

# Le thunking générique en Pascal

par [Alcatiz](#)>

Date de publication : 01/11/2005

Dernière mise à jour : 24/12/2006

Ce tutoriel explique comment utiliser le thunking générique en Pascal pour permettre à un programme Windows 16 bits d'appeler directement des fonctions de l'API 32 bits

- I - Le thinking générique
- II - Les fonctions WOW nécessaires au thinking générique
  - A - LoadLibraryEx32W
  - B - FreeLibrary32W
  - C - GetProcAddress32W
  - D - GetVDMPointer32W
  - E - CallProc32W
- III - Chargement des fonctions nécessaires au thinking
- IV - Le coeur du thinking : la fonction Appel32
- V - Un exemple simple : la fonction GetVersion
- VI - Conversion d'un handle de fenêtre : la fonction HWnd32
- VII - Un exemple plus complexe : la fonction CreateFile
- VIII - Un dernier exemple avec une structure : la fonction GetOpenFileName
- IX - Portabilité
- X - L'unité Win32

## I - Le thinking générique

Les versions de Windows de la famille NT (Windows NT, Windows 2000, Windows 2003 Server, Windows XP) permettent à des applications 16 bits d'être exécutées, grâce à la technologie **WOW** (Windows on Windows NT). La couche WOW relaie automatiquement les appels à l'API 16 bits aux fonctions correspondantes de l'API 32 bits.

Toutefois, il peut être nécessaire à une application 16 bits d'accéder directement à une fonction de l'API 32 bits (soit parce qu'elle n'existe pas dans l'API 16 bits, soit parce qu'elle remplace avantageusement une fonction 16 bits).

Un exemple : les fonctions de gestion de fichiers de l'API 16 bits (\_ICreat, \_IOpen, \_IRead, \_IWrite, \_ISeek...) sont implémentées dans l'API 32 bits et sont appelées par la couche WOW lorsque leur version 16 bits est appelée par un programme. Mais, comme ces dernières, les versions 32 bits de ces fonctions ne supportent pas non plus les noms longs de fichiers ! D'où l'utilité de pouvoir directement appeler les nouvelles fonctions du kernel 32 bits (CreateFile, ReadFile, WriteFile, SetFilePointer...).

Sur les plate-formes de la famille NT, la technique qui permet cet appel direct est appelé **thinking générique**.

D'autres plate-formes Windows 32 bits supportent d'autres techniques de thinking (thinking universel sur Win32s, par exemple). Toutefois, la portabilité du code qui fait l'objet de ce tutoriel est *relativement* bonne vers Windows 95, Windows 98 et Windows Me.

## II - Les fonctions WOW nécessaires au thinking générique

### A - LoadLibraryEx32W

**Function LoadLibraryEx32W (lpszLibFile : pChar; hFile, dwFlags : LongInt) : LongInt;**

Cette fonction permet de charger une librairie 32 bits.

Paramètres :

**lpszLibFile** : adresse d'une chaîne AZT contenant le nom de la librairie;

**hFile** : paramètre inutilisé; doit être à 0;

**dwFlags** : action à accomplir après le chargement de la librairie; doit être à 0 dans notre cas.

Valeur retournée :

Le handle 32 bits de la librairie.

### B - FreeLibrary32W

**Function FreeLibrary32W (hInst : LongInt) : LongBool;**

Cette fonction décharge une librairie 32 bits (chargée par LoadLibraryEx32W).

Paramètres :

**hInst** : le handle de la librairie.

Valeur retournée :

True si OK ou False si erreur.

### C - GetProcAddress32W

**Function GetProcAddress32W (hModule : LongInt; lpszProc : pChar) : Pointer;**

Cette fonction retourne l'adresse d'une fonction dans une librairie 32 bits.

Paramètres :

**hModule** : le handle de la librairie;

**lpszProc** : adresse d'une chaîne AZT contenant le nom de la fonction.

Valeur retournée :

L'adresse de la fonction ou Nil si erreur.

## D - GetVDMPointer32W

**Function GetVDMPointer32W (IpAddress : Pointer; fMode : Word) : Pointer;**

Cette fonction convertit une adresse segmentée 16:16 en adresse linéaire 32 bits.

(voir à ce sujet les chapitres 1.2.6 à 1.2.8 de l'excellent [tutoriel Asm/C/C++](#) de Beuss).

Paramètres :

**IpAddress** : le pointeur 16:16 à convertir;

**fMode** : le mode de conversion (*voir remarque ci-dessous*).

Valeur retournée :

Le pointeur converti en adresse linéaire 32 bits.

*Remarque concernant le mode de conversion* :

*La MSDN documente le paramètre **fMode** comme étant :*

- 0 si le paramètre `IpAddress` est une adresse en mode réel;
- 1 si `IpAddress` est une adresse en mode protégé 16 bits.

*D'autres sources documentent `fMode` comme étant **la taille de la structure dont on doit convertir l'adresse**.  
La pratique confirme cette interprétation.*

## E - CallProc32W

**Function CallProc32W (Param1, Param2 ... Param32 : LongInt; IpProcAddress : Pointer; fAddressConvert, nParams : LongInt) : LongInt;**

Cette fonction appelle directement une fonction de l'API 32 bits.

*Déclarée telle quelle, cette fonction déclenche immédiatement une erreur de compilation. Car contrairement au C et à d'autres langages, **il est impossible en Pascal de prototyper une procédure ou une fonction avec un nombre variable de paramètres.***

*Pour contourner le problème, nous allons devoir déposer nous-même tous les paramètres sur la pile. Ce sera le travail de la fonction **Appel32** que nous réaliserons plus loin.*

La fonction CallProc32W sera donc déclarée comme suit :

**Function CallProc32W (IpProcAddress : Pointer; fAddressConvert, nParams : LongInt) : LongInt;**

Paramètres :

**IpProcAddress** : l'adresse de la fonction à exécuter (typiquement obtenue par `GetProcAddress32W`);

**fAddressConvert** : masque de bits indiquant quels paramètres sont des adresses à convertir;

**nParams** : le nombre de paramètres de la fonction à exécuter.

Valeur retournée :

Dépend bien sûr de la fonction appelée !

La valeur retournée étant un entier 32 bits, un transtypage sera parfois nécessaire (par exemple, le résultat de la fonction `GetOpenFileName` du [chapitre VIII](#) est un booléen).



### III - Chargement des fonctions nécessaires au thinking

Pour utiliser la technique du thinking générique, il faut bien entendu que toutes les fonctions nécessaires soient chargées. Elles se trouvent dans la librairie **\System32\KRNL386.EXE**, dont le nom interne est **KERNEL**.

Les adresses des fonctions sont stockées dans des constantes typées de type Function.

```

Const (* Handle de la librairie KERNEL *)
  hInstanceKernell16 : tHandle = 0;

  (* Adresses des fonctions de thinking *)
  LoadLibraryEx32W : Function (lpszLibFile : pChar; hFile, dwFlags : LongInt) : LongInt = Nil;
  FreeLibrary32W : Function (hInst : LongInt) : LongBool = Nil;
  GetProcAddress32W : Function (hModule : LongInt; lpszProc : pChar) : Pointer = Nil;
  GetVDMPointer32W : Function (lpAddress : Pointer; fMode : Word) : Pointer = Nil;
  CallProc32W : Function (lpProcAddress : Pointer; fAddressConvert, nParams : LongInt) : LongInt
= Nil;

Begin
  (* Ouverture de la librairie KERNEL et chargement des fonctions *)
  hInstanceKernell16 := LoadLibrary('KERNEL');
  if hInstanceKernell16 >= 32
  then
    begin
      @LoadLibraryEx32W := GetProcAddress(hInstanceKernell16, 'LoadLibraryEx32W');
      @FreeLibrary32W := GetProcAddress(hInstanceKernell16, 'FreeLibrary32W');
      @GetProcAddress32W := GetProcAddress(hInstanceKernell16, 'GetProcAddress32W');
      @GetVDMPointer32W := GetProcAddress(hInstanceKernell16, 'GetVDMPointer32W');
      @CallProc32W := GetProcAddress(hInstanceKernell16, 'CallProc32W');
    end;

  { ----- Ici se situe le corps du programme ----- }

  (* Fermeture de la librairie en fin de programme *)
  if hInstanceKernell16 <> 0
  then
    FreeLibrary(hInstanceKernell16);
End.

```

## IV - Le coeur du thinking : la fonction Appel32

Comme nous l'avons vu précédemment lors de la présentation de la fonction **CallProc32W**, nous devons trouver un subterfuge pour contourner l'impossibilité de prototyper en Pascal une fonction avec un nombre variable de paramètres. Nous allons donc déposer nous-même les paramètres sur la pile, comme le ferait un compilateur C, et appeler la fonction CallProc32W comme si de rien n'était.

Le nombre de paramètres de CallProc32W est limité à **32**. Nous allons donc créer une structure de 32 enregistrements :

```
Type rCallProcParam = Record
    Valeur : LongInt;      (* DWord ou adresse *)
    Traduire : Boolean;   (* Indique s'il s'agit d'une adresse 16:16 à convertir
en adresse linéaire *)
end;
tCallProcParams = Array [0..31] of rCallProcParam;

Var CallProcParams : tCallProcParams;
```

Le champ **Valeur** représente soit une valeur entière de 32 bits, soit une adresse 16:16 à convertir en adresse linéaire. C'est le champ **Traduire** qui permet de différencier les deux possibilités et qui permet de construire le masque de bits **fAddressConvert** passé comme paramètre à CallProc32W.

La convention d'appel de CallProc32W étant **\_PASCAL**, les paramètres sont déposés sur la pile dans l'ordre de leur passage à la fonction 32 bits :

```
Function APPEL32 (NbParams : LongInt; AdresseProc : Pointer) : LongInt;
Var i : LongInt;      (* Indice *)
    Masque : LongInt; (* Masque de traduction des paramètres *)
Begin
    Masque := $00000000;
    for i := 0 to (NbParams - 1) do
    begin
        if CallProcParams[i].Traduire
        then
            Masque := Masque or ($00000001 shl i);
        CallParam := CallProcParams[i].Valeur;
        asm
            push word ptr [CallParam + 2]
            push word ptr [CallParam]
        end;
    end;
    APPEL32 := CallProc32W(AdresseProc, Masque, NbParams);
End;
```

Les deux paramètres de la fonction Appel32 sont :

**NbParams** : le nombre de paramètres à déposer sur la pile;

**AdresseProc** : l'adresse de la fonction 32 bits à exécuter.

Valeur retournée par la fonction :

La valeur retournée par la fonction 32 bits appelée.

Si l'on regarde attentivement le code, on remarque que chaque paramètre à déposer est d'abord copié dans une variable **CallParam** et qu'ensuite les deux mots qui constituent cette variable 32 bits sont déposés séparément sur la pile.

```
Var CallParam : LongInt;
```

CallParam est déclarée comme **variable globale**, autrement dit dans le segment DATA. Si elle était déclarée comme variable locale de la fonction Appel32 (ce qui pourrait sembler logique), elle serait stockée dans le segment STACK et il serait plus difficile, sans inclure trop de code Assembleur, d'aller rechercher ses deux double-mots sur la pile pour les y redéposer ensuite. Mais, si le coeur vous en dit, rien ne vous empêche de procéder de la sorte !

Vous pouvez également constater que, au fur et à mesure du dépôt des paramètres sur la pile, le masque de traduction est construit. Le principe est simple : si le 6ème paramètre est une adresse 16:16 à traduire en adresse linéaire, le 6ème bit du masque est mis à 1.

## V - Un exemple simple : la fonction GetVersion

Nous allons réaliser un programme "hybride" 16/32 bits, c'est-à-dire un programme 16 bits qui, lorsqu'il tournera sur un OS 32 bits, fera appel à la fonction 32 bits GetVersion ou, sinon, fera appel à la fonction 16 bits.

La fonction GetVersion est très simple puisqu'elle ne demande aucun paramètre en entrée :

### Function GetVersion : LongInt;

Résumons ce que le programme fait :

- 1 Il charge les adresses des fonctions de la couche WOW
- 2 Il essaie d'ouvrir le kernel 32 bits et de charger l'adresse de la fonction GetVersion 32 bits
- 3 Si cela réussit, il appelle cette fonction à l'aide de la fonction APPEL32
- 4 Si, par contre, une des étapes précédentes a échoué, le programme appelle la fonction GetVersion 16 bits

```

Program GETVER;

{ Appel de la fonction GetVersion de l'API 32 bits à l'aide de la technique du thinking générique.
  Code directement compilable par Borland Pascal for Windows 7.0 }

{ ----- Directives au compilateur ----- }

Uses WINCRT,           (* Console *)
     WINTYPES, WINPROCS; (* API 16 bits *)

Const (* Versions de Windows *)
     os_Inconnu = 0;
     os_Win_20 = 1;
     os_Win_30 = 2;
     os_Win_31 = 3;
     os_Win_32s = 4;
     os_Win_NT35 = 5;
     os_Win_9x = 6;
     os_Win_NT40 = 7;
     os_Win_2000_XP_2003 = 8;

{ ----- Types de données ----- }

Type (* Tableau de 32 paramètres à déposer sur la pile par la fonction APPEL32 *)
     rCallProcParam = Record
         Valeur : LongInt;      (* DWord ou adresse *)
         Traduire : Boolean;    (* Indique s'il s'agit d'une adresse 16:16 à traduire *)
     end;
     tCallProcParams = Array [0..31] of rCallProcParam;

{ ----- Constantes typées ----- }

Const (* Handles des kernels 16 et 32 bits *)
     hInstanceKernell16 : tHandle = 0;
     hInstanceKernel32 : LongInt = 0;

     (* Adresses des fonctions de la couche WOW *)
     LoadLibraryEx32W : Function (lpszLibFile : pChar; hFile, dwFlags : LongInt) : LongInt = Nil;
     FreeLibrary32W : Function (hInst : LongInt) : LongBool = Nil;
     GetProcAddress32W : Function (hModule : LongInt; lpszProc : pChar) : Pointer = Nil;
     GetVDMPointer32W : Function (lpAddress : Pointer; fMode : Word) : Pointer = Nil;
     CallProc32W : Function (lpProcAddress : Pointer; fAddressConvert, nParams : LongInt) : LongInt
 = Nil;

     (* Adresse de la fonction 32 bits GetVersion *)
     W32GetVersion : Function : LongInt = Nil;

     (* Version de Windows obtenue par le programme *)
     OSVersion : Byte = os_Inconnu;

```

```

{ ----- Variables globales ----- }

Var (* Paramètres de CallProc32W à déposer sur la pile *)
    CallProcParams : tCallProcParams;
    CallParam : LongInt;

    (* Version de Windows renvoyée par la fonction GetVersion *)
    Version : LongInt;

{ ----- Fonction de dépôt des paramètres sur la pile et exécution de la fonction CallProc32W
----- }

Function APPEL32 (NbParams : LongInt; AdresseProc : Pointer) : LongInt;
Var i : LongInt;      (* Indice *)
    Masque : LongInt; (* Masque de traduction des paramètres *)
Begin
    Masque := $00000000;
    for i := 0 to (NbParams - 1) do
        begin
            if CallProcParams[i].Traduire
            then
                Masque := Masque or ($00000001 shl i);
            CallParam := CallProcParams[i].Valeur;
            asm
                push word ptr [CallParam + 2]
                push word ptr [CallParam]
            end;
        end;
    APPEL32 := CallProc32W(AdresseProc,Masque,NbParams);
End;

(* ----- Programme principal ----- *)

BEGIN
    (* Chargement des adresses des fonctions de la couche WOW *)
    hInstanceKernel16 := LoadLibrary('Kernel');
    if hInstanceKernel16 >= 32
    then
        begin
            @LoadLibraryEx32W := GetProcAddress(hInstanceKernel16,'LoadLibraryEx32W');
            @FreeLibrary32W := GetProcAddress(hInstanceKernel16,'FreeLibrary32W');
            @GetProcAddress32W := GetProcAddress(hInstanceKernel16,'GetProcAddress32W');
            @CallProc32W := GetProcAddress(hInstanceKernel16,'CallProc32W');
            if @LoadLibraryEx32W <> Nil
            then
                begin
                    (* Chargement de l'adresse de la fonction 32 bits GetVersion *)
                    hInstanceKernel32 := LoadLibraryEx32W('Kernel32',0,0);
                    if hInstanceKernel32 <> 0
                    then (* On peut déjà dire qu'il s'agit d'un OS 32 bits *)
                        begin
                            @W32GetVersion := GetProcAddress32W(hInstanceKernel32,'GetVersion');
                            if @W32GetVersion <> Nil
                            then
                                begin
                                    Version := APPEL32(0,@W32GetVersion);
                                    if Version <> 0
                                    then
                                        begin
                                            case Lo(LoWord(Version)) of
                                                3 : if (Version and $80000000) = 0
                                                    then
                                                        OSVersion := os_Win_NT35
                                                    else
                                                        OSVersion := os_Win_32s;
                                                4 : if (Version and $80000000) = 0
                                                    then
                                                        OSVersion := os_Win_NT40
                                                    else
                                                        OSVersion := os_Win_9x;
                                                5 : OSVersion := os_Win_2000_XP_2003;
                                            end;
                                        end;
                                    end;
                                end;
                            end;
                        end;
                    end;
                end;
            if OSVersion = os_Inconnu

```

```
then (* Toujours pas de résultat : essai de l'API 16 bits *)
begin
  Version := GetVersion;
  if Lo(LoWord(Version)) = 3
  then
    begin
      case Hi(LoWord(Version)) of
        0 : OSVersion := os_Win_30;
        10 : OSVersion := os_Win_31;
      end;
    end
  else
    if Lo(LoWord(Version)) = 2
    then
      OSVersion := os_Win_20;
    end;
  Write('Version de Windows : ');
  case OSVersion of
    os_Inconnu : WriteLn('Impossible de la déterminer');
    os_Win_20 : WriteLn('2.0x');
    os_Win_30 : WriteLn('3.0x');
    os_Win_31 : WriteLn('3.1x');
    os_Win_32s : WriteLn('Win32s');
    os_Win_NT35 : WriteLn('NT 3.5x');
    os_Win_9x : WriteLn('95 / 98 / Me');
    os_Win_NT40 : WriteLn('NT 4.0');
    os_Win_2000_XP_2003 : WriteLn('2000 / XP / 2003 server');
  end;
  ReadLn;
  (* Fermeture des librairies ouvertes *)
  if (@FreeLibrary32W <> Nil) and (hInstanceKernel32 <> 0)
  then
    FreeLibrary32W(hInstanceKernel32);
  if hInstanceKernel16 <> 0
  then
    FreeLibrary(hInstanceKernel16);
  DoneWinCRT;
End.
```

## VI - Conversion d'un handle de fenêtre : la fonction Hwnd32

Beaucoup de routines de l'API Windows nécessitent comme paramètre un handle de fenêtre. Si nous transmettons un handle de fenêtre 16 bits à une fonction de l'API 32 bits, celle-ci ne s'exécutera pas convenablement. Nous allons donc réaliser une petite fonction de conversion de handle de fenêtre 16 bits en handle 32 bits.

L'astuce sera d'utiliser le couple de fonctions **SetCapture (16 bits) / GetCapture (32 bits)**. La fonction GetCapture est implémentée dans la librairie **User32**.

Nous ne reprendrons pas ici l'intégralité du code : cette fonction peut très bien être implémentée directement dans le source du 1er exemple (la fonction GetVersion).

```

Const hInstanceUser32 : LongInt = 0;
      W32GetCapture : Function : LongInt = Nil; (* Adresse de la routine 32 bits *)

Function Hwnd32 (Handle16 : Hwnd) : LongInt;
Var AncienHandle16 : Hwnd; (* Ancien handle 16 bits de la fenêtre *)
Begin
  Hwnd32 := 0;
  hInstanceUser32 := LoadLibraryEx32W('User32',0,0);
  if hInstanceUser32 <> 0
  then
    begin
      @W32GetCapture := GetProcAddress32W(hInstanceUser32,'GetCapture');
      if @W32GetCapture <> Nil
      then
        begin
          (* Capture de la souris en 16 bits *)
          AncienHandle16 := SetCapture(Handle16);
          (* Récupération du handle en 32 bits *)
          FillChar(CallProcParams,SizeOf(CallProcParams),0);
          Hwnd32 := APPEL32(0,@W32GetCapture);
          (* Restauration de la capture initiale en 16 bits *)
          if AncienHandle16 <> 0
          then
            SetCapture(AncienHandle16)
          else
            ReleaseCapture;
          end;
          (* Libération de la librairie User32 *)
          if (@FreeLibrary32W <> Nil) and (hInstanceKernel32 <> 0)
          then
            FreeLibrary32W(hInstanceUser32);
          end;
        end;
      end;
    end;
End;

```

## VII - Un exemple plus complexe : la fonction CreateFile

Toujours dans le cadre d'un programme "hybride" 16/32 bits, nous allons réaliser une fonction d'ouverture de fichier.

Nous considérerons que la version de Windows aura déjà été testée au moyen de la fonction GetVersion du 1er exemple.

Dans l'API 16 bits, la fonction d'ouverture de fichier est `_IOpen` :

**Function `_IOpen` (PathName : pChar; ReadWrite : Integer) : Integer;**

Dans l'API 32 bits, c'est `CreateFile` :

**Function `CreateFile` (lpFileName : pChar, dwDesiredAccess, dwShareMode : LongInt, lpSecurityAttributes : LPSECURITY\_ATTRIBUTES, dwCreationDisposition, dwFlagsAndAttributes : LongInt, hTemplateFile : tHandle) : tHandle;**

Nous nous trouvons ici en présence d'une série de paramètres à déposer sur la pile; l'un d'entre eux, `lpFileName`, est un pointeur.

```
Function WIN32_OPENFILE (NomFichier : pChar; var hFichier : LongInt) : Boolean;
Begin
  if OSVersion >= os_Win_NT40
  then (* Uniquement pour la famille NT *)
  begin
    if @W32CreateFile = Nil
    then
      begin
        if hInstanceKernel32 = 0
        then
          hInstanceKernel32 := LoadLibraryEx32W('Kernel32',0,0);
        if hInstanceKernel32 <> 0
        then
          @W32CreateFile := GetProcAddress32W(hInstanceKernel32,'CreateFileA');
        end;
      end;
    if @W32CreateFile <> Nil
    then
      begin
        FillChar(CallProcParams,SizeOf(CallProcParams),0);
        (* lpFileName *)
        CallProcParams[0].Valeur := LongInt(NomFichier);
        CallProcParams[0].Traduire := True;
        (* dwDesiredAccess *)
        CallProcParams[1].Valeur := Generic_Read;
        CallProcParams[1].Traduire := False;
        (* dwShareMode *)
        CallProcParams[2].Valeur := File_Share_None;
        CallProcParams[2].Traduire := False;
        (* lpSecurityAttributes *)
        CallProcParams[3].Valeur := 0;
        CallProcParams[3].Traduire := False;
        (* dwCreationDisposition *)
        CallProcParams[4].Valeur := Open_Existing;
        CallProcParams[4].Traduire := False;
        (* dwFlagsAndAttributes *)
        CallProcParams[5].Valeur := File_Attribute_Archive;
      end;
    end;
  end;
End;
```

```

        CallProcParams[5].Traduire := False;
        (* hTemplateFile *)
        CallProcParams[6].Valeur := 0;
        CallProcParams[6].Traduire := False;
        (* APPEL *)
        hFichier := APPEL32(7,@W32CreateFile);
    end
else
    hFichier := -1;
end
else (* OS non NT : appel de l'API 16 bits *)
    hFichier := _lOpen(NomFichier,of_ReadWrite or of_Share_Exclusive);
WIN32_OPENFILE := (hFichier <> -1);
End;

```

Nous affectons les paramètres un à un au tableau **CallProcParams**. Le tout premier (numéroté 0) est l'adresse du nom du fichier à ouvrir. Comme il s'agit d'une adresse, nous demanderons à la fonction CallProc32W de la traduire en adresse 32 bits, en donnant la valeur **True** au champ **Traduire**.

Nous n'utilisons pas d'attributs de sécurité dans notre exemple; nous affectons donc 0 au paramètre lpSecurityAttributes. La valeur 0 correspond à un pointeur NULL, le champ Traduire reste à False.

Tous les autres paramètres ne sont pas des adresses et donc le champ Traduire restera à False.

### Portabilité

Si vous observez le test initial du n° de version de Windows, vous constatez que la fonction CreateFile n'est utilisée que sous Windows NT, 2000, XP et 2003 Server. Pour toutes les autres versions de Windows, donc tant les versions 16 bits que la famille Windows 95, 98 et Me, c'est la fonction \_lOpen qui est appelée.

Pour les versions 16 bits, il est bien entendu logique d'utiliser l'API 16 bits. Mais pour la famille 95, 98 et Me, l'appel de CreateFile provoque une instabilité du système voire un plantage à court terme. Heureusement, la version 32 bits de \_lOpen gère très bien les noms longs de fichiers !

Le chapitre IX est d'ailleurs consacré à la portabilité de la technique du thinking générique.

## VIII - Un dernier exemple avec une structure : la fonction GetOpenFileName

Si vous avez été attentif(ve) jusqu'ici, vous aurez remarqué que nous n'avons pas encore utilisé la fonction **GetVDMPointer32W**. A quoi peut bien servir cette fonction de traduction d'adresses 16:16 en adresses 32 bits puisque nous pouvons déjà demander à la fonction CallProc32W de le faire ? La réponse sera illustrée dans l'exemple ci-dessous : on utilisera la fonction GetVDMPointer32W **pour traduire des adresses contenues dans des structures**.

En effet, CallProc32W peut traduire des adresses qui lui sont transmises comme paramètres. Mais, dans le cas d'adresses contenues dans une structure, dont l'adresse est elle-même transmise comme paramètre à CallProc32W, c'est impossible.

L'exemple que nous allons réaliser, toujours hybride 16/32 bits, utilisera la fonction GetOpenFileName :

**Function GetOpenFileName (var lpOFN : LOPENFILENAME) : Boolean;**

Cette fonction permet d'appeler la boîte de dialogue standard Windows d'ouverture de fichier.

Mis à part le handle de la fenêtre parent, hWndOwner, dont la taille passe de 16 bits à 32 bits, la structure est absolument identique dans l'API 16 bits et dans l'API 32 bits. C'est bien commode pour nous car nous allons pouvoir utiliser la structure 16 bits comme paramètre de notre fonction hybride et aller recopier tous les champs dans la structure 32 bits, moyennant bien sûr quelques conversions !

```

Type tW32OpenFileName = Record
    lStructSize : LongInt;          (* Taille de la structure *)
    hWndOwner : LongInt;           (* Handle de la fenêtre parent *)
    hInstance : LongInt;          (* Handle du modèle de dialogue *)
    lpstrFilter : pChar;           (* Adresse du filtre d'extension *)
    lpstrCustomFilter : pChar;     (* Adresse du filtre personnalisé *)
    nMaxCustFilter : LongInt;      (* Taille de lpstrCustomFilter *)
    nFilterIndex : LongInt;        (* Index dans le buffer lpstrFilter *)
    lpstrFile : pChar;             (* Buffer d'entrée/sortie du chemin complet *)
*)
    nMaxFile : LongInt;            (* Taille de lpstrFile *)
    lpstrFileTitle : pChar;        (* Adresse du buffer recevant le nom de
fichier *)
    nMaxFileTitle : LongInt;       (* Taille de lpstrFileTitle *)
    lpstrInitialDir : pChar;       (* Adresse du répertoire de départ *)
    lpstrTitle : pChar;            (* Titre du dialogue *)
    Flags : LongInt;               (* Options *)
    nFileOffset : Word;            (* Offset du nom de fichier dans lpstrFile *)
    nFileExtension : Word;         (* Offset de l'extension dans lpstrFile *)
    lpstrDefExt : pChar;           (* Extension par défaut *)
    lCustData : LongInt;           (* Bloc de données de la routine de hooking *)
*)
    lpfnHook : Pointer;            (* Adresse de la routine de hooking *)
    lpTemplateName : pChar;       (* Nom du modèle de dialogue *)
end;

Function WIN32_GETOPENFILENAME (var OFN16 : tOpenFileName) : Boolean;
Var OFN32 : tW32OpenFileName;    (* Structure d'interface avec le dialogue Windows *)
Begin
    if OSVersion >= os_Win_NT40
    then (* Famille NT *)
    begin
        if @W32GetOpenFileName = Nil
        then
        begin

```

```

        if hInstanceComDlg32 = 0
        then
            hInstanceComDlg32 := LoadLibraryEx32W('ComDlg32',0,0);
        if hInstanceComDlg32 <> 0
        then
            @W32GetOpenFileName := GetProcAddress32W(hInstanceComDlg32,'GetOpenFileNameA');
    end;
    if @W32GetOpenFileName <> Nil
    then
        begin
            FillChar(OFN32,SizeOf(OFN32),0);
            (* lStructSize *)
            OFN32.lStructSize := SizeOf(tW32OpenFileName);
            (* hWndOwner *)
            OFN32.hWndOwner := HWND32(OFN16.hWndOwner);
            (* hInstance *)
            OFN32.hInstance := OFN16.hInstance;
            (* lpstrFilter *)
            if OFN16.lpstrFilter <> Nil
            then
                OFN32.lpstrFilter :=
                    GetVDMPointer32W(OFN16.lpstrFilter,StrLen(OFN16.lpstrFilter));
                (* lpstrCustomFilter *)
                if OFN16.lpstrCustomFilter <> Nil
                then
                    OFN32.lpstrCustomFilter :=
                        GetVDMPointer32W(OFN16.lpstrCustomFilter,OFN16.nMaxCustFilter);
                    (* nMaxCustFilter *)
                    OFN32.nMaxCustFilter := OFN16.nMaxCustFilter;
                    (* nFilterIndex *)
                    OFN32.nFilterIndex := OFN16.nFilterIndex;
                    (* lpstrFile *)
                    if OFN16.lpstrFile <> Nil
                    then
                        OFN32.lpstrFile := GetVDMPointer32W(OFN16.lpstrFile,OFN16.nMaxFile);
                        (* nMaxFile *)
                        OFN32.nMaxFile := OFN16.nMaxFile;
                        (* lpstrFileTitle *)
                        if OFN16.lpstrFileTitle <> Nil
                        then
                            OFN32.lpstrFileTitle :=
                                GetVDMPointer32W(OFN16.lpstrFileTitle,OFN16.nMaxFileTitle);
                            (* nMaxFileTitle *)
                            OFN32.nMaxFileTitle := OFN16.nMaxFileTitle;
                            (* lpstrInitialDir *)
                            if OFN16.lpstrInitialDir <> Nil
                            then
                                OFN32.lpstrInitialDir :=
                                    GetVDMPointer32W(OFN16.lpstrInitialDir,StrLen(OFN16.lpstrInitialDir));
                                (* lpstrTitle *)
                                if OFN16.lpstrTitle <> Nil
                                then
                                    OFN32.lpstrTitle :=
                                        GetVDMPointer32W(OFN16.lpstrTitle,StrLen(OFN16.lpstrTitle));
                                    (* Flags *)
                                    OFN32.Flags := OFN16.Flags;
                                    (* nFileOffset *)
                                    OFN32.nFileOffset := OFN16.nFileOffset;
                                    (* nFileExtension *)
                                    OFN32.nFileExtension := OFN16.nFileExtension;
                                    (* lpstrDefExt *)
                                    if OFN16.lpstrDefExt <> Nil
                                    then
                                        OFN32.lpstrDefExt :=
                                            GetVDMPointer32W(OFN16.lpstrDefExt,StrLen(OFN16.lpstrDefExt));
                                        (* lCustData *)
                                        OFN32.lCustData := OFN16.lCustData;
                                        (* lpfnHook *)
                                        Move(OFN16.lpfnHook,OFN32.lpfnHook,SizeOf(OFN32.lpfnHook));
                                        if OFN32.lpfnHook <> Nil
                                        then
                                            OFN32.lpfnHook := GetVDMPointer32W(OFN32.lpfnHook,SizeOf(OFN32.lpfnHook));
                                            (* lpTemplateName *)
                                            if OFN16.lpTemplateName <> Nil
                                            then
                                                OFN32.lpTemplateName :=
                                                    GetVDMPointer32W(OFN16.lpTemplateName,StrLen(OFN16.lpTemplateName));

            FillChar(CallProcParams,SizeOf(CallProcParams),0);

```

```
        CallProcParams[0].Valeur := LongInt(@OFN32);
        CallProcParams[0].Traduire := True;
        (* APPEL *)
        WIN32_GETOPENFILENAME := APPEL32(1,@W32GetOpenFileName) <> 0;
    end
  else
    WIN32_GETOPENFILENAME := False;
  end
else (* Windows 16 bits ou 9x *)
  WIN32_GETOPENFILENAME := GetOpenFileName(tOpenFileName(OFN16));
End;
```

Analysons un peu notre exemple. La fonction `GetOpenFileName` n'a qu'un seul paramètre, dont nous transmettons l'adresse au tableau `CallProcParams`. Nous demandons la traduction de l'adresse au moyen du champ `Traduire`, comme déjà vu précédemment. `CallProc32W` va donc traduire cette adresse 16:16 en adresse 32 bits, mais pas les adresses contenues dans la structure `OFN32` ! C'est pour cette raison que nous effectuons la traduction nous-mêmes au moyen de **GetVDMPointer32W**.

Vous constatez que, comme expliqué lors de la présentation de la fonction `GetVDMPointer32W`, nous transmettons à cette dernière **la taille de la structure dont l'adresse doit être convertie**. Si nous nous en tenons à la documentation MSDN, à savoir un simple 0 ou 1, nous obtenons de beaux plantages.

Vous pouvez également voir que le handle de la fenêtre parent de la boîte de dialogue Windows est converti en handle 32 bits par la fonction `HWND32` que nous avons écrite plus haut.

## IX - Portabilité

Il est très important, lorsque l'on utilise la technique du thinking générique, de s'inquiéter de la version courante de Windows. Dans tous nos exemples, nous en avons d'ailleurs tenu compte.

Le thinking générique est une technique propre à la technologie NT mais peut également fonctionner avec les autres versions 32 bits de Windows (95, 98 et Me). Malheureusement, pas dans tous les cas ! Seule l'expérience pourra réellement vous renseigner sur la portabilité du thinking d'une fonction.

Comme je suis un garçon sympathique, voici les résultats que j'ai obtenus jusqu'à présent :

Fonction 32 bits	Portabilité Windows 95/98/Me ?
GetVersion	Oui
GetCapture	Oui
GetLastError	Oui
CreateFile	Non - utiliser _IOpen ou _ICreat
ReadFile	Non - utiliser _IRead
WriteFile	Non - utiliser _IWrite
SetFilePointer	Non - utiliser _ISeek
CloseHandle	Non - utiliser _IClose
GetShortPathName	Oui
GetCurrentDirectory	Oui
FindFirstFile	Oui
FindNextFile	Oui
FindClose	Oui
GetOpenFileName	Non - utiliser version 16 bits
GetSaveFileName	Non - utiliser version 16 bits
ChooseColor	Non - utiliser version 16 bits
CommDlgExtendedError	Non - utiliser version 16 bits

## X - L'unité Win32

Le source complet de l'unité **Win32**, implémentant plusieurs fonctions hybrides 16/32 bits et utilisant la technique du thinking générique est téléchargeable dans les sources de la section Pascal de Developpez.com :

[http://pascal.developpez.com/sources/fichiers/bpw/alcatiz/win32\\_100.zip](http://pascal.developpez.com/sources/fichiers/bpw/alcatiz/win32_100.zip)