

Mettre de l'ASM dans du VB

Voilà encore une utilisation des objets légers (voir mes autres tutoriaux) : appeler du code ASM depuis VB...

Pour faire simple, nous allons nous servir des codes sur les pointeurs de fonctions et plus précisément du code sur les fonctions stdcall.

- Nous mettrons notre code dans une variable globale à un module de type structure afin de ne pas avoir de problème de libération de ressource anticipée (causant un crash).
- Ensuite dans une fonction d'initialisation d'un type *<interface>* dont la seule méthode colle aux paramètres de la fonction ASM
 - nous appellerons la fonction de construction d'un objet pour pointeur stdcall
 - Nous l'affecterons à notre fonction

Règles de codage

- Déjà, et c'est presque inutile de le dire, on cherche à optimiser
- Le code sera rédigé de façon de respecter la convention stdcall.
 - La fonction prend **tous ses paramètres sur la pile**
 - **Elle les dépile avec l'instruction RET**
 - L'instruction **RET doit être la dernière** (sauf remplissage avec NOP ou INT 3)

Il faudra le compiler pour avoir le code machine des instructions de la fonction. Je rappelle que pour des besoins d'alignement (ajout de données dans le code), on doit ajouter des NOP entre deux instructions. Une instruction n'est pas sécable, elle doit rester continue.

Comment obtenir les instructions binaires d'un code (ou C sans appels externes) ?

Sous Visual C++ 6

D'abord, il nous faut un projet : menu *File|New* puis sélectionner *Win32 Application*, donner un nom dans le champ *Project Name* puis valider par OK. Vérifier que *An Empty Project* est sélectionné. Puis cliquer sur *Finish*.

Ensuite, il faut se mettre en Release pour avoir du code sans ajout (surtout si on veut compiler du C) : menu *Build|Set Active Project Configuration*. Choisir *Win32 Release*.

Ensuite, il faut ajouter un fichier main.cpp : menu *File|New* et choisir *C++ Source File* et taper main.cpp dans le champ *File Name*.

Dans le fichier copier ce code :

```
//ne sert à rien
int WinMain(long arg1,long arg2,long arg3,long arg4)
{
    _asm {
        //votre code sera ici
    }

    return 0;
}
```

//à remplacer par votre prototype pour mémoire, si vous souhaitez

```

_declspec(naked) void _stdcall MaFct(void)
{
    _asm {
        //ou ici suivant vos préférences
    }
}

```

Enfin, il faut ajouter la génération d'un fichier contenant le code ASM et le code machine en plus de la source : menu *Project|Settings* puis dans l'onglet *C/C++*, choisir *Listing Files* dans la liste déroulante *Category*, puis dans la liste *Listing File Type*, choisir *Assembly, Machine Code, and Source*. Valider par *OK*.

Taper votre code dans l'un des deux emplacements prévus à cet effet.

Compiler votre code avec le menu *Build|Compile main.cpp*.

Dans le répertoire *Release* de votre répertoire de projet, vous devez avoir un fichier *main.cod*. Repérer le début de votre code, chaque ligne de code doit avoir la syntaxe suivante :

```
Offset_code code_machine code_asm
```

Offset_code est l'offset du code machine depuis le début de la fonction. C'est un entier hexadécimal à 5 chiffres.

Code_machine sont des octets (donc par groupe de deux chiffres hexadécimaux) qui doivent être dans cette ordre en mémoire et qui représentent l'instruction ASM *code_asm*. Il peut y avoir 6 ou 7 octets.

Code_ASM est votre code ASM en texte.

Sous Visual C++ .Net

D'abord, il nous faut un projet : cliquer sur *Nouveau projet* puis dérouler *Projets C++*, cliquer sur *Win32* et choisir *Projet Win32* et donner lui un nom et valider. Dans *Paramètres de l'application*, cocher *Projet vide* et cliquer sur *Terminer*.

Ensuite, il nous faut un fichier *main.cpp*. Menu *Project|Ajouter un nouvel élément*. Cliquer *Visual C++* et choisir *Fichier C++ (.cpp)*. Donner lui le nom *main.cpp* et cliquer sur *Ouvrir*.

Dans le fichier copier ce code :

```

//ne sert à rien
int WinMain(long arg1,long arg2,long arg3,long arg4)
{
    _asm {
        //votre code sera ici
    }

    return 0;
}

```

//à remplacer par votre prototype pour mémoire, si vous souhaitez

```

_declspec(naked) void _stdcall MaFct(void)
{
    _asm {
        //ou ici suivant vos préférences
    }
}

```

}

Ensuite, il faut se régler en Release. Menu *Générer|Gestionnaire de configurations...* Dans *Configuration de la solution active*, choisir *Release*.

Enfin, il nous faut paramétrer la génération du listing. Menu *Projet|Propriétés de ...* Dérouler *C/C++*. Cliquer sur *Fichiers de sortie*. Dans la liste *Sortie de l'assembleur*, choisir *Assembleur, code machine et source (/FAs)* puis cliquer sur *Appliquer* puis *OK*.

Taper votre code dans l'un des deux emplacements prévus à cet effet.

Vérifier que vous êtes bien dans *main.cpp* et qu'il est enregistré. Compiler votre code avec le menu *Générer|Compiler*.

Dans le répertoire *Release* de votre répertoire de projet, vous devez avoir un fichier *main.cod* Repérer le début de votre code, chaque ligne de code doit avoir la syntaxe suivante :

Offset_code code_machine code_asm

Offset_code est l'offset du code machine depuis le début de la fonction. C'est un entier hexadécimal à 5 chiffres.

Code_machine sont des octets (donc par groupe de deux chiffres hexadécimaux) qui doivent être dans cette ordre en mémoire et qui représentent l'instruction ASM *code_asm*. Il peut y avoir 6 ou 7 octets.

Code_ASM est votre code ASM en texte.

Sous Dev-C++

Oubliez parce que ce n'est pas de l'asm normal, c'est de l'asm GNU...C'est un genre spécial...

Et après, comment fais-je pour placer ce code dans ma variable ?

« *J'ai mon fichier main.cod . J'en fais quoi ?* ».

Vous allez regrouper par groupe de 4 octets les codes machines et ajouter autant de CC (ou 90) qu'il faut pour compléter le dernier paquet.

Par exemple (exemple fictif) :

12 34 56 78 9A BC DE F5 10 00

Par groupe de 4 :

12 34 56 78 ; 9A BC DE F5 ; 10 00 CC CC

Maintenant passons au code lui-même :

Private Type asmCode

Code(0 To <taille nécessaire - 1>) **As Long**

End Type

Private m_Code **As** asmCode

Private m_stdfct **As** typStdCallFunctionCallerStack

Public Function Init() **As** <interface>

'on place le code

With m_CodeESP

```

.Code(0) = &HXXXXXXXX
'....
.Code(n) = &HXXXXXXXX
End With
'on demande un objet pour appeler le code ASM
Set Init = InitStdCallFunctionCallerStack(m_stdct, VarPtr(m_Code.Code(0)))
End Function

```

<taille nécessaire-1> est le nombre de groupe de 4 octets.
 &HXXXXXXXX sera remplacé par chaque groupe de 4 octets en remplissant en ordre inverse (convention little-endian).

Exemple : 12 34 56 78 donnera &H78563412

<interface> sera une interface définie dans une typelib qui dérive de *IUnknown* et qui a une seule méthode dont le prototype correspond à celui de la fonction ASM.

Il est important de noter les points suivants :

- Le code est contenu dans une structure et non dans un tableau, car un tableau est libéré avant les instances d'objet et donc on pourrait avoir le risque d'appeler du code qui n'est plus en mémoire et donc crash.
- On doit impérativement définir une interface qui colle à ce qu'attend la fonction ASM pour pouvoir l'appeler de VB
- Lorsque vous n'en avez plus besoin, faites un *Set objet = Nothing*. Ne le laissez pas faire à VB, ça évite les plantages aléatoires.
- Il ne faut jamais mettre autre chose que NOP (&H90) ou INT 3 (&HCC) (et surtout pas 00) pour terminer le dernier DWORD. Ceci permet de détecter (dans le cas de INT 3) tout comportement anormal.

Deux exemples :

L'allocation de mémoire dans la pile : une façon simple de se passer de Redim (et pas forcément de Redim Preserve)

Voilà un moyen d'optimisation des allocations dynamiques de mémoire (autres que Redim Preserve). En effet, allouer de la mémoire dans la pile est nettement plus rapide pour créer un buffer que de faire un Redim ou un Dim tableau qui alloue dans le tas. Le code suivant allouera dans le tas : Dim b(512) as Byte 'ou Redim b(512)

Je rappelle que la pile croît vers le bas. Donc pour allouer, il faut soustraire la taille demandée de ESP.

Le problème de l'allocation dans la pile est la *pagination* de la mémoire. Une page fait 4096 octets. Si l'on change de page ou que l'on alloue plus d'une page, il faut vérifier avec l'instruction TEST que la page est en mémoire. Dans le cas contraire et en l'absence de TEST, on assiste à un plantage sans message de Windows.

Il faudra donc que l'on vérifie que chaque page de l'allocation est bien présente. La fonction prendra un argument qui est la taille que l'on veut allouer. Il est impératif que la taille soit un multiple de 4, sinon la pile sera désalignée et plantage.

Le code d'allocation sera le suivant :

```

MOV ECX, [ESP + 4] //on charge la taille d'allocation

```

```

    LEA EAX, [ESP + 8] //on charge l'adresse de la fin de la zone allouée
Probe:
    CMP ECX, 0x1000    //a-t-on demandé plus d'une page (4096 octets) ?
    JB FinProbe       //il faut tester leurs existances en mémoire

    SUB ECX, 0x1000    //on descend d'une page
    SUB EAX, 0x1000    //on retire une page la taille demandée

    TEST [EAX], ECX    //on teste : si la page n'est pas chargée, le
                       //gestionnaire de mémoire la charge
                       //si on ne fait pas ce test et qu'une page n'est
                       //pas chargée
                       //le code plante de façon transparente (!!!)

    JMP Probe         //on recommence
FinProbe:
    SUB EAX, ECX      //on met le reste < une page

    TEST [EAX], ECX  //on teste, si jamais on a changer de page

    POP ECX          //on récupère l'adresse de retour
    MOV ESP, EAX     //on place l'adresse nouvelle de la pile dans ESP
    PUSH ECX         //on remet l'adresse de retour

    RET              //on retourne à l'appelant

```

Avec le code machine :

```

0003b 8b 4c 24 04 mov ecx, DWORD PTR [esp+4]
0003f 8d 44 24 08 lea eax, DWORD PTR [esp+8]
$Probe$8877:
00043 81 f9 00 10 00
00 cmp ecx, 4096 ; 00001000H
00049 72 0f jb SHORT $FinProbe$8878
0004b 81 e9 00 10 00
00 sub ecx, 4096 ; 00001000H
00051 2d 00 10 00 00 sub eax, 4096 ; 00001000H
00056 85 08 test DWORD PTR [eax], ecx
00058 eb e9 jmp SHORT $Probe$8877
$FinProbe$8878:
0005a 2b c1 sub eax, ecx
0005c 85 08 test DWORD PTR [eax], ecx
0005e 59 pop ecx
0005f 8b e0 mov esp, eax
00061 51 push ecx
00062 c3 ret 0

```

Attention certaines instructions sont en deux lignes.

La désallocation est plus simple, il suffit d'ajouter la taille à désallouer à ESP. Le code sera le suivant :

```

POP ECX //on retire l'adresse de retour
POP EAX //on retire le nombre d'octets à désallouer
ADD ESP, EAX //on désalloue l'espace dans la pile
PUSH ECX //on remet l'adresse de retour
RET

```

Avec le code machine :

```

00063 59 pop ecx
00064 58 pop eax
00065 03 e0 add esp, eax

```

```

00067      51          push  ecx
00068      c3          ret    0

```

Enfin, voici un code permettant d'obtenir l'adresse de la pile (ESP) à tout instant. Cela peut servir à vérifier qu'un appel de fonction par objet léger laisse la pile dans son état d'origine.

```

MOV EAX,ESP          //on copie ESP dans ECX
ADD EAX,4            //on ajoute 4 pour ne pas tenir compte
                    //de l'adresse de retour
RET                  //on retourne à l'appelant

```

Avec code machine :

```

00035      8b c4          mov    eax, esp
00037      83 c0 04       add    eax, 4
0003a      c3          ret    0

```

Cela donnera donc le code suivant :

```

Private Type asmCode
  Code(0 To 1) As Long
End Type

```

```

Private Type asmCode2
  Code(0 To 9) As Long
End Type

```

```

Private m_CodeESP As asmCode
Private m_stdfctESP As typStdCallFunctionCallerStack

```

```

Private m_CodeAlloc As asmCode2
Private m_stdfctAlloc As typStdCallFunctionCallerStack

```

```

Private m_CodeFree As asmCode
Private m_stdfctFree As typStdCallFunctionCallerStack

```

'crée un objet qui permet de récupérer ESP

```
Public Function InitGetESPStack() As ICallLongVoid
```

```
  'on place le code
```

```
  With m_CodeESP
```

```
    .Code(0) = &HC083C48B
```

```
    .Code(1) = &HCCCC304
```

```
  End With
```

```
  'on demande un objet pour appeler le code ASM
```

```
  Set InitGetESPStack = InitStdCallFunctionCallerStack(m_stdfctESP,
```

```
  VarPtr(m_CodeESP.Code(0)))
```

```
End Function
```

'crée un objet qui permet d'allouer de la mémoire dans la pile

```
Public Function InitAllocStack() As ICallLong1Long
```

```
  'on place le code
```

```
  With m_CodeAlloc
```

```
    .Code(0) = &H4244C8B
```

```
    .Code(1) = &H824448D
```

```

.Code(2) = &H1000F981
.Code(3) = &HF720000
.Code(4) = &H1000E981
.Code(5) = &H2D0000
.Code(6) = &H85000010
.Code(7) = &H2BE9EB08
.Code(8) = &H590885C1
.Code(9) = &HC351E08B
End With
'on demande un objet pour appeler le code ASM
Set InitIAllocStack = InitStdCallFunctionCallerStack(m_stdftAlloc,
VarPtr(m_CodeAlloc.Code(0)))
End Function

```

'crée un objet qui permet de désallouer de la mémoire dans la pile

```

Public Function InitIFreeStack() As ICallLong1Long
'on place le code
With m_CodeFree
.Code(0) = &HE0035859
.Code(1) = &HCCCCC351
End With
'on demande un objet pour appeler le code ASM
Set InitIFreeStack = InitStdCallFunctionCallerStack(m_stdftFree,
VarPtr(m_CodeFree.Code(0)))
End Function

```

L'appel de l'instruction CPUID

L'instruction CPUID permet d'obtenir certaines infos sur le CPU de votre PC. Lorsque EAX vaut 0 avant CPUID, il renvoie une chaîne authentifiant la marque du processeur. Lorsque EAX vaut 1, CPUID renvoie la version du processeur.

On peut stocker toutes ces infos dans une structure que l'on passera en paramètre.

On aura le code suivant :

```

/*
typedef struct {
    char VendorID[12];
    unsigned long ProcessorSignature;
    unsigned long MiscInfo;
} CPUINFO, *LPCPUINFO;
*/
//_declspec(naked) void _stdcall GetCpuInfo(LPCPUINFO lpInfo);
_asm {
    push ebx                //on doit toujours sauvegarder EBX

    xor eax,eax            //CPUID 0
    cpuid

    mov eax,[esp + 8]      //on récupère l'adresse de la structure
    mov [eax],ebx         //on les 4 DWORD de l'ID vendeur
    mov [eax + 4],edx
    mov [eax + 8],ecx
}

```

```

    mov eax,1                //CPUID 1
    cpuid

    mov ecx,[esp + 8]       //on récupère l'adresse de la structure
    mov [ecx + 12],eax      //on récupère les infos
    mov [ecx + 16],ebx

    pop ebx                //on restaure EBX

    ret 4
}

```

On aura le fichier ODL suivant :

```

[
    helpstring("Bibliothèque de CPUID"),
    uuid(A1AA8AB4-7839-4A21-B374-74B4307EA1C2)
]
library libCPUID {
    importlib("stdole2.tlb");

    typedef struct {
        unsigned char VendorID[12];
        long ProcessorSignature;
        long MiscInfo;
    } CPUINFO;

    [
        helpstring("Interface pour CPUID"),
        uuid(B359BF14-2A1A-4ABF-826F-4CF90C067D7F), odl
    ]
    interface ICpuID: stdole.IUnknown {
        void GetCPUInfo([in,out] CPUINFO* lpInfo);
    }
}

```

On aura le code VB suivant :

```

Private Type asmCode
    Code(0 To 9) As Long
End Type

Private m_Code As asmCode
Private m_stdfct As typStdCallFunctionCallerStack

Public Function InitCPUID() As ICpuID
    'on place le code
    With m_Code
        .Code(0) = &HFC03353
        .Code(1) = &H24448BA2
        .Code(2) = &H89188908
        .Code(3) = &H48890450
        .Code(4) = &H1B808
        .Code(5) = &HA20F0000
        .Code(6) = &H8244C8B
        .Code(7) = &H890C4189
        .Code(8) = &HC25B1059
    End With
End Function

```

```
.Code(9) = &HCCCC0004  
End With  
'on demande un objet pour appeler le code ASM  
Set InitCPUID = InitStdCallFunctionCallerStack(m_stdct, VarPtr(m_Code.Code(0)))  
End Function
```