fasm

flat assembler g
Programmation sous Windows

par Tomasz Grysztar

Édition du 19 Avril 2016

Traduction française

By Calendos
Forum https://flatassembler.net/
Texte original prélevé le 9 mai 2024
Traduction v2.2 (23/05/2024 à 11:33)

Table des matières

La version Windows de l'assembleur FASM a vu la naissance du package standard d'inclusions conçues pour permettre l'écriture des programmes destinés à l'environnement Windows.

Le package d'inclusions contient les entêtes pour la programmation Windows 32 et 64 bits dans le dossier racine et les inclusions spécialisées dans les sous-dossiers. En général, les entêtes incluent les fichiers spécialisés requis pour vous, bien que vous puissiez parfois préférer inclure certains des packages de macro-instructions vous-même (car peu d'entre eux ne sont pas inclus par certains ou même tous les entêtes).

Vous pouvez choisir parmi six entêtes pour Windows 32 bits, avec des noms commençant par `win32` suivis soit de la lettre `a` pour l'utilisation du codage ASCII, soit de la lettre `w` pour le codage WideChar. `win32a.inc` et `win32w.inc` sont les entêtes de base, `win32ax.inc` et `win32wx.inc` sont les entêtes étendus, ils fournissent des instructions de macro plus avancées, ces extensions seront discutées séparément. Enfin, `win32axp.inc` et `win32wxp.inc` sont les mêmes entêtes étendus avec la fonction activée de vérification du nombre de paramètres dans les appels de procédure.

Il existe six packages analogues pour Windows 64 bits, avec des noms commençant par `win64`. Ils fournissent en général les mêmes fonctionnalités que ceux de Windows 32 bits, avec quelques différences expliquées plus tard.

Vous pouvez inclure les entêtes comme vous le souhaitez, en fournissant le chemin complet ou en utilisant la variable d'environnement personnalisée, mais la méthode la plus simple consiste à définir la variable d'environnement `INCLUDE` pointant correctement vers le répertoire contenant les entêtes, puis à les inclure comme :

```
include 'win32a.inc'
```

Il est important de noter que toutes les macro-instructions, par opposition aux directives internes de l'assembleur *flat*, sont sensibles à la casse et que la casse minuscule est utilisée pour la plupart d'entre elles. Si vous préférez utiliser une autre casse par défaut, vous devez faire les ajustements appropriés avec la directive `fix`.

1. Entêtes de base

Les entêtes de base `win32a.inc`, `win32w.inc`, `win64a.inc` et `win64w.inc` incluent des déclarations d'équivalences et de structures Windows et fournissent l'ensemble standard de macro-instructions.

1.1. Structures

Tous les entêtes activent la macroinstruction `struct`, qui permet de définir des structures d'une manière plus similaire aux autres assembleurs que la directive `struc`. La définition de la structure doit commencer par la macroinstruction `struct` suivie de son nom et se terminer par la macro-instruction `ends`. Dans les lignes entre seules les directives de définition de données sont autorisées, les labels étant les noms purs des champs de la structure :

```
struct POINT
  x dd ?
  y dd ?
ends
```

Avec une telle définition la ligne :

```
point1 POINT
```

déclarera la structure `point1` avec les champs `point1.x` et `point1.y`, en leur donnant les valeurs par défaut – les mêmes que celles fournies dans la définition de la structure (dans ce cas, les valeurs par défaut sont toutes les deux des valeurs non initialisées). Mais la déclaration de structure accepte également les paramètres, dans le même nombre que le nombre de champs de la structure, et ces paramètres, lorsqu'ils sont spécifiés, remplacent les valeurs par défaut des champs. Par exemple :

```
point2 POINT 10,20
```

initialisera le champ `point2.x` avec la valeur 10 et `point2.y` avec la valeur 20.

La macro `struct` permet non seulement de déclarer les structures d'un type donné, mais définit également des labels pour les décalages de champs à l'intérieur de la structure et des constantes pour la taille de chaque champ et de la structure entière. Par exemple, la définition ci-dessus de la structure `POINT` définit les étiquettes `POINT.x` et `POINT.y` comme les décalages des champs à l'intérieur de la structure, et `sizeof.POINT.x`, `sizeof.POINT.y` et `sizeof.POINT` comme tailles des champs correspondants et de l'ensemble de la structure. Les étiquettes de décalage peuvent être utilisées pour accéder aux structures adressées indirectement, comme :

```
mov eax,[ebx+POINT.x]
```

lorsque le registre EBX contient le pointeur vers la structure `POINT`. Notez que la vérification de la taille du champ sera également effectuée avec un tel accès.

Les structures elles-mêmes sont également autorisées dans les définitions de structure, de sorte que les structures peuvent avoir d'autres structures en tant que champs :

```
struct LINE
  start POINT
  end   POINT
```

```
ends
```

Lorsqu'aucune valeur par défaut pour les champs de sous-structure n'est spécifiée, comme dans cet exemple, les valeurs par défaut de la définition du type de sous-structure s'appliquent.

La valeur de chaque champ étant un paramètre unique dans la déclaration de la structure, pour initialiser les sous-structures avec des valeurs personnalisées, les paramètres de chaque sous-structure doivent être regroupés en un seul paramètre pour la structure :

```
line1 LINE <0,0>,<100,100>
```

La déclaration ci-dessus initialise chacun des champs `line1.start.x` et `line1.start.y` avec 0 et chacun des champs `line1.end.x` et `line1.end.y` avec 100.

Lorsque la taille des données définies par une valeur transmise à la structure de déclaration est inférieure à la taille du champ correspondant, elle est complétée à cette taille avec des octets non définis (et lorsqu'elle est plus grande, l'erreur se produit). Par exemple :

```
struct FOO
  data db 256 dup (?)
ends

some FOO <"ABC",0>
```

remplit les quatre premiers octets de `some.data` avec des valeurs définies et réserve le reste.

À l'intérieur des structures, des unions et des sous-structures sans nom peuvent également être définies. La définition de l'union doit commencer par `union` et se terminer par `ends`, comme dans cet exemple :

```
struct BAR
  field_1 dd ?
  union
    field_2  dd ?
    field_2b db ?
  ends
ends
```

Chacun des champs définis à l'intérieur de l'union a le même décalage et ils partagent la même mémoire. Seul le premier champ d'union est initialisé avec une valeur donnée, les valeurs du reste des champs sont ignorées (cependant, si l'un des autres champs nécessite plus de mémoire que le premier, l'union est complétée à la taille requise avec des octets non définis). L'union entière est initialisée par le seul paramètre donné dans la déclaration de structure, et ce paramètre donne une valeur au premier champ d'union.

La sous-structure sans nom est définie de la même manière que l'union, ne commence que par la ligne `struct` au lieu de `union`, comme :

```
struct WBB
  word dw ?
  struct
    byte1 db ?
    byte2 db ?
  ends
ends
```

Une telle sous-structure ne prend qu'un seul paramètre dans la déclaration de la structure entière pour définir ses valeurs, et ce paramètre peut lui-même être le groupe de paramètres définissant chaque champ de la sous-structure. Ainsi, le type de structure ci-dessus peut être déclaré comme :

```
my WBB 1,<2,3>
```

Les champs à l'intérieur des unions et des sous-structures sans nom sont accessibles comme si ils étaient directement les champs de la structure parente. Par exemple, avec la déclaration ci-dessus, `my.byte1` et `my.byte2` sont des labels corrects pour les champs de sous-structure.

Les sous-structures et les unions peuvent être imbriquées sans limite de profondeur d'imbrication :

```
struct LINE
  union
    start POINT
    struct
      x1  dd ?
      y1  dd ?
    ends
  ends
  union
```

```
    end   POINT
    struct
      x2  dd ?
      y2  dd ?
    ends
  ends
ends
```

La définition de la structure peut également être basée sur certains des types de structure déjà définis et elle hérite, dans ce cas, de tous les champs de cette structure. Par exemple :

```
struct CPOINT POINT
  color dd ?
ends
```

définit la même structure que :

```
struct CPOINT
  x     dd ?
  y     dd ?
  color dd ?
ends
```

Tous les entêtes définissent le type de données CHAR, qui peut être utilisé pour définir des chaînes de caractères dans les structures de données.

1.2. Imports

Les macro-instructions d'importation aident à créer les données d'importation pour le fichier PE (généralement placées dans la section séparée). Il existe deux macro-instructions à cet effet. La première, qui est appelée library, doit être placée directement au début des données d'importation et définit à partir de quelles bibliothèques les fonctions seront importées. Elle doit être suivie de n'importe quelle quantité de paires de paramètres, chaque paire étant le label de la table des imports de la bibliothèque donnée, et la chaîne entre guillemets définissant le nom de la bibliothèque. Par exemple :

```
library kernel32,'KERNEL32.DLL',\
        user32,'USER32.DLL'
```

déclare importer des deux bibliothèques. Pour chacune, la table des imports doit ensuite être déclarée quelque part dans les données d'importation. Ceci est fait avec la macro-instruction import, qui a besoin du premier paramètre pour définir le label de la table (le même que celui déclaré précédemment dans la macro library), puis les paires de paramètres contenant chacune le label du pointeur importé et la chaîne citée définissant le nom de la fonction exactement comme exportée par la bibliothèque. Par exemple, la déclaration de bibliothèque ci-dessus peut être complétée avec les déclarations d'import suivantes :

```
import kernel32,\
       ExitProcess,'ExitProcess'

import user32,\
       MessageBeep,'MessageBeep',\
       MessageBox,'MessageBoxA'
```

Les labels définis par les premiers paramètres de chaque paire passée à la macro d'importation adressent les pointeurs au format double mot, qui après le chargement du PE sont remplis avec les adresses des procédures exportées.

Au lieu d'une chaîne entre guillemets pour le nom de la procédure à importer, un nombre peut être donné pour définir l'importation par ordinal, sous la forme suivante :

```
import custom,\
       ByName,'FunctionName',\
       ByOrdinal,17
```

Les macros d'import optimisent les données importées de sorte que seules les imports pour les fonctions qui sont utilisées quelque part dans le programme sont placées dans les tables d'importation, et si une table d'importation était vide de cette façon, la bibliothèque entière ne serait pas référencée du tout. Pour cette raison, il est pratique d'avoir la table d'importation complète pour chaque bibliothèque – le package contient de telles tables pour certaines des bibliothèques standard. Elles sont stockées dans les sous-répertoires APIA et APIW et importent les variantes ASCII et WideChar des fonctions API. Chaque fichier contient une table d'import, avec un label en minuscules identique au nom du fichier. Ainsi, les tables complètes pour l'import à partir des bibliothèques KERNEL32.DLL et USER32.DLL peuvent-elles être définies de cette façon (en supposant que votre variable d'environnement INCLUDE pointe vers le répertoire contenant le package d'inclusion) :

```
library kernel32,'KERNEL32.DLL',\
        user32,'USER32.DLL'

include 'apia\kernel32.inc'
include 'apiw\user32.inc'
```

1.3. Procédures (32-bit)

Les entêtes pour Windows 32 bits fournissent quatre macro-instructions pour l'appel des procédures avec des paramètres passés sur la pile. Le `stdcall` appelle directement la procédure spécifiée par le premier argument en utilisant la convention d'appel STDCALL. Le reste des arguments passés à la macro définissent les paramètres de la procédure et sont stockés sur la pile dans l'ordre inverse. La macro `invoke` fait de même, mais elle appelle la procédure indirectement, via le pointeur étiqueté par le premier argument. Ainsi `invoke` peut-elle être utilisée pour appeler les procédures via des pointeurs définis dans les tables d'import. Ainsi, la ligne suivante :

```
invoke MessageBox,0,szText,szCaption,MB_OK
```

est-elle équivalente à :

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

et les deux génèrent ce code :

```
push MB_OK
push szCaption
push szText
push 0
call [MessageBox]
```

Les directives `ccall` et `cinvoke` sont analogues à `stdcall` et `invoke`, mais doivent normalement être utilisées pour appeler les procédures qui utilisent la convention d'appel C où la trame de pile doit être restaurée par l'appelant.

Pour définir la procédure qui utilise la pile pour les paramètres et les variables locales, vous devez utiliser la macro-instruction `proc`. Dans sa forme la plus simple, elle doit être suivie du nom de la procédure, puis du nom de tous les paramètres qu'elle prend, comme :

```
proc WindowProc,hwnd,wmsg,wparam,lparam
```

La virgule entre le nom de la procédure et le premier paramètre est facultative. Les instructions de procédure doivent être écrites dans les lignes qui suivent et se terminer par la macro-instruction `endp`. La trame de pile est mise en place automatiquement à l'entrée de la procédure avec l'utilisation du registre EBP comme base pour accéder aux paramètres. Vous devez donc éviter d'utiliser ce registre à d'autres fins. Les noms spécifiés pour les paramètres sont utilisés pour définir des labels basés sur EBP, que vous pouvez utiliser pour accéder aux paramètres en tant que variables régulières. Par exemple, l'instruction `mov eax,[hwnd]` à l'intérieur de la procédure définie comme dans l'exemple ci-dessus, est rigoureusement équivalente à `mov eax,[ebp+8]`. La portée de ces labels est limitée à la procédure et vous pouvez donc utiliser les mêmes noms à d'autres fins en dehors de la procédure donnée.

Dans la mesure où tous les paramètres sont poussés sur la pile en tant que doubles-mots lors de l'appel de telles procédures, les labels des paramètres sont définis pour considérer que les données sont des doubles-mots par défaut. Mais vous pouvez spécifier la taille des paramètres si vous le souhaitez, en faisant suivre le nom de paramètre par deux-points, puis l'opérateur de taille. L'exemple précédent peut être réécrit de cette façon, ce qui est encore une fois équivalent :

```
proc WindowProc,hwnd:DWORD,wmsg:DWORD,wparam:DWORD,lparam:DWORD
```

Si vous spécifiez une taille inférieure au double mot, le label donné s'applique à la plus petite partie du double-mot entier stocké sur la pile. Si vous spécifiez une taille plus grande, comme le pointeur *far* d'un quadruple mot, les deux paramètres de double-mot sont définis pour contenir cette valeur, mais sont étiquetés comme une seule variable.

Le nom de la procédure peut également être suivi des mots-clé `stdcall` ou `c` pour caractériser la convention d'appel qu'elle utilise. Lorsqu'aucun type de ces types n'est spécifié, la valeur par défaut est utilisée, ce qui équivaut à STDCALL. Puis, de la même manière, le mot-clé `uses` peut suivre, et après cela la liste des registres (séparés uniquement par des espaces) qui seront automatiquement stockés à l'entrée dans la procédure et restaurés à la sortie. Dans ce cas, une virgule après la liste des registres et avant le premier paramètre est requise. Ainsi, l'instruction de procédure complète pourrait-elle ressembler à ceci :

```
proc WindowProc stdcall uses ebx esi edi,\
     hwnd:DWORD,wmsg:DWORD,wparam:DWORD,lparam:DWORD
```

Pour déclarer la variable locale, vous pouvez utiliser la macro-instruction `local`, suivie d'une ou plusieurs déclarations séparées par des virgules, chacune étant constituée du nom de la variable suivi de deux points et du type de variable – l'un des types standard (doit être en majuscules) ou le nom de la structure de données. Par exemple :

```
local hDC:DWORD,rc:RECT
```

Pour déclarer un tableau local, vous pouvez faire suivre le nom de la variable par la taille du tableau entre crochets, comme ainsi :

```
local str[256]:BYTE
```

L'autre manière de définir les variables locales est de les déclarer à l'intérieur du bloc commencé par la macro-instruction "locals" et se terminant par "endl". Dans ce cas, elles peuvent être définies comme des données régulières. Cette déclaration est l'équivalent de l'exemple précédent :

```
locals
  hDC dd ?
  rc RECT
endl
```

Les variables locales peuvent être déclarées n'importe où dans la procédure avec, pour seule impératif, qu'elles soient déclarées avant d'être utilisées. La portée des labels pour les variables définies comme locales est limitée à l'intérieur de la procédure, ce qui signifie que vous pouvez utiliser les mêmes noms à d'autres fins en dehors de la procédure. Si vous donnez des valeurs initialisées aux variables déclarées comme locales, la macro-instruction génère les instructions qui initialiseront ces variables avec les valeurs données et les met à la même position dans la procédure, où la déclaration est placée.

Le ret placé n'importe où à l'intérieur de la procédure, génère le code complet nécessaire pour quitter correctement la procédure, en restaurant notamment la trame de pile et les registres utilisés par la procédure. Si vous avez besoin de générer l'instruction ret brute, utilisez le mnémonique retn, ou faites suivre le ret du paramètre *number*, ce qui le fait également être interprété comme une instruction classique.

Pour récapituler, la définition complète de la procédure peut ressembler à ceci :

```
proc WindowProc uses ebx esi edi,hwnd,wmsg,wparam,lparam
  local hDC:DWORD,rc:RECT
  ; metre les instructions ici
  ret
endp
```

1.4. Procédures (64 bits)

Dans Windows 64 bits, il n'y a qu'une seule convention d'appel et donc, seules deux macro-instructions sont fournies pour les procédures d'appel. La convention fastcall appelle directement la procédure spécifiée par le premier argument en utilisant la convention standard du système Windows 64 bits. La macro invoke fait la même chose, mais indirectement, via le pointeur étiqueté par le premier argument. Les paramètres sont fournis par les arguments qui suivent et peuvent être de n'importe quelle taille jusqu'à 64 bits. Les macro-instructions utilisent le registre RAX comme stockage temporaire lorsqu'une valeur de paramètre ne peut être copiée directement dans la pile à l'aide de l'instruction mov. Si le paramètre est précédé du mot addr, il est traité comme une adresse et est calculé avec l'instruction lea – donc si l'adresse est absolue, elle sera calculée comme relative à RIP, évitant ainsi de générer un déplacement en cas de fichier avec des corrections.

Étant donné que, dans Windows 64 bits, les paramètres à virgule flottante sont transmis d'une manière différente, ils doivent être identifiés en précédant chacun d'eux de l'instruction float. Ils peuvent être de taille double ou quadruple. Voici un exemple d'appel de certaines procédures OpenGL avec des paramètres double précision ou simple précision :

```
invoke glVertex3d,float 0.6,float -0.6,float 0.0
invoke glVertex2f,float dword 0.1,float dword 0.2
```

L'espace de pile pour les paramètres est alloué avant chaque appel et libéré immédiatement après. Cependant, il est possible de n'allouer cet espace qu'une seule fois pour tous les appels à l'intérieur d'un bloc de code donné. Les macros frame et endf sont fournies à cet effet. Elles doivent être utilisées pour délimiter un bloc, à l'intérieur duquel le registre RSP n'est pas modifié entre les appels de procédure et ils empêchent chaque appel d'allouer de l'espace de pile pour les paramètres, car ce dernier est réservé une fois pour toutes par la macro frame puis libéré à la fin par la macro endf.

```
frame  ; allocation de l'espace de pile une seule fois
  invoke TranslateMessage,msg
  invoke DispatchMessage,msg
endf
```

La macro proc pour Windows 64 bits a la même syntaxe et les mêmes fonctionnalités que son homologue 32 bits (bien que les options stdcall et c ne soient d'aucune utilité dans son cas). Il faut cependant noter que, dans la convention d'appel utilisée dans Windows 64 bits, les quatre premiers paramètres sont passés dans des registres (RCX, RDX, R8 et R9), et que donc, même s'il y a un espace réservé pour eux au niveau de la pile et qu'il est étiqueté avec le nom fourni dans la définition de procédure, ces quatre paramètres n'y résideront pas initialement.

Il faut y accéder en lisant directement les registres. Mais si ces registres doivent être utilisés à d'autres fins, il est recommandé de stocker la valeur de ces paramètres dans la cellule mémoire qui leur est réservée. Le début d'une telle procédure peut ressembler à :

```
proc WindowProc hwnd,wmsg,wparam,lparam
  mov [hwnd],rcx
  mov [wmsg],edx
  mov [wparam],r8
  mov [lparam],r9
  ; maintenant les registres peuvent être utilisés à d'autres fins
  ; et les paramètres seront toujours accessibles plus tard
```

1.5. Personnalisation des procédures

Il est possible de créer un code personnalisé pour l'espace de travail d'une procédure lors de l'utilisation de la macro-instruction `proc`. Il existe trois variables symboliques, `prologue@proc`, `epilogue@proc` et `close@proc`, qui définissent les noms des macro-instructions que `proc` appelle à l'entrée, au retour (créé avec la macro `ret`) et à la fin de la procédure (fait avec la macro `endp`). Ces variables peuvent être redéfinies pour pointer vers d'autres macro-structures, de sorte que tout le code généré avec la macro `proc` puisse être personnalisé.

Chacune de ces trois macro-instructions prend cinq paramètres. Le premier fournit un label de point d'entrée de procédure, qui est également le nom de cette même procédure. Le second est un champ de bits contenant des flags, notamment le bit 4 est mis à 1 lorsque l'appelant est censé restaurer la pile, ou à 0 dans le cas contraire. Le troisième est une valeur qui spécifie le nombre d'octets que les paramètres de la procédure prennent sur la pile. Le quatrième est une valeur qui spécifie le nombre d'octets à réserver pour les variables locales. Enfin, le cinquième et dernier paramètre est la liste des registres séparés par des virgules, que la procédure a déclaré être utilisés et qui doivent donc être sauvegardés par le prologue puis restaurés par l'épilogue.

La macro de prologue, outre la génération de code qui met en place la trame de pile et le pointeur vers les variables locales, doit définir deux variables symboliques, `parmbase@proc` et `localbase@proc`. La première doit fournir l'adresse de base de l'emplacement des paramètres, la seconde, l'adresse de l'emplacement des variables locales – généralement par rapport au registre EBP/RBP, mais il est possible d'utiliser d'autres bases s'il peut être garanti que ces pointeurs seront valides à tout moment à l'intérieur de la procédure où des paramètres ou des variables locales sont accessibles. Il appartient également à la macro `prologue` de faire les alignements nécessaires pour une implémentation de procédure valide; la taille des variables locales fournies comme quatrième paramètre peut elle-même ne pas être alignée du tout.

Le comportement par défaut de `proc` est défini par les macros `prologuedef` et `epiloguedef` (dans le cas par défaut, il n'est pas nécessaire de fermer la macro, donc `close@proc` a une valeur vide). S'il est nécessaire de revenir aux valeurs par défaut après l'utilisation de certaines personnalisations, cela doit être fait avec les trois lignes suivantes :

```
prologue@proc equ prologuedef
epilogue@proc equ epiloguedef
close@proc equ
```

À titre d'exemple de prologue modifié, vous trouverez ci-dessous la macro-instruction qui implémente le prologue de détection de pile pour Windows 32 bits. Une telle méthode d'allocation doit être utilisée chaque fois que la zone des variables locales peut dépasser 4 096 octets.

```
macro sp_prologue procname,flag,parmbytes,localbytes,reglist
{ local loc
  loc = (localbytes+3) and (not 3)
  parmbase@proc equ ebp+8
  localbase@proc equ ebp-loc
  if parmbytes | localbytes
   push ebp
   mov ebp,esp
   if localbytes
    repeat localbytes shr 12
     mov byte [esp-%*4096],0
    end repeat
    sub esp,loc
   end if
  end if
  irps reg, reglist \{ push reg \} }

prologue@proc equ sp_prologue
```

Il peut être facilement modifié pour mettre en oeuvre toute autre méthode de détection de pile au choix du pro-

grammeur.

Les entêtes 64 bits fournissent un ensemble supplémentaire de macros prologue/épilogue, qui permettent de définir une procédure qui utilise RSP pour accéder aux paramètres et aux variables locales (le registre RBP est donc disponible pour tout autre usage par la procédure) et alloue également l'espace commun pour tous les appels de procédure effectués à l'intérieur, de sorte que les macros `fastcall` ou `invoke` n'ont pas besoin d'allouer elles-mêmes d'espace de pile. C'est un effet similaire à celui obtenu en plaçant le code à l'intérieur de la procédure dans un bloc de trame, mais dans ce cas, l'allocation d'espace de pile pour les appels de procédure est fusionné avec l'allocation d'espace pour les variables locales. Le code à l'intérieur de cette procédure ne doit en aucun cas modifier le registre RSP. Pour basculer vers ce comportement de `proc` 64 bits, utilisez les instructions suivantes :

```
prologue@proc equ static_rsp_prologue
epilogue@proc equ static_rsp_epilogue
close@proc equ static_rsp_close
```

1.6. Exports

La macro-instruction `export` construit les données d'exportation pour le fichier PE (elles doivent être placées soit dans la section marquée comme export, soit dans le bloc `data export`. Le premier argument doit être une chaîne entre guillemets définissant le nom du fichier de bibliothèque, et le reste doit être n'importe quel nombre de paires d'arguments, le premier de chaque paire étant le nom de la procédure définie quelque part à l'intérieur de la source, et le second étant la chaîne entre guillemets contenant le nom sous lequel cette procédure doit être exportée par la bibliothèque. L'exemple qui suit :

```
export 'MYLIB.DLL',\
       MyStart,'Start',\
       MyStop,'Stop'
```

définit la table exportant deux fonctions, qui sont définies sous les noms `MyStart` et `MyStop` dans les sources, mais qui seront exportées par la bibliothèque sous des noms plus courts. La macro-instruction s'occupe du tri alphabétique du tableau, ce qui est requis par le format PE.

1.7. Modèle d'Objet de Composant

La macro `interface` permet de déclarer l'interface du type d'objet COM. Le premier paramètre est le nom de l'interface, puis les noms consécutifs des méthodes doivent suivre, comme dans cet exemple :

```
interface ITaskBarList,\
          QueryInterface,\
          AddRef,\
          Release,\
          HrInit,\
          AddTab,\
          DeleteTab,\
          ActivateTab,\
          SetActiveAlt
```

La macro `comcall` peut ensuite être utilisée pour appeler la méthode de l'objet donné. Le premier paramètre de cette macro doit être le handle d'objet, le second doit être le nom de l'interface COM implémentée par cet objet, puis le nom de la méthode et les paramètres de cette méthode. Par exemple :

```
comcall ebx,ITaskBarList,ActivateTab,[hwnd]
```

utilise le contenu du registre EBX comme descripteur de l'objet COM avec l'interface `ITaskBarList` et appelle la méthode `ActivateTab` de cet objet avec le paramètre `[hwnd]`.

Vous pouvez également utiliser le nom de l'interface COM de la même manière que le nom de la structure de données, pour définir la variable qui contiendra le handle d'objet de type donné :

```
ShellTaskBar ITaskBarList
```

La ligne ci-dessus définit la variable dans laquelle le handle de l'objet COM peut être stocké. Après y avoir stocké le handle vers un objet, ses méthodes peuvent être appelées avec le `cominvk`. Cette macro n'a besoin que du nom de la variable avec l'interface affectée et du nom de la méthode comme deux premiers paramètres, puis des paramètres de la méthode. Ainsi, la méthode `ActivateTab` de l'objet dont le handle est stocké dans la variable ShellTaskBar comme défini ci-dessus peut-il être appelée de cette façon :

```
cominvk ShellTaskBar,ActivateTab,[hwnd]
```

qui fait la même chose que :

```
comcall [ShellTaskBar],ITaskBarList,ActivateTab,[hwnd]
```

1.8. Ressources

Il existe deux façons de créer des ressources. L'une consiste à inclure le fichier de ressources externe créé avec un autre programme et l'autre à créer manuellement une section de ressources. Cette dernière méthode, bien que ne nécessitant aucun programme supplémentaire pour être mise en oeuvre, est plus laborieuse, mais les entêtes standard fournissent une assistance – l'ensemble des macro-instructions élémentaires qui servent de briques pour composer la section des ressources.

La macro-instruction `directory` doit être placée directement au début des données de ressources créées manuellement. Elle définit les types de ressources qu'elle contient. Elle doit être suivie des paires de valeurs, la première de chaque paire étant l'identifiant du type de ressource, et la seconde le label de sous-répertoire des ressources de type donné. Cela peut ressembler à ceci :

```
directory RT_MENU, menus, \
          RT_ICON, icons, \
          RT_GROUP_ICON, group_icons
```

Les sous-répertoires peuvent être placés n'importe où dans la zone de ressources après le répertoire principal, et doivent être définis avec la macro-instruction `resource`, qui nécessite que le premier paramètre soit le label du sous-répertoire (correspondant à l'entrée dans le répertoire principal) suivi de trios de paramètres – dans chacune de ces entrées, le premier paramètre définit l'identifiant de la ressource (cette valeur est librement choisie par le programmeur et est ensuite utilisée pour accéder à la ressource donnée à partir du programme), le second spécifie la langue et le troisième est le label de ressource. Les équivalences standard doivent être utilisées pour créer des identifiants de langue. Par exemple, le sous-répertoire des menus peut être défini de cette façon :

```
resource menus, \
         1, LANG_ENGLISH+SUBLANG_DEFAULT, main_menu, \
         2, LANG_ENGLISH+SUBLANG_DEFAULT, other_menu
```

Si la ressource est d'un type pour lequel la langue n'a pas d'importance, l'identifiant de langue `LANG_NEUTRAL` doit être utilisé. Pour définir les ressources de différents types, il existe des macro-instructions spécialisées, qui doivent être placées dans la zone de ressources.

Les bitmaps sont les ressources avec l'identifiant de type `RT_BITMAP`. Pour définir la ressource bitmap, utilisez la macro-instruction `bitmap` avec le premier paramètre constituant le babel de la ressource (correspondant à l'entrée dans le sous-répertoire des bitmaps) et le second étant la chaîne entre guillemets contenant le chemin d'accès du fichier bitmap, comme :

```
bitmap program_logo, 'logo.bmp'
```

Il existe deux types de ressources liées aux icônes. L'identifiant `RT_GROUP_ICON` est le type de la ressource, qui doit être liée à une ou plusieurs ressources de type `RT_ICON`, chacune contenant une seule image. Cela permet de déclarer des images de différentes tailles et profondeurs de couleurs sous un identifiant de ressource commun. Cet identifiant, donné à la ressource de type `RT_GROUP_ICON` peut ensuite être passé à la fonction `LoadIcon`, et celle-ci choisira l'image de dimensions appropriées dans le groupe. Pour définir l'icône, utilisez la macro-instruction `icon`, le premier paramètre étant le libellé de la ressource `RT_GROUP_ICON`, suivi des paires de paramètres déclarant les images. Le premier paramètre de chaque paire doit être le label de la ressource `RT_ICON` et le second, la chaîne entre guillemets contenant le chemin d'accès au fichier d'icône. Dans la variante la plus simple, lorsqu'un groupe d'icônes ne contient qu'une seule image, cela ressemblera à :

```
icon main_icon, icon_data, 'main.ico'
```

où `main_icon` est le label de l'entrée dans le sous-répertoire de ressources pour le type `RT_GROUP_ICON`, et `icon_data`, le label de l'entrée de type `RT_ICON`.

Les curseurs sont définis de manière similaire aux icônes, avec les types `RT_GROUP_CURSOR` et `RT_CURSOR` et la macro `cursor`, qui prend des paramètres analogues à ceux pris par la macro `icon`. Ainsi, la définition du curseur peut-elle ressembler à ceci :

```
cursor my_cursor, cursor_data, 'my.cur'
```

Les menus ont le type de ressource `RT_MENU` et sont définis avec la macro-instruction `menu` suivie de quelques autres définissant les éléments à l'intérieur du menu. La macro-instruction `menu`, en tant que telle, ne prend qu'un seul paramètre – le label de la ressource. `menuitem` caractérise l'élément dans le menu. Il prend jusqu'à cinq paramètres, mais seulement deux sont nécessaires – le premier est la chaîne entre guillemets contenant le texte de l'élément, et le second est la valeur de l'identifiant (qui est la valeur qui sera retournée lorsque l'utilisateur sélectionnera l'élément donné dans le menu). `menuseparator` définit un séparateur dans le menu et ne nécessite aucun paramètre.

Le troisième paramètre facultatif de `menuitem` spécifie les flags de ressources du menu. Deux flags de ce type sont disponibles - `MFR_END` est le flag du dernier élément du menu donné, et `MFR_POPUP` indique que l'élément donné est le sous-menu, et les éléments suivants seront des éléments composant ce sous-menu jusqu'à ce que l'élément avec le flag `MFR_END` soit trouvé. Le flag `MFR_END` peut également être donné comme paramètre à `menuseparator` et est le seul paramètre que cette macro-instruction peut prendre. Pour que la définition du menu soit

complète, chaque sous-menu doit être fermé par l'élément avec le flag MFR_END. De même, le menu entier doit également être fermé de cette façon. Voici un exemple de définition complète de menu :

```
menu main_menu
     menuitem '&File',100,MFR_POPUP
              menuitem '&New',101
              menuseparator
              menuitem 'E&xit',109,MFR_END
     menuitem '&Help',900,MFR_POPUP + MFR_END
              menuitem '&About...',901,MFR_END
```

Le quatrième paramètre menuitem facultatif spécifie les flags d'état pour l'élément donné. Ces flags sont les mêmes que ceux utilisés par les fonctions API, comme MFS_CHECKED ou MFS_DISABLED. De même, le cinquième paramètre peut spécifier les flags de type. Par exemple, cela définira l'élément coché avec une case d'option :

```
menuitem 'Selection',102, ,MFS_CHECKED,MFT_RADIOCHECK
```

Les boîtes de dialogue ont le type de ressource RT_DIALOG et sont définies avec la macro-instruction dialog suivie par un nombre quelconque d'éléments définis avec l'élément dialogitem terminé par enddialog.

La boîte de dialogue peut prendre jusqu'à onze paramètres, les sept premiers étant requis. Le premier paramètre, comme d'habitude, spécifie le label de la ressource, le second s'intéresse à la chaîne entre guillemets contenant le titre de la boîte de dialogue. Les quatre paramètres suivants spécifient respectivement les coordonnées horizontales et verticales, la largeur et la hauteur de la fenêtre de la boîte de dialogue. Le septième paramètre décrit les flags de style pour la fenêtre de la boîte de dialogue. Le huitième, facultatif, spécifie les flags de style étendus. Le neuvième paramètre peut spécifier le menu de la fenêtre – il doit s'agir de l'identifiant de la ressource de menu, le même que celui spécifié dans le sous-répertoire des ressources de type RT_MENU. Enfin, les dixième et onzième paramètres peuvent être utilisés pour définir la police de la boîte de dialogue – le premier d'entre eux doit être la chaîne entre guillemets contenant le nom de la police, et le dernier le numéro définissant la taille de la police. Lorsque ces paramètres facultatifs ne sont pas spécifiés, la police par défaut MS Sans Serif de taille 8 est utilisée.

Cet exemple montre la macro-instruction dialog utilisant tous les paramètres à l'exception du menu (qui est laissé avec une valeur vide). Les paramètres optionnels sont dans la deuxième ligne :

```
dialog about,'About',50,50,200,100,WS_CAPTION+WS_SYSMENU,\
             WS_EX_TOPMOST, ,'Times New Roman',10
```

La macro-instruction dialogitem comporte huit paramètres obligatoires et un facultatif. Le premier paramètre doit être la chaîne entre guillemets contenant le nom de classe de l'élément. Le deuxième paramètre peut être soit la chaîne entre guillemets contenant le texte de l'élément, soit l'identifiant de ressource dans le cas où le contenu de l'élément doit être défini par une ressource supplémentaire (comme l'élément de la classe STATIC avec le style SS_BITMAP). Le troisième paramètre est l'identifiant de l'élément, utilisé pour identifier l'élément par les fonctions API. Les quatre paramètres suivants spécifient respectivement les coordonnées horizontales, verticales, la largeur et la hauteur de l'élément. Le huitième paramètre spécifie le style de l'élément et le neuvième, facultatif, caractérise les flags de style étendus. Voici un exemple de définition d'élément de dialogue :

```
dialogitem 'BUTTON','OK',IDOK,10,10,45,15,WS_VISIBLE+WS_TABSTOP
```

Et un exemple d'élément statique contenant un bitmap, en supposant qu'il existe une ressource bitmap d'identifiant 7 :

```
dialogitem 'STATIC',7,0,10,50,50,20,WS_VISIBLE+SS_BITMAP
```

La définition de la ressource de dialogue peut contenir n'importe quelle quantité d'éléments y compris nulle. Elle doit toujours se terminer par une macro-instruction enddialog.

Les ressources de type RT_ACCELERATOR sont créées avec une macro-instruction accelerator. Comme d'habitude, le premier paramètre est le label de la ressource. Les suivants sont organisés en trios de paramètres – les flags de raccourci suivis du code de touche virtuelle ou du caractère ASCII et la valeur de l'identifiant (qui est comme l'identifiant de l'élément de menu). Une simple définition de raccourci peut ressembler à ceci :

```
accelerator main_keys,\
            FVIRTKEY+FNOINVERT,VK_F1,901,\
            FVIRTKEY+FNOINVERT,VK_F10,109
```

Les informations de version correspondent à la ressource de type RT_VERSION et sont créées avec la macro-instruction versioninfo. Après le label de la ressource, le deuxième paramètre spécifie le système d'exploitation du fichier PE (généralement VOS__WINDOWS32), le troisième paramètre le type de fichier (les plus courants sont VFT_APP pour le programme et VFT_DLL pour la bibliothèque), le quatrième, le sous-type (généralement VFT2_UNKNOWN), le cinquième, l'identifiant de langue, le sixième, la page de codes et ensuite les paramètres de chaîne entre guillemets, étant les paires de nom de propriété et de la valeur correspondante. Les informations de version les plus simples peuvent être définies comme suit :

```
versioninfo vinfo,VOS__WINDOWS32,VFT_APP,VFT2_UNKNOWN,\
                LANG_ENGLISH+SUBLANG_DEFAULT,0,\
            'FileDescription','Description of program',\
            'LegalCopyright','Copyright et cetera',\
            'FileVersion','1.0',\
            'ProductVersion','1.0'
```

D'autres types de ressources peuvent être définis avec la macro-instruction resdata, qui ne prend qu'un seul paramètre – le label de la ressource – et peut être suivie de toutes les instructions définissant les données. resdata se termine par une macro-instruction endres, comme indiqué ci-après :

```
resdata manifest
  file 'manifest.xml'
endres
```

1.9. Encodage de texte

Les macro-instructions de ressources utilisent la directive du pour définir toutes les chaînes Unicode à l'intérieur des ressources. Dans la mesure où cette directive ne fait qu'étendre à 16 bits les caractères sur un octet en mettant à zéro l'octet supérieur, la macro-instruction du peut avoir besoin d'être redéfinie pour les chaînes contenant des caractères non ASCII. Pour certains encodages, les macro-instructions redéfinissant du pour générer correctement les textes Unicode sont fournies dans le sous-répertoire ENCODING. Par exemple, si le texte source est encodé avec la page de codes Windows 1250, une telle ligne doit être placée quelque part au début de la source :

```
include 'encoding\win1250.inc'
```

2. Entêtes étendus

Les fichiers win32ax.inc, win32wx.inc, win64ax.inc et win64wx.inc fournissent toutes les fonctionnalités des entêtes de base et incluent quelques fonctionnalités supplémentaires impliquant des macro-instructions plus complexes. De même, si aucun format PE n'est déclaré avant d'inclure les entêtes étendus, les entêtes le déclarent automatiquement. Les fichiers win32axp.inc, win32wxp.inc, win64axp.inc et win64wxp.inc sont les variantes d'entêtes étendus qui effectuent en plus la vérification du nombre de paramètres pour les appels de procédure.

2.1. Paramètres de procédure

Avec les entêtes étendus, les macro-instructions pour les procédures d'appel permettent plus de types de paramètres que les valeurs de double-mot caractérisant les entêtes de base. Tout d'abord, lorsque la chaîne entre guillemets est passée en paramètre à la procédure, elle est utilisée pour définir les données de chaîne placées parmi le code, et passe à la procédure le double-mot du pointeur vers cette chaîne. Cela permet de définir facilement les chaînes qui ne doivent pas être réutilisées, juste dans la ligne appelant la procédure qui nécessite des pointeurs vers ces chaînes, comme :

```
invoke MessageBox,HWND_DESKTOP,"Message","Caption",MB_OK
```

Si le paramètre est le groupe contenant des valeurs séparées par des virgules, il est traité de la même manière qu'un simple paramètre de chaîne entre guillemets.

Si le paramètre est précédé du mot addr, cela signifie que cette valeur est une adresse et que ladite adresse doit être passée à la procédure, même si cela ne peut pas être fait directement – comme dans le cas des variables locales, qui ont des adresses relatives au registre EBP/ RBP. En 32 bits, le registre EDX est utilisé temporairement pour calculer la valeur d'adresse et la transmettre à la procédure. Par exemple,

```
invoke RegisterClass,addr wc
```

dans le cas où wc est la variable locale à l'adresse EBP-100h, générera cette séquence d'instructions :

```
lea edx,[ebp-100h]
push edx
call [RegisterClass]
```

Cependant, lorsque l'adresse donnée n'est relative à aucun registre, elle est stockée directement.

Dans le cas du 64 bits, le préfixe addr est autorisé même lorsque seuls les entêtes standard sont utilisés, car il peut être utile y compris dans le cas des adresses régulières, car il applique le calcul des adresses relatives à RIP.

Avec les entêtes 32 bits, si le paramètre est précédé du mot double, il est traité comme une valeur 64 bits et transmis à la procédure comme deux paramètres 32 bits. Par exemple,

```
invoke glColor3d,double 1.0,double 0.1,double 0.1
```

va transmettre les trois paramètres 64 bits sous forme de six mots doubles à la procédure. Si le paramètre suivant double est l'opérande mémoire, il ne doit pas avoir d'opérateur de taille, le double fonctionne déjà comme changeur de taille.

Enfin, les appels aux procédures peuvent être imbriqués, c'est-à-dire que l'appel à une procédure peut être utilisé

comme paramètre d'une autre. Dans ce cas, la valeur renvoyée dans EAX/RAX par la procédure imbriquée est transmise en tant que paramètre à la procédure dans laquelle elle est imbriquée. Exemple d'une telle imbrication :

```
invoke MessageBox,<invoke GetTopWindow,[hwnd]>,\
                  "Message","Caption",MB_OK
```

Il n'y a aucune limite à la profondeur d'imbrication des appels de procédure.

2.2. Structuration du programme-source

Les entêtes étendus permettent certaines macro-instructions qui facilitent la structuration du programme. Les .data et .code ne sont que les raccourcis vers les déclarations de sections pour les données et le code. La macro-instruction .end doit être placée à la fin du programme, avec un paramètre spécifiant le point d'entrée du programme, et elle génère également automatiquement la section d'importation en utilisant toutes les tables d'importation standard. Dans Windows 64 bits, le .end aligne automatiquement la pile sur une limite de 16 octets.

La macro-instruction .if génère un morceau de code qui vérifie une condition simple au moment de l'exécution, et en fonction du résultat, continue l'exécution du bloc suivant ou l'ignore. Le bloc doit se terminer par .endif, mais plus tôt, la macro-instruction .elseif peut être utilisée une ou plusieurs fois et commencer le code qui sera exécuté sous une condition supplémentaire, lorsque les conditions précédentes n'ont pas remplies. La macro-instruction .else comme la dernière avant .endif pour commencer le bloc qui sera exécuté lorsque toutes les conditions sont fausses.

La condition peut être spécifiée à l'aide d'un opérateur de comparaison – tel que =, <, >, <=, >= ou <> – entre les deux valeurs, dont la première doit être un opérande de registre ou de mémoire. Les valeurs sont comparées comme des valeurs non signées, sauf si l'expression de comparaison est précédée du mot signed. Si vous ne fournissez qu'une seule valeur comme condition, elle sera testée par rapport à zéro et la condition ne sera vraie que si ce n'est pas le cas. Par exemple,

```
.if eax
  ret
.endif
```

génère les instructions, qui sautent le ret lorsque EAX est à zéro.

Il existe également des symboles spéciaux reconnus comme conditions : ZERO? est vrai lorsque le flag ZF est à 1, de la même manière que CARRY?, SIGN?, OVERFLOW? et PARITY? correspondent respectivement à l'état des flags CF, SF, OF et PF.

Les conditions simples comme ci-dessus peuvent être composées en expressions conditionnelles complexes à l'aide des opérateurs &, |, respectivement pour la conjonction et l'alternative, l'opérateur ~ pour la négation et les parenthèses. Par exemple,

```
.if eax<=100 & ( ecx | edx )
  inc ebx
.endif
```

va générer les instructions de comparaison et de saut qui entraîneront l'exécution du bloc donné uniquement lorsque EAX est inférieur ou égal à 100 et, qu'en même temps, au moins l'un des registres ECX et EDX n'est pas nul.

La macro-instruction .while génère les instructions qui répéteront l'exécution du bloc donné (terminé par la macro-instruction .endw) tant que la condition est vraie. La condition doit suivre le .while et peut être spécifiée de la même manière que pour le .if.

La paire de macro-instructions .repeat et .until définit le bloc qui sera exécuté à plusieurs reprises jusqu'à ce que la condition donnée soit remplie – cette fois, la condition doit succéder à la macro-instruction .until, placée à la fin du bloc, ainsi que le précise l'exemple suivant :

```
.repeat
  add ecx,2
.until ecx>100
```