

Les appels d'APIs en VB.Net

Quand on commence à faire de la programmation proche du système, donc avancée, on a souvent besoin d'API : fonctions proposées par le système d'exploitation. Ces fonctions sont stockées dans de bibliothèque de liens : les DLLs.

Il en existe trois catégories principales :

- User32.dll : toutes le fonctions en rapport avec les fenêtres
- Gdi32.dll : tout ce qui est en rapport avec les images
- Kernel32.dll : tout ce qui a rapport avec les mécanismes internes : informations système, système de fichier, mutex, sémaphores.

I. Déclaration

Les déclarations d'APIs se font comme suit :

```
{Private|Public} Declare [{Ansi|Unicode}] {Function|Sub} nom_fonction [Alias "vrai_nom_fonction"] Lib "nom_dll.dll" (Param1 As Type1, Param2 As Type2,... ParamN As TypeN) [As TypeRetour]
```

{Private|Public} : portée de la déclaration (privée ou publique)

{Ansi|Unicode} : indique si les chaînes sont traitées comme ANSI (1 octet) ou UNICODE (2 octets)

{Function|Sub} : fonction ou procédure, une API Windows est rarement une procédure (sauf CopyMemory)

[Alias "vrai_nom_fonction"] : permet de renommer la fonction si son vrai nom est, par exemple, trop long ou trop compliqué ([_fct@8](#)) ou pour importer des fonctions exportées par Index (dans ce cas l'alias sera *#index*).

Penser à regarder si une classe faisant la même chose que l'API n'existerait pas...

1. Les paramètres

Il existe deux types de paramètres : les paramètres par valeur et les paramètres par référence (pointeur).

Un paramètre par référence indique que l'on passe l'adresse de la variable à la fonction pour pouvoir modifier la variable à l'intérieur. Le nom d'un tel paramètre est précédé de ByRef.

Un paramètre par valeur indique que l'on passe le contenu de la variable à la fonction. On ne peut donc pas modifier la variable à l'intérieur de la fonction. Le nom d'un tel paramètre est précédé de ByVal ou rien (ByVal par défaut).

2. Le type de retour

Si c'est une Sub, alors il n'y a pas de type de retour.

Sinon, c'est presque toujours un Integer. **Tout type de retour de plus de 4 octets devrait être passé par référence en paramètre.**

3. Et si ma fonction n'est pas dans une liste

Alors là c'est un peu plus compliqué. Mais si vous êtes là, c'est que vous avez le prototype C de votre fonction.

Pour que VB puisse utiliser une fonction C comme une API, il faut :

- **Que la convention d'appel soit STDCALL (par défaut cdecl en C)**
- **Qu'elle soit exportée avec un extern « C » ou un def file**

II. Traduction des types

1 L'attribut MarshalAs

Cette attribut se place avant le paramètre : `<MarshalAs(...)> {ByRef|ByVal} nom As Type`.

Par exemple :

```
Public Sub M1 (<MarshalAs(UnmanagedType.LPWStr)> msg As String)
```

On peut aussi l'appliquer à un member de structure :

Par exemple :

```
< MarshalAs(UnmanagedType.LPWStr)> Public msg As String
```

Pour utiliser correctement l'attribut MarshalAs, il faut ajouter la clause, en début de classe :

Import System.Runtime.InteropServices

2 Les types standards

Voici un tableau de types courants et de leurs traductions VB6 :

Type en C	Type en VB
Char	ByVal Byte (mais considéré non signé)
Short	ByVal Short
Int	ByVal Integer
Long	ByVal Integer
unsigned char	ByVal Byte
unsigned short	ByVal Short (mais considéré signé)
unsigned int	ByVal Integer (mais considéré signé)
unsigned long	ByVal Integer (mais considéré signé)
BOOL	ByVal <MarshalAs(UnmanagedType.Bool)> <i>param</i> As Boolean
bool	(<MarshalAs(UnmanagedType.I1)> <i>param</i> As Boolean
Float	ByVal Single
Double	ByVal Double
ULONGLONG, LONGLONG	ByVal Long
char*	ByVal <MarshalAs(UnmanagedType.LPStr) param As String
Short*	ByRef Short
int*	ByRef Integer
long*	ByRef Integer
unsigned char*	ByRef Byte
unsigned short*	ByRef Integer (mais considéré signé)
unsigned int*	ByRef Long (mais considéré signé)
unsigned long*	ByRef Long (mais considéré signé)
BOOL*	ByRef <MarshalAs(UnmanagedType.Bool)> <i>param</i> As Boolean
float*	ByRef Single
double*	ByRef Double

ULONGLONG*, LONGLONG*	ByRef Long
BSTR	ByVal <MarshalAs(UnmanagedType.BStr)>param As String
BSTR*	ByRef <MarshalAs(UnmanagedType.BStr)>param As String
TCHAR*	ByVal <MarshalAs(UnmanagedType.LPCTSTR)>param As String
IUnknown*	ByVal <MarshalAs(UnmanagedType.IUnknown)> param As Object
IUnknown**	ByRef <MarshalAs(UnmanagedType.IUnknown)> param As Object
Interface*	ByVal {Object Interface}
Interface**	ByRef {Object Interface}
Structure	ByVal Structure
Structure*	ByRef Structure (avec Attributs éventuellement pour les tableaux)
Tableau C	Voir plus bas
SAFEARRAY(type)*	ByRef <MarshalAs(UnmanagedType.SafeArray, SafeArraySubType:=Type) param As Type
Currency de COM	<MarshalAs(UnmanagedType.Currency)> param As Decimal
Type param[taille]	<MarshalAs(UnmanagedType.LPArray, SizeConst=taille)> param() As Type
char param[taille]	<MarshalAs(UnmanagedType.LPArray, SizeConst=taille)> param() As String
Hxxx	Integer (représente un handle)

Pour les autres types, il est nécessaire de rajouter des attributs aux paramètres

1 Le type Any de VB6

<MarshalAs(UnmanagedType.AsAny)> param As Object

Mais il est préférable de déclarer plusieurs fonctions Declare avec les différents types possibles pour le paramètre.

2 Les types pointeurs

Il existe un type pointeur en VB.Net : System.IntPtr.

Note : Les fonctions VarPtr, ObjPtr et StrPtr n'existent plus et je déconseille d'essayer de les réécrire car les versions possibles ne marchent pas aussi bien.

Voir *Travailler avec les pointeurs*

3 Les types Structures

1 Attribut StructLayout

On peut changer le type de chaînes de caractères, l'alignement et la taille de la structure. :

<StructLayout(LayoutKind.Sequential, CharSet :=charset, Pack :=alignement, Size:=taille)>

Private Structure nom

....

End Structure

charset (facultatif) : CharSet.Unicode ou CharSet.Ansi

alignement (facultatif) : aligner tous les 1, 2, 4, 8, 16, 32, 64 ou 128 octets

taille (très facultatif): taille absolue de la structure en octet

Note sur le contenu des structures :

- Les chaînes de taille fixe dans les structures (CHAR/WCHAR/TCHAR nom[taille]) se traduisent en
<MarshalAs(UnmanagedType.ByValTStr, SizeConst:=taille)>param() As String

Le type de charset (char/ANSI ou wchar/UNICODE) est défini dans l'attribut StructLayout de la structure.

- Les chaînes char*, WCHAR*, TCHAR* sont remplacées par
<MarshalAs(UnmanagedType.LPStr)> *membre* As String,
<MarshalAs(UnmanagedType.LPWStr)> *membre* As String et
<MarshalAs(UnmanagedType.LPTStr)> *membre* As String.
- Les types pointeurs sont System.IntPtr. *Voir classe Marshal plus bas*
- Les tableaux de taille fixe *type nom[taille]* se traduisent par
<MarshalAs(UnmanagedType.ByValArray, SizeConst:=*taille*)> *param()* As *Type*
- Pour les autres types :

Type C	Type VB
char/BYTE	Byte (interprété non signé)
short/USHORT/WORD	Short
int, long/DWORD/ULONG	Integer
ULONGLONG	Long
Float	Single
double	Double
BOOL/BOOLEAN	Integer
bool	Byte

4 Passer un tableau type C : de type simples ou de structures

Pour passer un tableau de type C, on a UNE SEULE solution :

- <MarshalAs(UnmanagedType.LPArray)>ByRef *Type_des_cases_du_tableau* : dans ce cas, on passe la première case du tableau (alloué), en général, il y a un paramètre pour donner la taille du tableau que l'on passe

Par exemple :

```
Dim t(10) As Long
Res = Fct(t(0),10)
```

5 Passer un tampon chaîne type C

Pour passer un buffer chaîne C : char/wchar* buffer (int taille, suit en général), on utilise un ByVal String. **Il faut impérativement remplir la chaîne avec des caractères avant de la passer à la fonction.**

Par exemple :

```
Dim s as New String(20)
res = Fct(s,20)
```

On peut aussi utiliser la classe StringBuilder de la même façon.

6 Les pointeurs de fonctions de rappel (callback)

On peut utiliser le type « Delegate ». Pour cela :

- On écrit le **prototype** Delegate de la fonction callback : {Private|Public|Protected} Delegate Function *nomDelegate(params) As retour*
- Ecrire la déclaration de la fonction qui prend en paramètre le pointeur de fonction avec comme type pour ce paramètre pointeur de fonction *nomDelegate*.
- Créer une fonctions ayant le même prototype que le delegate sans Delegate :

```
{Private|Public|Protected} Function nom(params) As retour
```

...

```
End Function
```
- Lors de l'appel de la fonction déclarée, passer au paramètre pointeur de fonction : AddressOf *nom*

Note : *nomDelegate* n'a rien d'obligatoire : vous pouvez donner un autre nom au type delegate. Vous pouvez aussi avoir plusieurs fonctions du même prototype que le delegate.

7 Travail avec les pointeurs : l'objet Marshal

1 Le type IntPtr

Un IntPtr se construit avec un Integer ou avec un Long. On peut le convertir en Integer et Long. Pour initialiser le pointeur à NULL, on écrit : *ptr* = IntPtr.Zero.

2 Lecture et écriture direct dans les données d'un pointeur

Type peut prendre différentes valeurs : Byte, Int16, Int32, Int64 et IntPtr.

Pour lire directement en mémoire, on utilise les fonctions *Marshal.ReadType*. Il en existe deux (en réalité trois) versions :

```
Marshal.ReadType(ByVal pointeur As IntPtr) As Type
```

```
Marshal.ReadType(ByVal pointeur As IntPtr, ByVal offset As Integer) As Type
```

La première version renvoie la donnée pointée par *pointeur*. On l'utilise pour les données seules.

La seconde version renvoie la donnée pointée par *pointeur[offset]*. On l'utilise pour les données type tableau C.

Exemple :

```
Dim ptr As IntPtr = 'adresse trouvée quelque part...
```

```
Dim b As Byte = Marshal.ReadByte(ptr)
```

Pour écrire directement en mémoire, on utilise les fonctions *Marshal.WriteType*. Il en existe deux (en réalité trois) versions :

```
Marshal.WriteType(ByVal pointeur As IntPtr, ByVal valeur As Type) As Type
```

```
Marshal.WriteType(ByVal pointeur As IntPtr, ByVal offset As Integer, ByVal valeur As Type) As Type
```

La première version écrit *valeur* dans la zone pointée par *pointeur*. On l'utilise pour les données seules.

La seconde version écrit *valeur* dans la zone pointée par *pointeur[offset]*. On l'utilise pour les données type tableau C.

Exemple :

```
Dim ptr As IntPtr = 'adresse trouvée quelque part...
```

```
Dim b As Byte = Marshal.WriteByte(ptr,19)
```

8 Copier les données d'un pointeur vers un objet VB

1 Pour les types simples et les tableaux de types simples

La fonction *Copy* permet de copier des tableaux

- d'un pointeur vers un tableau VB

```
Overloads Public Shared Sub Copy(ByVal zone_source As IntPtr, ByVal tableau_destination() As Type, ByVal index_début_copie As Integer, ByVal nb_case_copie As Integer)
```

- d'un tableau VB vers un pointeur

```
Overloads Public Shared Sub Copy(ByVal tableau_source() As Type, ByVal index_début_copie As Integer, ByVal destination As IntPtr, ByVal nb_case_copie As Integer)
```

Pour la fonction Copy, *Type* peut prendre les valeurs suivantes : *Byte*, *Char*, *Double*, *Short*, *Integer*, *Long*, *Single*.

1 Les chaînes de caractères

Les fonctions PtrToStringAnsi, PtrToStringBSTR, PtrToStringUni permettent de copier une chaîne de caractères respectivement : ANSI (1 octet), BSTR (2 octets), UNICODE (2 octets).

Il existe deux versions de ces fonctions :

- prenant en paramètre un pointeur de type IntPtr et renvoyant un objet String contenant la chaîne entière
- prenant en paramètre un pointeur de type IntPtr et le nombre de caractère à copier (Integer) et renvoyant un objet String contenant la chaîne copiée

2 Les structures

La fonction PtrToStructure permet de copier une structure unique. Il en existe deux versions :

- **Sub PtrToStructure(ByVal ptr As IntPtr, ByVal structure As Object)**

Cette version permet de copier une structure pointée par *ptr* dans un objet *structure* existant

Exemple :

```
Dim ptr As IntPtr = 'une adresse quelconque
Dim s As New MaStructure
Marshal.PtrToStructure(ptr,s)
```

Function PtrToStructure(ByVal ptr As IntPtr, ByVal structureType As Type) As Object

Cette version alloue un nouvel objet structure de type *structureType* à partir des données *ptr*.

Exemple :

```
Dim ptr As IntPtr = 'une adresse quelconque
Dim s As MaStructure = CType(Marshal.PtrToStructure(ptr, GetType(MaStructure)),
```

MaStructure)

3 Les tableaux de structures

On peut faire une boucle en incrémentant le pointeur de la taille d'une structure : *ptr = New IntPtr(ptr.ToInt32 + taille)*.

2 Transformer un objet VB en pointeur

1 VarPtr, StrPtr et ObjPtr : GCHandle

Note : ces fonctions sont à utiliser en dernier recourt : penser aux fonctions de l'objet Marshal avant de faire un CopyMemory VarPtr. L'objet Marshal donne du code plus sûr.

VarPtr : déconseillée

La classe GCHandle permet de réécrire ces fonctions très utiles de VB6.

```
Public Function VarPtr(ByVal obj As Object) As IntPtr
    Dim g As GCHandle = GCHandle.Alloc(obj,GCHandleType.Pinned)
    Dim ptr As IntPtr = g.AddrOfPinnedObject()
    g.Free()
    Return ptr
```

End Function

Une fois que l'on a le pointeur vers la variable *destination* et le pointeur *source* (ou inversement), on peut faire une copie de mémoire avec CopyMemory comme avec VB6.

ATTENTION : tous les types (et surtout les structures avec ByValArray) ne sont pas Pinnable. Il se peut donc que cette fonction échoue.

Pour avoir les mêmes fonctionnalités :

```
Dim ptr As IntPtr = Marshal.AllocHGlobal(taille)  
'on appelle la fonction avec un paramètre ByVal IntPtr  
Dim attr As type = CType(Marshal.PtrToStructure(ptr, GetType(type)), type)  
Marshal.FreeHGlobal(ptr)
```

On peut aussi convertir un GCHandle (créé avec GCHandle.Alloc) en IntPtr et inversement avec GCHandle.op_Explicit.

StrPtr

Note : il est préférable d'utiliser les attributs de paramètres, Ansi ou Unicode avec le Declare et ByVal String.

Il existe les fonctions *StringToHGlobalAnsi* et *StringToHGlobalUni*. Elles permettent respectivement de renvoyer un pointeur vers une chaîne ANSI et UNICODE en mémoire. Elle prennent en paramètre un objet de type String :

```
Function StringToHGlobalType(ByVal chaîne As String) As IntPtr
```

Il est nécessaire de libérer la zone pointée (précédente) par le pointeur renvoyé afin de ne pas faire de fuite mémoire :

```
Sub FreeHGlobal(ByVal pointeur_vers_chaine As IntPtr)
```

ObjPtr

En avez vous vraiment besoin....

3 Les « autres » méthodes de l'objet Marshal

Marshaling avancé	GetManagedThunkForUnmanagedMethodPtr, GetUnmanagedThunkForManagedMethodPtr, NumParamBytes
Fonction de bibliothèque COM	BindToMoniker, GetActiveObject
Utilitaires COM	ChangeWrapperHandleStrength, CreateWrapperOfType, GetComObjectData, GetComSlotForMethodInfo, GetEndComSlot, GetMethodInfoForComSlot, GetStartComSlot, ReleaseComObject, SetComObjectData
Transformation des données	Managées en non managées : Copy, GetComInterfaceForObject, GetIDispatchForObject, GetIUnknownForObject, StringToBSTR, StringToCoTaskMemAnsi, StringToCoTaskMemAuto,

StringToCoTaskMemUni, StringToHGlobalAnsi,
StringToHGlobalAuto, StringToHGlobalUni,
StructureToPtr,
UnsafeAddrOfPinnedArrayElement

Non managées en managées : Copy,
GetObjectForIUnknown,
GetObjectForNativeVariant,
GetObjectsForNativeVariants,
GetTypedObjectForIUnknown,
GetTypeForTypeInfo, PtrToStringAnsi,
PtrToStringAuto, PtrToStringBSTR,
PtrToStringUni

Propriétés : SystemDefaultCharSize,
SystemMaxDBCSCharSize

Lecture et écriture directes

ReadByte, ReadInt16, ReadInt32, ReadInt64,
ReadIntPtr, WriteByte, WriteInt16, WriteInt32,
WriteInt64, WriteIntPtr

Gestion des erreurs

COM : GetHRForException,
ThrowExceptionForHR

Win32 : GetLastError, GetExceptionCode,
GetExceptionPointers

Utilitaires d'hébergement

Les deux : GetHRForLastWin32Error
GetThreadFromFiberCookie

Iunknown

AddRef, QueryInterface, Release

Gestion de la mémoire

COM : AllocCoTaskMem, ReAllocCoTaskMem,
FreeCoTaskMem, FreeBSTR

Win32 : AllocHGlobal, ReAllocHGlobal,
FreeHGlobal

Utilitaires d'appel de plate-forme

Les deux : DestroyStructure

Examen de la structure

Prelink, PrelinkAll, GetHINSTANCE

Informations de type

OffsetOf, SizeOf

GenerateGuidForType, GenerateProgIdForType,
GetTypeInfoName, GetTypeLibGuid,
GetTypeLibGuidForAssembly, GetTypeLibLcid,
GetTypeLibName, IsComObject,
IsTypeVisibleFromCom