une taille totale de 72 octets
ENDS
```

Dans la pratique, il a bien été constaté que le système écrivait dans la zone de comblement à + 44h lors de l'utilisation de PAINTSTRUCT dans certaines circonstances. Cela prouve l'importance de se conformer à ces règles (sinon vous pourriez être confronté à un écrasement inopiné des données immédiatement après la structure).

Notez que le *début* des structures doit également être aligné sur la limite naturelle du plus grand membre. Toutes les règles ci-dessus garantissent, par conséquent, que les QWords dans la structure sont toujours alignés selon une modularité de 4 octets.

6.4.2 Alignement et comblement automatiques des structures et de leurs membres

Comme nous l'avons vu, l'alignement correct des structures et de leurs membres est crucial pour le bon fonctionnement du code 64 bits. Malheureusement, les fichiers d'en-tête de Windows contenant les définitions de structure ne contiennent pas nécessairement les directives de comblement nécessaires pour atteindre un tel alignement.

C'est pourquoi, à partir de la version 0.56 (beta), GoAsm fait ce travail automatiquement pour vous en procédant comme suit :

a. GoAsm aligne toujours la structure elle-même sur la modularité de données correcte.
b. GoAsm effectue toujours un comblement, si nécessaire, pour garantir que les membres de la structure sont sur leur *frontière naturelle*. Ainsi, dans l'exemple de structure MSG ci-dessous, le comblement à +0Ch peut être laissé de côté car inséré automatiquement par GoAsm.
c. GoAsm ajoute toujours des octets de comblement à l'extrémité d'une structure de telle sorte que celle-ci se termine sur une *frontière naturelle*. Ainsi, dans l'exemple ci-dessous, le comblement à + 2Ch peut être laissé de côté car inséré automatiquement par GoAsm.
d. Les symboles créés lors de l'utilisation d'une structure sont automatiquement ajustés pour satisfaire l'alignement et le comblement qui sont appliqués.

```
MSG   DQ 0      ;+0h hWnd
      DD 0      ;+8h message
      DD 0      ;comblement pour garantir l'alignement de la suite
      DQ 0      ;+10h wParam
      DQ 0      ;+18h lParam
      DD 0      ;+20h time
      DD 0      ;+24h position de la souris (X)
      DD 0      ;+28h position de la souris (Y)
      DD 0      ;+2Ch comblement pour porter la taille globale de la structure à 48 octets
```

Vous pouvez constater par vous-même l'alignement et le comblement que GoAsm a ajoutés à votre code source si vous spécifiez /l dans la ligne de commande de GoAsm et que vous examinez le fichier listing résultant. Vous pouvez également faire le même constat avec le débogueur.

6.4.3 Structures - Tableau d'ensemble

- Si vous écrivez un code source destiné aux versions 32 et 64 bits de votre programme, il sera beaucoup plus facile d'utiliser l'assemblage conditionnel pour produire les structures correctes au moment

de l'assemblage. De même, au lieu de remplir les structures en utilisant des décalages, il est nettement préférable d'utiliser des noms de membres. En utilisant cette méthode, GoAsm calcule automatiquement le décalage approprié. Cette technique a été utilisée dans le fichier de démonstration Hello64World 3.

- Vous pouvez utiliser l'assemblage conditionnel pour changer de banque de structures en une seule fois. Celles-ci peuvent être contenues dans des fichiers d'inclusion comprenant respectivement des structures 64 bits et 32 bits.
- Dans la mesure où GoAsm aligne et comble automatiquement les structures, vous pouvez utiliser les définitions de structure 64 bits déjà disponibles dans les fichiers d'inclusion, ou vous pouvez constituer les vôtres à partir des fichiers d'en-tête Windows à l'aide de l'utilitaire **xlatHinc** de Wayne J. Radburn.

6.5 Choix des registres

- La chose principale à retenir est que tous les handles de Windows sont sur 64 bits de sorte que les API opèreront leur retour dans RAX plutôt que dans EAX.
- La même chose vaut pour les pointeurs de Windows. Par exemple, vous pouvez demander de la mémoire à Windows. L'adresse de cette mémoire sera retourné alors dans RAX et non dans EAX.

  Cela signifie donc que

```
ARG 4h,3000h,EDX,0
INVOKE VirtualAlloc     ; réserve et engage EDX octets de mémoire en lecture/écriture
MOV [EAX],66666666h     ; insère un nombre au début de cette mémoire
```

  est un mauvais codage 64 bits, alors que

```
ARG 4h,3000h,EDX,0
INVOKE VirtualAlloc     ; réserve et engage EDX octets de mémoire en lecture/écriture
MOV [RAX],66666666h     ; insère un nombre au début de cette mémoire
```

  est correct.

- Étant donné que tous les pointeurs vers des labels internes de données et de code sont sur 32 bits, il est possible, en théorie, d'utiliser les versions 32 bits des registres à usage général (EAX à ESP) pour tous ces pointeurs de telle sorte que, par exemple, vous pourriez utiliser MOV [ESI],AL au lieu de MOV [RSI],AL.

  Cependant, je déconseille cette pratique pour les cinq raisons suivantes :

  a. Cela signifie que vous devez garder une trace de ceux des pointeurs qui sont internes et de ceux qui sont externes. Vous devez prévoir que les externes seront sur 64 bits.
  b. Vous pouvez avoir besoin de deux ensembles de procédures qui sont souvent utilisés dans votre programme, l'une utilisant des pointeurs sous forme de registres 32 bits, l'autre utilisant des pointeurs sous forme de registres 64 bits.
  c. Les instructions de chaîne tels que LODSB, MOVSW, STOSD, CMPSQ et SCASB utilisent RSI et RDI dans un programme 64 bits en lieu et place de ESI et EDI. Et les préfixes de répétition REP, REPZ et REPNZ utilisent RCX au lieu de ECX.
  d. L'utilisation des versions 32 bits de ces instructions dans les codes de programme 64 bits produit un opcode plus grand que pour la version 64 bits. En effet, dans un programme 64 bits, MOV [RSI], AL est le codage par défaut et sa conversion en MOV [ESI], AL nécessite l'octet 67h d'override.
  e. Vous pouvez toujours utiliser le même code source pour produire un programme sur les plateformes 32 bits et 64 bits à condition que vous utilisiez uniquement les registres à usage général, RAX à RSP. En effet, lorsque vous utilisez le commutateur /x86 avec GoAsm, ces registres sont automatiquement considérés, en lieu et place, comme EAX à ESP.

  Vous pouvez automatiser les modifications nécessaires au code 32 bits pour passer au 64 bits en utilisant **AdaptAsm**.

- Si vous avez besoin d'utiliser les registres R8 à R15, rappelez-vous que R8 à R11 sont volatiles, c'est-à-dire qu'ils ne seront pas préservés par les API. Si vous utilisez les registres non-volatiles R12 à R15 à l'intérieur des procédures de fenêtre et de procédures callback vous devez vous assurer qu'ils sont res-

taurés après utilisation. Cela peut être fait en utilisant PUSH au début et POP à la fin de la procédure qui les utilise, ou en utilisant l'instruction USES.

- Lors du passage des paramètres à une API par INVOKE, vous devez garder présent à l'esprit que, dans la convention d'appel FASTCALL, les paramètres sont transmis à l'API successivement par les registres RCX, RDX, R8 et R9. Par conséquent, vous ne pouvez envisager de passer des paramètres dans des registres qui seront écrasés par GoAsm (vous obtiendriez un message d'erreur si vous essayiez de le faire).

  Par exemple, la formulation qui suit est erronée et affiche une erreur :

  ```
  INVOKE MessageBoxW,RDX,R8,R9,R10
  ```

  Elle est dans cette situation parce que, si elle était autorisée, elle se traduirait par :

  ```
  MOV R9,R10
  MOV R8,R9
  MOV RDX,R8
  MOV RCX,RDX
  ```

  où l'on peut voir en effet que le contenu des registres est écrasé avant d'être utilisé pour établir les paramètres.

  Une meilleure formulation pourrait être

  ```
  INVOKE MessageBoxW,R10,R9,R8,RDX
  ```

  Qui se traduirait par

  ```
  MOV R9,RDX
  MOV RDX,R9
  MOV RCX,R10
  ```

  Notez que Go Asm n'empêche pas le codage de `MOV R8,R8`

  Pour terminer, voici une formulation plus probante qui ne nécessite pas de code supplémentaire pour passer les paramètres car ceux-ci sont déjà dans les registres appropriés :

  ```
  INVOKE MessageBoxW,RCX,RDX,R8,R9
  ```

  Ce code se révèle donc très efficace.

Voir aussi **quelques conseils pour réduire la taille de votre code** qui a des implications supplémentaires dans votre choix des registres
et aussi **quelques pièges à éviter lors de la conversion du code source existant**.

6.6 Extension à zéro des résultats dans les registres 64 bits

Il convient d'être prudent lors de l'usage conjoint de registres 32 et 64 bits parce que le processeur peut modifier le contenu de la partie haute du registre 64 bits au terme d'une opération bien que rien ne le laisse présager. En effet, lors de l'écriture de résultats dans un registre 32 bits, le processeur va étendre le résultat sur l'ensemble des 64 bits du registre, avec pour conséquence de mettre inopinément à zéro les 32 bits de plus fort poids. Ainsi, par exemple :

```
MOV RAX,-1              ; charge RAX avec 0FFFFFFFF FFFFFFFFh
AND EAX,0F0F0F0Fh       ; (apparemment) agit seulement sur EAX
```

mais le processeur étend à zéro le résultat dans RAX. En d'autres termes, le DWord de poids fort de RAX sera mis à zéro dans cette opération alors que rien de tel n'a été explicitement demandé. D'où le résultat inattendu : RAX = 00000000 0F0F0F0Fh et non pas 0FFFFFFFF 0F0F0F0Fh comme on pouvait s'y attendre. Cela se produit indépendamment de la valeur du bit 31 de RAX et se distingue, en cela, du mécanisme d'extension de signe.

Un phénomène similaire se produit lors de l'utilisation d'autres instructions. En voici un exemple avec XOR :

```
MOV RAX,-1      ; charge RAX avec 0FFFFFFFF FFFFFFFFh
XOR EAX,EAX     ; (apparemment) met EAX à zéro
```

Contre toute attente, le résultat dans RAX est zéro.

Ce phénomène apparaît de la même manière avec l'instruction MOV :

```
MOV RCX,1111111111111111h
MOV ECX,88888888h
```

Le résultat, dans ce cas, est RCX=88888888h

Vous pouvez tirer avantage de l'extension à zéro de diverses manières. Quelques exemples en sont donnés dans la section **quelques conseils pour réduire la taille de votre code**. Considérez également l'exemple qui suit où la structure RECT (qui est de 4 DWords) contient des valeurs qui doivent être transmises à l'API *MoveWindow* en tant que QWords :

```
MOV RBX,ADDR RECT
MOV EAX,[EBX]        ; récupère x-pos
MOV ECX,[EBX+4]      ; récupère y-pos
MOV EDX,[EBX+8]      ; récupère right
SUB EDX,EAX          ; récupère width
MOV R8D,[EBX+0Ch]    ; récupère bottom
SUB R8D,ECX          ; récupère height
INVOKE MoveWindow,[hWnd],RAX,RCX,RDX,R8,0
```

Ici seuls des registres 32 bits sont utilisés pour extraire les informations de la structure RECT, mais nous savons que la partie haute des versions 64 bits de ces registres sera systématiquement mise à zéro.

Il est possible qu'il y ait une perte de rendement en se fondant sur le mécanisme d'extension à zéro. Une partie de la documentation suggère en effet que le processeur doit effectuer une opération supplémentaire pour mettre à zéro les bits de poids fort du registre.

6.7 Extension de signe des résultats dans les QWords

Vous pouvez légitimement vous interroger sur la différence qui pourrait exister entre les instructions suivantes :

```
MOV D[THING],12345678h
MOV Q[THING],12345678h
```

Ces codes produisent, contre toute attente, des résultats différents ! La version DWord, comme vous vous y attendez, place la valeur 12345678H dans le DWord du label THING. La version QWord fait la même chose, mais force également à zéro le DWord à THING+4. Techniquement, cette instruction pratique une *extension de signe* sur le résultat dans le QWord situé au label THING. Selon le même principe, si le bit le plus élevé du DWord de plus faible poids avait été à 1, la version QWord de l'instruction aurait rempli THING+4 avec 0FFFFFFFFh. En d'autres termes, les valeurs de 32 bits dans ces instructions sont considérées comme des nombres signés, et écrits en mémoire en conséquence.

```
MOV D[THING],12345678h     ;THING prend la valeur 12345678h (comme DWord)
MOV Q[THING],12345678h     ;THING prend la valeur 12345678h (comme QWord)
MOV D[THING],82345678h     ;THING prend la valeur 82345678h ie. -7DCBA988h (comme DWord)
MOV Q[THING],82345678h     ;THING prend la valeur 0FFFFFFFF 82345678h
                           ; c'est-à-dire -7DCBA988h (comme QWord)
```

La même chose se produit si vous utilisez un registre pour traiter la zone de données, par exemple :

```
MOV RSI,ADDR THING
MOV D[RSI],12345678h       ;THING prend la valeur 12345678h (comme DWord)
MOV Q[RSI],12345678h       ;THING prend la valeur 12345678h (comme QWord)
MOV Q[RSI],82345678h       ;THING prend la valeur 0FFFFFFFF 82345678h
                           ; c'est-à-dire -7DCBA988h (comme QWord)
```

Notez que vous ne pouvez pas mettre plus de 4 octets directement dans la mémoire en utilisant l'instruction MOV, même si vous utilisez le code 64 bits. La représentation qui suit fait donc apparaître une erreur :

```
MOV Q[THING],123456789ABCDEFh
```

Au lieu de cela, il convient de procéder comme suit pour atteindre ce résultat :

```
MOV RAX,123456789ABCDEFh
MOV [THING],RAX
```

6.8 Alignement automatique de la pile

Le pointeur de pile (RSP) doit être aligné sur une modularité de 16 octets lors d'un appel d'API. Avec certaines API cela n'a pas d'importance, mais avec d'autres, un mauvais alignement va provoquer une exception. Certaines vont gérer l'exception elles-mêmes et aligner la pile au besoin (au prix, cependant, d'une perte de performances). D'autres API (élaborées au minimum au début du x64) ne peuvent pas gérer l'ex-

ception et, à moins que vous n'exécutiez l'application sous le contrôle du débogueur, il en résultera une sortie de programme.

En raison de cette exigence, la documentation Win64 indique que vous ne pouvez appeler une API qu'à l'intérieur d'une trame de pile. En effet, il est supposé que c'est seulement à l'intérieur d'une trame de pile que l'on peut garantir qu'elle sera alignée correctement. Un appel hors trame de pile aura pour effet de désaligner la pile de 8 octets.

Cette exigence est très restrictive pour les programmeurs en assembleur, et occasionne de fortes migraines aux compilateurs. GoAsm résout à ce problème en proposant l'insertion d'un codage spécial avant et après chaque appel d'API (lorsque INVOKE est utilisé) pour veiller à ce que la pile soit toujours bien alignée au moment de l'appel. Cela soulage le programmeur en assembleur et signifie que :

- Les appels vers les API (en utilisant INVOKE) peuvent être faits partout dans votre code. Ils peuvent être construits à partir de procédures appelées par d'autres procédures sans se soucier du pointeur de pile.
- PUSH et POP peuvent être utilisés de la manière habituelle pour sauvegarder et restaurer les registres, les adresses de mémoire et les contenus de mémoire sans avoir à se soucier que cela met ou non la pile hors alignement.
- Vous pouvez utiliser le même code source à la fois pour les versions 32 bits et 64 bits de votre application (il n'y a aucune exigence pour l'alignement de la pile en 32 bits).

L'entête pour aligner la pile au moment de chaque appel d'API se traduit pas neuf octets supplémentaires par API, ce qui semble un faible prix à payer au regard du bénéfice escompté. A contrario, pour conserver une taille de code aussi réduite que possible, GoAsm exploite un certain nombre de possibilités pour optimiser le code, en particulier lors de l'insertion des paramètres. Voir la section **optimisations effectuées par GoAsm** pour plus de détails. Voir aussi **codage pour obtenir un alignement automatique de la pile**.

6.9 Utilisation du même code source en 32 et 64 bits

Le présent manuel décrit l'utilisation d'ARG et INVOKE dans la section traitant des **appels des API Windows en 32 bits et 64 bits** et l'utilisation de FRAME ... ENDF dans la section traitant des **trames de pile pour callback en 32 et 64 -bits**. Les constructions ARG/INVOKE et FRAME...ENDF de GoAsm permettent de gérer efficacement les changements de convention d'appel induits par la programmation 64 bits.

Si l'on réunit toutes ces considérations, ainsi que celles énoncées ci-dessus, il est parfaitement possible d'utiliser le même code source pour créer des exécutables pour les deux plates-formes 32 bits et 64 bits.

Pour mémoire, voici les règles qui doivent être suivies pour ce faire :

- Lors des appels d'API utiliser INVOKE dans votre code au lieu de CALL.
- Pour passer les paramètres aux API utiliser ARG dans votre code au lieu de PUSH ou alors, positionner les paramètres à la fin du INVOKE (immédiatement après le nom d'API).
- Utilisez FRAME... ENDF dans votre code lors de l'utilisation de données locales (LOCAL) ou de la collecte de paramètres envoyés à une procédure de fenêtre (ou à toute autre procédure callback similaire).
- Si vous souhaitez utiliser les nouveaux registres R8-R15, XMM8-XMM 15, ou les nouveaux registres adressables en 8, 16 ou 32 octets, assurez-vous qu'ils ne sont mentionnés que dans la portion du script source strictement réservée au code 64 bits lorsque vous utilisez l'assemblage conditionnel.
- Utilisez de préférence la forme 64 bits des registres à usage général (RAX, RBP, RBX, RCX, RDX, RDI, RSI et RER) pour les *pointeurs*. Lorsque GoAsm procèdera en effet à un assemblage en 32 bits, il réduira automatiquement ces registres à leur configuration 32 bits.
- Si vous avez utilisé PUSHFD et POPFD pour sauvegarder et restaurer les flags, changer pour PUSHF et POPF ou PUSH FLAGS et POP FLAGS.
- Veiller à ce que les structures, les tailles de données et les indicateurs de type soient corrects pour un usage alternatif 32/64 bits, si nécessaire en recourant à l'assemblage conditionnel.
- Utiliser le commutateur /x64 dans la ligne de commande pour créer un exécutable 64 bits, et /x86 pour créer un exécutable 32 bits.

Les outils "Go" vont faire le reste...

Notez que le commutateur /x86 ne doit pas être utilisé dans la ligne de commande pour un code source Win32 (le réserver uniquement à un code source 32/64-bit commutable).

Voir le fichier **Hello64World3** comme exemple de code source qui peut afficher un "Hello World" soit en Win32, soit en Win64.

6.10 Conversion d'un code 32 bits existant en code 64 bits

Compte tenu des considérations qui précèdent, voici ce que vous devez faire pour convertir un code source existant 32 bits en code source 64 bits.

- Changez tous les CALLs à destination d'API en INVOKE. Ne pas changer les CALLs autres que pour des APIs.
- Si vous avez utilisé PUSH pour envoyer des paramètres à une API dans votre source 32 bits, changer pour ARG. En revanche, ne pas utiliser ARG en lieu et place des PUSH qui ne seraient pas dédiés à cet usage.
- Changez tous les registres 32 bits à usage général utilisés comme pointeurs (qui sont donc entre crochets) par leurs homologues 64 bits (RAX, RBP, RBX, RCX, RDX, RDI, RSI et RSP). Cela vous permettra de conserver un code plus court, et de garantir que les pointeurs vers des *données externes* fonctionnent correctement. Pensez aussi à utiliser uniquement RSI, RDI et RCX avec les instructions de chaîne et les préfixes de répétition utilisant un format inférieur de ces registres. Voir, à ce sujet, le paragraphe **choix des registres**.
- Veillez à ce que les registres qui contiennent les handles du système et d'autres valeurs fournies par le système soient changés par leurs homologues 64 bits (RAX, RBP, RBX, RCX, RDX, RDI, RSI et RER).
- Ajustez l'usage des autres registres aux besoins. En général pour une autre utilisation, les registres existants fonctionneront parfaitement bien, mais ne mélangez pas l'utilisation de registres 32 bits et 64 bits en raison du mécanisme d'**extension à zéro** des résultats. Il n'y a pas besoin de modifier les PUSH et POP de registres. Ces changements sont effectués automatiquement par GoAsm parce que les opcodes sont les mêmes (par exemple PUSH EAX est considéré comme étant le même que PUSH RAX et réciproquement).
- Veillez à ce que les structures, la taille des données et les indicateurs de type soient corrects dans la perspective d'une utilisation 64 bits.
- Vérifiez que vos instructions JECXZ sont modifiées en JRCXZ, le cas échéant.
- Dans la mesure où un code 64 bits a tendance à être un peu plus grand que son homologue 32 bits, vous pourriez constater, lors du réassemblage de votre code en utilisant le commutateur /x64, que certains sauts courts doivent être réorganisés car devenus hors de portée.

Fort heureusement, le programme **AdaptAsm** permet de réaliser automatiquement une partie de ces travaux d'adaptation.

6.11 Utilisation de AdaptAsm.exe pour la conversion 64 bits

AdaptAsm est compris dans le pack GoAsm. J'ai écrit cet utilitaire à l'origine pour aider à convertir à la syntaxe GoAsm le code source créé sur d'autres assembleurs. Aujourd'hui, il est capable, outre les fonctions précitées, de convertir du code source 32 bits en code source 64 bits. Cela fonctionne aussi bien sur les scripts GoAsm que sur ceux d'autres assembleurs.

*Pour plus de détails sur les autres fonctionnalités de AdaptAsm voir le **paragraphe 4.17**.*

AdaptAsm s'utilise à partir de la simple ligne de commande suivante :

```
AdaptAsm [command line switches] inputfile[.ext]
```

Si aucune extension du nom du fichier d'entrée n'est spécifiée, l'extension .asm est présumée.
Si aucune extension du nom du fichier de sortie n'est spécifié, l'extension .adt est présumée.
Les commutateurs de la ligne de commande de AdaptAsm.exe sont :

```
/h   = affiche l'aide
/a   = adapte un fichier A386
/m   = adapte un fichier MASM
/n   = adapte un fichier NASM
/fo  = spécifie le chemin d'accès du fichier de sortie. Ex : AdaptAsm /fo GoAsm\adapted.asm
/l   = crée un fichier de sortie avec l'extension .log
/o   = supprimer l'alerte précédant toute écriture sur le fichier d'entrée
/x64 = adapte le fichier à la plateforme 64 bits
```

Ce que AdaptAsm fait lorsqu'il facilite l'adaptation d'un fichier en 64 bits en utilisant le commutateur /x64

Les CALLs en direction des API sont changés en INVOKE (les CALLs qui ne sont pas des appels d'API ne sont pas affectés).
AdaptAsm réalise cette conversion en consultant la liste des API dans les fichiers ".h.txt" dans le même dossier que AdaptAsm.exe. Voir la section sur les **fichiers "h.txt"** pour plus d'informations.
Cela fonctionne avec tous les types d'appels, même si la destination est entre crochets et même si elle dépend d'une définition (equate) ou d'un interrupteur, par exemple :

```
CALL ExitProcess        ; changé en INVOKE
CALL [ExitProcess]      ; changé en INVOKE
CALL INTERNAL_PROC      ; inchangé
CALL SendMessage        ; changé en INVOKE
CALL SendMessageA       ; changé en INVOKE
CALL SendMessageW       ; changé en INVOKE
CALL SendMessage##AW    ; changé en INVOKE
```

Change PUSH en ARG pour les paramètres envoyés à l'API. AdaptAsm y procède en comptant le nombre correct de paramètres à rebours du CALL et en le comparant avec le nombre correct de paramètres figurant dans la liste des API du fichier "h.txt" dans le même répertoire que AdaptAsm.exe. Voir le paragraphe sur les **fichiers "h.txt"** pour plus d'informations.
Voici quelques exemples simples :

```
PUSH EBX,0,1100h,[hMessTV]  ; PUSH est changé en ARG (et EBX changé en RBX)
CALL SendMessageA           ; CALL est changé en INVOKE
PUSH EBX,0                  ; PUSH est changé en ARG (et EBX changé en RBX)
PUSH 1100h                  ; PUSH est changé en ARG
PUSH [hMessTV]              ; PUSH est changé en ARG
CALL SendMessageA           ; CALL est changé en INVOKE
```

Vous avez peut-être préservé des registres au travers des appels d'API et ceux-ci ne sont pas affectés, par exemple :

```
PUSH EAX                    ; PUSH demeure inchangé (mais EAX est changé en RAX)
PUSH EBX,0,1100h,[hMessTV]  ; PUSH est changé en ARG (et EBX est changé en RBX)
CALL SendMessageA           ; CALL est changé en INVOKE
POP EAX                     ; POP demeure inchangé (mais EAX est changé en RAX)
```

Toutefois, si vous avez mélangé ces deux utilisations de PUSH, AdaptAsm montrera une erreur en changeant le PUSH en ARG et en notifiant le problème dans le fichier journal :

```
PUSH EAX,EBX,0,1100h,[hMessTV]   ; PUSH est changé en ARG (trop de paramètres)
CALL SendMessageA                ; CALL est changé en INVOKE
POP EAX                          ; restaure le registre EAX
```

Si AdaptAsm ne peut pas trouver tous les paramètres attendus, il affiche une erreur en changeant le CALL en INVOKE et en notifiant le problème dans le fichier journal. Par exemple :

```
CALL INTERNAL_PROC          ; inchangé
PUSH 0,1100h,[hMessTV]      ; PUSH est changé en ARG
CALL SendMessageA           ; CALL est changé en INVOKE (trop peu de paramètres)
```

Cela signifie que ce genre de chose qui pourrait être fait en 32 bits, sera affiché comme une erreur par AdaptAsm (et à juste titre puisque, dans l'assembleur 64 bits, chaque CALL doit immédiatement succéder aux paramètres) :

```
PUSH 0,EAX,14Eh,[hComboSev]     ; 14Eh = CB_SETCURSEL
PUSH 0,EAX,151h,[hComboSev]     ; 151h = CB_SETITEMDATA
CALL SendMessageA
CALL SendMessageA
```

Les registres à usage général 32 bits entre crochets sont remplacés par leurs homologues 64 bits afin qu'ils puissent être utilisés à la fois en assemblage 32 et 64 bits. Par exemple :

```
MOV EAX,[EAX+EBX]           ; changé en MOV EAX,[RAX+RBX]
```

```
MOV D[EBX*8+EBP], 8h          ; changé en MOV D[RBX*8+RBP], 8h
CALL [EBX]                    ; changé en CALL [RBX]
INVOKE ExitProcess,[EBX]      ; changé en INVOKE ExitProcess,[RBX]
PUSH [EBX]                    ; changé en PUSH [RBX] ou ARG [RBX]
POP [EBX]                     ; changé en POP [RBX]
```

Lorsqu'un pointeur est utilisé avec un registre à usage général 32 bits, le registre est changé par son équivalent 64 bits, comme, par exemple :

```
MOV EAX, ADDR THING           ; changé en MOV RAX, ADDR THING
CMP ESI, ADDR THING           ; changé en CMP RSI, ADDR THING
MOV EBP, OFFSET THING         ; changé en MOV RBP, OFFSET THING
LEA EAX, THING                ; changé en LEA RAX, THING
```

Bien que pas strictement nécessaire mais pour faire bonne mesure, les registres 32 bits à usage général après PUSH, POP et INVOKE sont changés en leur équivalent 64 bits comme, par exemple :

```
PUSH EAX, EBX                 ; changé en PUSH RAX, RBX
POP EBX, EAX                  ; changé en POP RBX, RAX
INVOKE ExitProcess, EBX       ; changé en INVOKE ExitProcess, RBX
```

Ce que AdaptAsm ne fait pas (et que vous devez faire à la main)

AdaptAsm ne peut pas décider à votre place quel registre à utiliser dans d'autres circonstances. Ce choix vous incombe au cas par cas en vous inspirant toutefois des règles et usages recensés dans la section **choix des registres** qui fournit quelques indications à ce sujet.

AdaptAsm ne garantit pas que les tailles des structures et des données sont correctes pour une utilisation en 64 bits, ni que les pointeurs vers les structures et les chaînes sont correctement alignés.

6.12 Les fichiers "h.txt" utilisés par AdaptAsm avec le commutateur /x64

Ces fichiers sont des fichiers texte contenant des listes d'API ainsi que le nombre de paramètres requis par chacune d'entre elles. AdaptAsm regarde à l'intérieur de son propre répertoire pour voir si des fichiers "h.txt" sont présents. Ces fichiers sont créés à partir de fichiers d'en-tête Microsoft en utilisant l'astucieux fichier javascript ApiParamCount.js, écrit par Leland M George de West Virginia, qui a bien voulu en autoriser l'usage dans le domaine public. Ce fichier js est livré avec AdaptAsm ainsi que quelques fichiers h.txt prêts à l'emploi contenant les API les plus couramment utilisées. Si votre programme utilise des API déclarées dans d'autres fichiers d'en-tête, vous pouvez constituer vos propres "h.txt" de ces fichiers en utilisant le fichier js. Il y a deux façons d'utiliser ce dernier :

- Soit glisser-déplacer le fichier d'en-tête sur le fichier js (un fichier h.txt sera constitué dans le même répertoire)
- Soit à partir de la ligne de commande en utilisant la commande suivante (par exemple):

  cscript ApiParamCount.js WinNT.h

  ou

  wscript ApiParamCount.js WinNT.h

  qui commande le démarrage du Windows Scripting Host qui gère les handles des fichiers JavaScript en dehors de la page Web environnements.

  Si vous devez télécharger le Windows Scripting Host vous pouvez l'obtenir à partir du site Microsoft.

Sinon, vous pouvez créer votre propre fichier h.txt ou modifier ceux qui existent déjà. Le format est le suivant :

- Le premier nom d'API doit commencer au début du fichier et les suivants systématiquement en début de ligne.
- De nouvelles lignes sont crées à l'aide d'un retour chariot (code Ascii décimal 13) suivi d'un saut de ligne (code Ascii décimal 10).
- Une virgule suit immédiatement le nom de l'API.
- Le nombre de paramètres requis par l'API suit immédiatement la virgule et est exprimé par un chiffre décimal en format Ascii. Si l'API ne prend pas de paramètres le nombre est égal à zéro.

6.13 Commutation utilisant x64 et x86 en assemblage conditionnel

En plus de pouvoir déclencher l'assemblage 64 ou 32 bits en spécifiant respectivement /x64 ou /x86 dans la ligne de commande de GoAsm, cet assembleur permet même à ces directives d'être testées dans l'assemblage conditionnel. Ainsi, par exemple, vous pouvez écrire une procédure de fenêtre distincte pour chaque plateforme et déterminer leur activation au moyen du test suivant :

```
WndProcTable:
#if X64
MOV EAX,ADDR MESSAGES     ; met en EAX la liste des messages à traiter
CALL GENERAL_WNDPROC64    ; appel du gestionnaire de message générique (version 64 bits)
#else
MOV EDX,ADDR MESSAGES     ; met en EDX la liste des messages à traiter
CALL GENERAL_WNDPROC      ; appel du gestionnaire de message générique (version 32 bits)
#endif
RET
```

Notez que les mots "x64" et "x86" ne sont pas sensibles à la casse.

Voici un autre exemple où l'on commute des fichiers d'inclusion porteurs de structures :

```
#if X64
#include structures64.inc
#else
#include structures32.inc
#endif
```

6.14 Quelques pièges à éviter lors de la conversion du code source existant

- **Vous oubliez que les paramètres d'API sont systématiquement des QWords.**
  Votre code source existant 32 bits aura été écrit selon l'hypothèse correcte que chaque paramètre est un DWord. Par exemple :

  ```
  ARG 4000h,[SYSTEM_INFO+4h],[MEMORY_END]
  INVOKE VirtualFree        ; libère une page de mémoire
  ```

  En 32 bits, ce codage est correct parce qu'il y a un DWord à [SYSTEM_INFO +4h] (le dword contient ici la taille des pages mémoire système (celle-ci supposant que la structure a été remplie au moyen d'un appel à l'API *GetSystemInfo*).
  En 64 bits, ce codage est mauvais parce que la valeur à +4h est toujours un DWord, mais vous envoyez maintenant un QWord à VirtualFree et pas seulement un DWord. Cela devrait donc être codé comme suit en remplacement :

  ```
  XOR RAX,RAX               ; RAX = 0
  MOV EAX,[SYSTEM_INFO+4h] ; récup. de la taille de page dans les 32 bits de poids faible de RAX
  ARG 4000h,RAX,[MEMORY_END]
  INVOKE VirtualFree        ; libère une page de mémoire
  ```

  Notez que, dans la pratique, MOV EAX met à zéro la partie supérieure de RAX de sorte que vous pouvez supprimer la première ligne de cet exemple !

  Un problème similaire se pose lors de l'interrogation du système et de la réception des informations dans les données. Votre code 32 bits existant peut bien ressembler à ceci :

  ```
  ARG 0,ADDR SIZEOF_WORKAREA,0,48  ;48 = SPI_GETWORKAREA (excluding tray)
  INVOKE SystemParametersInfoA     ; récupère la taille de la zone de travail dans SIZEOF_WORKAREA
  ```

  Ici, l'appel place une valeur de 32 bits dans le dword SIZEOF_WORKAREA, ce qui est correct. Cependant l'assemblage puis l'exécution du même code sur une plateforme 64 bits a pour effet d'écraser le DWord suivant en mémoire (un QWord est envoyé, pas un DWord). D'où la nécessité d'étendre SIZEOF_WORKAREA au format QWord.

- **Vous oubliez que tous les CALLs sont maintenant avec des valeurs 64 bits.**

  Cela peut arriver facilement lors de l'utilisation de tables permettant, par exemple, d'effectuer les CALL en direction de labels de code dûment répertoriés dans ces tables. Considérons le cas d'un tableau simple de labels suivant :

  ```
  DATA
  ```

```
Table DD CODELABEL,2h
CODE
CALL [Table]
```

ou

```
DATA
Table DD CODELABEL,2h
CODE
MOV RSI,ADDR Table
CALL [RSI]
```

Ces CALLs attendent une adresse de 64 bits qui va être constituée, contre toute attente, de l'adresse du label CODELABEL dans le DWord de poids faible et de la valeur 2 dans le Dword de poids fort. D'où une erreur plus que probable au moment de l'exécution. Cela provient évidemment de la table qui déclare des DWords au lieu de QWords. La solution pour les appels internes est donc de coder comme suit :

```
DATA
Table DQ CODELABEL,2h
CODE
CALL [Table]
```

ou

```
DATA
Table DD CODELABEL,2h
CODE
MOV RSI,ADDR Table
XOR RAX,RAX
MOV EAX,[RSI]
CALL RAX
```

Le code qui précède garantit que le DWord de poids fort de l'adresse 64 bits est bien égal à zéro. Cela fonctionne parce que tous les pointeurs vers des labels de donnée et de code interne sont à 32 bits.

- **Vous oubliez que tous les handles de Windows sont maintenant des valeurs 64 bits.**

  Dans Win64, les handles de système sont étendus à 64 bits de sorte qu'il est imprudent de supposer qu'ils pourraient toujours tenir dans 32 bits. Cela signifie donc que :

```
ARG 32512               ; IDC_ARROW (curseur classique en forme de flèche)
INVOKE LoadCursorA,0    ; récupère dans EAX le handle de ce curseur
MOV [WNDCLASS+28h],EAX  ; et le communique à WNDCLASS
```

  procède d'un mauvais codage 64 bits, alors que :

```
ARG 32512               ; IDC_ARROW (curseur classique en forme de flèche)
INVOKE LoadCursorA,0    ; récupère dans RAX le handle de ce curseur
MOV [WNDCLASS+28h],RAX  ; et le communique à WNDCLASS
```

  est correct.

- **Vous oubliez que tous les POPs sont maintenant des QWords.**

  Votre code source existant 32 bits peut effectuer des POP de Dword en mémoire. Par exemple :

```
DRAW_RECTANGLE:
PUSH [RECT],[RECT+4]  ; sauvegarde de la largeur et de la longueur du rectangle
                      ; code pour ajuster le rectangle
                      ; puis le dessiner
POP [RECT+4],[RECT]   ; restauration de la longueur et de la largeur
                      ; du rectangle pour un usage ultérieur
RET
```

  En 64 bits, une structure RECT est toujours de 4 DWords comme elle l'était en 32 bits. Toutefois, le deuxième POP dans le code ci-dessus effacera le deuxième DWord dans la structure parce que le POP *agit en fait sur 64 bits*, et non pas 32 bits.
  Il en résulte que le codage correct pour 64 bits doit s'écrire :

```
DRAW_RECTANGLE:
PUSH [RECT],[RECT+4]  ; sauvegarde de la largeur et de la longueur du rectangle
```

```
                    ; code pour ajuster le rectangle puis le dessiner
POP RAX             ; restauration de la longueur du rectangle pour un usage ultérieur
MOV [RECT+4],EAX    ; insertion d'un dword seulement
POP RAX             ; restauration de la largeur du rectangle pour un usage ultérieur
MOV [RECT],EAX      ; insertion d'un dword seulement
RET
```

6.15 Assemblage et édition de liens pour produire un exécutable

Pour créer un fichier d'objet 64 bits avec GoAsm, utiliser la ligne de commande :

```
GoAsm /x64 filename
```

où *filename* est le nom de votre fichier asm écrit soit comme un fichier source 64 bits soit comme un fichier source commutable 32/64 bits. Utilisez /x86 au lieu de /x64 lorsque vous assemblez un fichier source commutable 32/64 pour en faire une version 32 bits.

Le fichier objet créé par GoAsm peut être envoyé à GoLink ou à un autre linker de la manière habituelle.

GoLink détecte automatiquement si le fichier objet est en 32 ou 64 bits et crée, selon le cas, le type de fichier exécutable approprié.

Vous ne pouvez pas mélanger des fichiers objets 32 bits et 64 bits. GoLink affichera une erreur pour toute tentative faite en ce sens.

Vous ne devez pas nécessairement faire des exécutables 64 bits sur une machine 64 bits. En effet, les noms de DLL donnés à GoLink disent simplement au linker que les DLL contiennent les API utilisées par l'application, lesquelles API tendent à être identiques entre les deux plates-formes. Si votre application appelle des API spécifiques au système 64 bits cependant, cela ne fonctionne pas.

6.16 Quelques optimisations et améliorations apportées par GoAsm

GoAsm vise toujours à produire, à partir de votre script source, le code le plus compact possible. Dans le cas du 64 bits, GoAsm n'a pas encore pris en considération toutes les possibilités d'optimiser le code. En effet, il subsiste encore quelques inconnues, notamment les effets sur les performances du code optimisé sur x64.

Les optimisations et améliorations effectuées automatiquement par GoAsm sont listées ici pour vous aider lorsque vous regarderez le code produit par GoAsm dans le débogueur.

Optimisations et améliorations apportées par GoAsm dans tout le code

Aucune de celles-ci n'affecte les flags ou n'affecte négativement les performances.

- **MOV *64-bit register*, ADDR label** est changé en **LEA *64-bit register*, label**. Cela permet d'économiser 5 opcodes. Une différence importante entre les deux instructions est que la version MOV utilise une relocalisation absolue (donc en théorie, il faut laisser un espace pour une valeur de 64 bits à insérer par le linker). L'instruction LEA utilise en revanche l'**adressage relatif RIP** et donc peut faire le même travail en se contentant d'un espace de 32 bits pour l'adresse relative.

- **PUSH** ou **ARG ADDR Non_Local_Label** utilise également LEA ***ainsi que le registre R11*** de la manière suivante :

  ```
  LEA R11,ADDR Non_Local_Label
  PUSH R11
  ```

  *Voir l'**explication de cette méthode**. Notez que cela aura également lieu avec INVOKE lors de la mise en pile des arguments avec ADDR, **qui comprend également l'utilisation de pointeurs vers une chaîne ou une donnée brute** (ex. 'Hello' ou <'H','i',0>).*

  Ceci affecte les flags.

- **PUSH** ou **ARG ADDR Local_Label** sont codés de la manière suivante :

  ```
  PUSH RBP
  ADD D[RSP],+/-Displacement
  ```

Optimisations et améliorations additionnelles uniquement avec INVOKE

Celles-ci peuvent affecter les flags qui n'ont pas d'importance lors de l'appel d'APIs. Celles qui comptent sur l'**extension à zéro** peuvent nécessiter une autre opération de la part du processeur, mais il est supposé que cela n'a pas d'importance lors de l'appel d'une API. Il est plus important de réduire la taille du code.

- Un **paramètre sous forme d'un registre contenant zéro** peut avantageusement être initié comme tel par l'opération XOR *r32,r32* en lieu et place de MOV *r32*,0. Cela représente une économie comprise entre 7 et 8 octets.
- Un **paramètre sous forme de registre contenant un nombre** (valeur «immédiate») qui peut tenir en 32 bits gagne à être modifié pour utiliser un registre de 32 bits et économiser ainsi entre 1 et 5 octets selon le registre et le nombre.
- Un **paramètre sous forme de registre contenant –1** est obtenu en utilisant OR *registre 64 bits*,–1 d'où un gain de 6 octets.
- Si le paramètre est **déjà dans le registre correct** aucun autre code n'est émis parce qu'il est pas nécessaire.
- Le codage pour réaliser l'**alignement automatique de la pile** et mettre cette dernière en conformité avec la **convention d'appel FASTCALL** est le suivant (avec une instruction introduisant le nombre de paramètres *x* comme facteur déterminant) :

```
PUSH RSP              ; sauvegarde valeur courante de RSP en pile
PUSH [RSP]            ; on rajoute une copie de cette valeur en pile
AND SPL,0F0h          ; on ajuste RSP pour aligner la pile sur 1 para si ce n'est pas déjà fait
                      ;
                      ; paramètres traités ici
                      ;
SUB RSP,20h           ; on décale RSP pour créer un espace libre de 20h octets sur la pile
CALL TheAPI           ; appel de l'API
LEA RSP,[RSP+xxh]     ; rétablit RSP à la place correcte pour la suite
POP RSP               ; on rétablit RSP dans sa valeur d'origine
```

ou

```
PUSH RSP              ; sauvegarde valeur courante de RSP en pile
PUSH [RSP]            ; on rajoute une copie de cette valeur en pile
OR SPL,8h             ; on ajuste RSP pour aligner la pile sur 1 para si ce n'est pas déjà fait
                      ;
                      ; paramètres traités ici
                      ;
SUB RSP,20h           ; on ajuste RSP pour créer un espace libre de 20h octets sur la pile
CALL TheAPI           ; appel de l'API
LEA RSP,[RSP+xxh]     ; rétablit RSP à la place correcte pour la suite
POP RSP               ; on rétablit RSP dans sa valeur d'origine
```

6.17 Quelques conseils pour réduire la taille de votre code

Notez qu'il est possible certaines de ces optimisations nuisent à la performance.

- L'utilisation de registres 64 bits (RAX à RSP) en tant que pointeurs vers la mémoire (par exemple, l'instruction `MOV [RSI],AL`) permet de gagner 1 octet sur la longueur de l'opcode par rapport à la variante utilisant des registres 32 bits (par exemple `MOV [ESI],AL`). En effet, dans ces instructions, un octet d'override 67h est nécessaire pour la version 32 bits.
- Il en va différemment lorsque vous assignez des valeurs immédiates (nombres) à des registres. Dans ce cas, l'utilisation de registres élargis (RAX à RSP), de registres étendus (R8 à R15) ou de l'une des nouvelles méthodes adressage des registres, ajoutent au moins un octet à chaque instruction. Par exemple, `MOV RAX,23456h` occupe de 2 octets de plus que `MOV EAX,23456h`. Le contraste est encore plus saisissant en utilisant de grands nombres, et supérieurs en tout état de cause à 7FFFFFFFh, parce ceux-ci doivent être codés comme des nombres 64 bits si vous utilisez un registre 64 bits. Ainsi, par exemple, le codage de `MOV RAX,80234560h` occupe 5 octets de plus que celui de `MOV EAX,80234560h`. Si le nombre que vous souhaitez déplacer tient dans un octet, de plus grandes économies peuvent être

réalisées. Par exemple, le codage de `MOV AL, 88h` n'occupe que 2 octets, alors que celui de `MOV RAX, 88h` atteint 10 octets.

- DEC et INC (avec un registre) utilisent maintenant deux opcodes, alors que dans les processeurs 86, ces instructions se contentaient d'un seul. Cela étant, Intel recommande aujourd'hui de leur préférer respectivement SUB *registre*, 1 et ADD *registre*, 1 afin d'uniformiser le traitement des flags.
  En effet, les instructions INC et DEC ne modifient seulement qu'une partie des bits dans le registre des flags comparativement aux instructions ADD et SUB dont elles ne sont pourtant qu'un cas particulier. Selon Intel,[2] il peut même s'avérer particulièrement problématique dans certains cas d'en poursuivre l'usage.
- En programmation 64 bits, l'instruction `LEA register, Label` est de 5 opcodes plus courte que l'instruction `MOV register, ADDR Label` tout en atteignant le même résultat. Dans un code source GoAsm, cependant, vous pouvez utiliser l'une ou l'autre de ces deux formes puisque GoAsm sélectionne automatiquement la plus courte.
- `PUSH ADDR THING` se code sur 9 octets, alors que si vous utilisez, en remplacement, `LEA RAX, THING` suivi de `PUSH RAX`, le codage global se réduit à 8 octets mais présente l'inconvénient de modifier le contenu du registre de RAX.
- Mettre à zéro un registre en utilisant XOR. L'instruction `XOR RAX, RAX` occupe 3 octets, alors que l'instruction `MOV RAX, 0` occupe 10 octets parce qu'elle met en œuvre une valeur immédiate de 64 bits (nombre). Notez cependant que XOR affecte les flags, ce qui n'est pas le cas de MOV.
- L'instruction `XOR EAX, EAX` est encore plus courte avec seulement 2 octets et met à zéro l'ensemble du registre RAX en vertu du principe d'**extension à zéro** automatique des résultats.
- Une bonne façon de porter la valeur –1 dans un registre, est d'utiliser l'instruction `OR registre, -1` qui, dans le cas d'un registre 64 bits, occupe 4 octets, soit un gain de 6 octets par rapport à `MOV, -1`. Cependant, contrairement à MOV, l'instruction OR affecte les flags.
- Une comparaison dans la plage –80h à +7Fh n'occupe que 4 octets (par exemple, `CMP RDX, -80h` à `CMP RDX, 7Fh`), mais en dehors de cette plage, le codage s'effectue sur 7 octets. Par exemple, `CMP RDX, 80h` se code sur 7 octets.
- Vous pouvez toujours utiliser LEA pour réaliser certaines opérations arithmétiques intra-registre, par exemple `LEA RAX, [RAX+RAX*2]` qui multiplie RAX par trois. Ce code n'occupe que 4 octets.

Voir également les quelques conseils et astuces de programmation prodigués dans l'**annexe K** de ce document.

6.18 Références et liens concernant la programmation 64 bits

Information about the AMD64
AMD information for developers
Intel – Introduction to x64 Assembly
Intel – 64-bit programming

Newsgroups et forums :
64-bit assembler forum
Start64 forum
Forum developpez.com

[2] Intel® 64 and IA-32 Architectures – Optimization Reference Manual – §3.5.1.1 Use of the INC and DEC Instructions

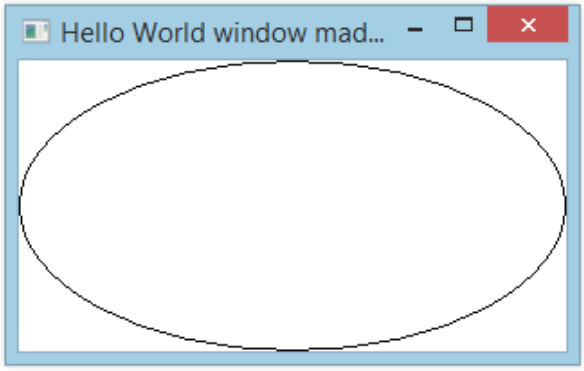
Annexe A
Exemples de programmes en assembleur GoAsm

Programme HelloWorld1.asm

Ce programme est dit "de console", c'est-à-dire qu'il est prévu pour fonctionner sous l'invite de commande MS-DOS. Son action se borne à afficher le message “Hello World (from GoAsm)” ainsi que le montre la copie d'écran ci-dessous (texte sur fond jaune et flèche rouge) :


```
;-------------------------------------------------------------------
; HelloWorld1 - copyright Jeremy Gordon 2002
; SIMPLE "HELLO WORLD" WINDOWS CONSOLE PROGRAM - for GoAsm
;
; Assemblage & Édition de liens :
;    GoAsm HelloWorld1 (produit un fichier PE COFF)
;    GoLink /console [-debug coff] helloworld1.obj kernel32.dll
; -debug coff n'est utilisé que s'il est souhaitable d'analyser le
; programme dans le débogueur
;
; Notez que les API GetStdHandle et WriteFile relèvent de kernel32.dll
;-------------------------------------------------------------------
;
DATA SECTION
;
RCKEEP DD 0     ; variable à usage général
;
CODE SECTION
;
START:
PUSH -11D                ; STD_OUTPUT_HANDLE
CALL GetStdHandle        ; récupère en EAX le handle du buffer d'écran actif
PUSH 0,ADDR RCKEEP       ; RCKEEP va récupérer la sortie d'API
PUSH 24D,'Hello World (from GoAsm)'     ; 24 = longueur de la chaîne
PUSH EAX                 ; handle du buffer d'écran actif
CALL WriteFile
XOR EAX,EAX              ; retour de la valeur zéro
RET
```

Programme HelloWorld2.asm



Le programme HelloWorld2.asm, écrit pour Windows 32 bits, permet de dessiner une ellipse dans un rectangle. La figure ci-contre montre le résultat obtenu sur l'écran.

On trouvera, dans la même annexe, trois variantes de ce programme :

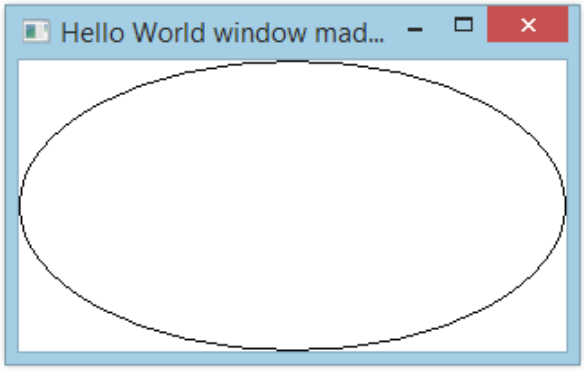
- **HelloWorld3.asm**, toujours en 32 bits, qui est une version plus structurée du même programme faisant usage de structures formelles, de trames automatiques de pile FRAME..ENDF, de USEDATA et USES, INVOKE, de définitions et de labels de nom réutilisables.
- **Hello64World2.asm** qui est la version 64 bits de **HelloWorld2.asm**.
- **Hello64World3.asm** qui est une version spéciale destinée à être compilée indifféremment pur les plate-formes 32 et 64 bits.

```
;-------------------------------------------------------------------
; HelloWorld2 - copyright Jeremy Gordon 2002
;
; SIMPLE "HELLO WORLD" WINDOWS GDI PROGRAM - for GoAsm
;
; Assemblage & Édition de liens :
;       GoAsm HelloWorld2 (produit un fichier PE COFF)
;       GoLink [-debug coff] HelloWorld2.obj user32.dll kernel32.dll gdi32.dll
; -debug coff n'est utilisé que s'il est souhaitable d'analyser le
; programme dans le débogueur
;-------------------------------------------------------------------
;
DATA SECTION
;
hInst DD 0      ; mémorise le handle du process lui-même
hDC   DD 0      ; mémorise le handle du device context
PAINTSTRUCT DD 16 DUP 0  ; structure accueillant les infos de Windows sur WM_PAINT
MSG DD 7 DUP 0  ; structure pour accueillir les messages de Windows comme suit:
        ; hwnd, +4=message, +8=wParam, +C=lParam, +10h=time, +14h/18=pt
WNDCLASS DD 10D DUP 0 ; structure pour les données manipulées par RegisterClass:
        ;  +0 style de classe de fenêtre (CS_)
        ;  +4 pointeur vers la procédure de fenêtre
        ;  +8 nb d'octets supplémentaires à allouer après la structure
        ;  +C nb d'octets supplémentaires à allouer après window instance
        ; +10 handle de l'instance de cette classe de fenêtre
        ; +14 handle de l'icône de classe
        ; +18 handle du curseur de classe
        ; +1C identifie la brosse d'arrière-plan de la classe
        ; +20 pointeur vers le nom de ressource pour le menu de la classe
        ; +24 pointeur vers la chaîne correspondant au nom de la classe de fenêtre
;******************** Table de messages Windows
;   (dans un programme plus élaboré, il y aurait beaucoup plus de messages à traiter)
MESSAGES DD (ENDOF_MESSAGES-$-4)/8      ;=nombre de messages à traiter
         DD  1h,CREATE,2h,DESTROY,0Fh,PAINT
ENDOF_MESSAGES: ; label utilisé pour déterminer le nombre de messages
;*****************************************
;
WINDOW_CLASSNAME DB 'WC',0  ; chaîne destinée à contenir le nom de la classe de fenêtre
;
```

```
;-----------------------------------------------------------------
CODE SECTION
;
INITIALISE_WNDCLASS:     ; prépare l'enregistrement de la classe de fenêtre
MOV EBX,ADDR WNDCLASS
MOV EAX,9                ; compteur = 10 paramètres à RAZ
L1:
MOV D[EBX+EAX*4],0       ; RAZ de chaque paramètre
DEC EAX
JNS L1
;**** additionner des infos à la classe de fenêtre pour toutes les fenêtres du programme ..
MOV EAX,[hInst]          ; donner le handle au process
MOV [EBX+10h],EAX        ; le constituer en tant que classe de fenêtre
PUSH 32512               ; curseur commun IDC_ARROW
PUSH 0
CALL LoadCursorA         ; on récupère dans EAX le handle de la flèche de curseur
MOV [EBX+18h],EAX        ; et on le met dans WNDCLASS
MOV D[EBX+1Ch],6D        ; initialise la couleur de fond dans COLOR_WINDOW+1
RET
;
;******************
CREATE:                  ; l'un des quelques messages traités par ce prog
XOR EAX,EAX              ; retourne zéro pour constituer la fenêtre
RET
;
DESTROY:                 ; l'un des quelques messages traités par ce prog
PUSH 0
CALL PostQuitMessage     ; sortie via la boucle de message
STC                      ; aller à DefWindowProc en plus
RET
;
; Le process qui suit dessine une ellipse dans le rectangle fourni par Windows
; sur le message WM_PAINT. Ce rectangle est la zone qui a besoin d'être mise à jour, par exemple
; au moment d'un redimensionnement ou si la fenêtre est découverte par une autre.
;
PAINT:
MOV EBX,ADDR PAINTSTRUCT
PUSH EBX,[EBP+8h]        ; EBP+8h=hwnd
CALL BeginPaint          ; obtient le contexte de périphérique à utiliser, initialise la peinture
MOV [hDC],EAX
PUSH [EBX+14h],[EBX+10h] ; extrémité basse, droite du rectangle
PUSH [EBX+0Ch],[EBX+8h]  ; extrémité haute, gauche du rectangle
PUSH [hDC]
CALL Ellipse             ; trace une ellipse dans le rectangle actualisé
PUSH EBX,[EBP+8h]        ; EBP+8h=hwnd
CALL EndPaint
XOR EAX,EAX
RET
;
;********** Ceci est une procédure de fenêtre générale qui, dans un programme
;********** ordinaire, traite tous les messages envoyés à la fenêtre
GENERAL_WNDPROC:         ; EAX peut être utilisé pour transmettre des informations au CALL
PUSH EBP                 ; utilise EBP pour éviter EAX, lequel peut contenir des informations
MOV EBP,[ESP+10h]        ; uMsg
MOV ECX,[EDX]            ; récupère le nombre de messages à traiter
ADD EDX,4                ; saute le dword contenant le nombre de messages
L2:
DEC ECX
JS >L3
CMP [EDX+ECX*8],EBP      ; voir si c'est le message correct
JNZ L2                   ; non
```

```
MOV EBP,ESP
PUSH ESP,EBX,EDI,ESI      ; on sauvegarde les registres tel que requis par Windows
ADD EBP,4                 ; on saute le dword contenant le code du message
; dès lors : [EBP+8]=hwnd, [EBP+0Ch]=uMsg, [EBP+10h]=wParam, [EBP+14h]=lParam,
CALL [EDX+ECX*8+4]        ; on appelle la procédure correspondant au message
POP ESI,EDI,EBX,ESP
JNC >L4                   ; nc=retourne la valeur en EAX - on n'appelle pas DefWindowProc
L3:
PUSH [ESP+18h],[ESP+18h],[ESP+18h],[ESP+18h]    ; permet le changement de ESP
CALL DefWindowProcA
L4:
POP EBP
RET
;
;****************** Ceci est la procédure de fenêtre courante
WndProcTable:
MOV EDX,ADDR MESSAGES     ; EDX pointe la liste des messages à traiter
CALL GENERAL_WNDPROC      ; appel du gestionnaire de message générique
RET 10h                   ; restauration de la pile comme requis par l'appelant
;
;*********************************************************************
START:
PUSH 0
CALL GetModuleHandleA     ; récupération du handle du process
MOV [hInst],EAX           ; enregistrement de ce handle au label de donnée hInst
CALL INITIALISE_WNDCLASS ; prépare l'enregistrement de la classe de fenêtre
;********** ajoutons maintenant des paramètres spécifiques à la fenêtre à construire
MOV D[EBX],1h+2h+40h      ;CS_VREDRAW+CS_HREDRAW+CS_CLASSDC (style de classe de fenêtre)
MOV D[EBX+4],ADDR WndProcTable  ; procédure de fenêtre
MOV D[EBX+24h],ADDR WINDOW_CLASSNAME    ; nom de la classe de fenêtre
PUSH EBX                  ; adresse structure avec données de classe de fenêtre
CALL RegisterClassA       ; enregistre la classe de fenêtre (window class)
PUSH 0,[hInst],0,0        ; propriétaire = bureau
PUSH 200D                 ; hauteur
PUSH 320D                 ; largeur
PUSH 50D,50D              ; position y, puis x
PUSH 90000000h  +0C00000h+40000h +80000h  +20000h     +10000h  ; style de fenêtre
;(POPUP+VISIBLE)+CAPTION+SIZEBOX+SYSMENU+MINIMIZEBOX+MAXIMIZEBOX
PUSH 'Hello World window made by GoAsm'  ; titre de la fenêtre
PUSH ADDR WINDOW_CLASSNAME      ; nom de la classe de fenêtre
PUSH 0                    ; style de fenêtre étendu
CALL CreateWindowExA      ; construit la fenêtre, retourne le handle dans EAX
;********************** on entre maintenant dans la boucle de message principale
L1:
PUSH 0,0,0
PUSH ADDR MSG
CALL GetMessageA          ; attente d'un message de Windows
OR EAX,EAX                ; on regarde si c'est WM_QUIT
JZ >L2                    ; oui, alors on va à L2
PUSH ADDR MSG
CALL TranslateMessage     ; dans le cas contraire, conversion du message en caractères si nécessaire
PUSH ADDR MSG
CALL DispatchMessageA     ; et envoi du message à la procédure de fenêtre
JMP L1                    ; après le message traité, boucle de retour pour un prochain
L2:
PUSH [hInst],ADDR WINDOW_CLASSNAME      ; le message était WM_QUIT
CALL UnregisterClassA     ; on fait en sorte que la classe soit supprimée
PUSH [MSG+8h]             ; sortie du programme (on envoie le contenu de wParam)
CALL ExitProcess          ; retour à Windows de la manière appropriée
```

Programme HelloWorld3.asm



Le programme HelloWorld3.asm, écrit pour Windows 32 bits, permet de dessiner une ellipse dans un rectangle. La figure ci-contre montre le résultat obtenu sur l'écran. **HelloWorld3.asm** est une version plus structurée de HelloWorld2.asm faisant usage de structures formelles, de trames automatiques de pile FRAME..ENDF, de USEDATA et USES, INVOKE, de définitions et de labels de nom réutilisables.

On trouvera, dans la même annexe, trois variantes de ce programme :

- **HelloWorld2.asm**, toujours en 32 bits, qui est une version moins structurée du même programme.
- **Hello64World2.asm** qui est la version 64 bits de **HelloWorld2.asm**.
- **Hello64World3.asm** qui est une version spéciale destinée à être compilée indifféremment pour les plateformes 32 et 64 bits.

```
;-----------------------------------------------------------------
;  HelloWorld3 - copyright Jeremy Gordon 2002
;
; "HELLO WORLD" WINDOWS GDI PROGRAM - programme de démonstration faisant
;  usage de structures formelles, de trames automatiques de pile FRAME..ENDF,
;  USEDATA et USES, INVOKE, de définitions et labels de nom réutilisables
;  dans la syntaxe GoAsm

;  Assemblage & Édition de liens :
;    GoAsm HelloWorld3 (produisant un fichier PE COFF)
;    GoLink [-debug coff] HelloWorld3.obj user32.dll kernel32.dll gdi32.dll
; -debug coff n'est utilisé que s'il est souhaitable d'analyser le
; programme dans le débogueur
;-----------------------------------------------------------------
;
; La première permet de spécifier un couple de structures formalisées
;
WNDCLASSEX STRUCT
cbSize          DD 30h  ;+0  taille de la structure (WNDCLASSEX seulement)
style           DD ?    ;+4  style de classe de fenêtre (CS_)
lpfnWndProc     DD ?    ;+8  pointeur de la procédure de fenêtre
cbClsExtra      DD ?    ;+C  nb d'octets supplémentaires à allouer après la structure
cbWndExtra      DD ?    ;+10 nb d'octets supplémentaires à allouer après l'instance de fenêtre
hInstance       DD ?    ;+14 handle de l'instance de cette classe de fenêtre
hIcon           DD ?    ;+18 handle de l'icône de classe
hCursor         DD ?    ;+1C handle du curseur de classe
hbrBackground   DD ?    ;+20 identifie la brosse d'arrière-plan de la classe
lpszMenuName    DD ?    ;+24 pointeur du nom de ressource pour la classe de menu
lpszClassName   DD ?    ;+28 pointeur vers la chaîne du nom de classe de fenêtre
hIconSm         DD ?    ;+2C handle de la petite icône
ENDS
;
RECT STRUCT
     left    DD ?
     top     DD ?
     right   DD ?
     bottom  DD ?
ENDS
;
```

```
PAINTSTRUCT STRUCT
hdc             DD ?
fErase          DD ?
rcPaint         RECT
fRestore        DD ?
fIncUpdate      DD ?
rgbReserved     DB 32 DUP ?
ENDS
;
; et maintenant quelques définitions pour une utilisation ultérieure :
;*** d'abord quelques styles de classe de fenêtre
CS_VREDRAW      = 1h
CS_HREDRAW      = 2h
CS_CLASSDC      = 40h
;*** et maintenant, certains styles de fenêtres
WS_POPUP        = 80000000h
WS_VISIBLE      = 10000000h
WS_CAPTION      = 0C00000h
WS_SIZEBOX      = 40000h
WS_SYSMENU      = 80000h
WS_MINIMIZEBOX  = 20000h
WS_MAXIMIZEBOX  = 10000h
;*** enfin, les messages auxquels nous allons avoir affaire
WM_CREATE       = 1h
WM_DESTROY      = 2h
WM_PAINT        = 0Fh
;*** et le reste...
COLOR_WINDOW    = 5
;-----------------------------------------------------------------
;
DATA SECTION
;
;-----------------------------------------------------------------
wcex WNDCLASSEX    ; établit la structure WNDCLASS dans DATA
hInst     DD 0     ; handle du process lui-même
MSG DD    7 DUP 0  ; structure contenant des messages de Windows comme
; suit : hWnd, +4=message, +8=wParam, +C=lParam, +10h=time, +14h/18=pt
;-----------------------------------------------------------------
;
CONST SECTION
;
;-----------------------------------------------------------------
;
WINDOW_CLASSNAME DB 'WC',0    ; chaîne destinée à contenir le nom
                              ; de classe de fenêtre
;
;-----------------------------------------------------------------
;
CODE SECTION
;
;-----------------------------------------------------------------
INITIALISE_WNDCLASS:      ; préparation de WNDCLASS pour la fenêtre
MOV EBX,ADDR wcex
ADD EBX,4                 ; saut au-delà du dword contenant la taille
MOV EAX,10
L1:
MOV D[EBX+EAX*4],0        ; comblement du reste de la structure avec des zéros
DEC EAX
JNS L1
;***** ajout d'éléments à la classe de fenêtre pour toutes
;***** les fenêtres dans le programme
```

```
MOV EAX,[hInst]          ; EAX = handle du process
MOV [wcex.hInstance],EAX ; on le reporte dans la structure de classe de fenêtre
PUSH 32512               ; IDC_ARROW valeur le la flèche de curseur ordinaire
PUSH 0
CALL LoadCursorA         ; on récupère, dans EAX, le handle de la flèche de curseur
MOV [wcex.hCursor],EAX   ; et on le répercute dans WNDCLASS
MOV D[wcex.hbrBackground],COLOR_WINDOW+1  ; on fixe la couleur de l'arrière-plan
RET
;
;******************
CREATE:                  ; le seul message effectivement traité dans ce programme
XOR EAX,EAX              ; retourne zéro pour faire la fenêtre
RET
;
DESTROY:                 ; l'un des rares messages traités par ce prog
PUSH 0
CALL PostQuitMessage     ; sortie via la boucle de message
STC                      ; aller à DefWindowProc
RET
;
;-------------------------------------------------------------
;
CONST SECTION
;
;-------------------------------------------------------------
;****************** Table des messages de fenêtre
;(Dans un véritable programme, cette table concernerait beaucoup plus de messages)
;
MESSAGES   DD WM_CREATE, CREATE       ; à chaque ligne,
           DD WM_DESTROY, DESTROY     ; le message puis l'adresse de code
           DD WM_PAINT, PAINT
;-------------------------------------------------------------
;
CODE SECTION
;
;-------------------------------------------------------------
;****************** Ceci est la procédure de fenêtre courante
WndProc:
FRAME hwnd,uMsg,wParam,lParam   ; établissement d'une trame de pile
;                               ; et obtention des paramètres
MOV EAX,[uMsg]                  ; récupération en EAX du msg envoyé par Windows
MOV ECX,SIZEOF MESSAGES/8       ; ECX, nb de messages de la table à tester
MOV EDX,ADDR MESSAGES           ; EDX = adresse de la table
L2:
DEC ECX
JS >.notfound                   ; saut car message non trouvé
CMP [EDX+ECX*8],EAX             ; on regarde si c'est le bon msg dans la table
JNZ L2                          ; non
CALL [EDX+ECX*8+4]              ; appel de la procédure correcte pour le msg
JNC >.exit
.notfound
INVOKE DefWindowProcA,[hwnd],[uMsg],[wParam],[lParam]
.exit
RET
ENDF                            ; fin de cette trame de pile
;
;*** et maintenant une procédure, en dehors du FRAME, qui doit adresser
;*** les données locales stockées sur la pile dans le FRAME WndProc
; Cette procédure trace une ellipse dans le rectangle fourni par Windows sur le
; message WM_PAINT. Ce rectangle est la zone qui a besoin d'être mise à jour, par
; exemple sur un redimensionnement ou si la fenêtre est découverte par une autre.
```

```
; le tracé est effectué en utilisant le contexte de périphérique fourni par Windows.
;
PAINT:
USEDATA WndProc                 ; utilisation des paramètres envoyés à WndProc
USES EBX,EDI,ESI                ; sauvegarde des registres comme requis par Windows
LOCAL lpPaint:PAINTSTRUCT, hDC  ; établit une zone de données locales
;
INVOKE BeginPaint, [hwnd],ADDR lpPaint  ; met en EAX le DC (Device Context) à utiliser
MOV [hDC],EAX                   ; sauvegarde du DC dans les données locales
INVOKE Ellipse, [hDC],[lpPaint.rcPaint.left],    \
                      [lpPaint.rcPaint.top] ,    \
                      [lpPaint.rcPaint.right],   \
                      [lpPaint.rcPaint.bottom]
;
;       d'autres procédures de dessin pourraient être écrites ici
;
INVOKE EndPaint, [hwnd],ADDR lpPaint    ; retourne DC à Windows
XOR EAX,EAX                     ; retourne Cf=0 et EAX = 0
RET
ENDU                            ; fin utilisant les paramètres envoyés à WndProc
;
;*****************************************************************
START:
;
INVOKE GetModuleHandleA, 0      ; récupération d'un handle pour le process
MOV [hInst],EAX                 ; on l'enregistre au label de donnée correspondant
CALL INITIALISE_WNDCLASS        ; on initialise la structure WNDCLASS
;
;********** on entre maintenant des paramètres spécifiques à la fenêtre à construire
;
MOV D[wcex.style],CS_VREDRAW+CS_HREDRAW+CS_CLASSDC
MOV [wcex.lpfnWndProc],ADDR WndProc                     ; procédure de fenêtre
MOV [wcex.lpszClassName],ADDR WINDOW_CLASSNAME          ; nom de classe de fenêtre
INVOKE RegisterClassExA,ADDR wcex                       ; mémorisation de la classe de fenêtre
INVOKE CreateWindowExA, 0, ADDR WINDOW_CLASSNAME,    \
        'Hello World window made by GoAsm',          \    ; chaîne correspondant au titre
        WS_POPUP|WS_VISIBLE|WS_CAPTION|WS_SIZEBOX    \    ; style de fenêtre
        |WS_SYSMENU|WS_MINIMIZEBOX|WS_MAXIMIZEBOX,   \
        50,50,                                       \    ; position-x puis position-y
        320,200,                                     \    ; largeur puis hauteur
        0,0,[hInst],0
;
;************************ on entre maintenant dans la boucle de message principale
;
.messloop
INVOKE GetMessageA, ADDR MSG,0,0,0
OR EAX,EAX      ; on regarde si c'est WM_QUIT (EAX = 0)
JZ >.quit       ; oui
INVOKE TranslateMessage, ADDR MSG
INVOKE DispatchMessageA, ADDR MSG
JMP .messloop   ; après le traitement du msg, boucle pour le suivant
.quit
INVOKE UnregisterClassA, ADDR WINDOW_CLASSNAME,[hInst]
INVOKE ExitProcess, [MSG+8h]
```

Programme HelloDialog.asm

Ce programme affiche des messages dans une boîte de dialogue modale (cf. figure 1 ci-dessous), puis dans une MessageBox lors de la sortie (figure 2 ci-dessous).
La boîte de dialogue modale est faite en utilisant un modèle dans les données au lieu du fichier de ressources habituel.
Pour cette raison, il utilise l'API *DialogBoxIndirectParamA* au lieu de *DialogBoxParam*, cette dernière étant dévolue à la constitution d'une boîte de dialogue modale par le biais d'un fichier de ressources. Nous sommes par ailleurs dans le cas d'un modèle étendu. L'en-tête utilise le format DLGTEMPLATE et les définitions de contrôle utilisent le format DLGITEMTEMPLATE.

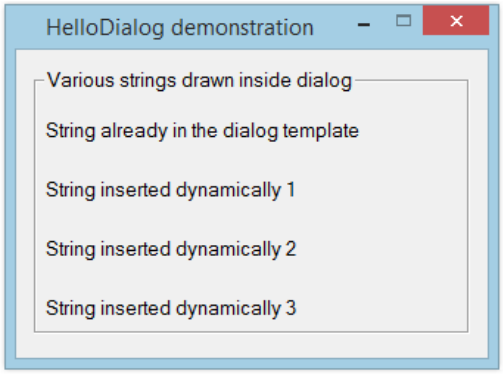


fig. 1 – première fenêtre

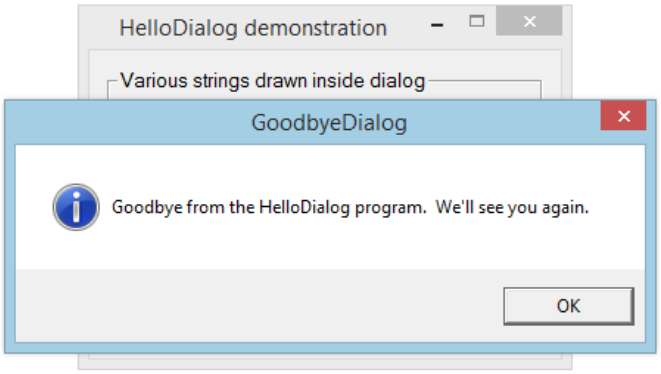


fig.2 – fenêtre en sortie de programme

```
;-------------------------------------------------------------------
;  HelloDialog - copyright Jeremy Gordon 2004
;
; SIMPLE programme de Dialogue
;
; aucune inclusion de fichier n'est utilisée car toutes les constantes
; sont insérées manuellement.
;
; Assemblage & Édition de liens :
;   GoAsm HelloDialog (produisant un fichier PE COFF)
;   GoLink [-debug coff] HelloDialog.obj kernel32.dll user32.dll
; qui produit HelloDialog.exe
; -debug coff n'est utilisé que s'il est souhaitable d'analyser le
; programme dans le débogueur
;
;-------------------------------------------------------------------
;
;*******************************************************************
;
DATA SECTION
;
ALIGN 4                 ; juste pour souligner que ce qui suit doit être aligné dword
;********************** Voici le modèle pour le dialogue
; Entête au format DLGTEMPLATE
DIALOG_TEMPLATE    DD 10000000h |0C00000h  |800h     |40h|  \
                   80h          |80000h    +20000h
                             ; style WS_VISIBLE+WS_CAPTION+DS_CENTER+DS_SETFONT
                             ;       DS_MODALFRAME+WS_SYSMENU+WS_MINIMIZEBOX
                   DD 0      ; style étendu
                   DW 5      ; nb d'items dans la boîte de dialogue
                   DW 0      ; position-x du coin supérieur gauche
                   DW 0      ; position-y du coin supérieur gauche
                   DW 160    ; largeur de la boîte
```

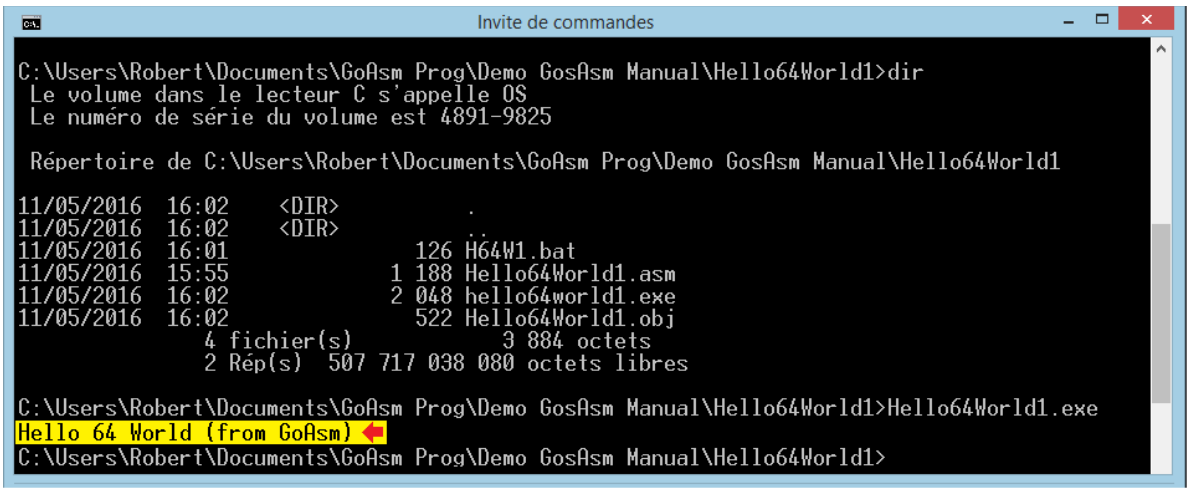
```
                DW 104    ; hauteur de la boîte
                DW 0      ; table de menu (pas de menu)
                DW 0      ; zone de classe (valeur par défaut)
                DUS 'HelloDialog demonstration',0  ; titre (DUS impose à
                                                   ; l'assembleur de convertir
                                                   ; la chaîne en Unicode
                                                   ; comme requis par Windows
                DW 10     ; taille de la police de caractères en points
                DUS 'Microsoft Sans Serif',0       ; vous pouvez changer la police
                                                   ; en déclarant, par exemple,
                                                   ; 'Arial' ou 'Times New Roman'.

;********************************* on constitue une boîte pour le groupe
; Contrôle au format DLGITEMTEMPLATE (y compris ceux qui suivent)
ALIGN 4                     ; le début doit être aligné dword
                DD 50000000h + 7h    ;(WS_CHILD+WS_VISIBLE)+BS_GROUPBOX
                DD 0      ; style étendu
                DW 6      ; position-x du coin supérieur gauche du contrôle
                DW 6      ; position-y du coin supérieur gauche du contrôle
                DW 148    ; largeur du contrôle
                DW 90     ; hauteur du contrôle
                DW 0      ; ID du contrôle
                DW -1     ; examen du suivant pour la classe
                DW 80h    ; contrôle de type "bouton"
                DUS 'Various strings drawn inside dialog',0
                DW 0      ; éléments à envoyer à DlgProc (0 = aucun)

;********************************* contrôle suivant
ALIGN 4                     ; le début doit être aligné dword
                DD 50000000h ;(WS_CHILD+WS_VISIBLE)
                DD 0      ; style étendu
                DW 10     ; position-x du coin supérieur gauche du contrôle
                DW 23     ; position-y du coin supérieur gauche du contrôle
                DW 140    ; largeur du contrôle
                DW 10     ; hauteur du contrôle
                DW 1h     ; ID du contrôle
                DW -1     ; examen du suivant pour la classe
                DW 82h    ; contrôle statique
                DUS 'String already in the dialog template',0
                DW 0      ; éléments à envoyer à DlgProc (0 = aucun)

;********************************* contrôle suivant
ALIGN 4                     ; doit être aligné dword
CONTROL_2:                  ;label permettant de modifier dynamiquement le contrôle
                DD 50000000h ;+0 (WS_CHILD+WS_VISIBLE)
                DD 0      ;+4 style étendu
                DW 10     ;+8 position-x du coin supérieur gauche du contrôle
                DW 43     ;+A position-y du coin supérieur gauche du contrôle
                DW 140    ;+C largeur du contrôle
                DW 10     ;+E hauteur du contrôle
                DW 2h     ;+10 ID du contrôle
                DW -1     ;+12 examen du suivant pour la classe
                DW 82h    ;+14 contrôle statique
                DW 30 DUP 0  ;+16 espace pour la chaîne Unicode terminée par zéro à ajouter
                DW 0      ; éléments à envoyer à DlgProc (0 = aucun)

;********************************* contrôle suivant
ALIGN 4                     ; doit être aligné dword
                DD 50000000h ;(WS_CHILD+WS_VISIBLE)
                DD 0      ; style étendu
                DW 10     ; position-x du coin supérieur gauche du contrôle
```

```
                DW 63     ; position-y du coin supérieur gauche du contrôle
                DW 140    ; largeur du contrôle
                DW 10     ; hauteur du contrôle
                DW 3h     ; ID du contrôle
                DW -1     ; examen du suivant pour la classe
                DW 82h    ; contrôle statique
                DW 0      ; cette chaîne sera ajoutée par INITIALISE_DIALOG
                DW 0      ; éléments à envoyer à DlgProc (0 = aucun)

;********************************** contrôle suivant
ALIGN 4                   ; doit être aligné dword
                DD 50000000h ;(WS_CHILD+WS_VISIBLE)
                DD 0      ; style étendu
                DW 10D    ; position-x du coin supérieur gauche du contrôle
                DW 83     ; position-y du coin supérieur gauche du contrôle
                DW 140    ; largeur du contrôle
                DW 10     ; hauteur du contrôle
                DW 4h     ; ID du contrôle
                DW -1     ; examen du suivant pour la classe
                DW 82h    ; contrôle statique
                DW 0      ; cette chaîne sera ajoutée par INITIALISE_DIALOG
                DW 0      ; éléments à envoyer à DlgProc (0 = aucun)
;
ALIGN 4
GOODBYE_MESSAGE    DB "Goodbye from the HelloDialog program.  We'll see you again.",0
DYNO_STRING        DB 'String inserted dynamically 1',0    ; 30 caractères incluant le zéro
;
;*********************************************************************
;*  CODE
;*********************************************************************
CODE SECTION
;
;************************************************** DÉBUT DU PROGRAMME
START:
PUSH 0
CALL GetModuleHandleA          ;récupération du handle de ce process en EAX
;
;****************************** Création de la boîte de dialogue et de ses contrôles
CALL INSERT_DYNAMICSTRING
PUSH 0                         ; valeur d'initialisation (non utilisée)
PUSH ADDR DlgProc              ; pointeur vers la procedure de dialogue
PUSH 0                         ; ce dialogue a le Bureau comme parent
PUSH ADDR DIALOG_TEMPLATE      ; adresse du modèle déclaré plus haut
PUSH EAX                       ; handle pour ce process en provenance de GetModuleHandleA
CALL DialogBoxIndirectParamA   ; retour de cette API uniquement à la clôture du dialogue
PUSH 0                         ; sortie avec le code zéro
CALL ExitProcess               ; sortie définitive
;
;************************************************** PROCÉDURE DE DIALOGUE
; On utilise ici FRAME..ENDF pour établir une zone de données locales et
; de récupération de paramètres.
;
DlgProc FRAME hDlg,uMsg,wParam,lParam
USES EBX,EDI,ESI               ; sauvegarde EBX, EDI et ESI comme requis par Windows
                               ; (par sécurité seulement)
MOV EAX,[uMsg]                 ; récupération du message dans EAX
CMP EAX,110h                   ; est-ce un message WM_INITDIALOG ?
JNZ >                          ; non, alors saut aux 2 points qui suivent
CALL INITIALISE_DIALOG
MOV EAX,1
JMP >.return                   ; retourne EAX <> 0 comme requis par Windows
```

```
:
CMP EAX,111h                    ; est-ce un message WM_COMMAND ?
JNZ >.false                     ; non
CMP W[wParam],2h                ; on regarde si sysmenu a été cliqué
JNZ >.false                     ; non

;********************** Oui, alors on dit"goodbye" avec une MessageBox
PUSH 40h                        ; mode icône d'information et bouton Ok
PUSH 'GoodbyeDialog'            ; titre
PUSH ADDR GOODBYE_MESSAGE       ; texte du corps de la MsgBox
PUSH [hDlg]                     ; propriétaire (ce dialogue)
CALL MessageBoxA

;*****************************
PUSH 1,[hDlg]
CALL EndDialog                  ; fin du dialogue
.false
XOR EAX,EAX                     ; retourne zéro comme requis
.return
RET
ENDF
;
; Notez que ce qui suit doit être exécuté avant que DialogBoxIndirectParam
; ne soit appelée. Il ne peut être réalisé sur le message WM_INITDIALOG

INSERT_DYNAMICSTRING:
;*************** ajout de la chaîne DYNO_STRING au 2ème contrôle statique
MOV EDI,ADDR CONTROL_2          ; addresse du contrôle statique dans EDI
ADD EDI,16h                     ; on fait pointer EDI sur l'adresse de la chaîne du contrôle
MOV ESI,ADDR DYNO_STRING        ; on fait pointer ESI sur l'adresse de la chaîne source
XOR EAX,EAX                     ; EAX = 0
;*************** chargement de la chaîne en la convertissant en Unicode en même temps
; Notez qu'on lit des octets successifs que l'on copie sur des mots successifs
; entraînant de facto la conversion en Unicode puisque, dans tous les cas, AH = 0
L0:
LODSB                           ; charge AL avec [ESI] et incrémente ESI d'une unité
STOSW                           ; copie AX en [EDI] et incrémente EDI de 2 unités
OR AL,AL                        ; on teste AL=0 marquant la fin de la chaîne à copier
JNZ L0                          ; non, alors on continue la copie
RET
;
; Sur WM_INITDIALOG vous pouvez faire beaucoup de travail pour ajouter des choses
; à la boîte de dialogue. Par exemple :

INITIALISE_DIALOG:
USEDATA DlgProc                 ; utilisation des données locales de la trame DlgProc
PUSH 'String inserted dynamically 2'
PUSH 3h                         ; identifiant du contrôle = 3h
PUSH [hDlg]                     ; utilisation du handle du dialogue dans les données locales
CALL SetDlgItemTextA
;************************* établissons maintenant le contrôle n° 4 d'une manière différente
PUSH 4h                         ; identifiant du contrôle = 4h
PUSH [hDlg]                     ; utilisation du handle du dialogue dans les données locales
CALL GetDlgItem                 ; récupération du handle du contrôle dans EAX
PUSH 'String inserted dynamically 3'
PUSH 0                          ; wParam n'est pas utilisé
PUSH 0Ch                        ; valeur constante pour le message WM_SETTEXT
PUSH EAX                        ; handle du contrôle issu du GetDlgItem qui précède
CALL SendMessageA               ;envoi du message WM_SETTEXT avec la chaîne
RET
```

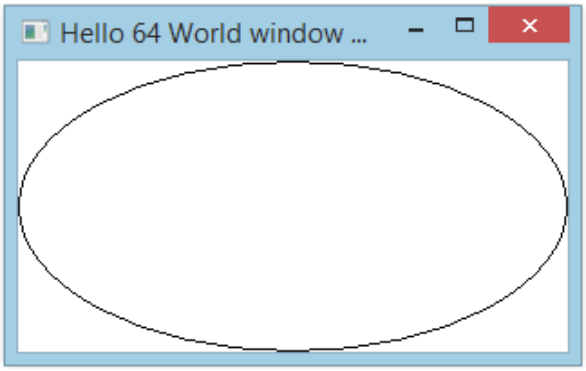
Programme Hello64World1.asm

Ce programme est écrit en 64 bits pour fonctionner en mode console. Ce mode est déclenché (voir plus loin) au niveau de l'édition de liens par l'utilisation du commutateur /console. La seule action de ce programme est d'afficher le message “Hello 64 World (from GoAsm)” (sur fond jaune dans la figure ci-dessous) :




```
;-------------------------------------------------------------------
;       Hello64World1 - copyright Jeremy Gordon 2005-6
;
;       SIMPLE "HELLO WORLD" WINDOWS CONSOLE PROGRAM - for GoAsm 64-bits
;
;        Assemblage & Édition de liens :
;          GoAsm /x64 Hello64World1 (produisant un fichier PE COFF)
;          GoLink /console [-debug coff] hello64world1.obj kernel32.dll
;        qui produit HelloDialog.exe
;        -debug coff n'est utilisé que s'il est souhaitable d'analyser le
;        programme dans le débogueur

;       Notez que les API GetStdHandle et WriteFile sont fournies par kernel32.dll
;-------------------------------------------------------------------
;
DATA SECTION
;
ALIGN 8                 ; alignement qword pour les données qui suivent
RCKEEP DQ 0             ; variable qword à usage général
Message DB 'Hello 64 World (from GoAsm)'
;
CODE SECTION
;
START:
ARG -11                 ; STD_OUTPUT_HANDLE
INVOKE GetStdHandle     ; récupération, dans RAX, du handle du buffer d'écran actif
;********************
ARG 0,ADDR RCKEEP       ; la variable RCKEEP reçoit la sortie de l'API
ARG 27                  ; longueur de la chaîne
ARG ADDR Message,RAX    ; RAX = handle du buffer d'écran actif
INVOKE WriteFile
XOR RAX,RAX             ; retour de zéro indiquant une exécution normale
RET

=======================================================
```

Programme Hello64World2.asm



Le programme Hello64World2.asm est la version 64 bits de HelloWorld2 décrit plus avant. Il consiste à dessiner une ellipse dans un rectangle. La figure ci-contre montre le résultat obtenu sur l'écran.

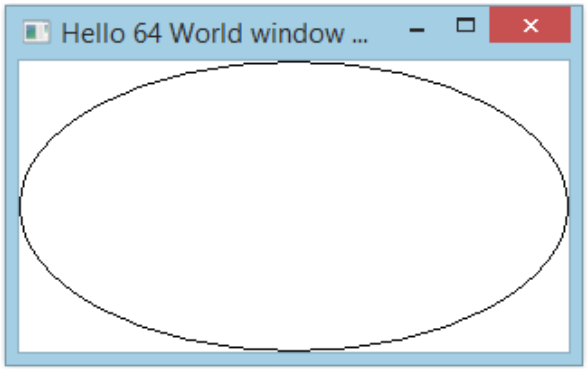
```
;-------------------------------------------------------------------
;
;      Hello64World2 - copyright Jeremy Gordon 2005-6
;
;      "HELLO WORLD" WINDOWS GDI PROGRAM - for 64 bits
;
;       Assemblage & Édition de liens :
;         GoAsm /x64 Hello64World2 (produisant un fichier PE COFF)
;         GoLink [-debug coff] Hello64World2.obj user32.dll kernel32.dll gdi32.dll
;       qui produit HelloDialog.exe
;       Notez que GoLink détecte automatiquement le fait qu'il s'agisse d'un fichier
;       à éditer en 64 bits
;       -debug coff n'est utilisé que s'il est souhaitable d'analyser le
;       programme dans le débogueur
;
;-------------------------------------------------------------------
;
;***************** Structure accueillant les infos de Windows sur WM_PAINT (version 64 bits)
;***************** pour l'API BeginPaint
PAINTSTRUCT STRUCT
            DQ 0         ; +0 hDC
            DD 0         ; +8 fErase
     left   DD 0         ; +C  gauche ┐
      top   DD 0         ; +10 haut    | RECT (coin supérieur gauche,
    right   DD 0         ; +14 droite  |     puis coin inférieur droit)
   bottom   DD 0         ; +18 bas     ┘
            DD 0         ; +1C fRestore     ┐
            DD 0         ; +20 fIncUpdate   | utilisés en interne par le système
            DB 32 DUP 0  ; +24 rgbReserved  ┘
            DD 0         ; bourrage permettant de porter la taille totale
                         ; de la structure à 72 octets
ENDS
;
DATA SECTION
;
ALIGN 8                  ; garantit que tous les éléments de données qui
                         ; suivent sont alignés sur un pas de 8 octets
hDC         DQ 0         ; pour mémoriser le handle du device context
PS PAINTSTRUCT

;***************** Structure accueillant les messages pour les API
;***************** GetMessageA, TranslateMessage et DispatchMessageA

MSG         DQ 0         ; +0 hWnd
```

```
            DD 0          ; +8 message
            DD 0          ; bourrage pour retomber sur un pas de 8 octets
   wParam   DQ 0          ; +10 wParam
            DQ 0          ; +18 lParam
            DD 0          ; +20 heure à laquelle le message a été posté
            DD 0          ; +24 position du curseur en coordonnées d'écran (1ère partie)
            DD 0          ; +28 position du curseur en coordonnées d'écran (2ème partie)
            DD 0          ; bourrage permettant de porter la taille totale
                          ; de la structure à 48 octets
;
;***************** Structure pour envoyer à RegisterClass les données mémorisées
;
WNDCLASSDD 1h+2h+40h      ; +0 style de classe de fenêtre (CS_VREDRAW+CS_HREDRAW+CS_CLASSDC)
            DD 0          ; +4 octets de comblement
            DQ WndProcTable       ; +8 pointeur vers la procédure de fenêtre
            DD 0          ; +10 nb d'octets supplémentaires à allouer après la structure
            DD 0          ; +14 nb d'octets supplémentaires à allouer après l'instance de fenêtre
    hInst   DQ 0          ; +18 handle de l'instance contenant la procédure de fenêtre
            DQ 0          ; +20 handle de l'icône de classe
  hCursor   DQ 0          ; +28 handle du curseur de classe
            DQ 6          ; +30 identifie la brosse de l'arrière-plan de la classe (6=COLOR_WINDOW+1)
            DQ 0          ; +38 pointeur vers le nom de ressource pour le menu de classe
            DQ WINDOW_CLASSNAME   ;+40 pointeur vers chaîne contenant le nom de la classe de fenêtre
;
;******************** Table des messages de fenêtre
;(Dans un véritable programme, cette table concernerait beaucoup plus de messages)
MESSAGES    DD (ENDOF_MESSAGES-$-4)/8  ; = nombre de messages à examiner
            DD  1h,CREATE,2h,DESTROY,0Fh,PAINT
ENDOF_MESSAGES: ;vlabel utilisé pour calculer le nombre de messages
;****************************************
;
WINDOW_CLASSNAME DB 'WC',0   ; chaîne contenant le nom de la classe de fenêtre
;
;----------------------------------------------------------------------
;
CODE SECTION
;
;*******************
CREATE:                      ; l'un des quelques messages traités par ce programme
XOR RAX,RAX                  ; retourne zéro pour fabriquer la fenêtre
RET
;
DESTROY:                     ; l'un des quelques messages traités par ce programme
INVOKE PostQuitMessage,0     ; sortie via la boucle de message
STC                          ; aller à DefWindowProc en plus
RET
;
; Le process qui suit permet de tracer une ellipse dans le rectangle fourni par
; Windows sur le message WM_PAINT. Ce rectangle est la zone qui a besoin d'être
; mise à jour, par exemple lors d'un redimensionnement ou si la fenêtre est
; découverte par une autre.

PAINT:
PUSH RDI,RBX                 ; sauvegarde des registres non-volatiles utilisés
MOV RDI,RCX                  ; sauvegarde hWnd pour un usage ultérieur
LEA RBX,PS                   ; récupération de paintstruct dans rbx
ARG RBX                      ; l'envoyer en tant que paramètre d'API
ARG RDI                      ; hWnd
INVOKE BeginPaint            ; récupération du device context à utiliser, initialisation du dessin
MOV [hDC],RAX
;***************** Utilisation du rectangle envoyé par le système dans
```

```
;***************** la structure de PAINTSTRUCT (PS) ..
XOR RAX,RAX
MOV EAX,[PS.bottom]
ARG RAX
MOV EAX,[PS.right]
ARG RAX
MOV EAX,[PS.top]
ARG RAX
MOV EAX,[PS.left]
ARG RAX
ARG [hDC]
INVOKE Ellipse              ; dessine l'ellipse dans le rectangle mis à jour
;*****************
INVOKE EndPaint,RDI,RBX     ; RDI = hWnd, RBX = paintstruct
POP RBX,RDI                 ; restauration des registres non-volatiles utilisés
XOR RAX,RAX                 ; retourne RAX = 0 et nc (ne pas appeler DefWindowProc)
RET
;
;*************************************************************************
; L'appel au label de code trouvé dans la table n'est pas destiné à CALL [EAX+R10D*8+4] puisque
; tous les calls à des adresses détenues dans des zones de mémoire pointées par des registres
; ou à des adresses détenues dans des registres eux-mêmes sont des appels à des adresses 64 bits
; en assemblage 64 bits.
; Mais la table ne contient que des adresses 32 bits. Ainsi, nous extrayons plutôt l'adresse
; de 32 bits dans la table à destination de R10D puis nous faisons un Call R10 sachant que
; le dword de poids fort de ce registre est nul.

GENERAL_WNDPROC64:          ; uMsg est dans RDX (actuellement EDX)
MOV R10D,[EAX]              ; on récupère le nombre de messages à traiter
ADD EAX,4                   ; on saute par-dessus le dword contenant la taille
L2:
DEC R10D
JS >L3                      ; message non trouvé dans la table
CMP [EAX+R10D*8],EDX        ; regardons si c'est le message correct
JNZ L2                      ; non
;RCX=hwnd, RDX=uMsg, R8=wParam, R9=lParam
PUSH R9,R8,RDX,RCX          ; sauvegarde des paramètres pour DefWindowProcA
MOV R10D,[EAX+R10D*8+4]     ; récupération de l'adresse de procédure correspondant au message
CALL R10D
POP RCX,RDX,R8,R9           ; restauration des paramètres pour DefWindowProcA
JNC >L4                     ; nc = valeur de retour dans EAX - pas d'appel de DefWindowProc
L3:
SUB RSP,20h                 ; on réserve de l'espace sur la pile pour les paramètres en registres
                            ; comme requis
CALL DefWindowProcA
ADD RSP,20h                 ; libération de cet espace initialement réservé sur la pile
L4:
RET
;
;*************************************************************************
;
;****************** Voici la procédure de fenêtre courante
WndProcTable:
MOV EAX,ADDR MESSAGES       ; EAX pointe la liste des messages à traiter
CALL GENERAL_WNDPROC64      ; appel du gestionnaire de message générique (version 64 bits)
RET
;
;*****************************************************************
START:
INVOKE GetModuleHandleA,0   ; récupération du handle du process
MOV [hInst],RAX             ; mémorisation de ce handle dans hInst
```

```
INVOKE LoadCursorA,0,32512  ; chargement dans EAX, handle de IDC_ARROW (flèche de curseur classique)
MOV [hCursor],RAX           ; mémorisation de ce handle dans hCursor (dans WNDCLASS)
;********** on enregistre maintenant la window class
INVOKE RegisterClassA,ADDR WNDCLASS     ; mémorisation de la window class
;********** création de la fenêtre
ARG 0,[hInst],0,0           ; owner=desktop
ARG 200D                    ; hauteur
ARG 320D                    ; largeur
ARG 50D,50D                 ; position y puis x
ARG 90000000h +0C00000h+40000h +80000h +20000h     +10000h
 ;(POPUP+VISIBLE)+CAPTION+SIZEBOX+SYSMENU+MINIMIZEBOX+MAXIMIZEBOX
ARG 'Hello 64 World window made by GoAsm'       ; titre de la fenêtre
ARG ADDR WINDOW_CLASSNAME   ; nom de window class
ARG 0                       ; style étendu
INVOKE CreateWindowExA      ; fabrication de la fenêtre
;********************** on entre maintenant la boucle de message principale
L1:
INVOKE GetMessageA,ADDR MSG,0,0,0
OR RAX,RAX                  ; on regarde si c'est WM_QUIT
JZ >L2                      ; oui
INVOKE TranslateMessage,ADDR MSG
INVOKE DispatchMessageA,ADDR MSG
JMP L1                      ; après le msg traité, boucle de retour pour un prochain msg
L2:                         ; cas où le message était WM_QUIT
ARG [hInst],ADDR WINDOW_CLASSNAME
INVOKE UnregisterClassA     ; on s'assure que la classe est supprimée
INVOKE ExitProcess,[wParam] ; wParam= code de sortie

=================================================
```

Programme Hello64World3.asm



Le programme Hello64World3.asm a également la même fonction que HelloWorld2, HelloWorld3 et Hello64World.asm en ce sens qu'il se borne à tracer une ellipse dans une rectangle.

Toutefois, le code source est écrit de telle sorte qu'il puisse servir indifféremment sur les plateformes 32 et 64 bits selon l'utilisation du commutateur /x86 ou /x64 sur la ligne de commande de GoAsm. On sait que l'éditeur de lien n'a besoin, quant à lui, d'aucune commande spécifique pour ce faire.

```
;-------------------------------------------------------------------
;
;       Hello64World3 - copyright Jeremy Gordon 2005-6
;
;       PROGRAM WINDOWS GDI "HELLO WORLD" COMMUTABLE Win32/64
;
;        Assemblage :
;          GoAsm /x64 Hello64World3 pour un exe 64 bits
;          GoAsm /x86 Hello64World3 pour un exe 32 bits
;       Édition de liens :
;          GoLink [-debug coff] Hello64World3.obj user32.dll kernel32.dll gdi32.dll
;       qui produit Hello64World3.exe
;       Notez que GoLink détecte automatiquement le fait qu'il s'agisse d'un fichier
;       à éditer en 32 ou 64 bits
;       -debug coff n'est utilisé que s'il est souhaitable d'analyser le
;       programme dans le débogueur

;-------------------------------------------------------------------
;
;  L'utilisation du commutateur x86 a pour effet que :
;  "x86" est un mot défini et identifiable en utilisant #if
;  Tous les registres étendus sont rempplacés par leurs homologues 32 bits
;  Par exemple,
;   RBX devient EBX
;   ARG RAX se traduit par PUSH EAX
;   ARG ADDR WNDCLASS se traduit par PUSH ADDR WNDCLASS
;   ARG 800h se traduit par PUSH 800h
;   ARG [hInst] se traduit par PUSH [hInst]
;   ARG 'String' ou ARG ADDR 'String' se traduit par PUSH 'String' (push de l'adresse de la chaîne)
;   INVOKE fonctionne en STDCALL
;   ARG doit être suivi par un INVOKE avant le CALL ou le RET suivants
;   FRAME...ENDF fonctionne en Win32
;
;  L'utilisation du commutateur x64 a pour effet que :
;  "x64" est un mot défini et identifiable en utilisant #if
;   ARG RAX (selon sa position) se traduit par MOV RCX ou RDX ou R8 ou R9,RAX ou PUSH RAX
;   ARG ADDR WNDCLASS (selon sa position) devient LEA RCX,WNDCLASS
;            ou LEA RDX ou R8 ou R9 ou PUSH ADDR WNDCLASS
;   ARG 800h se traduit par un MOV RCX ou RDX ou R8 ou R9,800h ou PUSH 800h
;   ARG [hInst] se traduit par un MOV RCX ou RDX ou R8 ou R9,[hInst] ou PUSH [hInst]
;   ARG 'String' ou ARG ADDR 'String' se traduit par un MOV RCX ou RDX ou R8 ou R9,ADDR 'String'
;            PUSH 'String' (met l'adresse de la chaîne dans le registre ou sur la pile)
;   INVOKE fonctionne en FASTCALL, offrant un espace réservé sur la pile pour les registres
;            contenant des paramètres au moyen de SUB RSP,20h avant le CALL et ADD RSP,20h après.
```

```
;          il corrige également la pile pour permettre les paramètres introduits par PUSH
;          il permet également aux paramètres de suivre les fonctions appelées comme en "C"
;   ARG doit être suivi par un INVOKE avant le CALL ou le RET suivants
;   FRAME...ENDF fonctionne en Win64
;
;-------------------------------------------------------------
;
RECT STRUCT
   left          DD ?
   top           DD ?
   right         DD ?
   bottom        DD ?
ENDS
;
;***************** Structure détenant les infos de Windows sur WM_PAINT (deux versions différentes)
;
#if x64
PAINTSTRUCT STRUCT          ; plateforme 64 bits
   hdc           DQ ?
   fErase        DD ?
   rcPaint       RECT
   fRestore      DD ?
   fIncUpdate    DD ?
   rgbReserved   DB 32 DUP ?
                 DD 0       ; bourrage pour atteindre une taille totale de 72 octets
ENDS
#else
PAINTSTRUCT STRUCT          ; plateforme 32 bits
   hdc           DD ?
   fErase        DD ?
   rcPaint       RECT
   fRestore      DD ?
   fIncUpdate    DD ?
   rgbReserved   DB 32 DUP ?
ENDS
#endif
;
DATA SECTION
;
;***************** Structure hébergeant le message
#if x64            ; ===== cas du 64 bits
ALIGN 8            ; impose un alignement sur un multiple de 8 octets pour la structure
MSG        DQ 0    ; +0 hWnd
           DD 0    ; +8 message
           DD 0    ; bourrage pour préserver l'alignement qword pour la suite
  wParam   DQ 0    ; +10 wParam
           DQ 0    ; +18 lParam
           DD 0    ; +20 time
           DD 0    ; +24 position de la souris (X)
           DD 0    ; +28 position de la souris (Y)
           DD 0    ; bourrage pour donner à la structure une taille de 48 octets
#else              ; ===== cas du 32 bits
MSG        DD 0    ; +0 hWnd
           DD 0    ; +4 message
  wParam   DD 0    ; +8 wParam
           DD 0    ; +C lParam
           DD 0    ; +10 time
           DD 0    ; +14 position de la souris (X)
           DD 0    ; +18 position de la souris (Y)
#endif
;
```

```
;***************** Structure à envoyer à RegisterClass qui détient les données
#if x64                  ; ===== cas du 64 bits
ALIGN 8                  ; impose un alignement sur un multiple de 8 octets pour la structure
WNDCLASS   DD 1h+2h+40h  ; +0 style de classe de fenêtre (CS_VREDRAW+CS_HREDRAW+CS_CLASSDC)
           DD 0          ;  bourrage pour retomber sur un alignement qword
           DQ WndProc    ; +8 pointeur vers la procédure de fenêtre
           DD 0          ; +10 nb d'octets supplémentaires à allouer après la structure
           DD 0          ; +14 nb d'octets supplémentaires à allouer après window instance
  hInst    DQ 0          ; +18 handle de l'instance contenant la procédure de fenêtre
           DQ 0          ; +20 handle de l'icône de classe
hCursor    DQ 0          ; +28 handle du curseur de classe
           DQ 6          ; +30 identifie la brosse d'arrière-plan de la classe (6=COLOR_WINDOW+1)
           DQ 0          ; +38 pointeur vers nom de ressource pour le menu de classe
           DQ WINDOW_CLASSNAME  ; +40 pointeur vers la chaîne du nom de classe de fenêtre
#else                    ; ===== cas du 32 bits
WNDCLASS   DD 1h+2h+40h  ; +0 style de classe de fenêtre (CS_VREDRAW+CS_HREDRAW+CS_CLASSDC)
           DD WndProc    ; +4 pointeur vers la procédure de fenêtre
           DD 0          ; +8  nb d'octets supplémentaires à allouer après la structure
           DD 0          ; +C  nb d'octets supplémentaires à allouer après l'instance de fenêtre
  hInst    DD 0          ; +10 handle de l'instance de cette classe de fenêtre
           DD 0          ; +14 handle de l'icône de classe
hCursor    DD 0          ; +18 handle du curseur de classe
           DD 6          ; +1C identifie la brosse d'arrière-plan de la classe (6=COLOR_WINDOW+1)
           DD 0          ; +20 pointeur vers nom de ressource pour le menu de classe
           DD WINDOW_CLASSNAME  ; +24 pointeur vers la chaîne du nom de classe de fenêtre
#endif
;+++
;******************** Table des messages Windows

; Cette table permet d'identifier un message (en fait, une valeur binaire sur un dword) et de
; fournir le pointeur approprié vers la routine associée. Dans un programme plus élaboré, il y
; aurait beaucoup plus de messages à traiter que les 3 qui sont proposés.

ALIGN 8                     ; garantit un alignement sur un multiple de 8 octets pour les call
                            ; (essentiel en 64-bits!)
MESSAGES   DD 1h,  CREATE   ; valeur du message, puis adresse de code 32 bits du label
           DD 2h,  DESTROY
           DD 0Fh, PAINT
;
;****************************************
;
ALIGN 8                  ; garantit un alignment sur un multiple de 8 octets de la chaîne
                         ; (essentiel en 64 bits !!)
WINDOW_CLASSNAME DB 'WC',0      ; chaîne contenant le nom de classe de fenêtre
;
;-----------------------------------------------------------------
;
CODE SECTION
;
;*******************
CREATE:                  ; l'un des quelques messages traités par ce programme
XOR RAX,RAX              ; retourne zéro pour autoriser la construction de la fenêtre
RET
;
DESTROY:                 ; l'un des quelques messages traités par ce programme
INVOKE PostQuitMessage,0 ; sortie via la boucle de message
STC                      ; va à DefWindowProc également
RET
;
```

```
;==================================== WINDOW PROC ==========================================
;**** Ceci est la procédure de fenêtre réelle qui fonctionne à la fois pour l'assemblage 32 bits
; et 64 bits. L'appel du label de code trouvé dans le tableau ne se fait pas par CALL [EDX+ECX*8+4]
; puisque tous les CALLs à des adresses détenues dans des zones mémoire pointées par des registres
; ou à des adresses détenues dans les registres eux-mêmes sont des CALLs à des adresses 64 bits
; dans l'assembleur 64 bits.
; Mais le tableau ne contient que des adresses 32 bits. Aussi, nous extrayons l'adresse de 32 bits
; du tableau pour la placer en ECX puis nous appelons RCX sachant que le dword de poids fort de
; RCX est nul.
;
WndProc:
FRAME hwnd,uMsg,wParam,lParam   ; établit la trame de pile et obtient les paramètres
;
MOV EAX,[uMsg]                  ; met en EAX le message envoyé par Windows
MOV ECX,SIZEOF MESSAGES/8       ; récupère le nombre de messages à examiner
MOV EDX,ADDR MESSAGES
L2:
DEC ECX
JS >.notfound
CMP [EDX+ECX*8],EAX             ; on regarde si c'est le message correct
JNZ L2                          ; non
MOV ECX,[EDX+ECX*8+4]
CALL RCX
JNC >.exit
.notfound
INVOKE DefWindowProcA,[hwnd],[uMsg],[wParam],[lParam]
.exit
RET
ENDF                            ; fin de la trame de pile
;
;**** Et maintenant une procédure, en dehors du FRAME, qui doit adresser les paramètres
;**** stockés sur la pile dans le FRAME WndProc
; Elle dessine une ellipse dans le rectangle fourni par Windows sur le message WM_PAINT. Ce
; rectangle est la zone qui a besoin d'être mise à jour, par exemple sur le redimensionnement
; ou si la fenêtre est découverte par une autre. Le dessin est fait en utilisant le contexte de
; périphérique fourni par Windows.
; Nous mettons en place des données LOCAL ici, spécifiques aux messages Windows
;
PAINT:
USEDATA WndProc                 ; utilise les paramètres envoyés à WndProc
USES RBX,RDI,RSI                ; sauvegarde les registres tel que requis par Windows
                                ; (pas vraiment nécessaire ici)
LOCAL lpPaint:PAINTSTRUCT,hDC   ; établit des données locales
;
INVOKE BeginPaint,[hwnd],ADDR lpPaint   ; récupère en EAX/RAX le DC à utiliser
MOV [hDC],RAX                           ; on le sauvegarde dans les données locales
INVOKE Ellipse, [hDC],[lpPaint.rcPaint.left],    \
                      [lpPaint.rcPaint.top] ,    \
                      [lpPaint.rcPaint.right],   \
                      [lpPaint.rcPaint.bottom]
;
;     d'autres procédures de dessin peuvent éventuellement être écrites ici
;
INVOKE EndPaint, [hwnd],ADDR lpPaint    ; retour du DC à Windows
XOR RAX,RAX                     ; retour d'un Carry nul et de RAX = 0
RET
ENDU                            ; fin de l'utilisation des paramètres envoyés à WndProc
;
```

```
;*****************************************************************
START:
INVOKE GetModuleHandleA,0    ; récupération du handle du process
MOV [hInst],RAX              ; mémorisation de ce handle dans le label de donnée hInst
INVOKE LoadCursorA,0,32512   ; récupération en EAX du handle deIDC_ARROW (curseur commun de flèche)
MOV [hCursor],RAX            ; enregistrement de ce handle au label de donnée hCursor (dans WNDCLASS)
;
;********** enregistrons maintenant la classe de fenêtre
INVOKE RegisterClassA,ADDR WNDCLASS     ; enregistrement de la classe de fenêtre
;
;********** Créons maintenant la fenêtre
ARG 0
ARG [hInst]
ARG 0
ARG 0        ; propriétaire=desktop
ARG 200D     ; hauteur
ARG 320D     ; largeur
ARG 50D      ; position y
ARG 50D      ; position x
ARG 90000000h +0C00000h+40000h +80000h +20000h     +10000h
 ;(POPUP+VISIBLE)+CAPTION+SIZEBOX+SYSMENU+MINIMIZEBOX+MAXIMIZEBOX
ARG 'Hello 64 World window made by GoAsm'       ; titre de la fenêtre
ARG ADDR WINDOW_CLASSNAME       ; nom de la classe de fenêtre
ARG 0                           ; style étendu
INVOKE CreateWindowExA          ; construction de la fenêtre
;******************** Entrons maintenant dans la boucle de message principale
L1:
INVOKE GetMessageA,ADDR MSG,0,0,0
OR RAX,RAX             ; on regarde si c'est WM_QUIT
JZ >L2                 ; oui
INVOKE TranslateMessage,ADDR MSG
INVOKE DispatchMessageA,ADDR MSG
JMP L1                 ; après un traitement de message, boucle de retour pour un autre message
L2:                    ; cas où le message était WM_QUIT
ARG [hInst]
ARG ADDR WINDOW_CLASSNAME
INVOKE UnregisterClassA         ; garantit que la classe est désenregistrée
INVOKE ExitProcess,[wParam]     ; wParam = code de sortie
```

Annexe B
Écriture d'un programme Windows élémentaire

Avez-vous déjà été frustré par une application qui ne fait pas exactement ce que vous voulez ? Eh bien, je me propose de vous montrer comment écrire vos propres applications et vous affranchir de ces inconvénients. C'est gratuit, enrichissant et, qui plus est, amusant !

Cela ne peut être un cours intensif en écriture de programme, et je vais devoir passer beaucoup de choses sous silence. Je vais supposer que vous en savez très peu sur la programmation informatique et commencer donc par les bases. À la fin de ce chapitre, vous saurez comment produire votre propre programme "Bonjour tout le monde". Il vous appartiendra, dès lors, de décider si vous voulez aller plus loin dans ce monde fascinant. Dans l'affirmative, vous pourrez lire les autres articles de ce document et faire de même sur d'autres sites.

Commençons par le commencement. Un programme contient des instructions à destination du processeur de l'ordinateur. Celles-ci forment le *code* du programme. Le processeur *exécute* les instructions de code une par une, ce qui est la raison pour laquelle le programme est qualifié d'*exécutable*. Lors du chargement du programme, Windows communique au processeur l'*adresse de départ* dans le code c'est à dire l'endroit où l'exécution doit commencer. A partir de là, il appartient au programme de porter l'exécution à l'endroit approprié de son code. Je reviendrai sur ce point important plus tard.

De quoi est constitué le code ? Il s'agit d'une série d'octets caractérisant les instructions à exécuter. Chaque instruction met en œuvre un ou plusieurs octets, chaque combinaison d'entre eux traduisant le plus souvent la très grande variété de modes d'adressage possibles. Lorsque le processeur a identifié une instruction, il l'exécute puis examine les octets suivants et ainsi de suite. Mais ce mode séquentiel est assez souvent remis en question par des sauts (JMP ou Jcc) ou des appels de sous-programme (CALL). Dans ce cas, le compteur d'instructions est incrémenté (ou décrémenté, selon le cas) d'un nombre d'octets correspondant à la valeur du saut quitte à revenir à une position quittée précédemment par un retour (RET).

Eh bien, que font les instructions ? Entres autres activités, elles déplacent des nombres ou des caractères de registre à registre, entre registre et espace mémoire ou entre deux espaces mémoire. Elles réalisent également des opérations arithmétiques et logiques entre ces différentes entités. Parmi ces registres, sept sont dévolus à un usage général et contiennent, chacun, *32 bits de donnée*. Ce format est nommé *dword* (littéralement : double mot). Chaque bit est activé ou désactivé (un ou zéro, “mis” ou “à zéro”) à un moment donné. C'est un nombre *binaire*. En 32 bits, si tous les bits sont activés (tous à 1), le nombre en décimal est 4 294 967 295 (2^{32}–1). Mais les nombres ne sont qu'un des aspects de ces représentations binaires puisque chaque octet peut également représenter un caractère selon la codification ASCII bien connue.

Après le code, on trouve généralement la section de données qui se caractérise, comme le code, par une série d'octets mais avec une finalité radicalement différente. Elle contient des valeurs numériques et des messages utilisés par la section de code décrite précédemment. Assez souvent, elle recueille le résultat d'opérations effectuées par les instructions.

Après les données, se situe une assez grande zone libre appelée le « tas » (*heap* en anglais) qui accueille à la fois les buffers et les différentes tables initiées par Windows pour le fonctionnement interne du programme.

On trouve enfin une zone de mémoire un peu particulière qui s'appelle la *pile*. Ce dispositif reçoit toutes les données transitoires ainsi que les variables locales. Bien qu'incluse dans la mémoire vive générale de l'ordinateur (RAM), elle se distingue par un mode de gestion inédit. Le stockage et le déstockage de l'information y est réalisé respectivement par une instruction PUSH (mise en pile) et une instruction POP (retrait de la pile). Ces 2 instructions n'ont pas de dispositif d'adressage explicite et fonctionne par analogie à une pile d'assiettes, chacune d'entre elles étant représentée par un Dword. Vous mettez une assiette sur la pile (PUSH) et vous la retirez le moment venu (POP). Élémentaire. Mais, si l'assiette qui vous chère est en-dessous d'une autre, il vous faut retirer cette dernière avant de pouvoir la retrouver. Cela diffère sensiblement d'un stockage dans la section de données à une adresse fixe. Conséquence immédiate de ce mode de fonctionnement : à mesure que l'on charge la pile de données par des PUSH successifs, l'adresse est décrémentée. Elle est incrémentée, en revanche, chaque fois que l'on retire un élément de la pile. Il résulte de cette structure un peu particulière que le sommet de la pile représente l'adresse extrême la plus haute de votre programme.

La pile est réservée par le système au moment du chargement du programme.

Dans le contexte de Windows l'une des choses les plus importantes que les instructions de code réalisent est d'appeler une *fonction de fenêtre*, c'est-à-dire de détourner l'exécution vers cette fonction. Les fonctions Windows sont appelées API (littéralement Application Programmer's Interface ou Interface de Programmeur d'Applications). La plupart des autres instructions de code sont utilisées pour préparer ces appels et en traiter le résultat.

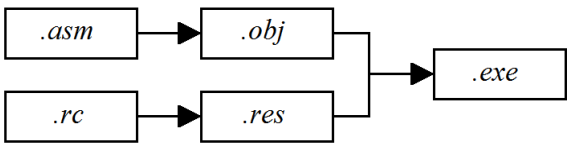
Ce sont ces appels aux API qui procureront à vos programmes des fonctionnalités extraordinaires. La possibilité d'appeler Windows de cette façon vous ouvre l'accès à une vaste gamme de processus qui comprendra l'interaction avec l'utilisateur, l'affichage d'écran approprié, l'impression, le traitement de fichiers, etc. Toutes ces actions sont effectivement conduites par Windows à la demande du programme.

Le processus de construction

Bien, mais comment élaborer un exécutable ? Vous avez besoin pour cela d'outils de développement qui sont tous disponibles en téléchargement à partir de mon site web. Ce sont des logiciels entièrement gratuits et sans date d'expiration. En voici un bref aperçu :

- **GoAsm** – l'assembleur. Ce programme prend en charge votre *script source* et le convertit en un *fichier objet*. Vous écrivez votre script source sous forme de texte avec votre éditeur de texte favori, par exemple le Bloc-Notes Windows (Notepad) ou Wordpad, en vous assurant qu'il enregistre le fichier sans caractères de contrôle dans le texte. Traditionnellement, l'extension de ce fichier est "*.asm*". Le script source contient les lignes de code incluant les instructions de code et de données constituant votre programme. Le fichier objet est un fichier sous forme codée qui peut être lu par l'*éditeur de liens* (ou *linker*) et qui est utilisé pour fournir l'exécutable final.
- **GoRC** – le compilateur de ressources. Ce programme prend en charge – s'il existe – votre script de ressources et le convertit en un fichier *res*. Le script de ressources est un autre fichier de texte brut, mais qui contient des instructions pour établir les contrôles de Windows pour nos programmes – menus, boîtes de dialogue, icônes, bitmaps et tables de chaînes. Traditionnellement, ce fichier reçoit l'extension *rc*. Il est sous forme codée et traité comme tel par l'éditeur de liens.
- **GoLink** – l'éditeur de liens. Ce programme prend en charge un ou plusieurs fichiers-objet et, s'il y a lieu, un fichier de ressources (*res*) dans le but de créer l'exécutable final.
- **GoBug** – le débogueur. Avec cet outil, vous pouvez observer votre programme en train d'exécuter les instructions une par une et voir comment chaque instruction affecte les registres et zones de mémoire.

Pour résumer, le processus de construction peut être représenté graphiquement comme suit :



Création d'un programme Windows "do-nothing"

Nous allons maintenant écrire un programme qui se charge, s'exécute et se termine... sans rien faire. L'objectif recherché ici est de détailler le processus de construction.

Action	**Explication**
Créez un nouveau répertoire dans votre lecteur C: appelé "prog".	Cela vous aide à organiser votre travail. Vous pouvez ainsi mettre tout votre travail et les outils de programmation dans ce répertoire ou dans des sous-répertoires créés à cet effet si vous préférez.
Copiez GoAsm, GoRC, GoLink et GoBug dans ce répertoire.	

Action	**Explication**

Ensuite, en utilisant votre éditeur de texte, créez un nouveau fichier et saisissez ces lignes :
`CODE SECTION`
`START:`
`RET`
Enregistrez le fichier ainsi constitué dans le répertoire "prog" et nommez-le "Nothing.asm".
	Voici l'intégralité du programme "do-nothing". La ligne "CODE SECTION" indique à l'assembleur que les lignes qui suivent ne contiennent que des instructions de code à défaut de rencontrer toute autre déclaration de section. "START" est un *label de code*. Il notifie à l'éditeur de liens que c'est à cet endroit que l'exécution du programme doit commencer. "RET" est un *mnémonique* (instruction processeur exprimée en mots explicites) qui signifie au processeur qu'il doit revenir au code appelant qui, dans ce cas, est Windows lui-même.

Maintenant, en utilisant toujours votre éditeur de texte, créez un nouveau fichier et tapez ces lignes :
`GoAsm Nothing`
`GoLink Nothing.obj /console`
`Pause`
Enregistrez ce fichier dans le dossier "prog" en l'appelant "gonothing.bat".
	Vous créez un *fichier batch* qui consiste en un petit fichier très utile s'exécutant ligne par ligne à partir de la fenêtre MS-DOS (invite de commande). La première ligne exécute l'assembleur GoAsm, en lui fournissant le fichier Nothing.asm (ici, l'extension asm n'est pas mentionnée car considérée comme implicite). Cela crée le fichier Nothing.obj qui est ensuite soumis à GoLink, pour créer le fichier exécutable Nothing.exe. Avec le commutateur `/console` sur sa ligne de commande, GoLink est par ailleurs invité à créer un programme de *console*, ce nom indiquant qu'il ne fait pas usage de l'interface utilisateur graphique Windows dans le sens où il n'utilise pas de fenêtres. Enfin, la dernière ligne *Pause* permet de suspendre l'exécution du fichier batch le temps, pour l'utilisateur, de découvrir le compte-rendu d'exécution des lignes qui précèdent sur l'écran.

Allez dans le menu "démarrer", "programmes", puis cliquez sur "MS-DOS" (noter que la position de cette commande peut différer selon les ordinateurs, par exemple, dans XP, on trouve *démarrer*, *tous les programmes*, *accessoires*, *invite de commande*). Maintenant, vous devriez voir ce que l'on appelle le "C:\>" d'invite dans la fenêtre MS-DOS (invite de commande). Tapez "cd prog" puis Entrée.	Si vous n'êtes pas habitué aux commandes DOS, ces manœuvres resteront un mystère pour vous. Mais disons ici, pour simplifier, que vous changez le répertoire en cours dans la fenêtre MS-DOS (invite de commande) en c:\prog, qui est le répertoire dans lequel votre travail de programmation réside.
Tapez "gonothing" puis Entrée.	Ici, vous exécutez le fichier batch *gonothing.bat*. Il l'exécutera ligne par ligne et vous pourrez voir ce qui se passe dans la fenêtre MS-DOS (invite de commande). Vous verrez les fichiers Nothing.obj et Nothing.exe successivement créés par GoAsm puis GoLink. *Si vous préférez, vous pouvez obtenir le même résultat en double-cliquant tout simplement sur le fichier batch dans l'Explorateur Windows.*
Finalement, tapez "Nothing" puis Entrée.	Pour terminer, vous exécutez le fichier exécutable *Nothing.exe* que vous venez de créer précédemment. Constatez, en tout cas, qu'il ne fait réellement... rien. L'invite DOS va simplement se déplacer d'une ligne vers le bas. Mais Windows a été très heureux avec ce programme – félicitations, vous venez de faire votre premier programme Windows !

Création d'un programme de console "Hello World"

Nous allons écrire un programme qui se charge à partir du prompt MS-DOS (commande) et affiche « Hello World » à l'écran. Il s'agit d'une version identique mais notablement plus commentée de **HelloWorld1.asm** décrit en annexe A. Son but essentiel est de décrire l'appel d'une API Windows. Ce type de programme est dit *de console* car il ne produit pas de fenêtres et, par conséquent, ne fait aucun usage de l'interface utilisateur graphique (connue sous l'acronyme « GUI »).

Le processus est le même que précédemment sauf que, cette fois, vous allez nommer le script source HelloWorld.asm, et le fichier batch, GoHello1.bat. N'oubliez pas de changer les instructions à destination de l'assembleur et de l'éditeur de liens dans le fichier batch de sorte qu'ils traitent les bons fichiers.

Action	**Explication**

Créez un nouveau fichier et tapez ces lignes :
`DATA SECTION`
`RCKEEP DD 0`
	La ligne "DATA SECTION" indique à l'assembleur que les instructions qui suivent sont des déclarations de données. "RCKEEP"est un *label de donnée*, autrement dit un nom indiquant un emplacement particulier dans les données.
	«DD» crée un DWord (4 octets) dans les données (littéralement "Déclare-Dword").
	"0" initialise le DWord à la valeur, zéro dans ce cas.

Ajoutons maintenant ces 4 lignes :
`CODE SECTION`
`START:`
`PUSH -11`
`CALL GetStdHandle`
	"CODE SECTION" indique à l'assembleur que les lignes qui suivent sont des instructions de code. "PUSH" met une valeur sur la pile. Celle-ci, dans notre cas, est –11 (exprimée en valeur décimale signée). Cette instruction est le seul paramètre à pousser en pile dans le cadre de l'appel de l'API *GetStdHandle*. Cette valeur est donnée par Microsoft et correspond à l'activation du buffer d'écran de la console active. Cette console est l'interface avec l'utilisateur, qui dans ce cas sera la fenêtre MS-DOS (également connue sous le nom d'"invite de commande".)
	Le "CALL" transfère le paramètre à une procédure, dans ce cas à l'API Windows *GetStdHandle*. Cette API reçoit ce paramètre et renvoie, en retour, un handle dans le registre EAX qui permettra d'écrire dans la fenêtre via une autre API.

Maintenant, ces 4 autres lignes :
`PUSH 0, ADDR RCKEEP`
`PUSH 24D, 'Hello World (from GoAsm)'`
`PUSH EAX`
`CALL WriteFile`
	Ici, nous poussons en pile les 5 paramètres nécessaires à l'appel de l'API Windows *WriteFile* avant d'appeler cette dernière. *WriteFile* écrit dans le handle qui lui est passé dans le dernier paramètre (toujours détenu par le registre EAX). Concrètement, elle écrit la chaîne qui est de 24 caractères. Nous donnons également l'adresse de RCKEEP qui est le dword de données déclaré plus tôt. Ceci est nécessaire car la fonction *WriteFile* répercutera dans ce dword le nombre de caractères que'elle a effectivement écrits.

Et enfin, ces 2 lignes en guise de conclusion :
`MOV EAX,0`
`RET`
	Ici, nous passons la valeur 0 dans le registre EAX pour notifier une fin de programme réussie et utilisons ensuite RET pour retourner au système (fermeture du programme).

Maintenant, vous pouvez assembler ce fichier et en procéder à l'édition des liens, en utilisant les mêmes techniques que précédemment, mais avec un léger changement. Votre fichier batch s'écrira désormais :
`GoAsm HelloWorld1`
`GoLink HelloWorld1.obj /console kernel32.dll`
Ici, au-delà des différents noms de fichiers dont la présence va de soi, vous avez inclus "kernel32.dll" dans la ligne de commande de GoLink. Cet insert commande à GoLink d'explorer le fichier système de Windows kernel32.dll lors de la l'édition de liens du fichier pour toutes les «inconnues», et notamment, les API Windows *GetStdHandle* et *WriteFile* qui sont indispensables à l'exécution du programme.
Enfin, l'exécution de ce dernier se traduira par l'affichage de la chaîne « Hello World » en regard de l'invite de commande de la fenêtre MS-DOS.	

Vous savez maintenant comment créer un programme simple en utilisant l'assembleur et le système d'exploitation Windows.

Je vous suggère de lire également avec attention la plupart des annexes qui suivent et notamment :

Annexe C – Pour les débutants... en programmation
Annexe D – Représentations binaires
Annexe E – Pour les débutants... en langage assembleur
Annexe F – Flags, sauts conditionnels, CMOVcc et SETcc
Annexe G – Pour les débutants... en Windows
Annexe H – Pour les débutants... en débogage symbolique
Annexe I – Comprendre... la Pile (prioritairement la partie n° 1)
Annexe J – Comprendre... la mémorisation inversée

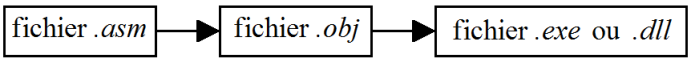
Je vous recommande enfin de porter une attention particulière aux scripts sources commentés de la série Hello World dans l'**annexe A**. Ensuite, vous pourrez passer aux sujets relevant du niveau intermédiaire et vous confronter à l'ensemble du présent manuel.

Annexe C
Pour les débutants... en programmation

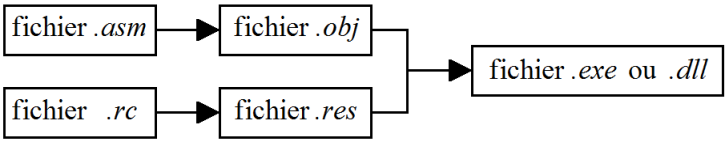
L'une des choses les plus obscures au commun des mortels est d'appréhender l'écriture et la création d'un programme que l'ordinateur soit en mesure d'exécuter. Dans l'annexe B “Écriture d'un programme Windows élémentaire” qui précède, nous avons levé une partie du mystère à ce sujet et vu notamment à quel point cet objectif était relativement simple à tenir, en particulier avec les outils de développement de l'assembleur. Mais ce n'était qu'un bref aperçu. Nous allons nous attacher, maintenant, à décrire le processus de construction des programmes et à examiner les fichiers impliqués de manière plus détaillée.

Construction d'un programme

Le process de programmation Windows utilisant l'assembleur SANS ressources est :



Le process de programmation Windows utilisant l'assembleur AVEC ressources est :



Les *ressources* sont archivées et traitées séparément dans le but d'accroître la facilité de programmation, et détiennent principalement le contenu des menus et des dialogues, mais aussi des composants tels que des bitmaps, des curseurs et des icônes. Normalement, un programme de *console* ne contient pas de ressources car il ne fait aucune utilisation des capacités graphiques de Windows. En revanche, un programme qui recourt à l'interface graphique Windows GUI ("Graphics User Interface") utilisera assez souvent les ressources bien que cela ne soit absolument pas nécessaire.

Le fichier *.asm*

Le fichier ASM est un fichier que vous créez et éditez en utilisant un éditeur de texte ordinaire, comme Paws que vous pouvez télécharger à partir de mon site web, **www.GoDevTool.com**, ou de programmes comme Notepad (Bloc-notes) ou Wordpad qui sont livrés avec Windows. Si vous utilisez ces deux derniers, vous devez vous assurer que vous enregistrez le fichier dans un format qui n'ajoute pas de caractères de contrôle ou de formatage autres que l'habituelle fin de ligne (retour chariot et saut de ligne). Ceci, parce GoAsm ne s'intéresse qu'au texte brut. Vous pouvez vous prémunir contre ces caractères non désirés en sauvegardant le fichier comme document «texte». Si vous n'adjoignez pas une extension au nom de fichier (l'extension désigne les caractères après le point), alors l'éditeur peut lui attribuer automatiquement une extension ".txt". Cependant, rien ne vous empêche de la changer en renommant le fichier (vous pouvez exécuter cette opération sur l'Explorateur Windows en pratiquant un clic-droit sur le nom et en sélectionnant la fonction “Renommer”).

Il se peut que vous ne puissiez visualiser l'extension du fichier sur votre ordinateur. Il s'agit, en ce cas, d'une question de paramétrage de l'Explorateur Windows. Pour ce faire, sélectionnez l'élément de menu "Affichage", "Options", "modifiez les options des dossiers et de recherche" puis sur l'onglet "Affichage" et, enfin, veillez à ce que la case "masquer les extensions des fichiers dont le type est connu" soit décochée. La procédure peut différer légèrement selon la version de Windows.

Il est de tradition chez les programmeurs d'attribuer à leurs scripts source une extension qui corresponde au langage dans lequel il est écrit. Par exemple, vous pourriez avoir un fichier assembleur appelé “myprog.asm”. De la même manière, vous trouverez généralement le code source écrit en langage “C” avec l'extension “.c” ou “.cpp” (pour “C ++”), “.pas” pour Pascal et ainsi de suite. Cependant, ces extensions

sont totalement neutres d'un point de vue strictement informatique. GoAsm accepte ainsi les fichiers de toute extension de même que les fichiers qui en sont dépourvus.

Le fichier *.asm* contient vos instructions pour le processeur en mots et nombres. Celles-ci sont converties en code exécutable successivement par l'assembleur puis par l'éditeur de liens. C'est ce code qui sera reconnu et exécuté par le processeur. On dit donc que le fichier *.asm* contient votre "code source" ou votre "script".

Lorsque vous que votre programme est prêt, vous passez le fichier *.asm* à l'Assembleur qui est un programme qui convertit les fichiers *.asm* en fichiers.*obj*. Vous pouvez utiliser, par exemple, mon assembleur GoAsm.

Le fichier *.obj*

C'est le fichier constitué par l'assembleur à partir d'un fichier *.asm*. L'assembleur prend les instructions dans le fichier *.asm* qui sont dans les mots et les nombres et les convertit dans le format d'objet COFF qui est le format attendu par l'éditeur de liens. L'assembleur concatène toutes vos instructions de *code* et de *donnée* dans le script source et les organise en sections de code et de données dans le fichier *.obj*. La section de code contient les instructions réelles de processeurs («opcodes») que le processeur exécute lorsque le programme est lancé. La section data contient des informations qui seront conservées en mémoire pendant que le programme est exécuté.

Vous ne pouvez pas réellement exécuter un fichier *.obj* en tant que programme car il n'est pas dans le format final de programme attendu par Windows (le format PE). Pour y parvenir, vous devez soumettre le fichier *.obj* à l'*éditeur de liens* (ou *linker*) dès lors que vous êtes prêt à finaliser votre programme. Parfois, le fichier *.obj* est qualifié de fichier «binaire» ou «bin» dans un souci de simplification. En effet, le fichier peut bien être considéré comme ne contenant maintenant que des chiffres. Mais si vous regardez à l'intérieur d'un fichier *.obj* vous y verrez toujours les chaînes de caractères qui étaient dans votre script source.

Le fichier *.rc*

Il s'agit ici d'un autre type fichier texte que vous constituez en utilisant un éditeur de texte. Ce fichier rassemble des instructions d'un format spécifique sous forme de mots et de chiffres que Windows utilise pour constituer les *ressources* de votre programme lorsqu'il est exécuté. Entrent notamment dans cette catégorie la plupart des menus, boîtes de dialogue et tables de chaînes. Vous pouvez également utiliser le fichier *.rc* pour nommer les fichiers qui doivent être chargés dans le fichier *.res* lorsque le compilateur de ressources est exécuté (comme, par exemple, des icônes et des curseurs bitmaps). Par conséquent, le fichier *.rc* peut correctement être décrit comme contenant de la *matière source*. Vous pouvez trouver plus d'informations sur les ressources dans le chapitre consacré à mon compilateur de ressources GoRC dans le volume 2.

Le fichier *.res*

Ce fichier est produit par le compilateur de ressources GoRC à partir d'un fichier *.rc*. Le compilateur de ressources formate les instructions dans le fichier *.rc* qui sont sous forme de mots et de nombres et les convertit en une forme prête pour l'insertion dans la section de ressources dans le fichier *.exe* final. Vous passez le fichier *.res* dans l'éditeur de liens lorsque vous êtes prêt à finaliser votre programme.

Le fichier *.exe*

C'est le fichier *exécutable* final qui peut être exécuté en tant que programme par Windows. Il est dans le format Portable Executable (PE). Il est constitué par l'éditeur de liens qui traite un ou plusieurs fichiers *.obj* et, éventuellement, un fichier *.res* et les combine dans le fichier *.exe* final. Le format PE exige également que le fichier *.exe* ait un en-tête avec des informations sur le fichier *.exe*. L'éditeur de liens fournit cette information. Pour élaborer les programmes Windows, vous aurez besoin d'utiliser d'un éditeur de liens capable de faire des fichiers PE comme, par exemple, GoLink décrit dans le volume 2.

Le fichier *.dll*

Il contient d'autres fonctions et données que votre fichier *.exe* peut utiliser quand il est en cours d'exécution. La plupart du temps vous n'aurez absolument pas besoin d'utiliser un fichier *.dll*. Toutefois, un tel fichier peut se révéler utile s'il contient des fonctions qui doivent être appelées par plus d'un fichier *.exe*. Au lieu de dupliquer le même code dans deux fichiers *.exe*, vous pouvez ne l'avoir qu'une seule fois dans un fichier

.dll. Windows utilise les fichiers *.dll* à grande échelle pour fournir aux applications Windows un accès aux API. Prenez le temps de consulter le répertoire Windows\system dans votre ordinateur – c'est là que les DLLs de Windows sont normalement entreposées.

A noter enfin que le volume 2 propose un chapitre détaillé sur le sujet.

Organisation de votre travail de programmation

Voici quelques suggestions concernant l'organisation de votre travail de programmation. Il existe d'autres moyens, y compris l'utilisation d'un IDE (Integrated Development Environment) pour exécuter les différents outils au mieux de vos intérêts. Mais si vous débutez, vous trouverez probablement plus facile de vous rallier, dans un premier temps, aux suggestions qui vont suivre, quitte à en changer au fur et à mesure de l'évolution de vos besoins et de vos idées.

Cela nécessite un effort de mise en place, mais une fois ce dernier accompli vous aurez en main une formule souple qui pourra être utilisée à maintes reprises et facilement modifiée selon les besoins.

Tout d'abord, il convient d'entreposer tous les travaux de programmation et les fichiers correspondants sur le disque dur dans un dossier spécifique qui pourrait se nommer PROG. Bref, d'un nom, qui permette de le séparer des autres travaux sur votre ordinateur.

Ensuite, concernant le démarrage d'un nouveau projet, je vous suggère de créer un nouveau sous-dossier de ce projet dans l'explorateur Windows. Conservez-y les fichiers sources du projet ainsi que tous les fichiers créés à partir de ces fichiers source dans le sous-dossier. Donc tous les fichiers.*asm*, *.rc*, *.obj*, *.res* et *.exe* relatifs au projet devront y prendre place.

Entreposez l'assembleur, le compilateur de ressources et l'éditeur de liens dans un autre sous-dossier dans le dossier PROG.

Ainsi, par exemple, si votre projet est appelé "Myprog", vos fichiers seront organisés comme suit :

- Dans le répertoire c:\prog\myprog – Myprog.asm, Myprog.rc, Myprog.obj, Myprog.res et Myprog.exe.
- Dans le répertoire c:\prog\utils – GoAsm.exe (l'assembleur), GoRC.exe (le compilateur de ressources) et GoLink.exe (l'éditeur de liens).

Ensuite, vous devez concevoir deux fichiers *batch* qui automatisent l'assemblage, la compilation et le processus d'édition de liens. Un fichier batch est un fichier texte ordinaire avec l'extension ".bat". Quand un tel fichier est exécuté à partir d'une ligne de commande MS-DOS ("invite de commande"), il exécute chaque ligne l'une après l'autre, comme si chacune avait été tapée séparément sur la ligne de commande. Les fichiers batch doivent être écrits en utilisant l'éditeur de texte et être entreposés dans le répertoire c:\prog\myprog. Outre les détails donnés ci-après, on trouvera en **Annexe M** un aperçu aussi complet que possible de ces fichiers.

Si vous mettez en œuvre des ressources, le premier fichier batch pourra être appelé "Gorc.bat" et aura la tâche de lancer le compilateur de ressources afin que le fichier Myprog.rc soit converti en Myprog.res. Dans ce cas, il y aura une seule ligne dans le fichier de commandes libellée comme suit :

```
c:\prog\utils\GoRC /r Myprog.rc
```

Si vous utilisez ce fichier et tapez "Gorc" puis Entrée à partir de la ligne de commande, ce sera comme si la ligne de commande complète avait été saisie manuellement.

Le fichier batch principal peut être appelé "Go.bat". Il déclenche l'assemblage de Myprog.asm puis confie le fichier *.obj* résultant et le fichier *.res* à l'éditeur de liens qui convertit le tout en un fichier *.exe*. Dans ce cas, le fichier batch comportera deux lignes :

```
c:\prog\utils\GoAsm Myprog.asm
c:\prog\utils\GoLink @command.fil
```

La première ligne initie l'assemblage qui se traduit par la création du fichier Myprog.obj.

La deuxième ligne active l'éditeur de liens. Vous pouvez voir que nous utilisons ici un fichier nommé "command.fil". Un tel fichier contient des instructions à destination de l'éditeur de liens. L'utilisation d'un fichier de commandes facilite le changement éventuel des paramètres passés à l'éditeur de liens. Vous constituez ce fichier en utilisant l'éditeur de texte. Encore une fois, il devra être dans le répertoire "c:\prog\myprog".

Le fichier de commandes peut contenir, par exemple, les lignes suivantes :

```
/debug coff
Myprog.obj
Myprog.res
Kernel32.dll
```

La première ligne est le commutateur `/debug coff` qui demande à l'éditeur de liens d'inclure les symboles dans l'exécutable final. Cela vous permettra notamment d'utiliser ultérieurement le débogueur symbolique GoBug. Les deux lignes suivantes nomment les fichiers d'entrée, dans ce cas, le fichier objet puis le fichier *.res* contenant les ressources (omettre la ligne *.res* si vous n'utilisez aucune ressource). La dernière ligne invite l'éditeur de liens à explorer le fichier système Kernel32.dll pour y identifier les appels d'API effectués dans le programme. Il se pourrait que votre programme effectue des appels en direction d'API localisées dans d'autres DLL système. Si tel est le cas, il est impératif que les DLL concernées soient nommées. Vous pouvez identifier celles-ci dans l'information sur Windows contenue dans le Software Development Kit, ou dans les fichiers d'en-tête de Windows. Habituellement, la plupart des appels d'API dont vous aurez besoin sont contenus dans une ou plusieurs des DLL suivantes (en plus de Kernel32.dll), qui devront, dans ce cas, être mentionnées dans le fichier de commandes :

```
user32.dll
gdi32.dll
COMCTL32.dll
COMDLG32.dll
OLEAUT32.dll
Winspool.drv
Hhctrl.ocx
```

L'éditeur de liens peut fonctionner plus rapidement si vous pouvez réduire la liste des DLL.

Note : les travaux ci-dessus concernent exclusivement GoLink. Avec d'autres éditeurs de liens, vous serez contraints d'utiliser des fichiers "lib" pour identifier la DLL pertinente. Si les fichiers lib ne sont pas disponibles, vous devez les constituer à partir de la DLL au moyen d'un outil spécial.

Lancement de l'Assembleur, du Compilateur de Ressources et de l'Éditeur de liens

Il existe deux façons d'exécuter les fichiers batch que vous avez créés :

1. La première est de double-cliquer sur l'icône de fichier appropriée dans l'Explorateur Windows (qui ouvre automatiquement une fenêtre de console pour donner les résultats).

2. La seconde est d'ouvrir vous-même une fenêtre MS-DOS (invite de commande) et d'exécuter les fichiers à partir de là. Pour ce faire, cliquez sur Démarrer, Programmes, invite MS-DOS. Dans XP c'est *démarrer*, *tous les programmes*, *accessoires*, *invite de commande*. Vous verrez alors probablement l'invite "C:\WINDOWS>". Cela signifie que le répertoire actif est le dossier "Windows" dans le lecteur c :. Modifiez ce répertoire en tapant "cd c:\prog\myprog" puis en validant avec la touche Entrée. L'invite doit maintenant afficher "C:\prog\myprog>". Pour la suite, on suppose que vos fichiers source sont en ordre que vous êtes maintenant prêt à exécuter les fichiers batch. "Gorc" lancera le compilateur de ressources (si vous l'utilisez), "Go" assemblera et procèdera à l'édition de liens du programme. Enfin "Myprog" va exécuter le fichier EXE.

Voir le chapître consacré au compilateur de ressources GoRC dans le volume 2 pour voir comment faire un fichier RC.

Voir, dans le présent volume 1, comment élaborer un fichier de travail *.asm*.

Voir le chapître consacré à l'éditeur de liens GoLink dans le volume 2 pour plus de précisions sur son utilisation.

Enfin, il existe plusieurs exemples de fichiers sur mon site que vous pouvez consulter et utiliser, en particulier les fichiers HelloWorld (également en annexe A de ce document) et les fichiers d'aide de Testbug.

Annexe D
Représentations binaires

D1 Systèmes de numération

Un système de numération est un ensemble de symboles permettant de représenter des nombres. Le système décimal que nous connaissons bien utilise 10 symboles (ou chiffres) : 0, 1, 2, 3, 4, 5, 6, 7, 8 et 9. Mais il est possible d'en imaginer d'autres tels que, par exemple, le binaire, l'octal, l'hexadécimal, etc.

Pour cela, chaque système de numération possède une base *b* à partir de laquelle il est possible d'écrire tous les nombres sous la forme suivante :

$$\text{Valeur décimale} = a_n \times b^n + a_{n-1} \times b^{n-1} + \cdots + a_1 \times b^1 + a_0 \times b^0$$

Dans laquelle : b = base (10 pour décimal, 2 pour binaire, 8 pour octal, 16 pour hexadécimal)
$0 \leq a_n < b$

Partant de là :

- le nombre décimal 193 s'écrit : $(1 \times 10^2) + (9 \times 10^1) + (3 \times 10^0) = 193$
- le nombre binaire 1101 s'écrit : $(1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 13$
- le nombre hexadécimal A57C s'écrit : $(10 \times 16^3) + (5 \times 16^2) + (7 \times 16^1) + (12 \times 16^0) = 42\ 364$

D2 Bits et représentation binaire

Un bit est un élément électrique d'une puce électronique de l'ordinateur qui peut être soit "on" ou "off". En termes de physique, nous avons affaire à une jonction de semi-conducteur qui est en mesure soit de produire une tension électrique quand elle est "on", soit de ne pas en produire du tout lorsqu'elle est "off". A l'état "on", elle est considérée comme ayant la valeur 1. En langage informatique, on dit alors que le bit est "mis" (traduction de "set"). Quand elle est "off", elle est considérée comme ayant la valeur zéro. En langage informatique le bit est donc mis à zéro («cleared»). Les bits peuvent être mis (à 1) ou à zéro (0) mais ne peuvent avoir d'autre état. La présence de 2 états suggère, bien évidemment, une représentation limitée à 2 symboles (0 et 1), d'où le recours naturel à la représentation binaire.

Deux ou plusieurs bits peuvent être associés pour représenter un nombre plus élevé. Lorsque des bits sont ainsi combinés, le bit sur la droite est le moins significatif et, s'il est "mis", il représente la valeur 1. Le bit immédiatement à gauche du précédent est plus significatif à un facteur 2 près. Lorsque ce bit est "mis" il représente la valeur 2. Voyons tout de suite un exemple : supposons que vous ayez un nombre formé de deux bits. Dans ce cas, il peut s'exprimer en binaire par 00, 01, 10 ou 11, correspondant en décimal respectivement à 0, 1, 2 ou 3.

D3 Octets, mots, double-mots, quadruple-mots et autres...

Les octets (bytes), mots (words), double-mots (dword) et quadruple-mots (qwords) sont des structures de donnée standard – au sens large du terme – utilisées en programmation. Le processeur fonctionnera avec la taille de donnée appropriée selon l'instruction en cours d'exécution.

Représentation des nombres entiers

Si n est le nombre de bits d'une donnée, le nombre maximum représentable correspond à $2^n–1$ en base décimale. Il s'agit, en l'occurrence, de nombres entiers en arithmétique non signée.

Parmi les différents formats les plus usuels, un octet contient 8 bits, un mot en contient 16 (2 octets), un dword, 32 bits (4 octets) et un qword, 64 bits (8 octets). Le tword de 80 bits (10 octets) existe également mais au niveau de l'unité arithmétique du processeur. Il est qualifié de "format en double-précision étendu". Enfin, les processeurs actuels possèdent quelques registres 128 bits (16 octets) et certaines instructions sont à même de manipuler directement des données de 128 bits.

Les formats les plus courants affichent les caractéristiques suivantes :

Format	Nb de bits	Nb d'octets	Valeur entière max (non signée)
Octet (byte)	8	1	$2^8–1 = 256$
Mot (word)	16	2	$2^{16}–1 = 65\,535$
Double-mot (dword)	32	4	$2^{32}–1 = 4\,294\,967\,295$
Quadruple-mot (qword)	64	8	$2^{64}–1 = 18\,446\,744\,073\,709\,551\,615$

On notera qu'il est courant de parler, pour simplifier le propos, de 64 kilo-octets (64 Ko) de capacité pour un mot et de 4 Gigaoctets (4 Go) pour un double-mot bien que les valeurs exactes affichent une différence sensible.

Tous les formats que nous venons de voir sont employés aussi en arthmétique signée. Le principe est simple : le bit de plus fort poids est dédié au signe (1 = négatif, 0 = positif). Corrélativement, la gamme de valeurs absolues quantifiables se voit divisée par 2.

Représentation des nombres en virgule flottante

Le processeur ne peut se cantonner à ne gérer que des nombres entiers et doit assez souvent recourir aux formats en virgule flottante. Ici, nous entrons dans un autre monde où les mêmes formats utilisés précédemment emploient désormais la structure arithmétique classique de la forme :

$$Nb = mantisse \times 2^{exposant}$$

Cette représentation est régie par la norme IEEE 754 relative aux nombres en virgule flottante. Ses principes de base peuvent être résumés dans le tableau suivant[3] :

Type de donnée	Longueur (bits)	Précision (bits)	Champ de définition approximatif des nombres	
			Binaire	Décimal
Demi-Précision	16	11	2^{-14} à 2^{15}	$3{,}1 \times 10^{-5}$ à $6{,}50 \times 10^{4}$
Simple-Précision	32	24	2^{-126} à 2^{127}	$1{,}18 \times 10^{-38}$ à $3{,}40 \times 10^{38}$
Double-Précision	64	53	2^{-1022} à 2^{1023}	$2{,}23 \times 10^{-308}$ à $1{,}79 \times 10^{308}$
Double-Précision Étendue	80	64	2^{-16382} à 2^{16383}	$3{,}37 \times 10^{-4932}$ à $1{,}18 \times 10^{4932}$

D4 Nombres hexadécimaux

Avant d'aller plus avant dans la représentation numérique hexadécimale, attardons-nous un instant sur une subdivision importante de l'octet : le *quartet* qui est en fait un demi-octet. Ce quartet contient quatre bits de données. Il se prête particulièrement bien, on le voit, à la numération hexadécimale puisqu'il peut représenter l'ensemble des nombres allant de 0 à 15. Dans ce cas, il est d'ailleurs d'usage de représenter les nombres 10 à 15 repectivement par les lettres A à F. Considérons par exemple le nombre hexadécimal 148C qui occupe deux octets. Le premier octet contient la valeur de 14h et le second 8Ch. Les 4 quartets contiennent respectivement les valeurs 1, 4, 8 et 0Ch.

Les programmeurs utilisent la représentation hexadécimale pour de multiples raisons. C'est un moyen pratique de représenter un nombre de manière plus compacte mais néanmoins compréhensible. La modularité par quartet permet de visualiser, du premier coup d'oeil, le format d'un nombre en tant qu'octet, word, dword ou qword. Elle permet également au programmeur de scruter certains bits et d'évaluer immédiatement leur état ce que n'autorise pas une représentation décimale. Enfin, leur usage facilite la compréhension et la mise en oeuvre d'instructions logiques telles que OR, AND, XOR, TEST ou BT, par exemple.

Les nombres hexadécimaux (ou Hex en abréviation) sont ainsi nommés parce qu'ils sont en base 16. Chaque symbole hexadécimal utilise les chiffres 0 à 9 ou les lettres A, B, C, D, E ou F, ces dernières caractérisant respectivement les valeurs 10, 11, 12, 13, 14 ou 15. Chaque symbole hexadécimal occupe quatre bits de données binaires.

[3] Tableau traduit du document "Intel® 64 and IA-32 Architectures – Software Developer's Manual" – Volume 1: Basic Architecture – Order Number: 253665-055US (June 2015)

Voici les valeurs qui peuvent être créées à partir de quatre bits et, pour chacune, leur équivalent en hexadécimal et en décimal :

binaire	**hex**	**décimal**
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Un octet peut être représenté par la juxtaposition de deux chiffres hexadécimaux, un mot par quatre chiffres hexadécimaux et un dword par huit chiffres hexadécimaux. Le tableau qui suit vous permet de mesurer le réel atout de la notation hexadécimale lorsque l'on représente de grands nombres qui deviennent dès lors difficiles à manier sous leur format décimal :

binaire	**hexa**	**décimal**	
10000000	80	128	**(byte)**
1000000000000001	8001	32 769	**(word)**
1111111111111111	FFFF	65 535	**(word)**
10000000000000000000000000000001	80000001	2 147 483 649	**(dword)**
11111111111111111111111111111111	FFFFFFFF	4 294 967 295	**(dword)**

D5 Nombres finis, négatifs, signés et en complément à 2

Dans le monde des mathématiques, les nombres sont censés pouvoir prendre des valeurs infiniment grandes.

Dans le monde de l'informatique, les nombres gérés par les ordinateurs sont nécessairement limités parce que la taille du bloc de données mis à disposition du stockage et du calcul est finie. Par exemple, un octet ne peut contenir un nombre supérieur à 255. Si, lors d'une opération sur un octet, vous ajoutez 1 à 255, le résultat devient nul. Dans ce cas particulier, fort heureusement, l'instruction positionne le flag de retenue à 1 indiquant de la sorte que le nombre résultant était trop grand pour le format de donnée utilisé. Vous pouvez savoir ainsi que le résultat n'était pas nul, mais égal en fait à 256.

Étant donné que la taille des données est finie, il est possible de considérer les données comme ayant deux valeurs à un moment donné, une valeur positive et une valeur négative.

Voici où je veux en venir : supposons que vous ayez un octet dont tous les bits sont à 1. Vous pouvez logiquement penser qu'il contient le nombre 0FFh (ou 255 en décimal). Mais vous pourriez également considérer qu'il s'agit du nombre –1, dont on peut dire qu'il correspond à la soustraction 256–1. Vous pouvez d'ailleurs prouver qu'il est correct d'assimiler ce nombre à -1 parce que si vous lui ajoutez 1, le résultat est nul, ce qui est correct au regard de la taille des données concernées (bien qu'il y ait une retenue).

En jargon de programmeur, le seul fait de considérer que le bloc de données puisse contenir un nombre négatif signifie que le nombre appartient *de facto* à la catégorie des "nombres signés". On parle aussi d'arithmétique signée dans ce cas. Que le nombre soit négatif ou non dépend alors de l'état du bit de plus fort poids (le plus à gauche) dans le bloc de donnée, qui est appelé le "bit de signe". Si ce bit est à 1, nous sommes en présence d'un nombre signé négatif. Par exemple, supposons qu'un octet contienne la valeur 82h. Si nous sommes censés évoluer en arithmétique signée, le bit de signe est à 1 de sorte que le nombre est en réalité –7Eh (ou –126 décimal) car résultant de la soustraction 100h–82h. De même, une valeur de 81h dans le même contexte correspond à –7Fh (ou –127 en décimal). Ou encore, une valeur de 80h corres-

pond à –80h (ou –128 en décimal). En revanche, une valeur de 7Fh caractérise avec certitude le nombre positif +7Fh puisque le bit de signe est présentement nul et ceci, que nous soyions en arithmétique signée ou non.

Le bit de signe est toujours celui qui se situe le plus à gauche. C'est donc le bit 7 dans un octet, le bit 15 dans un mot, le bit 31 dans un dword et le bit 63 dans un qword. Cette numérotation suppose que le bit le plus à droite soit le bit 0 dans tous les cas, ce qui correspond du reste à un usage largement répandu.

Au terme de l'exécution de la plupart des instructions arithmétiques, le **flag de signe** est positionné à 1 si le bit de signe est lui-même à 1. Evidemment, ce flag peut être testé pour voir si le résultat est négatif après que l'instruction a été exécutée.

Remarquons que, dans un octet en arithmétique signée, le nombre 0FFh correspond à –1h, 0FEh correspond à –2h, 0FDh correspond à –3h et ainsi de suite. On voit donc que cette séquence, bien que *décroissante* en apparence, est en réalité une progression si l'on considère la valeur absolue des nombres négatifs équivalents. Seule l'expérience en matière de manipulation de nombres hexadécimaux permet de saisir cette dualité au premier coup d'oeil.

Une façon simple de calculer la valeur absolue d'un nombre négatif (affichant donc un bit de signe à 1) est d'effectuer son complément en inversant tous ses bits, puis d'ajouter 1 au résultat obtenu. Voici une illustration de ce calcul :

```
0F0h en binaire est  11110000
 son complément est  00001111  → complément à 1
        qui est      0Fh
  on ajoute 1, soit  10h       → complément à 2
ce qui correspond à  16 en décimal
```

La première étape consistant à inverser les bits est nommée « complément à 1 ». L'ajout d'une unité à ce résultat permet d'obtenir ce que l'on appelle le « complément à 2 ». C'est la raison pour laquelle les nombres signés sont parfois dits en « complément à 2 ».

L'avantage décisif de ce système de numération est que beaucoup d'instructions processeur peuvent être utilisées à la fois pour les nombres signés et non-signés. Il appartient de ce fait au programmeur, et à lui seul, de définir le statut des nombres utilisés. Il y a quelques instructions, cependant, pour lesquelles cette polyvalence n'existe pas et, notamment les instructions de multiplication, de division et de décalage de bits à droite. Pour les deux premiers cas, le processeur propose des instructions de multiplication – IMUL – et de division – IDIV – spécialement dédiées à l'arithmétique signée à utiliser en lieu et place de MUL et DIV qui sont les versions non signées. Il y a trois raisons pour lesquelles cette distinction est nécessaire :

- En premier lieu, lors de la manipulation de nombres signés, le signe du résultat dépend de celui des opérandes. On connaît la règle, applicable à la fois à la multiplication et à la division : lorsque les 2 membres de l'opération sont de même signe, le résultat est positif ; lorsqu'ils sont de signe contraire, le résultat est négatif. Les instructions IMUL et IDIV reportent le bon signe sur le résultat alors que MUL et DIV sont incapables de le faire et ne peuvent opérer que sur des nombres positifs.
- En second lieu, le format des opérandes d'instructions de multiplication et division ainsi que celui du résultat de ces mêmes opérations ont, par nature, un gabarit différent. Par exemple, le résultat de la multiplication de 2 opérandes de 32 bits est restitué par la paire de registres EDX:EAX qui forme un ensemble de 64 bits. Si la multiplication met en oeuvre des opérandes signés, il est nécessaire que le signe du résultat soit positionné sur son bit 63 (ou le bit 31 du registre EDX dans ce cas particulier). Dans le cas de la division 32 bits, le dividende est présumé à 64 bits et hébergé par la paire de registres EDX:EAX si le diviseur est un registre ou un espace mémoire de format 32 bits. Le quotient est recueilli par le registre EAX et le reste, dans le registre EDX. Le signe du dividende sera sur le bit 31 de EDX ; celui du diviseur sera sur le bit 31 du registre ou emplacement mémoire qui l'héberge. Enfin, celui du quotient et du reste sera respectivement sur le bit 31 des registres EAX et EDX. On voit ainsi que les instructions signées veillent à ce que le bit de signe soit à la bonne place. On parle, dans ce cas, d'"extension de signe" d'un nombre.
- Enfin, comme nous allons le voir, les instructions signées garantissent que les flags de *carry* (retenue) et d'*overflow* (dépassement) sont correctement positionnés selon le signe du résultat.

Dans le cas de l'instruction décalage à droite SHR, il existe une variante spéciale (SAR) qui ne modifie pas le bit de plus fort poids de l'opérande tout en maintenant le bon signe dans le résultat. Une telle précaution

n'est pas nécessaire pour les instructions de décalage à gauche puisque le signe s'auto-corrige. Pour autant, il existe une instruction SAL pendante de SHL mais qui a, en réalité le même code processeur.

Cet exemple montre ce qu'il advient concrètement :

```
MOV AL,0FEh      ; met -2 dans AL
SAR AL,1         ; divise AL par 2 - le résultat dans AL est maintenant 0FFh, soit -1
MOV AL,0FEh      ; met -2 dans AL
SAL AL,1         ; multiplie par 2 - le résultat dans AL est maintenant 0FCh, soit -4
```

Les nombres signés ont aussi leur propre jeu d'instructions de **saut conditionnel signé**.

Annexe E
Pour les débutants... en langage assembleur

Qu'est-ce que le langage assembleur ?

Le langage assembleur est un langage de programmation. Les programmes y sont écrits sous forme littérale à l'aide d'un simple éditeur de texte en vue de constituer le "code source", également nommé "script". Ce code source est lu puis traduit par un "Assembleur" en fichier objet ; ce dernier est à son tour traité par un "Linker" – ou éditeur de lien – pour constituer le programme exécutable final.
Une des caractéristiques du langage assembleur est que chaque ligne du code source ne contient habituellement qu'une seule instruction à destination du processeur. Par exemple `MOV EAX,EDX` copiera le contenu du registre EDX dans le registre EAX. Nous découvrons ici l'instruction "MOV" qui est appelée "mnémonique", contraction de "mnémotechnique". Concrètement, cette instruction déplace (MOV(e) en anglais) le contenu d'un registre ou d'un emplacement mémoire vers un autre registre ou emplacement mémoire. Quoi que l'on puisse penser de cette symbolique minimaliste, on conviendra aisément qu'elle est plus expressive qu'une succession de 2 ou 3 octets binaires impossibles à mémoriser par un cerveau normalement constitué.
L'instruction, telle que nous venons de la découvrir, peut sembler assez basique si vous êtes habitué à un langage de haut niveau. Toutefois, lorsque vous commencerez à utiliser des API Windows dans vos programmes, vous côtoierez, sans vous en rendre vraiment compte, un langage de haut niveau (Windows lui-même). Ainsi, l'utilisation conjointe de Windows et de l'assembleur vous permettra-t-elle de manier un langage de haut niveau tout en conservant le contrôle total du processeur, autant dire LA combinaison parfaite !

Pourquoi un code de si bas niveau ?

Il y a plusieurs raisons pour lesquelles il peut être intéressant d'avoir recours à un langage de si bas niveau :

- Taille : en codant à ce faible niveau, vous pouvez réduire le code à un strict minimum inatteignable même par les compilateurs les plus performants.
- Vitesse : une taille de code réduite au strict nécessaire signifie que vos programmes se chargent et s'exécutent plus rapidement.
- Contrôle : vous contrôlez totalement le code, à la différence d'un compilateur. Cela signifie que vous savez exactement ce que le processeur est en train de faire lorsqu'il évolue à travers votre code. Cela permet d'obtenir un résultat probant et de repérer les erreurs plus facilement.
- Satisfaction : vous écrivez vous-même le programme. Il n'y a aucune intervention de quelque compilateur que ce soit.

Les instructions à votre disposition

Vous aurez besoin d'étudier les instructions qui constituent le cœur de votre assembleur et d'en assimiler parfaitement les tenants et les aboutissants. Cependant, si l'on s'attache aux plus essentielles dans le cadre d'une approche initiatique, on aboutit à la liste qui suit :

- **Instructions de registre**. Elles consistent à commander au processeur de déplacer des données ou effectuer des calculs en utilisant ses propres registres 32 bits. Il y a 6 ces registres à usage général appelés EAX, EBX, ECX, EDX, ESI et EDI. Des exemples de telles instructions sont :

```
MOV ESI,EBX         ; déplace le contenu du registre EBX dans le registre ESI
ADD EAX,EDI         ; additionne le contenu du registre EDI à celui du registre EAX
BT ECX,0            ; teste le bit 0 du registre ECX
CMP EDX,450         ; compare le contenu du registre EDX à la valeur 450
DIV ECX             ; divise EDX:EAX (entier long) par ECX
MUL ECX             ; multiplie EAX par ECX et place le résultat en EDX:EAX (entier long)
SHL EDX,4           ; décale les bits de EDX de 4 bits vers la gauche (multiplication par 16)
TEST EAX,8          ; teste le bit 3 du registre EAX (équivaut à un AND)
```

- **Instructions de pile**. La pile est une zone de mémoire fournie par Windows pour chaque programme en cours d'exécution destinée à être utilisée comme une zone de stockage temporaire. Des exemples de telles instructions sont :

```
PUSH EAX              ; pousse le contenu du registre EAX sur la pile
POP EDX               ; récupère de la pile dans EDX le dernier élément poussé sur la pile
PUSH 1000h            ; pousse la valeur hexadécimale 1000 sur la pile
MOV EBP,ESP           ; récupère la valeur courante du pointeur de pile dans le registre EBP
SUB ESP,30h           ; déplace le pointeur de pile pour réserver une zone de données locales
MOV D[EBP-20h],500h   ; insère la valeur hexadécimale 500 dans la zone de données locales
```

- **Instructions d'exécution**. Elles commandent au processeur d'interrompre l'exécution normale du flot d'instructions consécutives et de la poursuivre en un autre point du code. Les exemples d'instructions répondant à cette définition sont :

```
CALL MAKEWINDOW       ; exécute la procédure située au label MAKEWINDOW et revient ensuite
CALL EAX              ; exécute la procédure située au label dont l'adresse est contenue
                      ; dans le registre EAX et revient ensuite
RET                   ; achève une procédure et provoque un retour à l'appelant
JZ 4                  ; déplace l'exécution au label 4: si le résultat est nul
JC >.fin              ; déplace l'exécution au label .fin si le flag de Carry est à 1
JMP MAKEWINDOW        ; poursuit l'exécution au label MAKEWINDOW
LOOP 2                ; décrémente ECX puis effectue un saut au label 2: si ECX≠0
```

- **Instructions de mémoire**. Ces instructions, selon le cas, lisent une zone mémoire autre que la pile ou écrivent dessus. Typiquement, cette mémoire peut être dans la *section de données* (data section) propre de l'exécutable ou dans de la mémoire éventuellement allouée par Windows lors de l'exécution. En voici quelques exemples :

```
ADD EAX,[ESI]         ; additionne au contenu de EAX le contenu de la mémoire pointée par ESI
MOV EAX,[MYDATA]      ; copie dans EAX le contenu de la mémoire au label MYDATA
SUB D[MYDATA+64],10h  ; soustrait la valeur 10h au dword à l'adresse MYDATA plus 64 octets
CMP B[MYDATA+EDX*4],2     ; compare à 2 un octet situé dans le tableau MYDATA à l'offset EDX
LODSB                 ; copie dans AL l'octet de mémoire pointé par le registre ESI
STOSD                 ; copie le contenu de EAX dans le dword de mémoire pointé par EDI
```

- **Instructions de flags**. Les principaux indicateurs (ou flags) que vous allez utiliser sont Z (flag de zéro), C (flag de carry ou de retenue), S (flag de signe) et D (flag de direction). Ils sont dépositaires, en temps réel et automatiquement, du résultat de la plupart des opérations ou tests effectués par les instructions. A cela s'ajoutent certaines instructions spécifiques que vous pouvez utiliser pour modifier manuellement les flags du processeur :

```
STC                   ; met à 1 le flag de carry (retenue)
CLC                   ; met à 0 le flag de carry (retenue)
STD                   ; met à 1 le flag de direction pour LODS, STOS, CMPS, SCAS et MOVS
CLD                   ; met à 0 le flag de direction
```

- **Déclarations de mémoire**. Windows réserve de la mémoire à l'exécutable lors du démarrage de ce dernier. Les déclarations sont faites pour réserver de la mémoire dans la *section de données* ou la *section constante* si les données doivent être *initialisées*, c'est-à-dire qu'une valeur leur est assignée. Si les données ne sont pas destinées à être initialisées, la zone de données peut être réservée alors dans la *section de données non initialisées*. Dans ce cas, elle ne prend pas de place dans le fichier Exe. Au lieu cela, un espace dans la mémoire lui est allouée au moment du démarrage de l'exécutable.

  Quelques exemples de la façon dont la mémoire est déclarée (qui peut différer selon les assembleurs) :

```
DB 4                  ; déclare un octet et porte sa valeur initiale à 4
MYDATA DB 4           ; octet de valeur initiale 4 avec le label de donnée MYDATA
MYSTRUCT DD 16 DUP 0  ; 16 dwords tous mis à 0 et appelés MYSTRUCT
BUFFER DB 1024 DUP ?  ; 1024 octets nommés BUFFER en tant que données non initialisées
```

- **Déclarations de sections**. Instructions qui précisent à l'assembleur dans quelle section mettre le code source qui suit. L'assembleur considère la *section de code* en lecture seule et comme étant la seule exécutable. L'assembleur définit également des *sections de données* définies et non-définies en lec-

ture/écriture. Voici quelques exemples (des différences existent entre les différents assembleurs existants) :

```
CODE SECTION            ; tout ce qui suit cette déclaration est à marquer en
                        ;   lecture seule et exécutable (code)
DATA SECTION            ; tout ce qui suit cette déclaration est avec des attributs
                        ;  de lecture/écriture distincts des attributs de code
CONST SECTION           ; tout ce qui suit cette déclaration est dans une section
                        ;  avec des attributs en lecture seule
```

- **Commentaires**. Tout ce qui suit un point-virgule sera ignoré jusqu'à la fin de la ligne courante, vous permettant ainsi de décrire exactement, sous forme de commentaires, ce que votre code source est en train de faire et dans quel but.
- **Fonctions Windows**. Elles permettent au programmeur assembleur d'accéder à une vaste gamme d'API Windows (Applications Programming Interface). Il s'agit de code localisé dans le système d'exploitation Windows. En voici quelques exemples :

```
PUSH 12h                ; pousse en pile le code de la touche Alt sur le clavier
CALL GetKeyState        ; demande à Windows de mettre l'état de la touche Alt dans EAX
TEST EAX,80000000h      ; test si la touche Alt est pressée (bit 31=1 ?)
JZ >L22                 ; non, alors on va au label L22

.............

PUSH 24h                ; valeur hexa 24 = point d'interrogation + Boutons Oui et Non
PUSH ESI,EDI            ; adresse du titre, adresse du message
PUSH [hWnd]             ; handle de la fenêtre
CALL MessageBoxA        ; affiche la Message Box Windows avec la demande Oui/Non
CMP AL,7                ; On regarde si c'est le bouton Non qui a été cliqué par l'utilisateur
JNZ >L40                ; non, alors on va au label L40

.............

PUSH 0
PUSH ADDR FILE_DONE     ; on donne l'adresse de FILE_DONE pour y recevoir le résultat
PUSH ECX,EDX            ; ECX = nb d'octets à écrire, EDX= source de données,
PUSH ESI                ; ESI = handle du fichier
CALL WriteFile          ; écrit ECX octets de EDX vers ESI

.............

PUSH 808h,5h            ; 808 = bas et milieu rempli, 5 = élevé
PUSH EBX,EDX            ; EBX = RECT, EDX = contexte de périphérique
CALL DrawEdge           ; dessine un rectangle bordé spécial à l'écran

.............

PUSH 4h,3000h,ESI,0     ; 4h = mémoire en lecture/écriture, 3000h = réservation
CALL VirtualAlloc       ; réserve et engage ESI octets de mémoire en lecture/écriture

.............

PUSH 0,[hInst],0,0      ; param, handle du module, menu, personnel
PUSH 208,130,30,300     ; hauteur, largeur, y, x
PUSH 80C80000h          ; style (POPUP+CAPTION+SYSMENU)
PUSH EAX                ; EAX = adresse de la chaîne AsciiZ avec titre
PUSH 'LISTBOX'          ; mise en pile du pointeur de 'LISTBOX'
PUSH 0                  ; style étendu (aucun)
CALL CreateWindowExA    ; création d'une fenêtre listbox
```

..... ou, si vous préférez, vous pouvez utiliser INVOKE ..

```
   INVOKE CreateWindowExA,  0,'LISTBOX',EAX,80C80000h,300,30,130,208 \
                            0,0,[hInst],0

.............

   INVOKE ShowWindow, [hWnd], 1
```

Annexe F
Flags, sauts conditionnels, CMOVcc et SETcc

F1 Les Flags

Les flags sont constitués, chacun, d'un seul bit du registre 32 bits EFLAGS du processeur (RFLAGS en 64 bits). Il y a six *flags d'état* utilisés pour indiquer le résultat de certaines instructions et un *flag de direction* permettant de fixer le sens d'incrémentation automatique des adresses sur certaines instructions telles que MOVS, CMPS, SCAS, LODS et STOS. Certaines instructions comme CMP, TEST et BT se bornent à modifier certains de ces flags et ne font rien d'autre. D'autres instructions effectuent des opérations tout en modifiant tout ou partie de ces flags tandis que certaines n'ont aucune action sur eux. L'impact de chaque instruction sur les flags est parfaitement documenté par les fondeurs de processeur.

Les flags sont fréquemment utilisés pour changer le cours d'exécution d'une partie du code par le biais d'instructions de saut conditionnel. Ces instructions ont la particularité d'effectuer ou non un branchement vers une autre portion de code selon l'état d'un ou plusieurs flags. Seulement cinq parmi ces flags peuvent être utilisés de cette manière : *zéro*, *signe*, *retenue* et *parité*. Les sixième (*retenue auxiliaire*) et septième (*direction*) flags sont lus par d'autres instructions.

Voici plus d'informations sur les cinq flags qui peuvent être utilisés par les instructions de saut conditionnel :

Zf – flag de zéro

Prend la valeur 1 chaque fois que le résultat d'une opération prescrite par une instruction est nul dans le registre ou l'emplacement mémoire destination. Certaines instructions telles que CMP (comparaison) et TEST (test) permettent de positionner ce flag sans effectuer la moindre opération. Par exemple, CMP réalise une soustraction de l'opérande-source et de l'opérande-destination sans en donner le résultat et à seule fin d'activer certains indicateurs dont le flag de zéro. En ce sens, elle s'apparente à l'instruction SUB qui opère de la même manière mais fournit explicitement le résultat de la soustraction dans l'opérande destination. Par exemple :

```
CMP EAX,33h      ; met à 1 le flag de zéro si EAX=33h, mais ne modifie pas EAX
SUB EAX,33h      ; met à 1 le flag de zéro si EAX=33h, mais soustrait 33h à EAX
CMP EAX,EDX      ; met à 1 le flag de zéro si EAX=EDX
CMP EAX,[VALUE] ; met à 1 le flag de zéro si EAX=le nb pointé par VALUE
```

Le flag de zéro peut également être utilisé pour détecter l'atteinte d'une borne supérieure ou inférieure par un compteur. Par exemple

```
DEC EAX  ; met à 1 le flag de zéro si EAX est nul après l'instruction
         ; et le met à zéro dans le cas contraire
INC EAX  ; met à 1 le flag de zéro si EAX est nul après l'instruction
         ; et le met à zéro dans le cas contraire
```

Le flag de zéro peut également servir à contrôler la répétition d'instructions de chaîne telles que LODS, STOS, MOVS, SCAS, CMPS, INS et OUTS

- Le préfixe REP appliqué aux instructions **LODS**, **STOS**, **MOVS**, **INS** et **OUTS** répète ces instructions autant de fois que spécifié par le contenu du registre (E)CX/RCX. Concrètement, au terme de chaque exécution de l'instruction, l'adresse contenue dans chaque registre d'index[4] est incrémentée (ou décrémentée[5]), la répétition s'interromp lorsque (E)CX/RCX=0, situation marquée par le flag Zf.
- Les préfixes REPE/REPZ/REPNE/REPNZ appliqués à l'instruction **SCAS** permettent de comparer le contenu de (E)AX/RAX à l'adresse de mémoire pointée par (E)DI/RDI. SCAS est répétée tant que (E)AX/RAX=[(E)DI/RDI] avec REPE/REPZ ou tant que (E)AX/RAX≠[(E)DI/RDI] avec

[4] Ces registres d'index sont : (E)SI/RSI pour LODS et OUTS, (E)DI/RDI pour STOS et INS, (E)SI/RSI et (E)DI/RDI pour MOVS.

[5] Les registres d'index sont incrémentés automatiquement après chaque répétion de l'instruction si le flag de direction Df est nul (situation créée avec l'instruction CLD). Les registres d'index sont décrémentés automatiquement après chaque répétion de l'instruction si le flag de direction Df est à 1 (situation créée avec l'instruction STD).

REPNE/REPNZ. Au terme de chaque exécution de l'instruction, l'adresse contenue dans (E)DI/RDI est incrémentée (ou décrémentée) selon le positionnement du flag de direction.

- Les préfixes REPE/REPZ/REPNE/REPNZ appliqués à l'instruction **CMPS** permettent de comparer le contenu de l'emplacement mémoire pointé par (E)SI/RSI à celui pointé par (E)DI/RDI. CMPS est répétée tant que [(E)SI/RSI] = [(E)DI/RDI] avec REPE/REPZ ou tant que [(E)SI/RSI] ≠ [(E)DI/RDI] avec REPNE/REPNZ. Au terme de chaque exécution de l'instruction, les adresses contenues dans (E)SI/RSI et (E)DI/RDI sont incrémentées (ou décrémentées) selon le positionnement du flag de direction.

Il n'est pas rare qu'un retour d'API Windows (le plus souvent dans EAX) se solde par un échec d'exécution. Donc, vous aurez souvent besoin de tester cette éventualité. Lors du test de EAX, vous pouvez utiliser ces alternatives :

```
CMP EAX,0      ; voir si EAX = 0 (flag de zéro à 1 dans ce cas)
OR EAX,EAX     ; fait la même chose mais utilise 2 opcodes au lieu de 3
TEST EAX,EAX   ; fait également la même chose et utilise seulement 2 opcodes
```

Les versions 16 bits et 8 bits des instructions testent seulement les 16 ou 8 premiers bits de la zone de registre ou de la mémoire, respectivement, par exemple

```
CMP W[DAVID],0 ; voir si les 16 premiers bits de mémoire pointés
               ; par DAVID sont nuls
CMP B[SUE],0   ; voir si les 8 premiers bits de mémoire pointés
               ; par SUE sont nuls
OR DX,DX       ; voir si le registre DX est à zéro (registre 16 bits)
TEST DH,DH     ; voir si le registre DH est à zéro (registre 8 bits)
SUB B[VALUE],2 ; soustrait 2 à la valeur 8 bits pointée par VALUE (flag de
               ; zéro à 1 si le résultat de l'opération est nul)
DEC SI         ; flag de zéro à 1 si SI est nul (registre 16 bits)
INC B[COUNT]   ; flag de zéro à 1 si l'octet pointé par COUNT est nul
```

Dans la mesure où les flags sont très utiles lors du retour d'une routine pour identifier un éventuel échec d'exécution, vous devrez parfois en fixer préalablement le contenu. Il n'existe pas d'instruction dédiée à l'initialisation du flag de zéro. Vous pouvez cependant y parvenir simplement de la manière suivante :

```
CMP EAX,EAX    ; met à 1 le flag de zéro (EAX demeure inchangé)
SUB EAX,EAX    ; met à 0 le flag de zéro (provoque de surcroît : EAX = 0)
CMP EAX,EDX    ; met à 0 le flag de zéro lorsque EAX≠EDX
OR EAX,EAX     ; met à 0 le flag de zéro tant que EAX≠0
TEST EAX,EAX   ; même effet que OR EAX,EAX
```

Après exécution de l'instruction TEST, le flag de zéro est mis à 1 si le bit testé est nul. Notez, dans ce cas, que l'opérande de l'instruction permet de définir le (ou les) bit(s) testé(s). Par exemple : 1 = bit #0, 2 = bit #1, 4 = bit #2, 3 = bits #0 et #1. Retenir que, selon un principe analogue à CMP, TEST effectue un AND entre les opérandes source et destination, positionne le flag de zéro en conséquence mais ne fournit pas le résultat de l'opération.

Les exemples qui suivent illustrent le mécanisme de réflexion à associer aux instructions TEST et CMP pour bien en comprendre le fonctionnement :

```
MOV ECX,1      ; donne la valeur 1 au registre ECX
TEST ECX,1     ; AND ECX,1 est égal à 1 => le flag de zéro est nul
CMP ECX,1      ; SUB ECX,1 est égal à 0 => le flag de zéro est à 1

MOV EDX,0      ; EDX = 0
TEST EDX,1     ; AND EDX,1 est égal à 0 => le flag de zéro est à 1
CMP EDX,1      ; SUB EDX,1 est égal à  -1 => le flag de zéro est nul

MOV EBX,-1     ; EBX = -1 (0FFFFh en pratique => tous les bits à 1)
TEST EBX,-1    ; AND EBX, -1 est égal à 1 (TEST sur les 32 bits) => le flag
               ; de zéro est nul
CMP EBX,-1     ; SUB EBX, -1 est égal à 0 => le flag de zéro est à 1
```

Le flag de zéro est le seul utilisé avec les instructions de saut conditionnel JZ et JNZ (équivalent rigoureusement à JE et JNE). Par exemple :

```
JZ >L10        ; saut vers l'avant en L10 si le flag de zéro est égal à 1
JNZ L1         ; saut vers l'arrière en L1 si le flag de zéro est nul
```

Le flag de zéro est également utilisé dans le JA (sauter si supérieur à[6]), JB (saut si inférieur à) et les instructions équivalentes de saut conditionnel (JA = JNBE et JB = JNAE).

Il peut également être utilisé dans des boucles en utilisant des instructions spécifiques ou non. Voici quelques exemples assez classiques :

```
L1:
                ; autre code ici
CMP EDX,EAX
LOOPZ L1        ; décrémente ECX, poursuit la boucle jusqu'à ce que ECX = 0
                ; ou jusqu'à ce que EDX = EAX (lorsque le flag de zéro sera à 1)
;*******
L1:
                ; autre code ici
CMP EDX,EAX
LOOPNZ L1       ; décrémente ECX, poursuit la boucle jusqu'à ce que ECX = 0
                ; ou jusqu'à ce que EDX ≠ EAX (lorsque le flag de zéro sera nul)
;*******
L1:
                ; autre code ici
CMP EDX,EAX
JZ >L10         ; saut hors de la boucle si EDX=EAX (flag de zéro flag à 1)
LOOP L1         ; décrémente ECX, poursuit la boucle jusqu'à ce que ECX = 0
L10:
;*******
L1:
                ; autre code ici
CMP EDX,EAX
JNZ L1          ; poursuit la boucle jusqu'à ce que EDX = EAX (flag de zéro à 1)
```

On notera ici qu'il n'a pas été question ici des instructions de saut conditionnel JCXZ, JECXZ et JRCXZ fréquemment utilisées dans les boucles mais dont la particularité est de n'agir que sur le constat que le contenu des registres CX, ECX, RCX (selon le cas) est nul.

Sf – flag de signe

Prend la valeur 1 chaque fois que le bit le plus significatif (le plus à gauche) du résultat est égal à 1. La position de ce bit dépend de la taille des données. Dans un octet le bit le plus significatif est le bit 7 (8 bits des bits 0 à 7) ; dans un mot, c'est le bit 15 (16 bits des bits 0 à 15) ; dans un dword, c'est le bit 31 (32^{e} bit des bits 0 à 31) et dans un qword, c'est le bit 63 (64^{e} bit des bits 0 à 63). Donc, ce bit sera à 1 si le résultat de l'instruction est ≥ 80h pour un octet, ≥ 8000h pour un mot ou ≥ 80000000h pour un dword. Notez que dans les **nombres signés** le bit le plus significatif indique que le nombre est négatif (1) ou non (0).

Le flag de signe est modifié par INC et DEC alors que le flag de retenue ne l'est pas, donc le test de l'indicateur de signe est souvent utile dans les boucles. Par exemple :

```
L0:
;
DEC ECX         ; décrémente ECX d'une unité
JNS L0          ; boucle arrière vers L0 si ECX n'est pas encore à -1
```

Le flag de signe peut également être commodément utilisé dans des fonctions à multiples-actions comme, par exemple :

```
MULTI_ACTION:   ; à l'entrée, le registre AL contient l'action à accomplir
DEC AL          ; voir si AL = 0
JS >L0          ; oui
DEC AL          ; voir si AL = 1
JS >L1          ; oui
DEC AL          ; voir si AL = 2
```

[6] Notez que nous utilisons ici une formulation rigoureuse : "supérieur à" et "inférieur à" se rapportent à l'arithmétique non signée alors que "plus grand que" et "plus petit que" ont trait à l'arithmétique signée (cf. § F2).

```
JS >L2          ; oui
DEC AL          ; voir si AL = 3
JS >L3          ; oui
DEC AL          ; voir si AL = 4
JS >L4          ; oui
```

L'utilisation de l'indicateur de signe est un moyen pratique de voir si le bit de plus fort poids d'un registre est à 1 ou à 0. Un certain nombre d'instructions mettent ce flag à 1 sans modifier aucun registre comme, par exemple :

```
OR EDX,EDX      ; met le flag de signe à 1 si le plus haut bit de EDX est à 1
CMP EDX,EDX     ;   - idem -
TEST EDX,EDX    ;   - idem -

OR CL,CL        ; met le flag de signe à 1 si le plus haut bit de CL est à 1
CMP CL,CL       ;   - idem -
TEST CL,CL      ;   - idem -
```

Lors de la vérification du contenu de zones de mémoire cependant, vous ne pouvez adresser la mémoire qu'une fois par instruction de sorte que vous devez utiliser CMP. Par exemple :

```
CMP B[DATA44],0     ;met à 1 le flag de signe si le 8° bit de DATA44 est à 1
CMP W[DATA44],0   ; met à 1 le flag de signe si le 16° bit de DATA44 est à 1
CMP D[DATA44],0   ; met à 1 le flag de signe si le 32° bit de DATA44 est à 1
CMP B[DATA44+7],0 ; met à 1 le flag de signe si le 64° bit de DATA44 est à 1
CMP B[DATA44+9],0 ; met à 1 le flag de signe si le 80° bit de DATA44 est à 1
```

Noter que la position du bit de poids fort dans la zone de mémoire pointée par DATA44 dépend de la taille de données traitée par l'instruction au moyen de l'indicateur de type (B, W ou D présentement). En effet, les octets des données dans les zones de mémoire sont stockés dans l'ordre inverse, l'octet le moins significatif étant positionné en premier alors que l'octet le plus significatif figure en dernier à une adresse plus haute. On lira avec intérêt sur ce sujet l'Annexe J traitant la question de la "**mémorisation inversée**". L'instruction `CMP B[DATA44+7],0` teste le 8ème octet qui contient le 64ème bit. Il s'agit bien du signe, mais pour une taille de données de 64 bits.

Le flag de signe est principalement utilisé par les instructions de saut conditionnel JS et JNS comme, par exemple :

```
JS >L10         ; saut avant vers L10 si le flag de signe est à 1
JNS L1          ; saut arrière vers L1 si le flag de signe est à 0
```

Le flag de signe est également utilisé par JG (saut si plus grand que), JNG (saut si non-plus grand que) et les instructions équivalentes de saut conditionnel (JG = JNLE et JNG = JLE).

C*f – flag de retenue ou flag de carry*

Prend la valeur 1 chaque fois que le résultat de l'instruction est allé au-delà de la limite de la taille des données (c'est-à-dire qu'une « retenue » – "carry" en anglais – a été faite). Supposons par exemple que, dans une instruction de 8 bits, la valeur de 1 soit ajoutée à 255. Cela ne peut pas faire 256 puisque 255 est la limite structurelle de la donnée pour un octet. Donc, le résultat sera 0, mais le flag de retenue sera activé en prenant la valeur 1. Imaginons maintenant la situation où l'on soustraierait 4 à 2. Là encore, on constaterait l'activation du flag de retenue parce que le résultat passe en dessous de zéro qui est la limite inférieure de taille des données.

Le flag de retenue indique donc qu'un débordement est survenu lors de l'utilisation des nombres non signés. Voir la section concernant le **flag d'overflow** s'agissant des débordements lorsque des nombres signés sont utilisés.

Contrairement à d'autres flags d'état, il existe des instructions spécifiquement conçues pour positionner le flag de retenue directement :

```
STC             ; met à 1 le flag de retenue
CLC             ; met à 0 le flag de retenue
CMC             ; complémente le flag de retenue (inverse son état)
```

Par ailleurs, les instructions de manipulation de bits BT, BTS, BTR et BTC permettent de copier un bit spécifié dans le flag de retenue.

Compte tenu de la simplicité de mise en oeuvre des instructions STC, CLC et CMC, le flag de retenue est très utilisé pour signaler le résultat d'une fonction au processus appelant, par exemple :

```
CALCULATE2:
;
CMP EBX,ESI
JZ >.fail          ; saut au label .fail si EBX = ESI
CMP EAX,ESI
JZ >.success       ; saut au label .success si EAX = ESI
.fail
STC                ; met le flag de retenue à 1 pour faire apparaître un échec
RET
.success
CLC                ; met le flag de retenue à 0 pour faire apparaître un succès
RET
;
CALCULATE1:
CALL CALCULATE2
JC >L40            ; saut avant vers L40 s'il y a eu échec dans CALCULATE2
```

Notez que INC et DEC ne modifient pas le flag de retenue. Il en va de même pour les instructions de boucle. Ceci est utile si vous avez une boucle qui doit rendre compte de son résultat en utilisant le flag de retenue comme, par exemple :

```
.loop
;
CMP ESI,EDI        ; est-ce que ESI < EDI ? (flag de retenue à 1 si oui)
DEC ECX            ; voir s'il y a plus de boucles à exécuter
JNZ .loop          ; oui
RET                ; retour avec le résulutat de CMP ESI,EDI dans le
                   ; flag de retenue
```

Quelques instructions mettent toujours à zéro le flag de retenue. Il est utile de le savoir pour éviter le codage d'un CLC surabondant si vous souhaitez effacer le flag de retenue au terme de ces instructions. Il s'agit de AND, OR et TEST.

Certaines instructions ont une action déterminée partiellement ou en totalité par le flag de retenue :

- ADC *addition avec retenue*
- SBB soustraction avec retenue
- Les *sauts conditionnels* JC, JNC, JA, JNA, JAE, JNAE, JB, JBE, JNBE, JNB,
- Les instructions de *copie de donnée conditionnelle* CMOVC, CMOVNC, CMOVA, CMOVNA, CMOVAE, CMOVNAE, CMOVB, CMOVBE, CMOVNBE, CMOVNB.

Le flag de retenue est principalement utilisé avec les instructions de saut conditionnel JC et JNC en plus de celles mentionnées précédemment. Par exemple :

```
JC >L10            ; saut avant vers L10 si le flag de retenue est à 1

JNC L1             ; saut arrière vers L1 si le flag de retenue est à 0
```

Of – *flag de débordement* ou *flag d'overflow*

Pour appréhender le fonctionnement du flag de débordement (overflow), il est nécessaire de bien comprendre la structure des **nombres signés**. Le flag de débordement est utilisé pour identifier un éventuel débordement au terme d'une opération effectuée sur des nombres signés. Le flag de retenue ne peut pas être utilisé dans ce but ainsi que le montre l'exemple simple ci-dessous :

```
MOV AL,0FEh        ; charge AL avec 254 décimal (non-signé) ou -2 (signé)
ADD AL,4h          ; additionne 4 => AL contient maintenant 2h
```

Ici, on constate que le flag de retenue (carry) est mis à 1 parce que le résultat de 258, interprété au sens de l'arithmétique non-signée, est trop grand au regard de la limite de taille fatidique de 255. Mais, si l'on se place du point de vue de l'arithmétique signée, il n'y a pas débordement. La valeur de 2 prise par AL est le résultat de l'addition –2+4. Le flag de débordement est donc à 0, en toute logique, puisqu'il se détermine dans un contexte d'arithmétique signée.

Voici un autre exemple où **il y a débordement** dans un calcul signé :

```
MOV AL, 7Fh        ; charge AL avec 127 décimal
ADD AL, 4h         ; additionne 4 => AL contient maintenant 83h
```

Ici, le flag de retenue est nul parce que le résultat non signé 131 (83h) en AL est en-deçà de la limite de taille de donnée de 255. Mais, analysé dans un contexte d'arithmétique signée, il provoque un débordement parce que le résultat se situe au-delà des limites assignées de –127 à +128.

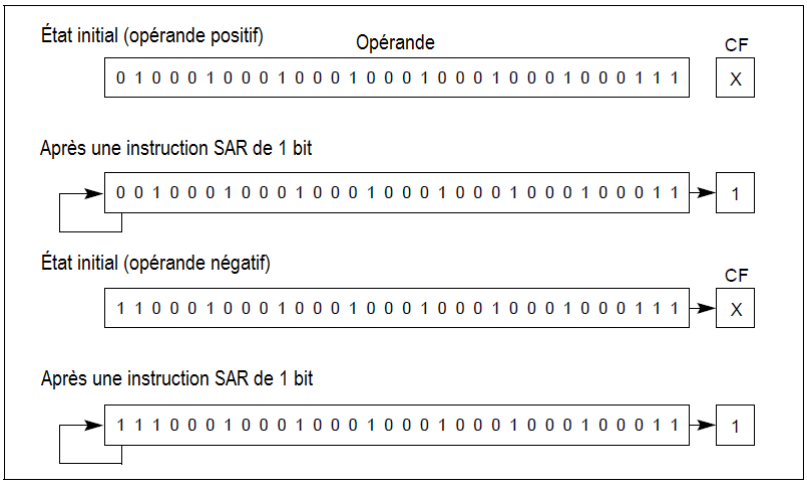
Donc, dans ce type d'opération arithmétique le processeur met à 1 le flag de débordement si le bit de signe change alors qu'il n'y a pas de retenue. Ceci est indépendant du flag de retenue comme on peut le voir dans l'exemple qui suit :

```
MOV AL, 7Fh        ; charge AL avec 127 décimal
INC AL             ; provoque un dépassement (retenue non affectée par INC)
;
MOV AL, 80h        ; charge AL avec -128 decimal
DEC AL             ; provoque un dépassement (retenue non affectée par DEC)
```

Dans les instructions de décalage binaires et *uniquement pour les opérations de décalage d'un seul bit*, le flag de débordement ne donne une indication valide que si le résultat signé est trop grand pour la taille des données. Par exemple :

```
MOV AL, 80h        ; AL = -128
SHL AL, 1          ; la multiplication par 2 cause un débordement
MOV AL, 0FEh       ; AL = -2
SHL AL, 1          ; la multiplication par 2 (-4) ne cause pas de débordement
MOV AL, 80h        ; AL = -128
SHL AL, 2          ; la multiplication par 4 ne signale pas de débordement
MOV AL, 0FEh       ; AL = -2
SAR AL, 1          ; la division par 2 (-1) ne provoque pas de débordement
```

SAR est une instruction spéciale de décalage binaire à droite qui a pour effet de maintenir le bon signe dans le résultat. Elle le fait en déplaçant à droite tous les bits *sauf* celui de plus fort poids qui détient le signe. L'emplacement ainsi libéré immédiatement à droite du bit de signe est comblé par une copie de ce même bit. Dans la mesure où l'instruction SAR revient à une division par deux, elle ne peut jamais déborder.



fonctionnement de l'instruction SAR[7]

En revanche, SHL est sujette à cet inconvénient et, dans les opérations de décalage binaire d'un bit, le flag de débordement est positionné de manière appropriée selon le résultat. Pour y parvenir, le processeur scrute une éventuelle égalité entre le bit de signe et le flag de retenue et met à zéro le flag de débordement si tel est le cas.

[7] La figure représentant le fonctionnement de l'instruction SAR est empruntée à l'ouvrage Intel® 64 and IA-32 Architectures – Software Developer's Manual – Volume 1 : Basic Architecture (Order Number: 253665-055US – June 2015)

Grâce à ce test, il est possible d'envisager une autre utilisation pour le flag de débordement (*notez que ces tests affectent le contenu du registre*) ainsi que le montre l'exemple qui suit :

```
SHL AL,1
JNO >L1         ; saut si les 2 bits de plus fort poids de AL sont les mêmes
SHL AL,1
JO >L1          ; saut si les 2 bits de plus fort poids de AL sont différents
SHL EAX,1
JNO >L1         ; saut si les 2 bits de plus fort poids de EAX sont les mêmes
SHL EAX,1
JO >L1          ; saut si les 2 bits de plus fort poids de EAX sont différents
```

Les instructions de rotation de bits fonctionnent de la même manière. Etant donné que l'instruction ROR décale tous les bits vers la droite en remplaçant le bit le plus élevé par le plus bas, ceci fournit un moyen de comparer le bit de plus fort poids et celui de plus faible poids d'une donnée. En voici une illustration (*notez que ces tests changent le contenu du registre*) :

```
ROR AL,1
JNO >L1         ; saut si le bit le plus bas et le plus haut de AL sont égaux
ROR AL,1
JO >L1          ; saut si le bit le plus bas et le plus haut de AL diffèrent
ROR EAX,1
JNO >L1         ; saut si le bit le plus bas et le plus haut de EAX sont égaux
ROR EAX,1
JO >L1          ; saut si le bit le plus bas et le plus haut de EAX diffèrent
```

L'instruction spéciale IMUL de multiplication signée met à 1 le flag de débordement si le résultat signé est trop grand pour la taille des données.

Le flag de débordement est principalement utilisé avec les instructions de saut conditionnel JO et JNO comme, par exemple :

```
JO >L10          ; saut avant vers L10 si le flag de débordement est à 1
JNO L1           ; saut arrière vers L1 si le flag de débordement est à 0
```

Les flags JG, JGE, JL, JLE, JNG, JNGE, JNL, JNLE sont également conditionnés par O*f* mais ce dernier est analysé conjointement avec S*f*. Par exemple :

- JNGE (saut court si non plus grand que ou égal) n'est actif que si les flags Of et Sf affichent un état différent.
- JGE (saut court si plus grand que ou égal) n'est actif que si les flags Of et Sf ont le même état.

P*f* – *flag de parité* ou *Parity flag*

Le flag de parité indique si la donnée traitée affiche un nombre de bits à 1 pair ou impair. Cet indicateur est à 1 si le nombre de bits à 1 est pair est à 0 si ce nombre est impair. Dans les communications série, le bit de parité est utilisé comme contrôle d'erreur élémentaire. Parallèlement à chaque octet envoyé, l'émetteur envoie un *bit de parité* qui indique au récepteur si l'octet qu'il vient d'envoyer est pair ou impair. Le procédé est trop sommaire pour identifier avec certitude un octet corrompu mais se révèle néanmoins intéressant dès lors qu'apparaît une série d'octets corrompus. Lorsque le contrôle de parité est utilisé de cette manière en transmissions série, il est d'usage, pour chaque octet, d'affecter 1 bit au contrôle de parité et les 7 autres à la transmission effective de la donnée.

Le flag de parité est principalement utilisé avec les instructions de saut conditionnel JP et JNP comme, par exemple :

```
JP >L10          ; saut avant vers L10 si le flag de parité est à 1

JNP L1           ; saut arrière vers L1 si le flag de parité est à 0
```

A*f* – *flag auxiliaire* ou *Auxiliary flag*

Le flag de retenue auxiliaire est utilisé en arithmétique BCD (Binaire Codé Décimal). Contrairement aux autres flags, celui-ci n'a strictement aucun effet sur les instructions de saut conditionnel. Au contraire, il ne peut être activé que par une instruction de calcul BCD, puis lu par l'instruction BCD suivante. L'arithmétique BCD est décrite dans le volume 2.

D*f* – *flag de direction* ou *Direction flag*

Le flag de direction est très important car il détermine le mode de fonctionnement des instructions de chaîne LODS, STOS, MOVS, SCAS, CMPS, INS et OUTS. Ces instructions utilisent en effet, séparément ou conjointement, le registre d'index (E)SI/RSI[8] pour pointer la donnée source (SI = *Source Index*) et son homologue (E)DI/RDI pour pointer la donnée destination (DI = *Destination Index*).

Or, ces registres d'index sont incrémentés ou décrémentés automatiquement au terme de l'exécution des instructions précitées de manière à être prêts à traiter la donnée suivante. C'est ici qu'intervient le flag de direction D*f* :

- Df = 1, les registres d'index sont décrémentés,
- Df = 0, les registres d'index sont incrémentés.

Deux instructions permettent de positionner ce flag :

- CLD (*CLear Direction*) → Df = 0 → le(s) registre(s) d'index sont incrémentés
- STD (*SeT Direction*) → Df = 0 → le(s) registre(s) d'index sont incrémentés

Deux exemples mettant en oeuvre `REP MOVSB` permettront de mieux comprendre ce mécanisme :

```
DATA SECTION
      msg1     DB 'on incrémente !'
      msg2     DB 'on décrémente !'
      msg1b    DB 15 dup 0
      msg2b    DB 15 dup 0
CODE SECTION
START:
CLD                         ; incrémentation automatique des adresses
MOV  ESI,ADDR msg1          ; ESI pointe msg1
MOV  EDI,ADDR msg1b         ; EDI pointe msg1b
MOV  ECX,15                 ; nombre de caractères à copier
REP  MOVSB                  ; après cette opération :
                            ; msg1b = 'on incrémente !'
                            ; ECX = 0, ESI = ADDR msg1+15, ESI = ADDR msg1b+15

STD                         ; décrémentation automatique des adresses
MOV  ESI,ADDR msg2+14       ; ESI pointe le dernier caractère de msg2
MOV  EDI,ADDR msg2b         ; EDI pointe le dernier caractère de msg2b
MOV  ECX,15                 ; nombre de caractères à copier
REP  MOVSB                  ; après cette opération :
                            ; msg1b = 'on incrémente !'
                            ; ECX = 0, ESI = ADDR msg2-1, ESI = ADDR msg2b-1
XOR  EAX,EAX
RET
```

F2 Les sauts conditionnels

En raison de la cohabitation de l'arithmétique signée et non signée, la qualification de la comparaison de 2 grandeurs peut poser quelques problèmes si elle n'est pas encadrée par un vocabulaire précis. Nous utiliserons donc, dans ce manuel, la convention de traduction suivante :

	Arithmétique NON signée		Arithmétique signée	
symbole	<	>	<	>
En français	Inférieur à	Supérieur à	Plus petit que	Plus grand que
En anglais	Below than	Above than	Less then	Greater than

[8] la représentation (E)SI/RSI regroupe, pour la commodité de l'exposé, les différents formats du registre d'index source : SI en 16 bits, ESI en 32 bits et RSI en 64 bits. Cette forme est également utilisée pour d'autres registres.

On rappelle ici la signification de la représentation abrégée des flags :

- **Zf** flag de zéro (*zero flag*)
- **Sf** flag de signe (*Sign flag*)
- **Cf** flag de retenue (*Carry flag*)
- **Of** flag de débordement (*Overflow flag*)
- **Pf** flag de parité (*Parity flag*)
- **Af** flag de retenue auxiliaire (*Auxiliary flag*)

Les instructions de saut conditionnel NON signées

Instruction	Alternative	Action
JZ	JE	Saut si Zf = 1 (saut si égal)
JNZ	JNE	Saut si Zf = 0 (saut si non égal)
JC	JB ou JNAE	Saut si Cf = 1 (saut si "inférieur à" ou "non-supérieur à" ou "égal")
JNC	JNB ou JAE	Saut si Cf = 0 (saut si non "inférieur à" ou "supérieur à" ou égal)
JA	JNBE	Saut si ni Cf, ni Zf ne sont à 1 (ie. Saut si "supérieur à" ou si non "inféreur à" ou égal). Voir la note concernant JNA ci-dessous.

JNA	JBE	Saut si soit Cf, soit Zf = 1 (c'est-à-dire si saut "non-supérieur à" ou si "inférieur à" ou égal). Ceci est utile pour vérifier après une soustraction qu'un registre reste supérieur à 1. Par exemple, vous pourriez utiliser ceci dans le codage de protection qui suit dans lequel il est supposé que EDI pointe la fin d'une chaîne dans BUFFER :
		`MOV EDX,ADDR BUFFER`
		`SUB EDI,EDX ; récupère la longueur de chaîne dans EDI`
		`JNA >.error ; la longueur est trouvée négative ou nulle`

JP		Saut si Pf = 1
JNP		Saut si Pf = 0
JECXZ		Saut si ECX = 0
JCXZ		Saut si CX = 0

Les instructions de saut conditionnel signées

Instruction	Alternative	Action
JS		Saut si Sf = 1 (saut si le signe du nombre est négatif)
JNS		Saut si Sf = 0 (saut si si le signe du nombre est positif)
JO		Saut si Of = 1 (dépassement à l'issue d'une opération)
JNO		Saut si Of = 0 (pas de dépassement à l'issue d'une opération)
JL	JNGE	Saut si Sf et Of sont différents (saut si "plus petit que" or "non-plus grand que" ou "égal"). Ce test spécial est nécessaire pour les nombres considérés comme signés parce que JA (saut si "supérieur à") et JB (saut si "inférieur à") ne travaillent pas avec ces nombres. Par exemple, si AL est à 1, alors une instruction JB placée immédiatement après `CMP AL,-1` n'effectuera pas de saut puisque Cf = 0. En revanche, JL effectuera ce saut parce que, justement, Cf = 0 et qu'il agit en arithmétique signée, considérant que –1 est *plus petit que* 1.
JNL	JGE	Saut si Sf et Of sont égaux (saut si "non-plus petit que" ou "plus grand que ou égal"). Voir la note ci-dessus concernant JL.
JLE	JNG	Saut si Sf et Of sont différents et que ZF = 1 (saut si "plus petit que ou égal" ou "non-plus grand que")
JLNE	JG	Saut si Sf et Of sont égaux et que Zf = 0 (saut si "non-plus petit que ou égal" ou "plus grand que")

F3 Instructions CMOVcc et SET cc

Passées les améliorations classiques telles que l'augmentation de la vitesse d'horloge et du parallélisme des opérations, les fondeurs de processeurs ont introduit des améliorations beaucoup plus subtiles où la vitesse d'exécution est subordonnée à un ensemble de petites actions de nature parfois assez complexes, le tout assorti de mémoires-cache, de *multi-threading* et autres concepts tout aussi étranges. Bref, il existe désormais, pour chaque famille de processeurs, de véritables "pavés" souvent plutôt bien faits, si l'on se limite à Intel et AMD, qui détaillent les multiples "recettes" que se doit de connaître désormais tout programmeur soucieux de produire des applications hautement optimisées.

Un focus particuler est fait notamment dans ces ouvrages sur les branchements. D'une manière générale, il est recommandé d'en faire aussi peu usage que possible et, en tout état de cause, de les limiter à trois – voire moins – par fenêtre de code de 16 octets consécutifs. Fort heureusement, 2 instructions ont été introduites dans ce but et nous verrons qu'elles permettent de supprimer de nombreux sauts jugés incontournables jusqu'ici.

F3.1 Les instructions CMOVcc

Plutôt que de réaliser un MOV dans le cadre d'un branchement conditionnel, une nouvelle instruction baptisée CMOV pour la circonstance – *Conditionnal MOVe* –, propose un MOV dont l'exécution est subordonnée à l'état des flags selon l'exacte grille de lecture qu'en font les sauts conditionnels.

On trouve de la sorte : CMOVA, CMOVAE, CMOVNA, CMOVNAE, CMOVB, CMOVBE, CMOVNB, CMOVNBE, CMOVC, CMOVNC, CMOVE, CMOVNE, CMOVG, CMOVGE, CMOVNG, CMOVNGE, CMOVL, CMOVLE, CMOVNL, CMOVNLE, CMOVO, CMOVNO, CMOVP, CMOVPE, CMOVNP, CMOVS, CMOVNS, CMOVZ, CMOVNZ, CMOVPO.

Sur le plan sémantique, le mnémonique résulte de la contraction de MOV et Jcc, le J étant éliminé. Il est de la forme : CMOVcc *dest, srce*

Par exemple : CMOVA EAX,EBX signifie que le contenu du registre EBX est copié dans EAX si, et seulement si, les flags ont un état tel qu'un branchement conditionnel JA pourrait être activé dans cette situation.

Voici maintenant 2 exemples qui montrent comment supprimer un saut conditionnel avec un MOV :

1° exemple	**2° exemple**

Avant optimisation
`TEST ECX, ECX ; on teste ECX = 0`
`JNZ Suite , saut vers Suite si ECX ≠ 0`
`MOV EAX, EBX ; EAX = EBX si ECX = 0`
`⋮`
`Suite:`
	Avant optimisation
	`CMP ECX, 5 ; on teste ECX > 5`
	`JB Suite , saut vers Suite si ECX < 5`
	`MOV ECX, 5 ; ECX est limité à 5`
	`⋮`
	`Suite:`

Après optimisation
`TEST ECX, ECX ; on teste ECX = 0`
`CMOVZ EAX, EBX ; réalise le MOV si le flag`
`; Zf est égal à 1. Le label`
`; Suite peut être supprimé.`
	Après optimisation
	`CMP ECX, 5 ; on teste ECX > 5`
	`CMOVA ECX, 5 ; ECX est plafonné à 5`

F3.2 Les instructions SETcc

Elles sont de la forme SETcc *dest.* et ont pour particularité de mettre l'opérande destination à 0 ou 1 selon le positionnement des flags d'état. Cet opérande est constitué d'un octet qui peut être un registre ou une adresse mémoire. Le positionnement opéré par cette instruction est strictement de même nature que le mode de branchement effectué par les sauts conditionnels dont il emprunte la forme.

D'un point de vue sémantique, le mnémonique résulte de la contraction de SET et Jcc, le J étant éliminé.

De même que pour CMOVcc, on trouve : SETA, SETAE, SETNA, SETNAE, SETB, SETBE, SETNB, SETNBE, SETC, SETNC, SETE, SETNE, SETG, SETGE, SETNG, SETNGE, SETL, SETLE, SETNL, SETNLE, SETO, SETNO, SETP, SETPE, SETNP, SETS, SETNS, SETZ, SETNZ, SETPO.

Par exemple, dans :

```
CMP     EAX,12
SETE    DL   ; implique DL = 1 si EAX = 12 (sinon DL = 0)
SETAE   DL   ; implique DL = 0 si EAX = 10 (sinon DL = 1)
```

Par rapport à CMOVcc, on peut remarquer que l'instruction SETcc a la particularité de fédérer sur 1 seul bit – celui de plus faible poids de la destination – tout résultat de comparaison.

Voici un exemple de suppression de saut réalisée avec SETcc :

Ecriture traditionnelle avec des sauts conditionnels

```
XOR EBX, EBX    ; EBX = 0
CMP EAX,10      ; est-ce que EAX > 10 ?
JBE InfEgal     ; saut à InfEgal si EAX ≤ 10
MOV EBX,115     ; EBX = 115 si EAX > 10
JMP Superieur   ; vers fin du calcul

InfEgal:
MOV EBX,110     ; EBX = 110 si EAX ≤ 10
Superieur:      ; EBX = 115 si EAX > 10
```

Suppression du saut par introduction de STcc

```
XOR EBX, EBX    ; EBX = 0
CMP EAX, 10     ; est-ce que EAX > 10 ?
SETBE BL        ; BL = 1 si EAX ≤ 10
                ; BL = 0 si EAX > 10
SUB EBX, 1      ; EBX = 11...11 ou 00...00 selon le contenu de BL
AND EBX,5       ; 115-110
ADD EBX,110     ; EBX = 110 ou 115
```

On voit ici que tous les sauts ont été supprimés.

Annexe G
Pour les débutants... en Windows

Cet article est destiné à ceux qui débutent en programmation sous Windows.

Il décrit les éléments du système d'exploitation Windows qu'il est essentiel de connaître avant de pouvoir écrire un programme significatif. Bien sûr, Windows est bien plus volumineux que ce qui est décrit ici, et vous aurez besoin de beaucoup plus d'informations avant de pouvoir écrire des programmes.

Windows – La mise sous contrôle...

Windows prend le contrôle de l'ordinateur presque totalement dès le moment où l'on met en route ce dernier et ce, jusqu'à ce qu'on l'éteigne. Une application ne peut donc s'exécuter qu'avec l'autorisation et l'aide de Windows et sous son contrôle. Dès lors, Windows peut assister l'utilisateur avec l'ergonomie et les performances attrayantes qu'on lui connaît : prévisibilité et cohérence de l'interface utilisateur, aptitude (en apparence tout au moins) à exécuter plusieurs programmes à la fois (multitâches) et robustesse du système en cas d'échec d'une application.

..... du matériel

La plupart des micro-puces qui fonctionnent avec l'unité centrale de traitement sont programmables. Par exemple, la carte d'affichage a besoin de connaître les bonnes vitesse de balayage, résolution et couleurs. Les circuits d'entrée/sortie de l'imprimante doivent connaître le port imprimante à utiliser et la vitesse à laquelle les données doivent être transférées. Les puces intégrées au clavier doivent connaître le taux de répétition à utiliser. La communication réelle avec ces périphériques doit être contrôlée – chacun aura sa propre zone de mémoire à utiliser et devra savoir à quoi s'attendre et quand. Windows exécute toutes les tâches de base que vous attendez normalement d'un système d'exploitation. Vous, le programmeur d'application, pouvez alors vous concentrer sur le travail réellement créatif.

Mais Windows prend également le contrôle total de la lecture/écriture avec tous les composants externes. Ceci présente des avantages importants pour le programmeur de l'application.

Par exemple, pour imprimer un document, l'application peut se borner à indiquer à Windows l'endroit où le document réside en mémoire ainsi que sa taille. Windows prendra les dispositions pour qu'il soit imprimé, en utilisant le pilote d'imprimante approprié pour l'imprimante en cours d'utilisation, et en le mettant dans la bonne position dans la file d'attente d'imprimante, laquelle pourrait également contenir d'autres travaux d'impression provenant d'autres applications. Le travail d'impression est toujours effectué en mode graphique. Windows indique à l'imprimante l'emplacement exact où chaque point constituant l'image finale imprimée doit aller sur le papier. Les avantages pour le programmeur d'application sont très importants. Il est libéré de la tâche fastidieuse de fournir des pilotes d'imprimante à son application et d'écrire des algorithmes graphiques. Il peut « penser graphique » dès le départ sans avoir à se soucier des complications qui en découlent.

Windows réalise un travail encore plus sophistiqué avec l'affichage à l'écran, qui est également en mode graphique. Ici, Windows doit souvent faire face à plusieurs applications en essayant d'afficher simultanément leur sortie sur l'écran. Il peut y avoir plusieurs fenêtres à tout moment, et pour l'utilisateur, certaines d'entre elles semblent se chevaucher de sorte que certaines informations sont invisibles. Windows a la lourde responsabilité de la construction de l'image finale de l'écran en arbitrant entre ces différentes sorties d'application qui revendiquent, à juste titre, le droit d'apparaître sur l'écran. Il le fait à partir de l'enregistrement actualisé de chaque fenêtre et de sa connaissance de la priorité, de l'ordre de l'écran, du type et du style de chaque fenêtre. Une grande partie de votre travail de programmation impliquera de définir correctement ces facteurs dans votre propre programme pour garantir que Windows produise la sortie d'écran correcte.

Le corollaire au contrôle que Windows exerce sur le matériel périphérique est qu'une application ne peut pas (ou du moins ne devrait pas) accéder directement aux périphériques. Mais cela ne présente généralement aucun inconvénient puisque d'une part, Windows se caractérise par une grande polyvalence, d'autre part les ordinateurs modernes accomplissent la plupart des tâches à une vitesse satisfaisante.

..... de toutes les applications

Que se passe-t-il lorsque l'utilisateur clique sur l'icône représentant votre programme avec la souris ? Tout d'abord, Windows sait exactement où la souris a été cliquée et quelle icône était dans le champ du pointeur de la souris (curseur) à ce moment. Il sait aussi, d'après ses listes de "raccourcis" et de "propriétés", quel programme démarrer si cette icône particulière est cliquée.

Comment le programme est-il lancé à partir de cette action ? Windows commence par *charger* le programme en lisant le fichier correspondant puis l'installe en mémoire vive. Enfin, Windows *appelle* tout simplement le programme. Autrement dit, le processeur est invité à poursuivre l'exécution à partir de l'adresse du point d'entrée du programme. En conséquence, la façon la plus simple pour le programme de s'achever consiste à retourner au système avec une banale instruction RET.

..... du microprocesseur

Comme nous l'avons vu précédemment, pour démarrer le programme, le registre EIP du processeur reçoit l'adresse du point d'entrée par Windows. Windows a également un contrôle sur la valeur de tous les autres registres du processeur. Windows conserve ceux-ci en mémoire dans une zone de mémoire appelée *contexte de registre* (register context). Windows peut (et ne se gêne pas pour y recourir souvent), arrêter le processeur, mémoriser la valeur courante des registres, et enfin charger le processeur avec d'autres données pour exécuter un autre programme à la place pendant un certain temps. Si bien que cet autre programme se voit affecter *une tranche de temps* par Windows. Une fois terminée, Windows peut alors restaurer les registres dans votre programme qui reprend alors l'exécution à partir de là où elle avait été interrompue, car il y avait alors une *tranche de temps*.

Nous venons de toucher du doigt la manière dont fonctionne le multitâches sous Windows. Chaque programme en cours d'exécution dans le système se voit affecter une part de temps processeur. L'utilisateur peut avoir ainsi l'impression que plusieurs programmes s'exécutent simultanément alors que ce n'est que du temps partagé séquentiel, s'agissant d'ordinateurs à processeur unique. Windows détermine la longueur de la tranche de temps en fonction de différentes priorités. Par exemple, une opération de lecture/écriture de disque se verra normalement attribuer une priorité très élevée et pourra même empêcher d'autres programmes de s'exécuter jusqu'à ce que l'opération soit terminée.

Un programme peut demander à Windows de démarrer un autre *thread*. Windows va alors attribuer à ce thread ses propres tranches de temps ainsi que ses propres valeurs de registres et de pile. Le nouveau thread donnera l'impression de s'exécuter en même temps que le thread principal du programme. Ceci est utile si un programme doit se poursuivre avec une seule activité (par exemple un long calcul) tout en continuant à dialoguer avec l'utilisateur. Le programme peut donc apparaître à l'utilisateur comme exécutant deux ou plusieurs choses à la fois.

C'est ce qu'on appelle le *multi-threading*.

..... de la mémoire et des données

Votre programme peut faire beaucoup plus que simplement manipuler des données dans les registres du processeur. À tout moment, votre programme aura ses propres données en mémoire. Celles-ci seront soit dans la mémoire établie pour l'adressage direct, soit sur la pile. Alors, comment Windows préserve-t-il ces données lors de la commutation de tranche de temps entre les programmes ?

La réponse est que Windows conserve une *carte mémoire* de toutes les données du programme. Cela veut dire qu'il sait exactement où, dans la *mémoire physique* de l'ordinateur, les données du programme sont maintenues à tout moment. Windows conserve cette carte de mémoire dans une zone appelée *mémoire de contexte*. Si la mémoire physique est épuisée, Windows va utiliser le disque dur pour conserver, si nécessaire, les données du programme. C'est la raison pour laquelle, dans les systèmes disposant de faibles ressources en mémoire, il y a beaucoup plus d'activité du disque que dans les systèmes mieux pourvus en mémoire.

Lorsque le programme a besoin d'accéder à ses données, il doit le faire en utilisant une *adresse virtuelle*. Cela signifie que l'adresse de la zone mémoire ne correspond pas, en fait, à l'adresse réelle des données dans la mémoire physique. Windows informe le processeur du lieu dans lequel se situent les zones de mémoire nécessaires. Il le fait en donnant au processeur, dans le registre CR3, l'adresse de sa table de répartition des pages correspondant au programme.

..... de l'interface-utilisateur

La principale raison pour laquelle Windows a été développé était, en premier lieu, de fournir une interface cohérente et compréhensible à l'utilisateur. L'idée était que ce dernier puisse se déplacer sans difficulté d'un ordinateur à un autre, une fois familiarisé avec le système d'exploitation Windows. Plus important pour le programmeur Windows, chaque application interagirait avec l'utilisateur de manière similaire et selon une disposition familière. Par conséquent, les utilisateurs auraient un bon point de départ pour développer un nouveau programme, ce qui réduirait la formation et le temps d'apprentissage de façon spectaculaire. Ces objectifs ont été atteints avec un grand succès, et il y a maintenant des millions d'utilisateurs d'ordinateurs dans le monde entier utilisant des applications Windows qui trainaient initialement une réputation d'« analphabète informatique ».

Windows a atteint cette uniformité en proposant des applications avec des composants standard à inclure dans leurs programmes. Les exemples les plus évidents sont les menus qui apparaissent juste sous la barre de titre d'une application, les boîtes de dialogue où l'utilisateur peut faire des choix, les boutons avec texte que l'utilisateur peut cliquer, les barres de défilement pour déplacer le contenu d'une fenêtre vers le haut ou vers le bas, les boutons de la barre d'outils contenant de petites images et les fichiers d'aide standardisés.

Ces composants standards sont collectivement regroupés sous l'appellation "*Graphical User Interface*" (interface graphique) ou, plus simplement, GUI.

Comment une application les utilise-t-elle ? Par appel d'une "*Applications Programming Interface*" ou API. Il s'agit de procédures proposées par Windows qui peuvent être appelées par votre programme au moyen d'un nom spécifique. Toutes les API sont contenues dans des fichiers nommés *Direct Linked Libraries* ou DLLs. Ces dernières sont des fichiers exécutables pourvus de l'extension Dll. Vous en verrez beaucoup dans le répertoire Windows\System sur votre ordinateur. Elles contiennent du code pour l'*export*, c'est-à-dire que ce code est disponible pour être importé dans votre programme lorsque vous souhaitez utiliser l'interface graphique Windows.

..... et des fichiers

Les Dlls sont constamment révisées par les développeurs de Windows qui ajoutent de nouvelles API ou modifient les existantes. Elles sont distribuées à l'utilisateur de diverses manières, parfois dans un service pack Windows, ou lorsque vous chargez la nouvelle version d'un programme Microsoft (la méthode préférée a été d'inclure de nouvelles versions des Dlls via Internet Explorer). L'utilisateur peut ne pas réaliser ce qui se passe et pour vous, en tant que programmeur, cela ne fait aucune différence. Votre programme continuera toujours à appeler la même API pour faire la même chose, mais il pourrait éventuellement le faire de manière légèrement différente.

Windows conserve trace de la localisation de tous les fichiers qui sont importants pour son propre fonctionnement ou pour le pilotage des périphériques. Lorsque votre programme est *installé*, Windows impose que cette action produise un enregistrement approprié dans la *base de registre*, qui est une base de données maintenue par Windows sur la configuration du système et des applications qui se déroulent sous son contrôle. Windows nécessite également qu'une application au comportement irréprochable place les fichiers supportant une nouvelle application dans des répertoires correctement nommés de sorte qu'ils soient plus rapides et plus faciles à trouver. Ce travail est dévolu à un programme d'installation, qui est généralement livré avec le programme et intitulé Setup.exe.

Système – communication de l'application

Comme nous l'avons vu plus haut, le système Windows contrôle tous les aspects importants de l'ordinateur ainsi que les applications en cours d'exécution à un moment donné. Pour atteindre ce niveau de contrôle, un système de *communication* entre le système et l'application est indispensable.

Une application voudra communiquer avec le système quand elle aura besoin de savoir quelque chose à propos de l'interface graphique, par exemple la taille d'une fenêtre particulière ou celle d'une chaîne de texte particulière dans une police spécifique. Aussi, quand une application souhaite utiliser les fonctionnalités d'une API, elle doit être en mesure de dire très précisément à l'API comment fonctionner.

Les méthodes habituelles de notification de l'application au système sont les suivantes :

- **Données sur la pile**. Avant de faire un appel d'API, vous poussez les données nécessaires sur la pile (PUSH) à destination de l'API pour son usage. Les données sont toujours des valeurs numériques au format DWord, mais peuvent souvent être des pointeurs vers des structures contenant plus de données, ou des chaînes de texte.

- **Messages**. Vous envoyez un message au système en appelant l'API *SendMessage*. Le message lui-même se réduit en fait à une valeur numérique au format DWord poussée sur la pile, mais vous pouvez également envoyer jusqu'à 3 dwords de données avec le message.

Le système va vouloir communiquer avec l'application pour lui donner les résultats d'un appel d'API, ou pour l'informer que quelque chose se passe dans l'interface graphique (GUI) ou que quelque chose d'important se passe dans le système lui-même.

Les méthodes habituelles de communication à partir du système vers l'application sont les suivantes :

- Au retour d'une API, le système dépose habituellement une valeur dans le registre EAX donnant le résultat de l'appel d'API.
- Parfois, au retour d'une API, le système laisse des données dans la mémoire à un emplacement spécifié par le programme lors de l'appel de cette API. Cet emplacement peut, par exemple, avoir été spécifié par l'application au moyen d'un PUSH de pointeur sur la pile avant l'appel de l'API.
- Messages à l'application. Ils sont émis par le système lorsqu'il *appelle* l'application. Lorsque cela se produit, le système envoie également des données sur la pile. L'application aura préalablement informé le système de l'adresse, dans son code, d'une procédure où ces appels peuvent être faits en prévison d'un traitement. Cette procédure, très importante est nommée le plus souvent *call-back*, *procédure de fenêtre* ou, plus simplement, *WndProc*.

Handles et contextes de périphériques

Tous les "objets" avec lesquels Windows travaille ont des *handles*. Ces objets peuvent être des fenêtres, des contrôles, des menus, des dialogues, des processus, des threads, des zones mémoire, des écrans, des imprimantes, des fichiers, des disques durs, et même des polices, des pinceaux et stylos utilisés pour le dessin et l'écriture. Un handle est une valeur DWord que l'application demande à Windows. L'application utilisera le handle lorsqu'elle voudra communiquer avec Windows pour utiliser ou modifier l'objet concerné.

Tous les appareils qui affichent ou, plus généralement, produisent une sortie ont des *contextes de périphériques*. Le contexte de périphérique est une zone de mémoire entretenue par Windows qui contient des informations sur la façon dont l'appareil doit afficher sa sortie. Ainsi, une fenêtre particulière aura-t-elle un contexte de périphérique qui contient des informations sur la police qui doit être utilisée dans la fenêtre et la couleur dans laquelle tout dessin ou écriture devront apparaître dans ladite fenêtre. Pour sa part, une imprimante aura un contexte de périphérique contenant des informations sur les capacités de l'imprimante, la taille du papier, les couleurs disponibles et ainsi de suite.

Les types d' exécutables

Un « exécutable » est un fichier qui contient un code qui peut être exécuté par le processeur. Intéressons-nous, dans l'immédiat, à deux d'entre eux ; ceux qui ont l'extension de fichier *exe* (une application) et ceux avec l'extension *dll* (bibliothèque de liens dynamiques).

Pour pouvoir être qualifié d'exécutable Windows, le fichier doit être en format PE (Portable Executable). Comme son nom l'indique, ce type de fichier est destiné à être portable entre différentes plates-formes, de sorte qu'il puisse fonctionner sur un ordinateur équipé indifféremment avec un processeur Intel, AMD, MIPS, Alpha, Powerpc, ou RISC, par exemple, et à condition bien sûr que le système d'exploitation Windows soit chargé et que le fichier PE soit dans sa version correcte pour ces processeurs.

Windows sait que l'exécutable est un fichier PE parce que la signature « PE » apparaît dès le début du fichier. Un fichier non-PE, par exemple un exécutable DOS, n'a pas cette signature et Windows peut alors prendre toutes dispositions appropriées pour son exécution.

Une Dll est utilisée si son code ou ses données doivent être partagés par plusieurs applications. Windows utilise les Dlls pour stocker le code de ses API. La Dll est alors réputée avoir des *exports*. Cela réduit considérablement la taille de l'Exe individuel, car on sait dès lors que le code sera disponible dans une DLL et utilisable en tant que tel par l'Exe. L'un des champs importants du fichier PE est précisément la liste des *imports*. Il s'agit d'une liste de fonctions sur lesquelles l'Exe repose et qu'il peut avoir besoin d'appeler pendant son fonctionnement. Cette liste détient également le nom de la Dll contenant la fonction. Lors du chargement du Exe, Windows vérifie que toutes les fonctions et Dlls sont disponibles. Sinon, il ne lance pas le programme. Vous pouvez tester vous-même cette situation en essayant de lancer un programme destiné uniquement à Windows NT sur Windows 98. Il y a des chances pour que quelques-unes des fonctions ou

DLLs soient absentes. Cela peut également se produire si votre application appelle une API qui est disponible uniquement dans des versions ultérieures de Windows.

Vous pouvez éviter ce genre de problème, soit en fournissant différentes versions de votre programme exécutables spécialisées selon les versions de Windows potentiellement en possession des utilisateurs, soit en programmant de telle sorte que l'API appropriée soit appelée lors de l'exécution en fonction de la version Windows installée. L'identification de la version du Windows résident peut être obtenue au moyen de l'API *GetVersionEx*. Fort de cette information, vous pouvez alors appeler l'API qui convient à cette version du système d'exploitation. Vous devez veiller à ne pas appeler l'API de la manière habituelle, sinon elle apparaît dans la liste des imports, ce qui pourrait arrêter votre programme en cours d'exécution dès le début. Au lieu de cela, réalisez cet appel en utilisant les API *LoadLibrary* (qui charge la Dll concernée si ce n'est pas déjà le cas) et *GetProcAddress* (qui trouve l'adresse de l'API concernée dans la Dll).

Tout comme Windows utilise des Dlls vous pouvez également écrire de tels modules, généralement pour les expédier avec le programme que vous avez décidé de diffuser. Vous pouvez procéder ainsi si vous expédiez plus d'un programme et que la Dll détient du code ou des données partagés. Vous pourriez également recourir à une Dll à seule fin de réduire la taille des inévitables mises à jour. Dans ce cas, au lieu de mettre à jour l'ensemble du fichier Exe, vous n'auriez affaire qu'aux éléments susceptibles d'être mis à jour régulièrement dans une Dll.

Annexe H
Pour les débutants... en débogage symbolique

Je ne saurais trop vous recommander d'apprendre à utiliser un débogueur. Si vos programmes franchissent victorieusement l'étape de l'assemblage et de l'édition de liens, mais refusent pour autant de fonctionner, vous resterez dans un brouillard opaque quant à la raison de cette situation jusqu'à ce que vous vous décidiez à placer vos programmes défaillants sous le contrôle du débogueur. Souvent, une session de débogage affichera l'erreur instantanément et pourra vous épargner un temps fastidieux passé à essayer différentes choses et à tenter parfois des manœuvres aussi hasardeuses que risquées. Lors de la programmation pour Windows, vous avez besoin d'un bon débogueur Windows conçu pour ce travail.

Voici les différents points traités dans cette annexe :

H1 Qu'est-ce qu'un débogueur ?

Un débogueur est un logiciel qui permet d'exécuter un autre programme (le «debuggee») dans des conditions particulières et étroitement contrôlées. Il vous permet de progresser pas à pas dans le code du programme. Le processeur exécute une seule instruction à la fois, et vous pouvez visualiser immédiatement son effet sur les registres et flags, la pile et les zones mémoire du programme examiné. Habituellement, un débogueur vous permet de choisir de tracer à l'intérieur des CALL ou de les exécuter sans en analyser le contenu. Un débogueur vous permet également de définir des points d'arrêt où vous pouvez arrêter l'exécution et procéder à l'analyse à partir de là, ou d'exécuter le programme suspect jusqu'à ce que quelque chose aille mal.

H2 A quoi servent les symboles ?

Les symboles sont les *labels* de données et de code de votre script source. Un label de *données* se rapporte à une adresse dans la section de données et existe lorsque la donnée est déclarée. Par exemple,

```
SmallBuffer DD 20h DUP 0
```

crée un tampon de 32 dwords initialisés à zéro, avec le label "SmallBuffer". Ce tampon peut alors être invoqué sous ce nom tout au long de votre code source. Un label de code se rapporte à une adresse dans la section de code et peut être libellé comme suit :

```
Procedure66:
```

Ceci permettra d'établir le label "Procedure66" dans la section de code. La procédure peut alors être appelée en utilisant la forme de code

```
CALL Procedure66
```

Un label de données ou de code ne doivent pas être au début de la zone de données ou de code. Vous pouvez diviser les zones de données de telle sorte que le label ne porte que sur une partie de celui-ci, et un label de code peut être sur une ligne dans la section de code.

H3 Qu'est-ce qu'un débogueur symbolique ?

Un débogueur symbolique connaît l'adresse des symboles et est capable de les afficher dans le désassemblage. Ici, par exemple, le débogueur visualise, dans le désassemblage, le label de code du début de la procédure et un second label quelques lignes plus loin. Les références de données dans le désassemblage sont montrées par label de données. Un débogueur symbolique

peut également utiliser les labels de code pour permettre à l'utilisateur d'établir des points d'arrêt, et peut afficher le contenu de la mémoire par référence à des labels de données. Les symboles sont connus par le débogueur soit parce que les informations de symbole sont intégrées dans l'exécutable, soit par ce qu'elles sont conservées dans un fichier séparé. Cela se fait au moment de l'édition de liens et est réalisé par le linker lequel, si on le lui demande, va trier les labels dans les fichiers objets et les mettre dans le fichier exécutable (ou dans un fichier séparé) pour pouvoir être lus par le débogueur en tant que symboles.

H4 Déboguage en Win32 - Besoins spéciaux

Un débogueur Win32 doit être spécialement conçu pour surveiller et traiter les événements du système, les messages et les actions. En particulier, il doit être en mesure d'observer et de rendre compte des messages échangés entre le programme à déboguer et le système d'exploitation, puisque c'est le principal moyen d'interaction entre les deux. Les messages peuvent être envoyés par le système, soit directement au programme, soit par le biais de la file de messages (prélevée dans la boucle de messages). Lors de l'attente d'un retour de *GetMessage* dans la boucle de message, le débogueur doit être capable de capter et de surveiller une exécution ailleurs dans le programme induite par les messages envoyés à des procédures de fenêtre. Le débogueur doit également être en mesure de déboguer des applications multithreads, de surveiller l'activité de chaque thread et de montrer l'interaction entre les threads. Il doit pouvoir afficher des zones de mémoire et de pile virtuelles et des valeurs de registres et identifier les zones de mémoire uniques pour le système d'exploitation. Il doit être capable de retracer l'exécution à l'intérieur des Dlls. Il doit être en mesure de traiter les exceptions à analyser causées par le programme et permettant à ce dernier de poursuivre son exécution sans crash. Il doit pouvoir signaler les erreurs indiquées par les APIs. Enfin, le débogueur doit être capable de fermer le programme à analyser de manière "propre".

H5 Techniques spéciales lors du débogage dans Win32

Voici quelques-unes des actions que vous devez pouvoir conduire avec un débogueur Win32 bien conçu :

- Mise en place d'un point d'arrêt pour exécuter le programme jusqu'à un message quelconque ou jusqu'à un message particulier, l'un et l'autre dans le cadre d'une procédure de fenêtre particulière.
- Visualiser la séquence et le détail des messages.
- Étude de l'exécution de nouveaux threads et examen de la manière dont Windows partage le temps processeur entre eux.
- Affichage du contenu de la pile en détail.
- Supervision des erreurs d'API.
- Changement de la valeur des registres (ordinaires, à virgule flottante ou MMX) ou des flags à l'exécution pour corriger d'éventuelles erreurs ou vérifier votre code.
- Possibilité de suspendre l'exécution des boucles continues.
- Visualiser et vérifier les opcodes produits par votre assembleur ou compilateur, et faire de même avec tout code exécutable sous forme de mnémoniques assembleur.
- Visualisation en tant que code ou de donnée des zones mémoire et de mémoire représentées par les symboles qui sont chargés.
- Visualisation du contexte mémoire du programme testé ainsi que ses Dlls (les images exécutables) tels que chargés en mémoire.
- Recherche du contexte mémoire du programme et la mémoire partagée pour les chaînes ou valeurs spécifiques et affichage des résultats.
- Visualisation des ressources du programme et de ses Dlls telles que chargées en mémoire et (sous réserve du respect du copyright), extraction des ressources utiles à partir de fichiers exécutables.

H6 Autres techniques utilisables (en substitution au débogage)

H6.1 Examen de votre code

Souvent, il n'y a pas d'autre solution que de scruter attentivement votre code source pour y identifier l'endroit où l'erreur est survenue. Il vous sera beaucoup plus facile de progresser dans le développement et le test de votre code source si vous procédez de manière incrémentale. En d'autres termes ajoutez de petits morceaux de code, puis astreignez-vous à les tester immédiatement plutôt que procéder par grandes sections, qui peuvent contenir plus d'une défaillance et même interagir.

H6.2 Retracer les étapes avant le bug

Essayez de repérer la séquence d'événements qui a précédé le bug chaque fois que le défaut se produit. Puis reprenez cette séquence de nouveau pour vérifier que le défaut se produit bien à cet endroit. Essayez d'isoler le défaut en supprimant certaines des étapes ou en prenant d'autres mesures. Obtenez, de la sorte, une séquence aussi courte que possible. Ce processus vous aidera à identifier le coupable le plus probable tout en réduisant le nombre de procédures que vous devez vérifier.

H6.3 Ralentissez le système au moment de l'exécution pour les problèmes de dessin

Si vous avez un problème de dessin Windows dans lequel vous soupçonnez que Windows dessine quelque chose à l'écran puis le masque aussitôt en dessinant quelque chose d'autre par-dessus, essayez de ralentir le programme en insérant le processus suivant à un endroit approprié :

```
PUSH 1000
CALL Sleep
```

Cette courte séquence appelle l'API *Sleep* (veille) et introduit une temporisation de 1000 millisecondes. L'API va agir uniquement sur le thread qui l'appelle, de sorte que, si vous soupçonnez qu'un autre thread est en train d'écraser le matériau du premier thread, cette technique le mettra en évidence. [*Notez qu'il n'est pas recommandé, dans une application Windows, d'avoir plus d'un thread responsable du dessin à l'écran ou de la gestion des fenêtres car cela est susceptible d'induire le système en erreur*].

Si l'écrasement apparait au sein de votre propre code en réponse au message WM_PAINT, insérez alors l'appel à l'API *Sleep* à des endroits appropriés dans votre processus de dessin pour découvrir exactement où l'écrasement se produit.

Windows va parfois dessiner par-dessus ce que vous avez essayé de dessiner à l'écran. Cela peut se produire, par exemple, si le système est persuadé qu'une autre fenêtre devrait apparaître au sommet du dessin (c'est-à-dire une autre fenêtre qui est plus élevée dans la hiérarchie-Z) ou si le système estime que la zone que vous avez dessinée est "invalide"(nécessite d'être peinte parce qu'elle s'est déplacée ou a été découverte). Gardez présent à l'esprit que le système ne fera pas de dessin plus loin dans une fenêtre jusqu'à ce que le thread qui a créé cette fenêtre ne soit revenu de son WndProc ou jusqu'à ce que DefWndProc soit appelé. Ainsi, vous pouvez insérer l'API *Sleep* avant de faire cela.

H6.4 Masquer des parties de votre code

Voici une autre technique utile pour identifier des problèmes d'écrasement par écriture, que ce soit à l'écran ou dans la mémoire. Elle consiste à éliminer provisoirement un code suspect en plaçant tout simplement un point-virgule au début de la ligne de code concernée, de sorte que l'assembleur en ignore le contenu, assimilé dès lors à un commentaire. Il suffit de voir alors si ce retrait corrige le problème. Bien sûr, vous devez être prudent et veiller notamment à ce que toutes les valeurs de registre et de mémoire nécessaires soient fournies aux lignes de code en aval, et que l'équilibre de la pile soit maintenu.

L'avantage de cette technique tient au fait qu'elle est facile à mettre en œuvre tout comme l'est le rétablissement à la situation antérieure.

H6.5 Ajouter un testeur visible à votre code au moment de l'erreur

La méthode consiste à afficher à l'écran une fenêtre de test lorsqu'une partie particulière du code est atteinte. Pour ce faire, on ajoute au programme en cours de développement le code et les données ci-dessous qui définissent le testeur dont il est question. La fenêtre de ce module affiche en hexadécimal la valeur du registre EAX. Le testeur enregistre tous les registres et flags ce qui le rend donc totalement transparent vis-à-vis du programme qu'il scrute. Si le testeur est appelé à nouveau, il ajoute une autre ligne à la fenêtre du testeur avec la valeur de EAX à ce moment. Chaque ligne est numérotée et peut être consultée par simple défilement. Il convient toutefois d'être prudent sur deux points. En premier lieu, si vous rencontrez des problèmes de peinture et de dessin, il se peut que le testeur aggrave la situation dans la mesure où il agit lui-même sur l'affichage selon des modalités identiques. En second lieu, lors de l'insertion du testeur dans un thread secondaire, il ne faut surtout pas l'appeler directement à partir de ce thread. Au lieu de cela, envoyez un message à la fenêtre principale et appelez le testeur à partir de là comme indiqué plus loin après le listing du testeur.

```
;******* LISTING DU TESTEUR *******
DATA SECTION
;
TESTER_MSG DD 7 DUP 0
```

```
;hWnd, +4=message, +8=wParam, +C=lParam, +10h=time, +14h/18h=pt
hTester DD 0
TesterThreadId DD 0
TEST_COUNTER DD 0
sHEXb DB '0123456789ABCDEF'
TEST_MESS DB 18D DUP 0
;
CODE SECTION
;
TESTER:                         ; le programme en cours de test appelle ce label, EAX contenant
                                ; la valeur à afficher en hexa et en décimal
PUSHFD                          ; sauvegarde des flags pour les restaurer ultérieurement
PUSH EAX,EBX,ECX,EDX,EDI,ESI    ; sauvegarde des registres qui vont être utilisés
MOV ESI,EAX                     ; sauvegarde du nombre à écrire dans ESI
CMP D[hTester],0                ; voir si le testeur de fenêtre existe déjà
JNZ >L200                       ; oui
MOV D[TEST_COUNTER],0           ; on force le compteur à démarrer à partir de zéro
CALL GetDesktopWindow           ; préparation pour que parent = bureau (vérifie que les
                                ; messages parents ne sont pas envoyés à l'application
                                ; et que la fenêtre est correctement dessinée)
PUSH 0,0,0,EAX
PUSH 208D,130D,30D,30D          ; hauteur, largeur et position de la fenêtre
PUSH 4C80000h+200000h+10000h+20000h+40000h      ; WS_CHILD+CAPTION+SYSMENU+WS_VSCROLL
                                                ;+WS_HSCROLL+WS_MINIMIZEBOX+SIZEBOX
PUSH 'Tester window'            ; titre chargé avec l'extension PUSH (voir manuel GoAsm)
PUSH 'LISTBOX'                  ; classe
PUSH 1h+8h                      ; WS_EX_DLGMODALFRAME + WS_EX_TOPMOST
CALL CreateWindowExA            ; constitution fenêtre avec retour du handle dans EAX
OR EAX,EAX                      ; une erreur s'est-elle produite ?
JZ >L202                        ; oui
MOV [hTester],EAX               ; mémorisation du handle
PUSH 8h,EAX                     ; prêt à afficher mais non activé
CALL ShowWindow
PUSH 12D
CALL GetStockObject             ; récupération du handle de ANSI_VAR_FONT détenu par Windows
PUSH 0,EAX,30h,[hTester]        ; 30h=WM_SETFONT
CALL SendMessageA               ; définition de la police de caractères pour la listbox
MOV EAX,ESI                     ; récupération du nombre à écrire
L200:
CALL WRITE_NUMBER
L202:
POP ESI,EDI,EDX,ECX,EBX,EAX     ; restauration de tous les registres affectés
POPFD                           ; restauration des flags
RET
;
WRITE_TESTCOUNTER:              ; écriture nombre décimal en EAX dans [EDI]
MOV EDI,ADDR TEST_MESS          ; cette fonction est appelée par WRITE_NUMBER
XOR EDX,EDX
XOR ECX,ECX                     ; ECX est utilisé comme un compteur
MOV EBX,10D                     ; EBX = 10 (décimal)
L100:
DIV EBX                         ; division EDX:EAX par 10 avec quotient dans EAX, reste en EDX
PUSH EDX                        ; mémorisation du reste en pile
INC ECX                         ; on incrémente le compteur de ceux qui sont en pile
XOR EDX,EDX
CMP EAX,EDX                     ; on regarde s'il y a quelque chose de plus à faire
JNZ L100                        ; oui
L101:
POP EAX                         ; inversion de l'ordre des chiffres utilisant la pile
ADD AL,48D                      ; conversion en ascii
STOSB                           ; écriture du chiffre dans le buffer
```

```
LOOP L101
MOV EAX,' .'
STOSD                          ; écriture nombre hexa en EAX dans [EDI]
RET
;
TEST_HEXWRITE:                 ; écrit en hexa le contenu de EAX dans [EDI]
PUSH ECX
MOV EBX,ADDR sHEXb
MOV ECX,8                      ; 8 caractères à afficher
L130:
ROL EAX,4                      ; AL = quartet supérieur de EAX obtenu par permutation
MOV DL,AL
AND EDX,0Fh                    ; on ne conserve que le quartet le moins significatif
MOV DL,[EBX+EDX]               ; DL = caractère correct extrait de la table
MOV [EDI],DL                   ; on écrit le caractère
INC EDI                        ; on pointe l'emplacement du caractère suivant
LOOP L130                      ; boucle jusqu'à ce que ECX = 0
POP ECX
RET
;
WRITE_NUMBER:                  ; affichage à l'écran du nombre contenu dans EAX
INC D[TEST_COUNTER]            ; incrémentation du compteur
CMP D[TEST_COUNTER],1000D      ; on regarde si le compteur dépasse maintenant 1000
JNA >L112                      ; non
MOV D[TEST_COUNTER],1          ; on dépasse 1000, alors on réinitialise le compteur
L112:
MOV EDI,ADDR TEST_MESS
MOV EAX,[TEST_COUNTER]
CALL WRITE_TESTCOUNTER         ; écriture du nombre en EAX en [edi] sous format décimal
MOV EAX,ESI                    ; restauration du nombre du programme sous test
CALL TEST_HEXWRITE             ; écriture du nombre en EAX en [edi] sous format hexa
MOV B[EDI],'h'                 ; caractère 'h' de fin marquant que la valeur est hexa
MOV B[EDI+1],0                 ; caractère NULL à la fin
PUSH ADDR TEST_MESS
PUSH 0,180h,[hTester]          ; 180h = LB_ADDSTRING
CALL SendMessageA
MOV EDI,EAX                    ; mémorisation de l'index en EDI
CMP EDI,999D                   ; voir si l'index dépasse 999 (indexé à zéro)
JNA >L118                      ; non
PUSH 0,0,182h,[hTester]        ; 182h = LB_DELETESTRING
CALL SendMessageA              ; suppression de la toute première entrée (on s'assure
                               ; qu'il n'y a pas surcharge)
L118:                          ; défile maintenant pour garantir que le dernier mis est visible
DEC EDI                        ; décrémentation de l'index
PUSH 0,EDI                     ; EDI pointe la chaîne permettant de garantir la visibilité
PUSH 197h,[hTester]            ;197h = LB_SETTOPINDEX
CALL SendMessageA              ;scroll de la listbox maintenant
L123:
PUSH [hTester]
CALL UpdateWindow              ; garantit que Windows affiche des changements
RET                            ; dès qu'il est prêt
```

Précautions particulières si vous appelez le testeur à partir d'un thread secondaire

Si vous appelez le testeur directement à partir d'un thread secondaire (c'est-à-dire d'un thread autre que celui qui a construit la fenêtre principale de votre application), le système va être désorienté et vous rencontrerez des problèmes. Au lieu de cela, envoyez un message défini par l'utilisateur à la fenêtre principale de votre application et appelez le testeur à partir de la procédure de fenêtre principale. Utilisez le code suivant pour envoyer un message défini par l'utilisateur à la fenêtre principale de votre application :

```
THREAD_TESTER:
PUSH EAX,ECX,EDX
```

```
PUSH 6,EAX,411h,hWnd
CALL SendMessageA
POP EDX,ECX,EAX
RET
```

Ce code envoie le message 411h à la procédure de la fenêtre principale (dont le handle est mémorisé dans hWnd) de la fenêtre. Notez que les messages d'utilisation définis doivent être de valeur 400h ou plus. Le nombre (en EAX) à écrire dans la fenêtre du testeur est envoyé à la procédure de fenêtre sur la pile comme wParam. Ici aussi, la valeur de lParam est fixée à 6 pour une identification plus poussée. Ce code est approprié dans la mesure où il n'est pas important que le thread secondaire doive attendre l'apparition de la fenêtre du testeur (*SendMessage* ne retourne pas jusqu'au retour effectif de la procédure de la fenêtre). Si cela constitue un problème, utilisez *PostMessage* à la place.

H6.6 Affectation d'une touche du clavier pour visualiser la fenêtre du testeur au cours du développement de votre application

On peut enfin faire en sorte qu'une touche spécifique du claver, identifiée comme telle par WM_KEYDOWN, provoque l'appel du testeur. Vous pouvez charger dans EAX n'importe quelle valeur de la mémoire que vous voulez étudier. Lorsque votre programme est terminé et prêt à être publié, assurez-vous que ce code est supprimé. En tout cas, il constitue un moyen rapide pour afficher les valeurs de mémoire sans avoir à démarrer le débogueur. Encore une fois, vous pouvez appeler le testeur autant de fois que vous le souhaitez, les résultats défilant tout simplement, en ce cas, dans la fenêtre. Cette technique peut également être utilisée pour vérifier les résultats de votre codage plus en détail. Par exemple, arrangez-vous pour appeler la fonction en cours de test lorsque la touche est pressée, en lui passant différentes valeurs. Lire le résultat pour vérifier qu'elle fonctionne bien.

Annexe I
Comprendre... la Pile

Tous les programmes font un usage intensif de la pile au moment de l'exécution. Si vous programmez dans un langage de haut niveau, vous ne pouvez absolument pas avoir conscience que le compilateur en fait usage. Mais en tant que programmeur assembleur vous ne pouvez vous en abstraire dans la mesure où la pile est l'un des principaux outils à votre disposition. En l'utilisant directement, vous pouvez réaliser son potentiel élevé dans vos programmes. Bien que vous puissiez programmer en assembleur sans savoir quoi que ce soit à propos de la pile, il est utile (et même recommandé) d'en appréhender le fonctionnement.

Dans la **partie 1**, vous trouverez des informations qui vous seront indispensables si vous envisagez de vous lancer sérieusement dans la programmation en assembleur.

La **partie 2**, d'un abord plus complexe, n'est pas indispensable mais elle propose une excursion en profondeur de ce dispositif.

I1 Partie 1

I1.1 Caractéristiques et avantages de la pile

Fondamentalement, la pile est une zone de dword (zones de données 32 bits) en mémoire que votre application peut utiliser pour stocker temporairement les données lors de son exécution. Elle a certaines caractéristiques et des avantages réels par rapport aux autres types de stockage de mémoire (sections de données et zones de mémoire lors de l'exécution) :

- Le processeur écrit et lit sur la pile très rapidement, car il est optimisé pour cet usage.
- Les instructions élémentaires PUSH et POP peuvent être utilisées pour écrire et lire sur la pile. Elles sont très compactes : un seul octet pour la manipulation de registres, cinq octets lors de l'utilisation de labels de mémoire ou de pointeurs vers des adresses de mémoire.
- Sous Windows, la pile est agrandie dynamiquement lors de l'exécution par blocs de 4K. Ceci permet d'éviter le gaspillage de mémoire.

I1.2 Usages courants de la pile

La pile peut être utilisée pour :

- Préserver la valeur de registres dans des fonctions

Exemple :

```
PUSH EDI                ; sauvegarde en pile du handle du fichier
CALL CALCULATE          ; réalisation de calculs (utilise EDI)
POP EDI                 ; restaure le handle du fichier
CALL CLOSE_FILEHANDLE   ; fermeture du handle du fichier contenu dans EDI
```

- Préserver des données en mémoire

  Exemple : supposons que vous ayez soigneusement calculé le nombre de widgets et que vous vouliez écrire des détails sur ceux-ci à la fois sur l'écran et dans un fichier. Vous pouvez utiliser le code suivant :

```
PUSH [NOOF_WIDGETS]     ; mémorisation en pile du nombre de widgets
L2:
CALL REPORT_WIDGET      ; écrit les détails du widget sur l'écran
DEC D[NOOF_WIDGETS]     ; décrémente le nombre de widgets
JNZ L2                  ; continue avec le suivant tant qu'il n'est pas nul
POP [NOOF_WIDGETS]      ; restaure le nombre de widgets
CALL WRITETO_FILE       ; et effectue un report similaire sur le fichier
```

- Déplacer des données dans la mémoire sans utiliser de registres

  Exemple : Supposons que vous vouliez déplacer le nombre de widgets de l'exemple précédent à un autre label de mémoire. Vous pouvez utiliser :

```
MOV EAX,[NOOF_WIDGETS]
MOV [COPYOF_NOOF_WIDGETS],EAX
```

  Mais vous pouvez tout aussi efficacement utiliser le code suivant qui évite toute utilisation de registre :

```
PUSH [NOOF_WIDGETS]
POP [COPYOF_NOOF_WIDGETS]
```

  Dans la mesure ou le registre EAX n'est plus mis à contribution, il ne perd donc pas sa valeur et peut donc être affecté à un autre usage.

- Renverser l'ordre de données :

  Exemple : Vous pouvez mettre à profit le mécanisme “Last In, First Out” à la base du fonctionnement de la pile pour inverser l'ordre de données. On en trouve une utilisation pratique dès lors que l'on se propose d'afficher à l'écran une valeur en format décimal. Voir l'exemple du I1.8 plus loin.

- L'utilisation de la pile par CALL et RET
- L'utilisation de la pile pour passer des paramètres

I1.3 Le Pointeur de Pile : registre ESP

Le registre ESP (littéralement "Extended Stack Pointer ") pointe l'emplacement mémoire correspondant au sommet de la pile. C'est le point où les instructions qui utilisent la pile (PUSH, POP, CALL et RET) effectuent leurs opérations avec la mémoire. A quelques exceptions près (réservation d'une zone de mémoire sur la pile, par exemple), ESP n'est géré que par les instructions PUSH, POP, CALL et RET et le programmeur intervient fort peu sur ce registre. Il en sera question un peu plus loin.

Le registre EBP (littéralement "Extended Base Pointer") est traditionnellement positionné par le programmeur à un endroit et à un moment particuliers sur la pile, de sorte que les données puissent y être lues et écrites au moyen de l'*adressage de base d'index* (*base index addressing*). Par exemple, dans l'instruction `MOV EAX,[EBP+8h]`,le registre EBP est utilisé comme un index pointant une zone de la pile et cette instruction copie un mot situé 8 octets plus bas dans la pile dans le registre EAX. L'utilisation traditionnelle de EBP à cet effet est l'héritage direct de la programmation 16 bits. Dans ce mode, utilisant la technique de mémoire segmentée, on trouvait un registre BP d'une taille de 16 bits qui, comme tous les registres d'index, était attaché à un registre de segment également de 16 bits. Par défaut, BP était attaché au registre de segment SS (Stack Segment), ce dernier pouvant être modifié par *override*. Aujourd'hui, sous 32 bits, les segments ne sont plus exploités et chaque programme fonctionne dans ses propres 4 Go d'espace d'adressage. EBP peut donc pointer l'ensemble de cet espace et ne se limite plus à traiter la pile. Il peut donc maintenant être utilisé comme un registre à usage général bien que le poids de son passé le cantonne au traitement de zones particulières de la pile principalement pour accéder aux paramètres transmis aux fonctions et routines callback et pour traiter les données locales.

I1.4 PUSH et POP de données sur la pile

La pile peut être assimilée à la desserte de plateau-repas d'un traiteur. Cet ustensile fonctionne selon le principe “dernier entré, premier sorti”. La dernier plateau poussé sur la desserte en utilisant l'instruction PUSH sera le premier retiré en utilisant l'instruction POP. Le pointeur de la pile dans ESP pointe toujours sur cette plaque supérieure.

Regardons cela sous une forme plus traditionnelle. Supposons que la valeur de ESP soit

64FE3Ch et vous ayez les instructions suivantes dans votre code source :

```
PUSH 2
PUSH [hWnd]
PUSH ADDR STRING
```

A l'issue de ces trois instructions, ESP serait 64FE30h (soit 12 octets ou 3 dwords de moins) et la pile affiche alors l'activité suivante :

ESP est ici →	64FE30h	Adresse de STRING
	64FE34h	Valeur contenue par hWnd
	64FE38h	Le nombre 2
	64FE3Ch	

Notez que chaque instruction **PUSH** commence par *réduire* la valeur de ESP de 4 octets, puis mémorise la donnée mentionnée en tant qu'opérande au nouvel endroit pointé par ESP.

Voyons maintenant ce qu'il en est avec les POP. Avec les mêmes valeurs sur la pile, utilisons les instructions suivantes :

```
POP EAX
POP EBX
POP ECX
```

L'intruction **POP** réalise l'inverse du PUSH : le contenu de l'adresse pointée par ESP est copié dans l'opérande puis ESP se voit incrémenteé de 4 octets. Examinons l'évolution du comportement de la pile :

	64FE30h	Adresse de STRING
	64FE34h	Valeur contenue par hWnd
	64FE38h	Le nombre 2
ESP est ici →	64FE3Ch	

La première chose à noter ici est que, après ces trois instructions, ESP revient à 64FE3Ch, soit la valeur initiale de l'exemple précédent. Cela signifie que ESP a été rétabli à l'équilibre. Il s'agit ici d'un concept important car la stabilité du programme en dépend.
Le registre EAX contient maintenant l'adresse de STRING, EBX reçoit la valeur détenue par hWnd et ECX reçoit le nombre 2. Donc, les données stockées sur la pile en ont été extraites dans l'ordre inverse de celui dans lequel elles y avaient été poussées.
Notez également que les données sur la pile demeurent toujours présentes bien qu'ayant été extraites. Cette situation résulte du fait que l'instruction POP n'écrit pas sur la pile et se contente d'en lire simplement la donnée dans la deuxième partie de l'instruction (appelée "opérande").

I1.5 Préservation de la valeur des registres dans les fonctions

Les programmes écrits en assembleur sont rapides car ils utilisent des registres autant que possible. Cela signifie souvent cependant que le contenu des registres à un moment donné doit être conservé en prévision d'une utilisation ultérieure. Ainsi, par exemple, supposons un handle de fichier en EDI, et qu'après avoir effectué quelques calculs utilisant également EDI vous auriez éventuellement besoin de fermer ledit handle du fichier détenu initialement par EDI. Voici comment opérer les sauvegardes nécessaires :

```
PUSH EDI                ; sauvegarde du handle du fichier
CALL CALCULATE          ; calculs utilisant EDI
POP EDI                 ; restauration du handle du fichier
CALL CLOSE_FILEHANDLE   ; fermeture du handle du fichier contenu en EDI
```

Comme alternative, vous pourriez également envisager de préserver EDI au sein de la procédure de calcul elle-même, par exemple :

```
CALL CALCULATE          ; calculs utilisant EDI mais sauvegardant sa valeur initiale
CALL CLOSE_FILEHANDLE   ; fermeture du handle du fichier contenu en EDI
```

où la fonction CALCULATE présenterait l'allure suivante :

```
CALCULATE:
```

```
PUSH EDI     ; sauvegarde du handle du fichier
   .
   .         ; code utilisant EDI et modifiant donc sa valeur
   .
POP EDI      ; restauration du handle du fichier
RET
```

Une autre raison pour laquelle un registre peut devoir être préservé doit être envisagée dans le contexte où une fonction particulière est appelée de l'extérieur (par une autre fonction dans le même programme, d'un autre programme ou par le système). Dans la plupart des cas, assurez-vous que EBP, EBX, EDI et ESI sont préservés. C'est assurément l'exigence d'un programme C dans le cas où il appelle une routine écrite en assembleur ; c'est également l'exigence de procédures callback appelées par Windows. Un exemple d'une telle procédure callback est la classique procédure de fenêtre qui est utilisée par le système pour transmettre des messages à une fenêtre dans l'application. Dans de telles circonstances, vous devez vous assurer que ces registres sont préservés en utilisant, par exemple :

```
PUSH EBP,EBX,EDI,ESI
   .
   .           ; votre code s'écrit ici
   .
POP ESI,EDI,EBX,EBP
```

Bien sûr, si votre code ne change pas ces registres, rien de vous empêche d'omettre les PUSH et POP concernés, mais il peut être considéré de bonne pratique que de les maintenir en l'état dans l'éventualité où vous augmenteriez ultérieurement votre code et oublieriez de veiller à ce que ces registres soient préservés. Notez comment les POP sont tous dans l'ordre inverse des PUSH : cela est dû au principe "dernier entré, premier sorti" qui est la nature-même de la pile. A noter également dans le code ci-dessus que les registres sont positionnés dans l'ordre alphabétique. Cela n'est qu'un moyen mnémotechnique destiné à faciliter le contrôle de cohérence des PUSH et POP.

Notez que GoAsm met à votre disposition l'instruction USES qui préserve et restaure automatiquement les registres selon une écriture simplifiée.

I1.6 Préservation des données en mémoire

Tout comme vous pouvez conserver la valeur d'un registre à l'aide de la pile, vous pouvez utiliser cette dernière pour conserver les données en mémoire. Supposons par exemple que vous ayez soigneusement calculé le nombre de widgets et que vous vouliez écrire des détails sur les widgets à la fois à l'écran et dans un fichier, vous pouvez utiliser le code suivant :

```
PUSH [NOOF_WIDGETS]   ; mémorisation en pile du nombre de widgets
L2:
CALL REPORT_WIDGET    ; écriture des détails du widget à l'écran
DEC D[NOOF_WIDGETS]   ; décrémentation du nombre de widgets
JNZ L2                ; on continue avec le suivant tant que le nombre de widgets n'est pas nul
POP [NOOF_WIDGETS]    ; on restaure le nombre de widgets de départ
CALL WRITETO_FILE     ; et on fait un report similaire en direction du fichier
```

I1.7 Déplacement de données en mémoire sans utiliser de registres

Supposons que vous vouliez déplacer le nombre de widgets de l'exemple précédent à un autre label de mémoire. Vous pouvez utiliser :

```
MOV EAX,[NOOF_WIDGETS]
MOV [COPYOF_NOOF_WIDGETS],EAX
```

Mais vous pouvez tout aussi efficacement utiliser le code suivant qui évite toute utilisation de registre :

```
PUSH [NOOF_WIDGETS]
POP [COPYOF_NOOF_WIDGETS]
```

Dans la mesure ou le registre EAX n'est plus mis à contribution, il ne perd donc pas sa valeur et peut donc être affecté à un autre usage.

Notez également que l'on peut déplacer facilement des données en mémoire avec l'instruction MOVS.

I1.8 Renversement de l'ordre des données

Vous pouvez mettre à profit le mécanisme “Last In, First Out” (dernier entré, premier sorti) à la base du fonctionnement de la pile pour inverser l'ordre de données. On en trouve une utilisation concrète dès lors que l'on se propose d'afficher à l'écran une valeur en format décimal. Voici un exemple (où EAX détient la valeur binaire à afficher en décimal et EDI, la position dans la mémoire du tampon qui contiendra la chaîne à afficher):

```
XOR EDX,EDX    ; mets EDX à zéro
XOR ECX,ECX    ; mets ECX à zéro (utilisé comme compteur)
MOV EBX,10     ; EBX contient toujours la valeur 10
CLD            ; incrémentation automatique de EDI avec STOSB
L2:
DIV EBX        ; Div EDX:EAX par 10 = quotient en EAX, reste en EDX
PUSH EDX       ; on place le reste sur la pile
INC ECX        ; compte le nombre d'empilages réalisés
XOR EDX,EDX    ; mets EDX à zéro
CMP EAX,EDX    ; il y a-t-il quelque chose d'autre à faire ?
JNZ L2         ; oui
L3:            ; inverse maintenant l'ordre des chiffres à afficher
POP EAX        ; retire de la pile l'enregistrement le plus récent
ADD AL,48      ; conversion de ce chiffre en nombre ASCII
STOSB          ; mémorisation du chiffre ASCII dans le buffer
LOOP L3        ; on continue la boucle tant que ECX n'est pas nul
```

Attardons-nous un instant sur ce code. Supposons que la valeur initiale de EAX à traiter soit le nombre décimal 123. La première division par 10 met 12 dans EAX (quotient) et 3 dans EDX (reste). La valeur 3 est poussée en pile par `PUSH EDX`. La seconde division de 12 par 10 met 1 en EAX et 2 dans EDX. La valeur 2 est poussée en pile de la même manière. La troisième division de 1 par 10 met zéro dans EAX et 1 dans EDX. Le reste 1 est poussé en pile à son tour. L'instruction `CMP EAX, EDX`[9] constate alors que le quotient est nul (EAX = 0), marquant ainsi la fin des divisions successives et le code est projeté au label L3. Le registre ECX affiche la valeur 3. Il a compté le nombre de divisions et, corrélativement, le nombre de chiffres du nombre traité. Chacun de ces 3 chiffres est maintenant extrait de la pile à son tour. S'agissant de valeurs binaires, on y ajoute la valeur 48, les transformant de la sorte, en caractères ASCII qui sont stockés dans la mémoire tampon et prêts à être affichés à l'écran ultérieurement.

I1.9 Comment CALL et RET utilisent la pile

L'instruction CALL est beaucoup utilisée en programmation. Elle a pour fonction de détourner l'exécution des instructions vers une procédure particulière (ou « fonction ») située en un autre point du programme. Lorsque cette procédure est terminée, une instruction particulière (RET) renvoie l'exécution juste après le CALL à l'origine de ce détournement. L'appel de procédures contribue à préserver la clarté de votre code source en limitant notamment la répétitions de blocs de code identiques. En voici un exemple :

```
MOV EAX,EDX
CALL CALCULATE_ASSETS
MOV [ASSETS],EAX        ; sauvegarde en mémoire le résultat de la procédure appelée par CALL
```

On devine aisément qu'un travail important est effectué par la procédure CALCULATE_ASSETS, mais il n'est pas nécessaire de se soucier de la façon dont elle fonctionne lorsqu'on regarde cet extrait du script source.

L'utilisation des CALLs contribue également à rendre votre code modulaire. La procédure ci-dessus peut également être utilisée par d'autres programmes. Si vous le voulez, vous pouvez la considérer comme un « objet ». Fondamentalement, elle est bien un objet de programmation orientée objet.

Alors, comment le processeur sait-il où poursuivre le traitement en retour de l'appel ? Eh bien, c'est très simple : il insère l'adresse de retour sur la pile !

Observons la pile lorsque cela se produit.

[9] On a préalablement mis EDX à zéro par un `XOR EDX,EDX` ce qui fait que la comparaison revient en fait à `CMP EAX,0`. Cette technique de codage, outre qu'elle rend le code plus compact, améliore la rapidité d'exécution. Si le quotient est nul, cela signifie que la série de divisions par 10 est terminée et que l'on dispose, dès lors, de tous les restes permettant de reconstituer tous les chiffres significatifs du nombre initial.

Supposons que la valeur du registre ESP soit à nouveau 64FE3Ch et que vous ayez les instructions suivantes dans votre code source :

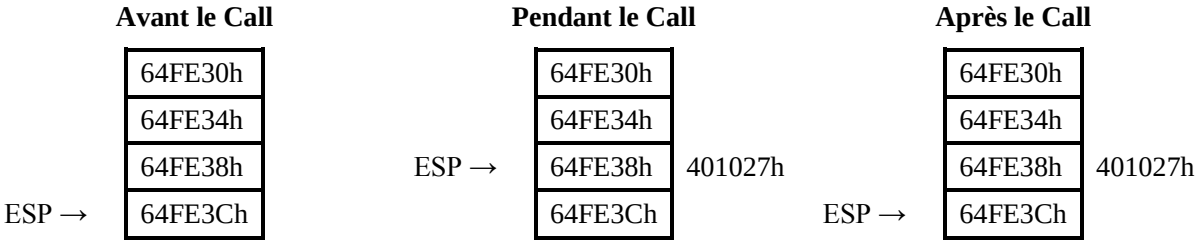
```
401020: MOV EAX,EDX
401022: CALL CALCULATE_ASSETS
401027: MOV [ASSETS],EAX      ; sauvegarde en mémoire le résultat de la procédure
                              ; appelée par CALL
```

Nous avons ajouté ici l'adresse du pointeur d'instructions (EIP) à gauche du code pour mieux illustrer ce qui se passe. Après la première instruction, ESP est évidemment encore à 64FE3Ch, la pile n'ayant pas changé car une instruction MOV ne modifie pas la pile de quelque façon que ce soit. Mais quand l'instruction suivante `CALL CALCULATE_ASSETS` est exécutée, le processeur commence par pousser sur la pile l'adresse de retour 401027h, puis détourne l'exécution du programme au label `CALCULATE_ASSETS`. Le code de la procédure s'exécute alors jusqu'à rencontrer une instruction RET (retour à l'appelant) qui retire de la pile la valeur stockée précédemment et la place dans le pointeur d'instructions, permettant au programme de reprendre son cours initial. Voici la forme que prend la procédure ainsi décrite :

```
CALCULATE_ASSETS:
                  ; code varié ici
RET               ; retour à l'appelant
```

L'instruction RET provoque en effet un POP en EIP, autrement dit ce qui est à [ESP] est chargé dans EIP (le pointeur d'instruction) et ESP est augmenté de 4 octets.

Considérons donc la pile avant, pendant et après les instructions qui précèdent :



Remarquez comment, après le CALL, ESP est rétabli à l'équilibre.

I1.10 Importance de l'équilibre de la pile

Nous avons vu comment une procédure peut être appelée et comment l'adresse de retour d'exécution est stockée sur la pile. Cela dit, il faut considérer que les procédures elles-mêmes font souvent appel à d'autres procédures, qui en appellent également d'autres et ainsi de suite. Vous pourriez avoir, par exemple :

```
CALCULATE_ASSETS:
CALL CALCULATE_FIXEDASSETS
RET                          ; retour à l'appelant
```

et

```
CALCULATE_FIXEDASSETS:
 .
 .                       ; code varié ici
 .
CALL GET_COSTVALUES
CALL ADJUSTFOR_DEPRECIATION
ADD ESP,4                ;place ESP en déséquilibre
RET
```

Ici, chaque partie de la tâche est divisée en différentes composants. Maintenant, supposons que la procédure CALCULATE_FIXEDASSETS ajoute 4 à ESP par erreur. Si cela se produit, lorsque l'instruction RET sera exécutée, le pointeur d'instruction EIP se verra chargé avec une valeur erronée et le programme partira dans une direction imprévisible avant de se bloquer.

Pendant qu'une procédure s'exécute, il est courant que ESP soit modifié (par exemple lors de la réservation d'un espace de données sur la pile), mais il est néanmoins essentiel de veiller à ce que l'équilibre de la pile soit rétabli lorsqu'une procédure est sur le point de se terminer. Lorsque des paramètres lui ont été envoyés

via la pile, l'équilibre peut être rétabli avec l'instruction RET xx ou en réajustant ESP en fonction du nombre de ces paramètres.

L'équilibre de la pile est également important pour le retour à Windows au terme d'une application, même la plus minimaliste. Dans le genre, l'application qui suit est assez remarquable :

```
START:
RET
```

Où START est le point d'entrée de l'application. En fait, normalement, Windows *appelle* votre application à partir de Kernel32.dll, et donc un simple RET termine le programme de la manière la plus heureuse qui soit.

I1.11 Utilisation de la pile pour passer des paramètres

Structurellement, les API Windows s'attendent à recevoir leurs paramètres sur la pile. Lorsque vous appelez une API vous devez donc pousser les paramètres sur la pile (instructions PUSH) de telle sorte qu'ils puissent être récupérés par elle. Il en va ainsi dans le code qui suit :

```
PUSH 1,[hButton]
CALL EnableWindow   ; autorise le bouton
```

Dans cet exemple, vous poussez sur la pile la valeur 1 en premier (flag ENABLE), suivi par le handle de la fenêtre que vous souhaitez activer. Windows utilise la convention d'appel STDCALL du C pour ses API ce qui fait qu'au retour de l'API la pile sera ramenée à l'équilibre. Cette convention implique également que EBP, EBX, ESI et EDI soient toujours restaurés à leur valeur précédente par l'API.

Un autre aspect de la convention STDCALL est que les paramètres sont toujours poussés *de droite à gauche*. Les spécifications pour l'API *EnableWindow* données dans le Software Development Kit (SDK) deWindows apparaissent en effet comme suit :

```
BOOL WINAPI EnableWindow(
  _In_ HWND hWnd,
  _In_ BOOL bEnable
);
```

Si l'on traduit cela en assembleur, vous avez besoin de lire les paramètres de droite à gauche pour construire correctement l'empilage des paramètres. Cela peut être un peu plus facile si vous utilisez l'instruction INVOKE au lieu du CALL, puisqu'il vous suffit alors de placer les paramètres dans l'ordre dans lequel ils apparaissent dans le SDK :

Avec CALL	**Avec INVOKE**

`PUSH 1`
`PUSH [hButton]`
`Call EnableWindow`
	`INVOKE EnableWindow, [hButton], 1`

Voilà tout ce que vous devez savoir sur la passation de paramètres sur la pile pour le moment, mais la compréhension de la pile est abordée de façon plus détaillée dans la partie 2 qui suit.

I2 Partie 2

Dans la présente **partie 2**, vous trouverez des informations qui ne sont pas une lecture essentielle pour les programmeurs en assembleur, mais peuvent être d'intérêt.

I2.1 La pile et la notion d'espace d'adressage virtuel

La valeur dans ESP est une *adresse virtuelle*. Si elle est, par exemple, d'une valeur de 64FE3Ch au démarrage, nous ne parlons pas ici d'une adresse dans la mémoire physique réelle. Pour obtenir l'adresse dans la mémoire physique réelle, le système a besoin de convertir (ou « cartographier ») le 64FE3Ch selon ses propres enregistrements internes. Par exemple, cette adresse pourrait être en réalité 2FE3Ch dans la mémoire physique réelle. Une adresse virtuelle est donc juste une représentation pratique d'une position en mémoire. On dit souvent que chaque application fonctionne dans son propre espace d'adressage virtuel. En théorie, toute la gamme d'adresses 32 bits (zéro à 4 Go) est disponible pour chaque application. Dans la pratique, ce n'est pas le cas, mais il est vrai que chaque application en cours d'exécution sur le système peut utiliser la même plage d'adresses virtuelles. Il n'y a pas de conflit entre elles parce que le système sait quelle application adresse la mémoire à tout moment et peut donc pointer l'application à la bonne place dans la mémoire physique. Donc, à un moment donné, il est probable qu'il existe plusieurs applications avec la

même valeur de ESP. Mais, en réalité, cette valeur pointera effectivement vers des parties différentes de la mémoire physique.

I2.2 La pile au démarrage : contenu

Dans Windows le thread principal se voit allouer sa propre zone de pile par le système lors de son chargement. Le système lui-même utilise ce thread et la zone de pile pour ses propres besoins avant d'appeler l'adresse du point d'entrée du programme. Vous pouvez le voir dans le débogueur. Démarrez votre programme jusqu'à l'adresse de départ et scrutez la valeur de ESP. Maintenant, ouvrez une fenêtre d'inspection de la valeur de ce registre. Vous pourriez vous attendre à ce que cette valeur se situe en bas de la zone de mémoire, mais tel n'est pas le cas. Si vous faites défiler l'inspecteur au fond de la zone de mémoire (faites le défiler jusqu'à l'adresse la plus élevée), vous voyez qu'il y a déjà eu beaucoup d'activité dans la pile où le système s'est préparé pour son appel de l'adresse du point d'entrée du programme. Il est intéressant de constater (dans Windows 98 en tout cas) que la dernière valeur sur la pile avant que l'application ait été appelée est une adresse de retour dans Kernel32.dll. Ceci indique qu'une fonction dans Kernel32.dll a appelé l'application. En raison de cette adresse de retour, il est possible d'utiliser un simple RET pour mettre fin à un processus, plutôt que d'appeler l'API *ExitProcess*. Bien sûr, cela ne fonctionne que si la pile est en équilibre de telle sorte que l'exécution du code se poursuive dans la fonction de l'appelant dans Kernel32.dll.
Un peu plus loin en bas de la pile, nous pouvons voir le nom du fichier de l'application, et beaucoup plus bas, que l'adresse du propre gestionnaire d'exception du système pour le thread principal de l'application a été mise sur la pile. Toutes ces choses montrent que la propre zone pile de l'application (et le propre thread) est utilisée par le système pour préparer l'appel de l'application.

I2.3 La pile au démarrage : dimension

Dans Windows, lorsque de la mémoire est réservée pour les besoins d'une application, une plage d'adresses virtuelles *est allouée* par le système. Cette allocation réserve ces adresses à l'usage exclusif de l'application. Si cette dernière requiert plus de mémoire, les mêmes adresses ne peuvent pas être réutilisés. Aucune mémoire physique n'est effectivement utilisée jusqu'à ce que la mémoire soit *réservée*. Ce n'est qu'à ce stade que les adresses virtuelles qui ont été allouées sont mappées sur la ou les zones de mémoire physique dont le système dispose.
Il est évident que pour que cet arrangement fonctionne, le système a besoin de connaître la taille maximale de mémoire contiguë qui peut être engagée. Celle-ci constitue alors la plage d'adresses allouée.
Lors de la réservation de mémoire pour la pile, la même démarche est mise en œuvre. Au moment du démarrage d'une application le système a besoin de connaître la quantité de mémoire à allouer à la pile, et l'ampleur de la réservation en première instance. Ces deux paramètres sont contenus dans l'exécutable à +48h et +4Ch respectivement dans l'en-tête optionnel (pour comprendre exactement où cela se trouve dans le fichier exécutable, vous devez connaître le format de fichier PE). Comme on le voit ci-dessous, ces paramètres s'appliquent non seulement au thread principal de l'application, mais également à de nouveaux threads générés par l'application.
La plupart des éditeurs de liens utilisent respectivement 1MB et 4K par défaut (la taille normale de page) pour ces valeurs. Avec GoLink vous pouvez modifier les valeurs par défaut en utilisant respectivement les commutateurs **/stacksize** et **/stackinit** (voir le manuel de GoLink pour connaître leur utilisation).

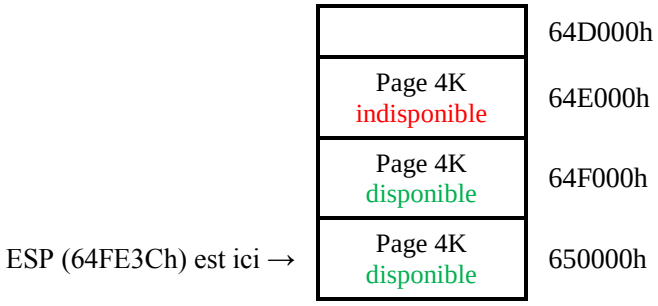
I2.4 Agrandissement de la pile au moment du lancement

Le système détecte si une application tente de lire ou d'écrire en dehors de la zone de pile réservée en ayant recours à la gestion d'exceptions. Si la tentative se situe dans la zone de pile autorisée, la nouvelle mémoire sera engagée selon les besoins (en W9x). Même si une tentative est faite pour agrandir la pile au-delà de la zone allouée, sous NT (mais pas W9x) le système va essayer d'allouer plus de mémoire, mais ce ne sera pas possible si les adresses virtuelles alors nécessaires ont été alloués à d'autres des zones de mémoire.

I2.5 Zone de pile autorisée utilisable

La pile n'est pas considérée comme appropriée à la mémorisation de grandes quantités de données et cette limitation est confortée par Windows au travers de son mécanisme d'exception. Dans W9x la zone de pile utilisable autorisée est entre la valeur de ESP courante et la frontière de page qui suit, plus la taille de la page.

Par exemple, si ESP est 64FE3Ch, que la limite de page suivante est 64F000h et qu'une page supplémentaire (qui est généralement fixé à 4K par le système) s'impose, cela vous emmène à 64E000h :



Donc, si ESP est à 64FE3Ch, vous constaterez que l'instruction

```
MOV D[ESP-1E40h],0
```

provoque une exception, parce l'adresse résultante pointée sur la pile correspond à 64DFFCh et que cette valeur se situe dans une région indisponible car non réservée par le système.

Et vous ne pouvez pas non plus contourner cela en déplaçant ESP. Dans W9x le système vous permet de déplacer ESP seulement jusqu'à la limite de page suivante + la taille d'une page moins quatre octets. Par exemple, si ESP est 64FE3C une simple instruction sera autorisée à déplacer ESP de 1E38h (soit 7836 octets en décimal). Cela signifie que l'instruction

```
SUB ESP,1E38h
```

ramène ESP à 64E004h et qu'elle est autorisée. En revanche, l'instruction

```
SUB ESP,1E3Ch
```

va provoquer une exception. La différence de 4 octets dans la position qui déclenche l'exception suggère qu'il existe deux mécanismes de protection différents au travail ici.

De ce qui précède, il pourrait sembler que la taille des données à mettre sur la pile est limitée à 4K, mais cela est faux. Il existe deux façons d'éviter que ces exceptions ne se produisent et permettent ainsi d'utiliser la pile pour des zones de données plus importantes.

La première façon consiste à déplacer ESP *selon un processus incrémental*. Cela permet de garantir que le système engage la mémoire progressivement comme prévu. Le code suivant crée en toute sécurité une zone de 40K octets sur la pile :

```
MOV ECX,10
L0:
SUB ESP,1000h
MOV D[ESP],0
LOOP L0
```

Ici, le système est invité à réserver dix blocs 4K de mémoire de pile. Au terme de cette opération, ESP pointe ensuite le sommet de cette zone de pile. Ce processus n'est pas particulièrement rapide puisque le système doit réserver de la mémoire dix fois de suite.

Une méthode plus rapide consiste à charger le système de réserver une zone d'emblée plus grande que la zone de mémoire habituelle pour la pile lorsque l'application est chargée. Avec GoLink vous pouvez le faire en utilisant le commutateur /stackinit.

Par exemple :

```
/stackinit 0A000
```

veillera à ce que 40K de mémoire soient réservés sur la pile au démarrage. Vous pourrez alors déplacer ESP en toute sécurité en utilisant l'instruction :

```
SUB ESP,0A000h
```

qui vous octroie 40K de mémoire sur la pile entièrement à votre disposition.

I2.6 Utilisation de la pile pour mémoriser des flux de données

Lorsque des précautions sont prises, la pile peut être utilisée pour mémoriser des flux de données assez conséquents. Les principes à retenir sont les suivants :

- Toujours restaurer ESP à l'équilibre lorsque vous en avez terminé avec la pile.

- Ne jamais écrire sur l'adresse [ESP], sauf si vous avez soustrait au moins 4 octets de la valeur originale de ce registre qui, rappelons-le, détient l'adresse de retour de la procédure. Ne jamais écrire sur l'adresse [ESP+*n*], sauf si un nombre suffisant d'octets a été soustrait de ESP pour éviter d'écraser d'autres données importantes.
- Si vous ne déplacez pas ESP vers le haut de la zone de données, alors vous devez écrire les données *dans l'ordre inverse*, c'est-à-dire selon des adresses consécutivement décroissantes. Cela peut se faire de diverses manières, la plus classique consistant à mettre à 1 le flag de direction par le biais de l'instruction STD, puis à utiliser l'instruction MOVS, par exemple :

```
MOV ECX,8000
MOV EDI,ESP
SUB EDI,4
STD             ; met à 1 le flag de direction (décrémentation auto de ESI et EDI)
REP MOVSD       ; copie ECX dwords de [ESI] à [EDI]
CLD             ; RAZ flag de direction
```

  Cette portion de code copie 8000 dwords dans la pile. Notez que `SUB EDI,4` évite l'écriture sur [ESP] qui détient l'adresse de retour de la procédure. Il n'y a pas de problème d'agrandissement de la mémoire dans la mesure où l'écriture est décrémentale, de sorte que le système crée correctement de nouvelles zones de mémoire 4K lorsqu'il en a besoin.
- Si vous ne positionnez pas ESP au sommet de la zone de données, vous devez prendre les précautions visées dans le paragraphe précédent. Après avoir fait cela, vous pouvez écrire dans la pile *en direction de l'avant*.

I2.7 La pile dans les applications multi-threads

Chaque thread de votre application a ses propres registres et sa propre pile. Autrement dit, lorsque le système donne du temps processeur à un thread, il bascule sur le *contexte de registre* (*register context*) qui lui est spécifique. Celui-ci contient toutes les valeurs de registres correspondant à l'achèvement du dernier temps-processeur exécuté sur le thread. Étant donné que les registres comprennent ESP, la valeur de ce dernier sera également correctement commutée de telle sorte que la zone appropriée de mémoire physique sera utilisée par le thread comme sa pile. Il en résulte qu'un thread peut compter sur le fait qu'il peut utiliser sa pile comme une zone discrète de la mémoire qui ne sera pas perturbée par d'autres threads. Vous pouvez le constater dans le débogueur. Vous pouvez également observer que ESP change toujours considérablement lors de l'exécution des changements d'un thread à l'autre.

Quand un thread démarre, une zone de pile lui est allouée. A titre d'exemple, il a été trouvé sous W98 que la pile du thread principal d'une application se situait à 64FE3Ch et que, lorsqu'un nouveau thread a été lancé, sa pile se situait à 75FF9Ch. Dans un autre test, à l'occasion de la création de six nouveaux threads, il a pu être constaté que leurs piles ont commencé respectivement à 19DEF9Ch, 1AFFF9Ch, 1C1FF9Ch, 1D3FF9Ch, 1E5FF9Ch et 1F7FF9Ch. Vous pouvez donc en déduire que le système sépare l'adresse virtuelle de chaque zone de pile par 128KB plus de la zone de 1Mo par défaut. Ceci est probablement un aménagement pour laisser de la place à l'usage propre du système de la pile et aussi une certaine marge de manœuvre supplémentaire. La modification d'allocation de la taille de la pile à 200000h (2MB) en utilisant le commutateur /stacksize puis la création de six nouveaux threads se sont traduits par des zones de pile séparées par 128KB en plus des 2Mo.

I2.8 La trame de pile et les données locales

Une trame de pile est une zone spécifique de la pile qui héberge l'adresse de retour d'une fonction ainsi que les données utilisées par cette fonction sans risque d'écrasement parce que le contenu de ESP a été décrémentée d'une valeur correspondant justement à la taille de ces données. Ces dernières sont conservées dans une trame de pile et sont appelées *données locales*. Elles sont prévues uniquement pour une utilisation dans la trame de pile concernée et n'ont pas vocation à être traitées par le programme en général. Considérons cet exemple simple :

```
PROCEDURE1:
SUB ESP,20h       ; constitution d'un espace sur la pile pour les données locales
;                 ; utilisation de la zone de données locales
CALL PROCEDURE2
;                 ; retour de la PROCEDURE2
```

```
;                ; on continue à utiliser la zone de données locales
ADD ESP,20h      ; rétablissement de ESP à l'équilibre
RET
```

et

```
PROCEDURE2:
PUSH EAX,EBX,ECX
;                ; réalisation de divers calculs
POP ECX,EBX,EAX
RET
```

Ici, la trame de pile est créée par l'instruction `SUB ESP,20h` qui a pour effet de réduire la valeur de ESP de 32 octets créant ainsi un espace pour 8 dwords sur la pile. Maintenant, du fait que ESP a été déplacé, tout ce qui arrive dans PROCEDURE2 ne peut remplacer ces 8 dwords. Vérifions ce point visuellement en formant l'hypothèse que ESP prend la valeur 64FE38h au début de PROCEDURE1 :

	64FE08h	Contient la valeur en ECX insérée par PROCEDURE2
	64FE0Ch	Contient la valeur en EBX insérée par PROCEDURE2
	64FE10h	Contient la valeur en EAX insérée par PROCEDURE2
ESP pointe le début de PROCEDURE2→	64FE14h	Contient l'adresse de retour de PROCEDURE2
Trame de pile de PROCEDURE1	64FE18h à 64FE34h	8 dwords pour les données locales
	64FE38h	Contient l'adresse de retour de PROCEDURE1

I2.9 Adressage des données locales

Note : l'adressage des données locales est automatisé en utilisant les directives FRAME..END avec GoAsm et PROC..ENDP avec MASM.

Puisque ESP pointe vers le haut de la zone de données locale, vous pouvez traiter les données en utilisant ESP. Ainsi, dans l'exemple ci-dessus le premier dword des données locales serait disponible à [ESP] immédiatement après le `SUB ESP, 20h`. Mais l'utilisation de ESP pour pointer les données locales sur la pile peut se révéler périlleuse parce ESP se déplacera à chaque CALL, PUSH ou POP au sein de la procédure. Pour cette raison, il est de pratique courante que l'on y substitue le registre EBP qui est exempt de tout désagrément de ce type. La valeur initiale de EBP est habituellement fixée au début de la trame de la pile vers le bas des données locales et ne sera pas modifiée avant que l'exécution ne quitte la trame de pile. De cette façon, vous pouvez être certain que les données locales peuvent toujours être traitées en utilisant un décalage par rapport à EBP.

De cette manière, le code pour une trame de pile classique ressemble à ceci :

```
TypicalStackFrame:
PUSH EBP         ; sauvegarde la valeur de EBP qui va être altérée      ┐
MOV EBP,ESP      ; EBP = valeur courante du pointeur de pile            │ "prologue"
SUB ESP,0Ch      ; création d'un espace pour les données locales        ┘
;                ; POINT "X"
;
;                ; code de la procédure (ne modifie jamais EBP)
;
MOV ESP,EBP      ; restaure le pointeur de pile à sa valeur d'origine   ┐
POP EBP          ; restaure la valeur de EBP                            │ "épilogue"
RET              ; retourne à l'appelant en ajustant le pointeur de pile ┘
```

Dans cet exemple, nous avons déplacé le pointeur de pile de 12 octets. Au point "X" la pile par référence à EBP ressemble à ceci :

	EBP–10h	Le prochain PUSH ira ici
ESP ici au point "X"→	EBP–0Ch	Espace pour les données locales
	EBP–8h	Espace pour les données locales
	EBP–4h	Espace pour les données locales
	EBP	Sauvegarde de EBP
	EBP+4h	Adresse de retour de TypicalStackFrame

Maintenant, au travers de la trame de pile, quoi qu'il puisse arriver à ESP, les données locales seront toujours accessibles à [EBP-4h], [EBP-8h] et [EBP-0Ch].

Notez comment ESP est rétabli à l'équilibre automatiquement par l'utilisation de `MOV ESP,EBP` juste avant le retour au programme appelant.

En réalité, cette fonction dévolue à EBP peut surprendre car n'importe quel registre peut en faire autant. Mais EBP est traditionnellement utilisé à cet effet et votre code n'en sera que plus compréhensible vis-à-vis des autres si telle est votre préoccupation.

I2.10 Accès aux paramètres à partir de la pile

Nous avons vu déjà comment passer des paramètres sur la pile à d'autres procédures. Maintenant, nous allons voir comment utiliser les paramètres passés aux procédures de l'intérieur desdites procédures. Fondamentalement, ces paramètres sont plus bas dans la pile de sorte qu'ils ne risquent pas d'être écrasés dans des circonstances de fonctionnement normales. Pour cette raison, il n'est absolument pas nécessaire de les récupérer ni de les sauvegarder. Lors de l'entrée dans une procédure, ESP pointe l'adresse de retour de la procédure (insérée par CALL). Les paramètres seront donc successivement à [ESP + 4h], [ESP + 8h], [ESP + 0Ch] et ainsi de suite, selon leur nombre. Mais il peut être difficile de repérer les paramètres avec ESP de manière fiable car ce registre ne va pas arrêter de changer au gré des PUSH, POP ou CALL qui peuvent être rencontrés dans le code de la procédure. C'est là que s'impose l'utilisation de EBP pour pointer les paramètres indépendamment des aléas du code.

Si vous avez le code de prologue :

```
PUSH EBP        ; sauvegarde la valeur de EBP qui va être altérée      ┐
MOV EBP,ESP     ; EBP = valeur courante du pointeur de pile            │ "prologue"
SUB ESP,0Ch     ; création d'un espace pour les données locales        ┘
```

Lorsque ESP est copié dans EBP, il est de 4 octets inférieur en valeur par rapport au début de l'appel (cela est dû au premier `PUSH EBP`). Par conséquent, les paramètres peuvent maintenant être consultés aux adresses [EBP+8h], [EBP+0Ch], [EBP+10h] et ainsi de suite, selon le nombre de paramètres.

I2.11 Utilisation de la pile dans les procédures callback de Windows

Les deux techniques qui viennent d'être exposées (constitution d'un espace pour les données locales et paramètres d'adressage) sont nécessaires dans les procédures callback de Windows. La procédure callback la plus souvent rencontrée dans les programmes Windows est la procédure de fenêtre. C'est dans son cadre que Windows envoie des "messages" et qu'il en attend la réponse correcte. Windows *appelle* la procédure de fenêtre à l'aide du propre thread du programme. Cela se produit généralement lorsque le programme se trouve dans la boucle de message, soit en attente d'un retour de l'API *GetMessage*, soit pendant l'exécution de l'API *DispatchMessage*.

Heureusement, dans GoAsm, vous pouvez utiliser FRAME..ENDF pour récupérer les paramètres envoyés par Windows et les adresser selon leur nom. FRAME..ENDF vous permet tout aussi facilement de constituer des zones de données locales adressables nominativement. Et vous pouvez également préserver les registres ainsi que restaurer la pile à l'équilibre automatiquement. Le présent manuel vous renseignera complètement sur ces différentes fonctionnalités ou vous pouvez également vous reporter utilement à la **partie n° 1** de cette annexe.

Annexe J
Comprendre... la mémorisation inversée

Cette question revêt une certaine importance si vous projetez d'examiner des données, de la mémoire ou, plus généralement, votre ordinateur à l'aide d'un débogueur.

Fondamentalement, les processeurs Intel ne conservent pas les données en mémoire dans l'ordre auquel vous vous attendez. Au lieu de cela, les données sont *stockées en inverse* sur la base d'*un octet pour un octet*.

Voyons cela plus en détail : un seul octet est stocké évidemment en mémoire à l'adresse attendue. On s'en serait douté... Mais un mot (deux octets) sera stocké en mémoire avec l'octet le moins significatif à l'adresse du mot en mémoire et l'octet le plus significatif à l'adresse immédiatement supérieure.

Pour une séquence d'octets, on parle parfois de format "little-endian" parce que l'octet le moins significatif est en premier et le plus significatif en second. L'inverse est appelé "big-endian" et est utilisé par d'autres processeurs.

Exemple du mot (Word)

Supposons donc que vous ayez un mot de valeur 248Ch qui doit être stocké à 400000h. Il sera stocké de la façon suivante :

8Ch à l'adresse 400000h
24h à l'adresse 400001h.

Lorsque la mémoire à 400000h est lue dans un registre 16 bits (soit un mot) par une instruction de type word, les octets seront lu dans le sens inverse afin que la valeur 248Ch soit chargée dans le registre.

Exemple du double-mot (Dword)

Un dword (quatre octets) sera stocké dans la mémoire avec l'octet le moins significatif à l'adresse du dword en mémoire, puis les deux prochains octets et enfin avec l'octet le plus significatif à la quatrième adresse vers le haut.

Par exemple, supposons que vous ayez un dword de valeur 12345678h devant être stocké à l'adresse 400000h. Il sera stocké de la façon suivante :

78h à l'adresse 400000h
56h à l'adresse 400001h
34h à l'adresse 400002h
12h à l'adresse 400003h

Lorsque la mémoire à 400000h est lue dans un registre 32 bits par une instruction dword, les octets sont lus dans le sens inverse afin que la valeur 12345678H soit chargée dans le registre.

On notera que les quartets de l'octet ne sont inversés à aucun moment.

A présent, vous vous interrogez peut-être sur les raisons de cette pratique pour le moins exotique. Ne serait-il pas beaucoup plus facile de stocker des données dans le même ordre que nous les visualisons ? Eh bien, ce serait bien pour nous, mais pas pour l'ordinateur. Et ce n'est pas quelque chose que vous pouvez reprocher au langage assembleur. Toutes les données quelle que soit la langue utilisée sont stockées de cette manière sur les processeurs Intel. Normalement, c'est transparent pour le programmeur et vous ne devez pas vous inquiéter pas à ce sujet.

Annexe K
Quelques conseils et astuces de programmation

K1 Quelques conseils de programmation

K1.1 Usage traditionnel des registres

Certaines instructions utilisent des registres particuliers pour effectuer certaines tâches ; d'autres s'avèrent plus rapides si certains registres sont utilisés ; dans les premiers processeurs, enfin, seuls certains registres étaient multi-usages. Ces trois constats auxquels il convient d'associer l'utilisation traditionnelle des registres établie par les programmeurs en assembleur au fil des ans, ont établi une sorte de corpus de règles informel sur la façon dont les registres doivent être utilisés. Si vous vous conformez à ces quelques règles, la lisibilité et la portabilité de votre code s'en ressentiront sensiblement.

- Utilisez EAX pour transmettre des données à une procédure et pour retourner des données à l'issue d'une procédure à destination du code appelant. Les API Windows elles-mêmes utilisent EAX pour retourner une valeur à l'appelant. Les formats AL, AX et EAX peuvent également être privilégiés dans la mesure du possible pour recevoir des données de la mémoire et en envoyer, car ils travaillent un peu plus rapidement que d'autres registres. Par exemple utiliser `MOV AL,[ESI]` de préférence à `MOV DL,[ESI]`. De même, si vous avez besoin d'utiliser ADD, AND, ADC, CMP, MOV, OR, SUB, TEST, XCHG, XOR avec une valeur immédiate (c'est-à-dire un nombre comme dans `MOV AL,23h`) utiliser AL, AX ou EAX si vous pouvez car l'instruction utilise moins d'opcodes que si vous utilisiez un autre registre.
- Utilisez le registre EDX comme sauvegarde de EAX si ce dernier est déjà en cours d'utilisation.
- Utilisez ECX comme compteur. Ce registre est en effet utilisé directement en tant que tel par certaines instructions. Par exemple, JECXZ est une instruction spéciale qui effectue un saut si ECX est nul. De même, les instructions LOOP, SCAS et MOVS utilisent toutes ECX comme compteur.
- Utilisez EBX pour héberger des données générales ou pour adresser la mémoire comme, par exemple `MOV EAX,[EBX]` ou `MOV [EBX],EDX`.
- Utilisez ESI dès lors que vous devez lire de la mémoire. Par exemple : `MOV EAX,[ESI]` et EDI si vous devez écrire dans la mémoire. Par exemple, `MOV [EDI],EAX`. Ceci est cohérent avec les pratiques d'adressage des instructions LODSD, STOSD et MOVSD mais ne constitue en aucun cas une obligation de codage.
- Utilisez l'un des registres de base ou d'index dans les instructions utilisant un adressage complexe de la mémoire comme, par exemple, `MOV EAX,[`*memptr*`+ESI*4+ECX]`

- N'utilisez jamais ESP autrement qu'en tant que pointeur de pile, sauf si vous avez une routine qui n'a pas du tout d'activité de pile. Dans ce cas, vous pouvez sauvegarder la valeur de ESP en mémoire et la restaurer avant de retourner à l'appelant de la routine.
- Traditionnellement, EBP est utilisé pour adresser les données locales sur la pile dans les routines callback. EBP et sa composante BP en 16 bits peuvent être utilisés en tant que registres généraux en programmation Windows, mais vous devez être très prudent si vous utilisez des trames de pile (FRAME dans GoAsm) ou des données locales (LOCAL dans GoAsm). En effet, les paramètres de trame de pile et les données locales sont adressés à l'aide de EBP *associé à un offset positif ou négatif*. Si EBP est modifié, les paramètres et données locales ne pourront plus être adressés de manière simple tant qu'il n'aura pas été rétabli à sa valeur précédente. Notez qu'en code 16 bits, BP était conçu pour agir en relation avec le segment de la pile, sauf s'il était utilisé avec un remplacement de segment (*override*). Mais en Windows 32 bits, EBP peut être utilisé pour traiter une partie de la zone de mémoire "flat" de 4GB.
- Les registres de segment CS, DS et SS sont encore utilisés par Windows, même dans sa version 32 bits, de sorte que vous devez impérativement éviter de les utiliser. De même, vous ne pouvez pas utiliser ES, FS ou GS non plus. En Windows 98 et ses versions ultérieures, toute tentative contrevenant à ces exigences provoque une exception.
- Si vous avez besoin d'utiliser des registres ordinaires pour héberger une information 64 bits, utilisez le couple de registres EDX:EAX dans lequel EDX détient les bits les plus significatifs. Cela concorde avec les instructions de décalage 64 bits SHLD et SHRD, avec CDQ, MUL/IMUL et DIV/IDIV.

K1.2 Utilisation de Windows : registres et pile

Vous pouvez compter sur le fait que toutes les API Windows sauvegardent et restaurent les registres EBP, EBX, EDI et ESI. Par conséquent utilisez ces registres pour héberger les handles et pointeurs qui doivent être utilisés plus d'une fois à travers des séquences d'appels d'API.
Tout comme Windows préserve la valeur de ces registres dans les API, vous devez également veiller à ce qu'ils soient maintenus dans vos procédures callback. Je recommande que, dans chacune de vos procédures callback et de fenêtre, vous enregistriez ces registres au début de la procédure et que vous les restauriez à la fin.
Vous pouvez également compter sur les API Windows pour la restauration de la pile sous réserve d'avoir fourni en entrée le nombre exact de paramètres requis. Cependant, je suis tombé sur une exception à cette règle (wsprintf). Mais, ceci est documenté dans le SDK Windows.

K1.3 Mise en pile des registres par ordre alphabétique

Si vous avez besoin de préserver une série de registres dans une procédure particulière, il est conseillé de les pousser en pile (PUSH) dans l'ordre alphabétique. De cette façon, vous pouvez facilement vérifier que les POP sont dans l'ordre alphabétique inverse, préservant ainsi la restauration des registres avec leur valeur d'origine. Par exemple :

```
PUSH EBP,EBX,EDI,ESI
  .
  .             ;votre code vient ici
  .
POP ESI,EDI,EBX,EBP
```

Si vous préférez, vous pouvez utiliser en GoAsm l'instruction USES qui préservera et restaurera automatiquement les registres pour vous.

K1.4 Protection du code

Il est important de concevoir des programmes aussi robustes que possible en réduisant au minimum le risque d'un accident de programme ou d'une boucle infinie. Voici quelques façons d'agir en ce sens :

- Avant d'utiliser les préfixes de répétition REP, REPZ ou REPZ qui utilisent ECX comme registre de compte à rebours, et avant l'utilisation de LOOP, LOOPZ ou LOOPZ, toujours s'assurer que ECX n'est pas à zéro. Si l'instruction est effectuée avec ECX = 0, elle fera en effet 4294967296 opérations ! Si vous voulez être doublement prudent, vous pouvez également tester le bit de plus fort poids de ECX et vous assurer qu'il n'est pas à 1, moyen simple de prévenir une valeur anormalement élevée. On y parvient simplement au moyen de l'instruction `OR ECX,ECX` suivie de `JS> L2`.

- Eviter une division par zéro en vous assurant préalablement que le diviseur n'est pas nul (registre ECX dans `DIV ECX` par exemple) à défaut de quoi une exception est générée avec tous les troubles qui peuvent s'ensuivre pour votre programme.
- Chaque fois que vous pratiquez une division avec un diviseur 32 bits (DIV ECX par exemple) et que vous n'utilisez que EAX pour le dividende, forcez EDX à zéro. Dans le cas d'une division avec un diviseur 16 bits (DIV CX par exemple) et que vous n'utilisez que AX pour le dividende, forcez DX à zéro.
- MUL peut provoquer une exception de débordement si les valeurs qui lui sont données sont trop élevés. Si ces valeurs sont dans les registres, vérifier qu'elles ne sont pas trop élevées avant d'appeler l'instruction MUL.
- Avant d'essayer de lire ou écrire dans un emplacement mémoire adressé par registre, vérifiez que le contenu de ce dernier n'est pas nul. Comme précaution supplémentaire, vous pouvez vous aider de l'API Windows *IsBadReadPtr* pour réaliser ce diagnostic.
- Enfin, protégez-vous du code en utilisant la gestion des exceptions. Voir mon article très détaillé sur ce sujet disponible à partir de mon **site**.

K1.5 Développement incrémental du code

Lors de l'écriture d'un nouveau code et dès que vous avez terminé une partie restreinte de celui-ci, le tester en temps réel dans toutes les conditions possibles et aussi, si nécessaire, exécutez-le en pas-à-pas au travers du débogueur. De cette façon, si votre programme fait quelque chose d'inattendu vous pouvez être raisonnablement certain que la faute en est imputable au code que vous venez d'écrire. Si vous abandonnez le test jusqu'à ce que vous ayez écrit un autre code, la faute sera plus difficile à trouver.

K1.6 Fonctions réutilisables

Prenez en compte le fait que certaines fonctions que vous écrivez pour votre programme peuvent être utiles dans d'autres parties dudit programme ou dans les futurs programmes que vous écrirez. Vous allez alors vous retrouver avec un certain nombre de modules réutilisables qui assurent des tâches spécifiques, comme, par exemple, le chargement d'une chaîne à destination de la mémoire, l'écriture en mémoire d'une valeur décimale, la division par dix, le chargement d'une police de dialogue, ou le chargement d'un nouveau titre de fenêtre. Donnez aux fonctions noms expressifs tels que LOAD_STRINGEDI ou DECIMAL_WRITE ou DIVIDE_BY_TEN et ainsi de suite. Vous pouvez également faire en sorte que ces modules déclarent eux-mêmes les données qu'ils utilisent pour les rendre encore plus indépendants de leurs appelants. En programmation orientée objet on qualifierait ces modules d'"objets". Donc, vous êtes maintenant dans la POO ! Pour faciliter cette approche modulaire, contentez-vous dans la mesure du possible d'une l'utilisation traditionnelle des registres, et veillez à sauvegarder et restaurer tous les registres utilisés par le module.

K1.7 Divisez vos fonctions si elle sont trop volumineuses

L'approche modulaire de la programmation consiste également à faciliter la lisibilité de votre code en faisant notamment en sorte que le nom des petites fonctions appelées soit suffisamment explicite et vous aide à comprendre ce que font les grandes sections de code. Si vos fonctions deviennent trop grandes pour être facilement compréhensibles, divisez-les en plus petites fonctions à appeler en donnant à chaque fonction un nom décrivant ce qu'elle fait. Je recommanderais de faire de même si la fonction n'est appelée qu'une seule fois. En tout état de cause, on retiendra qu'un appel à fonction ne génère qu'une perte de vitesse insignifiante. En règle générale, si l'un de vos sauts au sein d'une fonction (autre que les sauts pour quitter la fonction) ne peut pas être codé en utilisant la forme courte (hors de la plage +127 / –128 octets), c'est un signe que votre fonction gagnerait à être divisée en plus petites sections.

K1.8 Valeurs et flags en retour de fonctions

Traditionnellement EAX est utilisé pour restituer une valeur en retour de l'exécution d'une fonction, ce que font notamment les API Windows. En assembleur il est également habituel, si le travail d'une fonction est de trouver une adresse mémoire, que celle-ci soit retournée dans ESI, EDI ou EBX.

Il arrive aussi que les fonctions aient besoin de positionner un flag particulier lors de leur retour pour indiquer un résultat de traitement. Les usages en la matière sont les suivants :

Retour Cf = 1 (Carry) pour montrer qu'une erreur est survenue.
Retour Cf = 0 (not Carry) pour qualifier un succès.
Retour Zf = 0 ou 1 (Zéro ou non Zéro) pour afficher le résultat d'une action.

Les flags peuvent également être utilisés pour inviter l'appelant à ne pas prendre d'autres mesures. Voici un exemple de la fonction GENERAL WNDPROC dans le programme de démo HelloWorld2 :

```
CALL [EDX+ECX*8+4]        ; appel de la procédure appropriée pour le message
JNC >L4                   ; Cf = 0 = ne pas appeler DefWindowProc
PUSH [ESP+18h],[ESP+18h],[ESP+18h],[ESP+18h]
CALL DefWindowProcA
L4:
```

K1.9 Qualité des descriptions et commentaires

Rappelez-vous qu'il peut vous arriver de vous référer à un code des années après que vous l'ayez écrit. C'est pourquoi, au tout début du script source, il vous est recommandé de décrire les fonctionnalités du programme, d'expliquer comment il fonctionne, comment il doit être assemblé et soumis à l'éditeur de liens. Décrivez également chaque fonction ainsi que son fonctionnement, si ce n'est pas évident. Ajouter des commentaires à une ligne dans le script source s'il apparaît que son action ne va pas de soi. Ajouter des commentaires et des descriptions aux déclarations de données et aux modèles de structure.

K1.10 Ordre des déclarations dans le script source

Bien que GoAsm soit un assembleur à une seul passe, il n'impose pas que les déclarations de données soient dans un endroit particulier du script source bien qu'il soit habituel de déclarer des données avant le code qui en fait usage. En fait, GoAsm n'impose cette précédence qu'aux définitions et modèles de structure qui doivent donc être déclarés avant toute utilisation à défaut de quoi GoAsm ignorera tout de leur qualité.
Les données sont idéalement alignées sur une modularité correspondant à la taille des données. Ceci peut être réalisé au moyen de la directive ALIGN. Cependant, en cas de données de tailles diverses, un bon alignement sera obtenu automatiquement si l'ordre suivant est respecté dans les déclarations de données, à partir de l'ouverture de la section de données :
QWords, dwords, mots, octets, chaînes. Les Twords sont mieux alignés sur une modularité de 8 octets, mais un nombre impair de Twords déclarés au début de la section de données bouleverse l'alignement du reste des données, car ils sont de 10 octets chacun.
Une déclaration de données non-initialisées n'affectera l'alignement en aucune manière puisqu'il ne s'agit que d'une réservation d'espace et qu'elle ne concerne que la section .bss.

K1.11 Meilleure direction des sauts conditionnels

Sur le Pentium III et versions ultérieures, le processeur décide de mettre en cache la destination d'un saut conditionnel dans votre code en fonction de sa direction. La règle utilisée par le processeur est que les destinations de sauts vers l'avant ne sont pas mises en cache, tandis que les destinations de sauts arrière le sont. Il en résulte que votre code sera plus rapide si vous observez les règles suivantes :

- Lors de la vérification des erreurs, par exemple dans le cas du test de la valeur dans EAX après un appel d'API, toujours sauter vers l'avant pour sortir en cas de défaillance.
- Dans les boucles, la boucle doit toujours faire *un saut arrière* vers le début de la boucle.
- Dans les boucles, lors du test permettant de mettre fin à la boucle, toujours sauter *vers l'avant* pour sortir de la boucle.

K1.12 Minuscules ou majuscules ?

Cette question relève, en grande partie, de choix personnels. Cependant, ma propre expérience en matière de programmation Windows m'a conduit à adopter ce qui suit pour rendre le script source aussi lisible que possible :

- Les mnémoniques et noms de registres sont toujours soit en majuscules, soit en minuscules mais ce choix doit demeurer cohérent tout au long du fichier.
- Les labels de code sont toujours en majuscules seulement. Cela distingue ces labels lorsqu'ils sont appelés à partir d'une API Windows qui combinera majuscules et minuscules. Par exemple, si vous observez cette règle, vous saurez que CALL COMPARESTRING est un appel à l'une de vos propres procédures et que CALL CompareStringA est un appel d'API Windows.
- Les labels de données et les noms de pointeur décrits dans le SDK Windows doivent être utilisés en respectant la casse spécifiée, tandis que les autres labels doivent être en majuscules seulement. Encore une fois cela accroit la lisibilité du script source parce que les labels de données de Windows et les noms de pointeur sont bien connus de tous les programmeurs Windows. Vous savez par exemple

que hwnd, hAccel ou szWindowName ont une signification particulière pour Windows et sont décrits dans le SDK, alors que MBTITLE et MBMESSAGE sont spécifiques à votre programme.

K1.13 Sauvegardez votre travail régulièrement

Ceci est prudent non seulement en cas de défaillance du disque, mais il y a aussi une autre raison. En programmation vous devez prendre parfois la décision de changer radicalement la façon dont tout ou partie de votre programme fonctionne. Vous apportez, de ce fait, des modifications majeures à votre code. Cependant, à la fin de cette démarche, vous pourriez décider de revenir au codage d'origine. Alors conservez une copie de toutes les versions du script jusqu'à ce que vous soyez certain d'être satisfait de votre changement radical !

K2 Quelques tours de main

K2.1 Positionnement des flags

L'état des flags reflète normalement le résultat de l'exécution d'une instruction particulière mais il est souvent nécessaire de les prépositionner manuellement afin de rendre ce résultat pertinent. En dehors de CLC, CMC et STC permettant de forcer le flag de Carry à 0 ou à 1 et de CLD et STD permettant de faire de même avec le flag de direction, vous pouvez utiliser les instructions suivantes qui ne changent pas les registres concernés et qui tiennent en deux opcodes chacune :

```
CMP EAX,EAX       ; met à 1 le flag de zéro (EAX est inchangé)
CMP EAX,EDX       ; met à 0 le flag de zéro si EAX et EDX ont un contenu différent
OR EAX,EAX        ; met à 0 le flag de zéro si EAX est différent de zéro
TEST EAX,EAX      ; même effet que OR EAX,EAX
```

K2.2 Test de zéro

Voici différentes manières de tester une valeur nulle :

```
JECXZ >L1         ; 2 opcodes
OR ECX,ECX        ; 2 opcodes
JZ >L1            ; et 2 opcodes de plus
;
TEST ECX,ECX      ; 2 opcodes
CMP ECX,0         ; 3 opcodes
```

K2.3 Initier une valeur dans un registre ou en mémoire

Voici différentes manières de mettre un registre à une valeur donnée :

```
XOR EAX,EAX       ; réalise EAX = 0 avec 2 opcodes
SUB EAX,EAX       ; réalise EAX = 0 avec 2 opcodes
AND EAX,0         ; réalise EAX = 0 avec 3 opcodes
MOV EAX,0         ; réalise EAX = 0 avec 5 opcodes
;
XOR EAX,EAX
INC EAX           ; réalise EAX = 1 avec un total de 3 opcodes
MOV EAX,1         ; réalise EAX = 1 avec 5 opcodes
;
OR EAX,-1         ; réalise EAX = -1 avec 3 opcodes
XOR EAX,EAX
DEC EAX           ; réalise EAX = -1 avec un total de 3 opcodes
MOV EAX,-1        ; réalise EAX = -1 avec 5 opcodes
;
XOR EAX,EAX
MOV AL,66h        ; réalise EAX = 66h avec 4 opcodes
MOV EAX,66h       ; réalise EAX = 66h avec 5 opcodes
```

et lorsqu'on utilise de la mémoire :

```
XOR EAX,EAX    ; EAX = 0 avec 2 opcodes
MOV [ESI],EAX  ; met à 0 la mémoire située à ESI avec 4 opcodes
```

```
MOV D[ESI],0   ; met directement à 0 la mémoire située à ESI avec 6 opcodes
;
XOR EAX,EAX    ; EAX = 0 avec 2 opcodes
MOV [HELLO],EAX    ; met HELLO à zéro en utilisant 7 opcodes
MOV D[HELLO],0 ; met HELLO à zéro en utilisant 10 opcodes
```

K2.4 Utilisation de INC et DEC en lieu et place de ADD et SUB

```
INC ESI,ESI       ;incrémente ESI deux fois avec 2 opcodes
ADD ESI,2         ;increment ESI deux fois avec 3 opcodes
;
DEC ESI,ESI       ; décrémente ESI deux fois avec 2 opcodes
SUB ESI,2         ; décrémente ESI deux fois avec 3 opcodes
```

Cela étant, Intel recommande aujourd'hui d'éviter désormais l'utilisation des instructions INC *registre* et DEC *registre* et de leur préférer respectivement SUB *registre*, 1 et ADD *registre*, 1 afin d'uniformiser le traitement des flags.
En effet, les instructions INC et DEC ne modifient seulement qu'une partie des bits dans le registre des flags comparativement aux instructions ADD et SUB dont elles ne sont pourtant qu'un cas particulier. Selon Intel,[10] il peut même s'avérer particulièrement problématique dans certains cas d'en poursuivre l'usage.

K2.5 Utilisation des registres 16 bits

Dans la plupart des cas, l'utilisation d'un registre de 16 bits au lieu d'un registre à 8 ou 32 bits va ajouter un opcode supplémentaire à votre code. En effet, l'assembleur doit générer l'octet de modification de taille (66h) avant l'instruction.

K2.6 Multiplications utilisant LEA

Voici quelques exemples de multiplication rapide utilisant l'instruction LEA :

```
LEA EAX,[EAX+EAX*2]   ; multiplie EAX par 3 avec 3 opcodes et 1 cycle d'horloge
LEA EAX,[EAX+EAX*4]   ; multiplie EAX par 5 avec 3 opcodes et 1 cycle d'horloge
LEA EAX,[EAX+EAX*8]   ; multiplie EAX par 9 avec 3 opcodes et 1 cycle d'horloge
;
LEA EAX,[EAX+EAX*4]   ; multiplie EAX par 5
LEA EAX,[EAX*2]       ; le résultat final est la multiplication par 10 (total 6 opcodes)
;
LEA EAX,[EAX+EAX*4]   ; multiplie EAX par 5
SHL EAX,1             ; le résultat final est la multiplication par 10 (total 5 opcodes)
;
LEA EDX,[EAX*2]       ; EDX = 2 * EAX
LEA EAX,[EAX+EAX*8]   ; multiplie EAX par 9
LEA EAX,[EAX*8]       ; maintenant, le résultat dans EAX est EAX * 72
SUB EAX,EDX           ; maintenant, le résultat dans EAX est EAX * 70
```

K2.7 Utilisation d'un registre plus d'une fois

Pour copier dans un registre le contenu d'une adresse mémoire, il est parfaitement admis de lui charger les données pointées par lui-même, par exemple :

```
MOV EAX,[EAX]
```

K2.8 Neutraliser une ligne en la transformant en commentaire

Lors du développement de votre code, il peut être nécessaire de neutraliser provisoirement une ligne d'instruction sans la supprimer de sorte qu'elle puisse être restaurée facilement ultérieurement en cas de besoin. Pour cela, il suffit de placer un simple point virgule en début de ligne, de manière à la transformer en commentaire :

```
MOV EBX,ADDR WORTHYNESS
;MOV EDX,[EBX+14h]      ; ligne neutralisée par sa mise en commentaire
```

[10] Intel® 64 and IA-32 Architectures – Optimization Reference Manual – §3.5.1.1 Use of the INC and DEC Instructions

```
MOV EDX,[EBX+10h]       ; ligne remplaçant la ligne ci-dessus mise en commentaire
                        ; pour effectuer des tests transitoires
```

K2.9 Tests précédant l'utilisation d'instructions MMX, SSE ou SSE2

Avant d'utiliser des mnémoniques MMX, SSE ou SSE2, toujours vérifier que le processeur les accepte en mettant en œuvre l'instruction CPUID. Si tel n'est pas le cas, utiliser un code de remplacement utilisant des registres ordinaires.

K2.10 Assemblage et édition de liens avec les symboles de débogage

Pour faciliter le débogage au cours du développement mettre le commutateur sur la ligne de commande du linker destiné à produire une sortie de débogage. Consulter le manuel de votre éditeur de liens pour voir comment procéder. Vous pouvez utiliser mon linker GoLink pour ce faire et, ensuite, mon débogueur GoBug. GoAsm passe automatiquement tous les symboles à l'éditeur de liens pour l'inclusion dans la sortie de débogage. Puis, lorsque votre programme est terminé, vous pouvez produire une version finale de l'exécutable sans sortie de débogage.

K2.11 Recherche d'erreurs dans votre programme

Consulter le manuel relatif à mon utilitaire GoBug dans le volume 2 de cette édition.

Annexe L
Normalisation des procédures Callback Win32

Cette annexe décrit ma méthode favorite de codage des procédures CALLBACK dans un grand programme en assembleur pour Windows 32. J'y décris tout d'abord ce que sont les procédures callback Win32, puis y expose quelques exemples de code.

Au moment de l'exécution, le système Win32 appelle votre programme selon un schéma immuable et cadré. Les procédures que vous fournissez au système à appeler sont nommées procédures CALLBACK.

Voici des exemples où elles sont utilisées :

1. Pour gérer une fenêtre que vous avez créée. Dans ce cas, le système va envoyer de nombreux messages à la procédure de fenêtre pour la gestion de celle-ci. La procédure de fenêtre est le label de code que vous fournissez lorsque vous vous déclarez votre classe de fenêtre (en appelant l'API *RegisterClass*). Par exemple, le message WM_SIZE est envoyé par le système lorsque la fenêtre est redimensionnée.

2. Pour informer le propriétaire d'une fenêtre-fille d'événements survenus dans cette fenêtre. Par exemple, WM_PARENTNOTIFY (avec un code de notification) est envoyé à la Procédure de Fenêtre du propriétaire d'une fenêtre lorsque la fenêtre-fille est en cours de création ou de destruction, ou si l'utilisateur clique sur un bouton de la souris alors que le curseur est en dehors de la fenêtre-fille.

3. Pour informer le propriétaire d'un contrôle commun d'événements survenant dans le contrôle. Par exemple, si vous créez un bouton dans votre fenêtre, la procédure de fenêtre correspondant à cette fenêtre reçoit le message BN_CLICKED si ledit bouton est cliqué.

4. Les messages envoyés à un dialogue que vous avez créé. Ce sont des messages relatifs à la création du dialogue ainsi que des différents contrôles associés. La procédure de dialogue est informée des événements survenant dans les contrôles.

5. Si vous "sur-classez" ou "sous-classez" un contrôle commun, vous recevez des messages pour que ce contrôle commun se comporte comme une procédure de hameçonnage, tout en retenant que votre procédure de fenêtre conserve la responsabilité de les transmettre au contrôle.

6. Si vous créez des procédures "Hook" (hameçonnage) vous pouvez intercepter les messages sur le point d'être envoyés à d'autres fenêtres. Le système appellera votre procédure d'hameçonnage et ne passera le message qu'au retour de celle-ci.

7. Vous pouvez demander au système de fournir à votre programme les informations à envoyer à une procédure CALLBACK. Les exemples sont EnumWindows (énumérer toutes les fenêtres de niveau supérieur) ou EnumFonts (énumérer toutes les polices de caractères disponibles).

Dans les cas 1 à 5 ci-dessus, juste avant que le système n'appelle la procédure CALLBACK, le système pousse (PUSH) 4 dwords sur la pile (c'est-à-dire 4 "paramètres"). Traditionnellement, les noms donnés à ces paramètres sont :

hWnd = handle de la fenêtre en cours d'appel
uMsg = numéro de message
wParam = paramètre envoyé avec le message
lParam = autre paramètre envoyé avec le message.

Le nombre de paramètres envoyés aux procédures d'hameçonnage et aux procédures callback varie – voir le SDK Windows.

Dans la mesure où votre procédure de fenêtre (ou de dialogue) devra réagir d'une certaine manière en fonction du message envoyé, votre code devra détourner l'exécution vers l'endroit approprié selon le message.

Les programmeurs en C ont l'avantage de pouvoir coder cela simplement, en utilisant "switch" et "case".

Les programmeurs en assembleur utilisent diverses techniques. Peut-être la pire de toutes, lorsqu'il y a beaucoup de messages à traiter, consiste-t-elle à élaborer la chaîne de comparaisons décrite ci-dessous (en format GoAsm) :

```
MOV EAX,[EBP+0Ch]   ; mettre en EAX le numéro du message
CMP EAX,1h          ; voir si c'est WM_CREATE
JNZ >L2             ; non
XOR EAX,EAX         ; garantit que EAX prend la valeur zéro en sortie
JMP >L32            ; c'est fini
L2:
CMP EAX,116h        ; voir si c'est WM_INITMENU
JNZ >L4             ; non
CALL INITIALISE_MENU
JMP >L30             ; code de sortie correct
L4:
CMP EAX,47h          ; voir si c'est WM_WINDOWPOSCHANGED
JNZ >L8
```

et ainsi de suite .....

Pour éviter ces longues chaînes de comparaison, les programmeurs en assembleur ont développé diverses techniques. Vous pouvez en voir de nombreux exemples sur différents sites de code assembleur utilisant des sauts conditionnels, des macros ou une indexation par table. Je ne veux pas comparer ces différentes méthodes, mais simplement mettre en avant ma préférence actuelle, qui je crois, possède les avantages suivants :

1. Elle fonctionne sur tous les assembleurs
2. Elle est modulaire, à savoir que le code pour chaque fenêtre peut être concentré dans une partie particulière de votre code source
3. Il est facile de suivre à partir du code source quel message provoque quel résultat
4. La même fonction peut facilement être appelée à partir de procédures de fenêtre différentes

Ma méthode est illustrée par la procédure de fenêtre très simple qui suit (en format GoAsm) :

```
WndProc:
MOV EDX,OFFSET MAINMESSAGES
CALL GENERAL_WNDPROC
RET 10h
```

où les messages et les fonctions (spécifiques à cette procédure de fenêtre particulière) sont définis dans un tableau comme celui-ci :

```
;-------------------------------------------------------
DATA SECTION           ; assembleur pour mettre ce qui suit dans la section DATA
;-------------------------- WNDPROC messages de fonctions
MAINMESSAGES   DD (ENDOF_MAINMESSAGES-$-4)/8 ;=nombres de messages
               DD  312h,HOTKEY,116h,INITMENU,117h,INITMENUPOPUP,11Fh,MENUSELECT
               DD  1h,CREATE,2h,DESTROY,  410h,OWN410,411h,OWN411
               DD  231h,ENTERSIZEMOVE,47h,WINDOWPOSCHANGED,24h,GETMINMAXINFO
               DD  1Ah,SETTINGCHANGE,214h,SIZING,46h,WINDOWPOSCHANGING
               DD  2Bh,DRAWITEM,0Fh,PAINT,113h,TIMER,111h,COMMAND
               DD  104h,SYSKEYDOWN,100h,KEYDOWN,112h,SYSCOMMAND
               DD  201h,LBUTTONDOWN,202h,LBUTTONUP,115h,SCROLLMESS
               DD  204h,RBUTTONDOWNUP,205h,RBUTTONDOWNUP
               DD  200h,MOUSEMOVE,0A0h,NCMOUSEMOVE,20h,SETCURSORM
               DD  4Eh,NOTIFY,210h,PARENTNOTIFY,86h,NCACTIVATE,6h,ACTIVATE
               DD  1Ch,ACTIVATEAPP
ENDOF_MAINMESSAGES:    ; label utilisé pour déterminer le nombre de messages
```

```
;-------------------------------------------------------
CODE SECTION                    ; assembleur pour mettre ce qui suit dans la section CODE
;-------------------------------------------------------
```

et où chacune des fonctions ici sont des procédures, par exemple :

```
CREATE:
XOR EAX,EAX     ; garantit le retour avec EAX=0 et Cf=0
RET
```

et où GENERAL_WINDPROC s'écrit comme suit :

```
GENERAL_WNDPROC:
PUSH EBP
MOV EBP,[ESP+10h]               ; récupère uMsg dans EBP
MOV ECX,[EDX]                   ; récupère le nombre de messages à traiter
ADD EDX,4                       ; sauter après le dword donnant la taille
L33:
DEC ECX
JS >L46                         ; flag de signe = 1 => message non trouvé
CMP [EDX+ECX*8],EBP             ; voir si c'est le message correct
JNZ L33                         ; non
MOV EBP,ESP
PUSH ESP,EBX,EDI,ESI            ; sauvegarde les registres comme requis par Windows
ADD EBP,4                       ; on saute le Dword contenant le nombre de messages
;maintenant, [EBP+8]=hWnd,[EBP+0Ch]=uMsg,[EBP+10h]=wParam,[EBP+14h]=lParam
CALL [EDX+ECX*8+4]              ; appel de la procédure correspondant au message
POP ESI,EDI,EBX,ESP
JNC >L48                        ; NC = ne pas appeller DefWindowProc EAX = code de sortie
L46:
PUSH [ESP+18h],[ESP+18h],[ESP+18h],[ESP+18h]      ; ESP change au fur et à mesure des PUSHs
CALL DefWindowProcA
L48:
POP EBP
RET
```

NOTES :

1. Au lieu de donner la valeur courante du message, vous pouvez, bien sûr, donner le nom d'une EQUATE. Par exemple (dans la syntaxe Asm) :

```
WM_CREATE EQU 1h
```

ou si vous préférez

```
WM_CREATE=1h
```

Ou, enfin

```
#define WM_CREATE 1h
```

vous permet d'écrire `WM_CREATE`, `CREATE` au lieu de `1h`, `CREATE` dans les déclarations de MAINMESSAGES si vous le jugez plus approprié.

2. Il est tentant de positionner la table de messages dans la section de code. Ceci est parfaitement possible parce que la seule différence que fait le système Win32 entre la section de code et la section de données tient au fait que la zone mémoire correspondant à la section de code est utilisable en lecture seule, alors que la section de données l'est en lecture/écriture. Cependant, il peut en résulter une perte de performance parce que le processeur est conçu pour lire les données plus rapidement à partir de la section de données. J'ai effectué quelques tests à ce propos et mis en évidence que la présence de la table dans la section de code plutôt que dans la section de données ralentit considérablement l'exécution ainsi qu'en témoignent les résultats suivants :

processeur 486	22% à 36% plus lent
processeur AMD-K6-3D	78% à 193% plus lent

Ces essais ont été réalisés sur une table de 60 messages et les valeurs trouvées ont trait à la rapidité d'exploration de la table pour trouver le message.

3. Les noms de procédure ne doivent pas être des noms d'import d'API pour éviter toute confusion ! Par exemple modifier légèrement *SetCursor* pour éviter toute confusion avec l'API *SetCursor*.

4. Si une fonction restitue le flag de carry Cf à 1, la procédure de fenêtre appellera *DefWindowProc*. Un retour avec Cf à zéro traduira simplement un retour au système avec le code de retour dans EAX. Certains messages doivent être traités de cette manière.

5. Vous pouvez envoyer un paramètre de votre initiative à GENERAL_WNDPROC en utilisant EAX. Ceci est utile si vous souhaitez identifier une fenêtre particulière. Par exemple :

```
SpecialWndProc:
MOV EAX,OFFSET hSpecialWnd
MOV EDX,OFFSET SPECIALWND_MESSAGES
CALL GENERAL_WNDPROC
RET 10h
```

6. L'instruction `ADD EBP,4` juste avant le CALL à la fonction (`CALL [EDX+ECX*8+4]`) a pour objet de garantir que EBP pointe les paramètres sur la pile de la même manière que si la procédure de fenêtre avait été saisie normalement. Ceci est destiné à garantir que la fonction sera compatible si elle est appelée par une procédure de fenêtre ordinaire écrite en assembleur, par exemple :

```
WndProc:
PUSH EBP
MOV EBP,ESP     ;now [EBP+8]=hWnd,[EBP+0Ch]=uMsg,[EBP+10h]=wParam,[EBP+14h]=lParam
```

7. Une procédure normalisée pour traiter les messages d'une procédure de dialogue peut également être créée de la même manière, sauf qu'elle doit retourner TRUE (EAX = 1) si le message est traité et FALSE (EAX = 0) s'il ne l'est pas, et ne pas appeler *DefWindowProc*. La même méthode de codage peut être appliquée à des hameçonnages et de recenseurs CALLBACKS bien ceux-ci puissent varier.

[Adapté de l'article publié, pour la première fois, sur Asm Journal No. 5]

Annexe M
Fichiers Batch

Les fichiers batch sont un héritage du DOS. Bien que théoriquement étrangers au monde Windows, ils permettent de lancer des séquences de commandes programmées en mode console. À ce titre, ils sont très utiles pour simplifier les opérations d'assemblage puis d'édition de liens, celles-ci pouvant comporter de nombreux paramètres dont la saisie répétée lors du processus de mise au point peut se révéler fastidieuse.

Comme les fichiers *asm*, les fichiers *bat* ne doivent contenir aucun caractère de contrôle et peuvent donc s'écrire avec un simple éditeur de texte du style Bloc-Notes de Windows.

Les fichiers *batch* mettent en œuvre une batterie d'instructions dont nous n'allons détailler ici que les plus essentielles pour mener à bien les tâches dont il est question ici.

M1 Un premier aperçu de fichier batch

Mais auparavant, voici un fichier batch nommé *aslink.bat* destiné à automatiser le processus d'assemblage/édition de liens :

```
rem *** Assemblage + Édition de liens ***
goasm %1.asm %1.obj
golink %1.obj %1.exe user32.dll gdi32.exe
pause
```

On le lance par la commande suivante en mode console :

```
C:\GOASM\aslink test
```

Cette commande prescrit à *aslink.bat* la tâche d'assembler le fichier *test.asm* puis d'en assurer l'édition de liens en une seule opération. Notons que l'extension *.asm* n'est pas mentionnée au lancement de *aslink* car reconstituée automatiquement dans son processus.

Examinons maintenant le fichier batch ligne par ligne :

Ligne 1 `REM *** Assemblage + Édition de liens ***`

Elle commence par l'instruction REM qui indique que tous les caractères qui suivent, jusqu'à la fin de ligne sont seulement des commentaires à destination du lecteur du listing et n'ont, de ce fait, aucune fonction dans le processus.

Ligne 2 `goasm %1.asm %1.obj`

Cette ligne voit l'assembleur GoAsm entrer en action. Le terme %1 recueille, par construction, le premier paramètre de la ligne de commande de *aslink*, soit le mot *test* dans notre cas. De ce fait, la ligne 2 équivaut à la formulation `goasm test.asm test.obj`.

Ligne 3 `golink %1.obj %1.exe user32.dll gdi32.exe`

Selon les principes énoncés à la ligne précédente, cette formulation revient à écrire `golink test.obj test.exe user32.dll gdi32.dll`. En clair, GoLink procède à l'édition des liens de *test.obj* pour produire l'exécutable final test.exe. Pour ce faire, il entre en relation avec les DLL *user32.dll* et *gdi32.dll* afin d'y rechercher les adresses d'exécution nécessaires à certaines API du programme.

Ligne 4 `pause`

Cette instruction à pour effet de suspendre l'exécution ligne par ligne du fichier batch et d'attendre la pression d'une touche quelconque pour continuer. Cette suspension prend toute son importance dans l'environnement Windows où le simple fait de double-cliquer sur le fichier *aslink.bat* à partir de l'Explorateur de fichier déclenche l'ouverture fugitive d'une fenêtre DOS, la très brève exécution de aslink.bat et un retour quasi-immédiat à l'Explorateur. Autant dire que nous ne voyons rien de ce qui se passe pendant l'exécution du fichier batch et notamment d'éventuelles erreurs intervenues à l'une ou l'autre des étapes de ce processus. C'est ici qu'intervient l'instruction *pause* qui maintient l'affichage de la fenêtre DOS et de toutes les informations qu'elle contient tant que l'utilisateur n'a pas décidé de la poursuite des opérations.

En ce sens qu'il se propose d'appliquer un traitement indifférencié aux scripts source *asm*, un tel processus batch reste assez théorique car il propose un cadre de traitement rigide, s'agissant notamment de la liste des DLL à solliciter lors de la phase d'édition de liens qui dépend étroitement des API sollicitées par le programme.

M2 Instructions et configurations utilisables dans les fichiers batch

Les instructions de commande sont destinées au processeur cmd.exe généralement localisé dans le répertoire c:\Windows\System32\.

M2.1 Espaces dans les noms de fichiers, répertoires et sous-répertoires

Si l'usage de l'espace était naguère proscrit dans les noms de fichiers, répertoires et sous-répertoires, cette restriction est désormais levée sous Windows et sans inconvénient aucun. Les commandes du DOS accessibles par Windows ont suivi cette évolution et acceptent également l'espace avec, toutefois, une possibilité de déconnexion de cette fonction.

L'autorisation de l'espace relève du *mode étendu* de *cmd.exe* généralement installée par défaut. On peut toutefois déconnecter cette facilité et retrouver ainsi les pratiques du bon vieux DOS mais il nous faut agir, pour cela au niveau de la base de registres. Il convient de se référer au site Microsoft si cette voie est choisie. Sachez, en tout cas, qu'elle est réservée aux programmeurs expérimentés.

M2.2 Les principales instructions

Cd (Chdir) Change le répertoire courant ou affiche le nom du répertoire courant si la commande est lancée sans aucun paramètre.

Syntaxe

cd rep1\nouveau rep : le répertoire courant devient *nouveau rep* si *rep1* puis *nouveau rep* sont des répertoires en aval du répertoire courant avant l'application de **cd**.

cd.. le répertoire courant devient le répertoire amont avant l'application de **cd**.

cd le répertoire courant devient le répertoire racine

cd affiche le répertoire courant incluant tous les répertoires amont ainsi que le nom de lecteur

Cls Efface l'écran et remet le curseur en haut de l'écran

Del Efface le ou les fichiers spécifiés. Le nom des fichiers peut être spécifié en utilisant des caractères génériques. Cette commande doit être utilisée avec prudence car son exécution n'est précédée d'aucun avertissement comme dans Windows. De même, le mécanisme de sauvegarde en Corbeille est inactif car n'agissant que dans le cadre de Windows.

Syntaxe

del machin.txt : efface le fichier *machin.txt* dans le répertoire courant.

del machin.txt bidule.ttg : efface les fichiers *machin.txt* et *bidule.ttg* dans le répertoire courant.

del m*.txt efface tous les fichiers avec l'extension *txt* dont le nom commence par la lettre *m* dans le répertoire courant.

del *.txt efface tous les fichiers avec l'extension *txt* dans le répertoire courant.

del *.* efface tous les fichiers du répertoire courant. Lorsqu'elle est validée, cette commande envoie opportunément le message de demande de confirmation "êtes-vous sûr (O/N) ?". Mais c'est le seul cas...

Dir Affiche le nom des fichiers et sous-répertoires contenus dans le répertoire courant. L'option **/p** accolée à cette commande permet de suspendre le défilement si l'affichage nécessite plus d'une page, l'utilisateur étant invité à actionner une touche quelconque pour accéder à la page suivante.

Echo **echo** permet d'activer (**echo on**) ou de désactiver (**echo off**) l'affichage des différentes lignes d'un fichier batch. Cet affichage est actif par défaut.

La commande, en elle-même, peut ne pas être affichée en stipulant **@echo**.

Si l'on préfère un affichage personnalisé par ligne, il suffit de mettre le symbole @ devant toute commande.

Enfin, on peut obtenir un saut de ligne dans l'affichage en mettant un point à la fin d'**echo**.

Exemples

echo off : neutralise l'affichage de toutes les lignes du fichier batch à partir de cette commande mais affiche néanmoins cette dernière.

@echo off : neutralise l'affichage de toutes les lignes du fichier batch à partir de cette commande mais n'affiche pas cette dernière.

echo tout est parfait : affiche le texte **tout est parfait**.

echo. provoque un saut de ligne du prompt.

Dir Affiche le nom des fichiers et sous-répertoires contenus dans le répertoire courant. L'option **/p** accolée à cette commande permet de suspendre le défilement si l'affichage nécessite plus d'une page, l'utilisateur étant invité à actionner une touche quelconque pour accéder à la page suivante.

Pause Suspend l'exécution du fichier batch avec affichage d'un message invitant l'utilisateur à actionner une touche quelconque pour poursuivre l'exécution du batch. Cette instruction est particulièrement utile lorsque l'on lance des commandes DOS en mode console Windows car, la plupart du temps, l'exécution du fichier batch est trop fugitive pour avoir le temps de prendre connaissance des messages affichés (notifications d'erreurs ou compte-rendu d'exécution).

Ren (Rename) Renomme le fichier mentionné avec le nom spécifié immédiatement après.

Exemple

Ren file.txt file2.txt : renomme le fichier *file.txt* en *file2.txt*.

Nous nous en tiendrons là dans cette énumération des commandes DOS susceptibles d'être utilisées dans les fichiers batch. Les commandes citées sont de loin les plus courantes et il est toujours possible de recourir, en cas de besoin, à une documentation exhaustive sur internet.

M2.3 Le fichier Config.fil

Le fichier Config.fil est utilisable avec GoLink dont il décrit les paramètres de la ligne de commande. On en trouvera une description détaillée dans l'**Annexe C - Organisation de votre travail de programmation**.

Index Alphabétique

Retour au sommaire

Notes relatives au document
Sources, ajouts et modifications

Le présent ouvrage est une traduction française aussi fidèle que possible de documents originaux proposés sur le site http://www.godevtool.com/. Le tableau ci-dessus en donne les liens sous une forme directement activable. Dans quelques cas, assez rares, des ajouts ou des modifications ont été opérés. La liste exhaustive est proposée après le tableau.

Sources

Sections du volume 1	Sources
Introduction, chapitres 1 à 5 et index	View the GoAsm manual
Chapitre 6	Writing 64-bit programs
Annexes	
Annexe A - Exemples de programmes	
Programme HelloWorld1.asm	Simple Windows console program
Programme HelloWorld2.asm	Simple Windows GUI program
Programme HelloWorld3.asm	Simple Windows GUI program
Programme HelloDialog.asm	Simple Dialog program
Programme Hello64World1.asm	Hello 64World 1
Programme Hello64World2.asm	Hello 64World 2
Programme Hello64World3.asm	Hello 64World 3
Annexe B – Écriture d'un programme Windows élémentaire	Quick start to .. writing a simple Windows prog
Annexe C – Pour les débutants... en programmation	For those new to programming
Annexe D – Représentations binaires	Understand bits, binary and bytes
	Understand hex numbers
	Understand finite, negative, signed and 2's complement numbers
Annexe E – Pour les débutants... en langage assembleur	For those new to assembly language
Annexe F – Flags, sauts conditionnels, CMOVcc et SETcc	Understand flags and conditional jumps
Annexe G – Pour les débutants... en Windows	For those new to Windows
Annexe H – Pour les débutants... en débogage symbolique	For those new to symbolic debugging
Annexe I – Comprendre... la Pile –Partie 1	Understand the stack – Part 1
Partie 2	Understand the stack – Part 2
Annexe J – Comprendre... la mémorisation inversée	Understand reverse storage
Annexe K – Quelques conseils et astuces de programmation	Some programming hints and tips
Annexe L – Normalisation des procédures Callback Win32	Standardized window and dialog procedure
Annexe M – Fichiers Batch	*ajouté*

Ajouts et modifications

Introduction, chapitres 1 à 5
Paragraphe3.15.5 : ajout d'un diagramme représentant l'organisation des bits dans un Tword (source Intel).

Annexe A : Exemples de programmes
Les 7 fichiers .asm décrits dans cette annexe bénéficient systématiquement de commentaires traduits en français. Ils sont assortis, dans chaque cas, d'une copie d'écran de la fenêtre la plus explicite et d'un petit exposé introductif.

Annexe B : Écriture d'un programme Windows élémentaire
La portion de l'introduction décrivant les différentes sections d'un programme a été étoffée.

Annexe D : Représentations binaires
Le tableau ci-dessus mentionne que cette annexe regroupe 3 tutoriels du site Godevtool. Y figurent également de notables ajouts. Parmi ces derniers, une introduction sur les systèmes de numération en général et le système binaire en particulier, un exposé succinct sur la représentation des nombres entiers et en virgule flottante.

Annexe F : Flags, sauts conditionnels, CMOVcc et SETcc
Flag Of : pour faciliter la compréhension du mécanisme de fonctionnement de l'instruction SAR, un diagramme issu de documentation Intel a été ajouté et traduit.
Sauts conditionnels : ajout d'un paragraphe décrivant les conventions de traduction en français pour les comparaisons entre grandeurs signées et non-signées.
Instructions CMOVcc et SETcc : une description de ces 2 instructions a été ajoutée car relevant maintenant du mécanisme de sauts conditionnels. Elle a été empruntée, pour l'essentiel, à des documents Intel.

Annexe H : Pour les débutants... en débogage symbolique
Le listing du testeur a été intégré au texte. Le sous-programme TEST_HEXWRITE a été simplifié par la mise en œuvre d'une boucle.

Annexe K : Quelques conseils et astuces de programmation
Conformément à la documentation Intel, des restrictions ont été émises quant à l'utilisation des instructions INC et DEC en lieu et place de ADD et SUB, respectivement (§ K2.4)

Annexe M : Fichiers Batch
Section rédigée par le traducteur.