Wie erstelle ich eine DLL in Assembler?

Diese zusammengestellten Grundlagen sollen dabei helfen eine DLL in Assembler zu erstellen. Grundsätzliche Assemblerkenntnisse werden vorausgesetzt, da es sich nicht um einen Assemblerkurs handelt. Es wird nur gezeigt wie der Aufbau des Assemblerprogrammes für eine DLL aussehen muß und welche Voraussetzungen erforderlich sind um dieses Programm als DLL zu compilieren.

Alle Ausführungen und das mitgelieferte Beispiel beziehen sich auf die 32-Bit Windowsversionen. Das heißt ab Windows95 aufwärts und alle entsprechenden Profanversionen ab 5.0-32.

Profan ist zwar eine, wie ich meine, sehr gute und einfach zu benutzende Programmsprache mit sehr großem Funktionsumfang. Doch können klarerweise nicht alle Möglichkeiten die Windows bietet inkludiert sein bzw. können manche Funktionen nicht oder nur sehr umständlich in Profan programmiert werden. Hier bietet sich eine DLL an.

DLLs können aber nicht nur in Assembler sondern auch in anderen Sprachen erstellt werden (C++, Delphi, ...)

Eine DLL, in Assembler geschrieben, kann neben den reinen Assemblerinstruktionen auch Windows-API Aufrufe benutzen.

Zur Verwendung von **APIs** gibt es auf meiner H.P. auch einen Workshop.

Im Anhang befindet sich ein Literaturverzeichnis.

Diese Version 3 ist eine Erweiterung der vorigen Versionen.

Gerhard Putschalka 30.8..2003 gerhard-putschalka@gmx.at

Homepage http://www.gerhard-putschalka.xprofan.com/index.htm

Ein (ganz kurzes) Vorwort

Obwohl es immer Assembler heißt gibt es doch (geringe) Unterschiede ob man mit Masm32 (Microsoft) oder mit TAsm (TurboAssembler von Borland) arbeitet. Darüber hinaus gibt es noch weitere Assembler.

Grundsätzlich sollten alle Assembler-Compiler zum Erstellen einer DLL geeignet sein.

Das Beispiel in diesem Workshop wurde für Verwendung mit Masm32 erstellt.

Und ohne weitere Vorrede steigen wir gleich direkt ein.

Wie muß das Gerüst des Programmes für eine DLL aussehen?

```
   .386
   .model flat, stdcall
   option casemap :none   ; case sensitive
```
; bei casemap :none wird festgelegt, daß die Groß- und Kleinschreibung entscheidend ist (case sensitive).

; Um Groß- und Kleinschreibung zu ignorieren kann
```
   option casemap :all   ; case not sensitive
```
; angegeben werden. Ich bevorzuge case senitive

Nach diesem Einleitungsteil kommen die **Datenbereiche**, von denen es 3 Arten gibt. Sie werden eingeleitet mit
```
.data
.data?
.const
```

Alle 3 Arten sind wahlweise.

```
.data
```
; hier werden Bereiche definiert, die mit Daten initialisiert sein können und die auch inhaltlich verändert werden können
; z.B.:

```
Wert1   dd   25
Wert2   dd   ?
Text    db   "ABCDEFG",0"
```

```
.data?   ; dieser Datenbereich ist nicht zwingend erforderlich
```
; In diesem Bereich können keine Werte zugeordnet werden. Werte können nur im Programmablauf eingesetzt bzw. geändert ; werden. Der Vorteil ist, daß dieser Bereich erst bei der Ausführung zugeordnet wird.
```
Adress1    dd   ?
Adress2    dd   ?
Adress3    dd   ?
Text01     db   50 dup(?)  ; Variable für 50 Bytes
```

```
.const?   ; dieser Datenbereich ist nicht zwingend erforderlich
```
; hier werden Konstanten definiert. Diese können im Programmablauf nicht verändert werden. Natürlich könnten diese Konstanten auch im `.data` *Bereich definiert sein.*
; z.B.:
```
Text1   db   "TestDLL",0"
```

Zuletzt der **Befehlsbereich**
```
.code
; hier werden die Programmfunktionen angegeben.
```

; Die Prozedur LibMain wird immer beim Starten und Beenden der DLL durchlaufen.
```
LibMain   proc hInstDLL:DWORD, reason:DWORD, unused:DWORD
```

; die hier eventuell benutzten Instruktionen werden auf Seite 12 beschrieben.

```
          mov  eax,1
          ret
LibMain   Endp
```

; und hier werden, als Prozeduren, die einzelnen Funktionen eingebaut die vom Hauptprogramm aufgerufen werden.

; Als Beispiel die Funktion1 die zwei LongInt-Werte erwartet diese addiert und als Ergebnis an das aufrufende (Profan-) Programm zurückgibt:

```
Funktion1 proc
          push  ebp           ; base pointer Register sichern (auf den Stack)
          mov   ebp,esp       ; stack pointer in base pointer Register laden
          mov   eax,[ebp+8]   ; Wert des ersten Parameters in eax setzen
          add   eax,[ebp+12]  ; Wert des zweiten Parameters zu eax addieren
          pop   ebp           ; base pointer Register zurückholen (vom Stack)
          ret   8             ; xx = Stackpointer rücksetzen = Anzahl Parameter x 4
Funktion1 endp


Funktion2 proc
          push  ebp
          mov   ebp,esp
```

; hier werden die Instruktionen, die die Funktion ausführen, eingesetzt

```
          pop   ebp
          ret   xx          ; xx = Stackpointer rücksetzen = Anzahl Parameter x 4
Funktion2 endp
```

; und der Abschluss:

```
          End       LibMain
```

Was passiert in Funktion1?

Wird eine Funktion (aus Profan) aufgerufen so werden gewisse Daten auf den Programmstack gestellt. Dieser Stack ist ein interner Speicherbereich, der u.a. bei Prozedurenaufrufen benutzt wird (eine Funktion ist im Prinzip eine Prozedur).

Wie sieht der Stackinhalt beim Eintritt in die Funktion aus?

```
esp+8        DD    Adresse des zweiten Wertes
esp+4        DD    Adresse des ersten Wertes
esp+0        DD    Rücksprungadresse zum aufrufenden Programm (= zu Profan)
```

`esp` ist das Register, das auf die aktuelle Adresse des Stacks zeigt (**E**xtended **S**tack **P**ointer).

Zuerst wird das base pointer Register (`ebp`) mit push auf den Stack gesichert, weil es anschließend verändert wird.
Durch den Befehl `push` wird der Stackpointer, das ist der Wert im Register `esp`, um 4 Stellen erhöht (das Register `ebp` ist ein 4-Byte Register).

Daher sieht der Stack nun so aus:

```
esp+12       DD    Adresse des zweiten Wertes
esp+8        DD    Adresse des ersten Wertes
esp+4        DD    Rücksprungadresse zum aufrufenden Programm (= zu Profan)
esp+0        DD    Inhalt des gesicherten Registers ebp

         mov   ebp,esp      ; stack pointer in base pointer Register laden
```

Der Stackpointer wird in das **E**xtended **B**ase **P**ointer Register übertragen, mit dem nun adressiert wird.
Damit ist auch klar, daß Parameter1 nun mit [ebp+8] und Parameter2 nun mit [ebp+12] zu adressieren sind.

```
       mov   eax,[ebp+8]  ; Wert des ersten Parameters in Register eax setzen
       add   eax,[ebp+12] ; Wert des zweiten Parameters zu Register eax addieren
```

; am Ende der Prozedur wird wieder das `ebp` *Register" rückgesichert (für jedes* `push` *muß es ja auch ein* `pop` *geben)*

```
       pop   ebp
```

; Damit zeigt der Stackpointer wieder auf die Rücksprungadresse zum aufrufenden Programm wohin mit dem Befehl **ret** *; wiedernach Profanm zurückgesprungen wird.*

```
       ret   8
```

Um die nun nicht mehr benötigten Parameterwerte vom Stack zu löschen wird bei dem Befehl **ret** noch die Anzahl der zu löschenden Stellen angegeben. Da pro Parameter ein Dword (DD, oder in Profan ein LongInt, also 4 Bytes) benutzt wird ergibt das im Beispiel mit 2 Parametern 8 Stellen.

Der Wert, der sich beim Rücksprung in das aufrufende Programm im Register `eax` befindet kann in Profan als Ergebnis abgefragt werden:

```
Let Ergebnis& = @Funktion1(Zahl1&,Zahl2&)
```

Bevor ich auf Details eingehe setzen wir voraus die DLL ist fertig programmiert und wir wollen sie compilieren.

Was brauchen wir dazu?

Zuerst den Compiler. Masm32 in der Version 6. Inzwischen gibt es bereits die Version 8. Download siehe Anhang.

Um die DLL erfolgreich compilieren zu können brauchen wird noch eine Definitionsdatei. Sie kann so aussehen (und kann mit einem Texteditor als Datei mit der Dateierweiterung `.DEF` erstellt werden):

```
LIBRARY DLL1.dll
EXPORTS Funktion1
EXPORTS Funktion2
```

Bei `LIBRARY` ist der Name der zu erstellenden DLL einzutragen.
Für jede aufzurufende Funktion muß eine Zeile `EXPORTS` bestehen mit dem Namen der Funktion. Wichtig ist dabei, daß der Name genauso groß/klein geschrieben wird wie in der Assemblersource.

Setzen wir voraus der Compiler befindet sich auf D: im Verzeichnis masm32; die Source hat den Namen DLL1.asm und die DLL soll den Namen DLL1.dll haben.

Erstellen wir zum Compilieren eine .bat Datei (DOS läßt grüßen!) mit den 2 folgenden Anweisungen.

```
D:\masm32\bin\ml  /c  /coff  DLL1.asm
D:\masm32\bin\Link  /SUBSYSTEM:WINDOWS  /DLL  /DEF DLL1.def  DLL1.obj
```

Mit Doppelklick auf diese Datei compilieren wir die Source. Damit wird, wenn die Umwandlung erfolgreich ist, eine Datei DLL1.obj erstellt, die mit dem Linker als DLL gelinkt wird.
Und damit wäre die DLL gebrauchsfertig. Die Dll1.obj kann gelöscht werden.

Wird eine Umwandlungsliste gewünscht, die bei Umwandlungsfehlern oft hilfreich ist, so ist die erste Zeile der .bat Datei zu ändern:

```
D:\masm32\bin\ml  /FlDLL1.lst  /c  /coff  DLL1.asm
```

Damit wird eine Textdatei DLL1.lst erstellt, die mit WordPad angesehen werden kann. Die Erweiterung `.lst` kann auch `.txt` lauten.

Einfachere Form einer Assemblerfunktion.

Da die Schnittstelle zwischen Profan und der Masm-DLL standardisiert ist (`stdcall`) können die Prozeduren einfacher gehalten werden. Statt mit dem eher umständlichen Stackpointer zu hantieren können die Parameter auch anders verarbeitet werden:

```
Funktion1  proc Parm1:DWord, Vari2:DWord
           mov   eax,Parm1
           add   eax,Vari2
           ret
Funktion1  endp
```

Die Parameter die an die Funktion übergeben werden sind in der `proc` Anweisung, in der selben Reihenfolge wie im Profanaufruf, anzugeben. Hier sind noch die Parametertypen anzugeben. `Varaiblenname:Type` .Bei aufeinanderfolgenden Variablen gleicher Type könnte die Zeile auch so aussehen:

```
Funktion1  proc Parm1, Vari2:DWord
```

In der Beispiel-DLL habe ich alle Funktionen nach dem einfacheren Schema gemacht. Ist übersichtlicher und weniger zu schreiben.
Auch hier wird der Wert in `eax` als Ergebnis an Profan zurückgemeldet. Wollten wir das Ergebnis (die Summe der Addition) (auch) im 2. Parameter an Profan zurückgeben so schreiben wir

```
Add   Vari2, eax
```

statt

```
add   eax,Vari2
```

Damit wird der Wert von Parm1 zum Wert in Vari2 addiert und an Profan zurückgegeben. Er bleibt aber auch in eax unverändert bestehen.

Um die Funktionen der Dll aufzurufen sind folgende Anweisungen im Profanprogramm notwendig:

' zuerst die Funktionen dem Profanprogramm bekanntgeben (hier nur Bezug auf Funktion1!):

```
Def Addieren(2)  ! "DLL1.dll","Funktion1"  ' 2 Werte addieren
```

Nach "Def" kommt der Funktionsname wie er in Profan angesprochen wird. Die Zahl in Klammern gibt an wieviele Parameter an die Funktion zu übergeben sind. Das Rufzeichen weist hin daß es sich um die 32-Bit Version handelt (also ab Window95 aufwärts). Dann kommt, **nach einer Leerstelle,** der Name (ggf. inkl. Pfad) der Dll und zuletzt der Name der Funktion wie er in der DLL angegeben ist. Groß/Kleinschreibung beachten! Der Name der DLL kann auch in einer Stringvariablen stehen (dann natürlich ohne Apostrophen).

Aufgerufen wird die Funktion:

```
@Addieren(Zahl1&,Zahl2&)
```

oder wenn die Funktion ein Ergebnis liefern kann:

```
Let Ergebnis& = @Addieren(Zahl1&,Zahl2&)
```

Das Zeichen @ kann auch weggelassen werden.

Während Strings nur über Bereichsvariablen an eine Funktion zu übergeben sind, können Zahlen direkt, oder in einer numerischen Variablen als Integer, Longinteger, oder in einer Bereichsvariablen (als Integer, Longinteger) übergeben werden (die DLL muß entsprechend unterschiedlich handeln). Es besteht zwar die Möglichkeit statt der Bereichsvariablen auch eine Stringvariable für die DLL zu verwenden doch sollte man das eher unterlassen. Es ist zwischen Profan Versionen vor 7.x und ab 7.x ein wesentlicher Unterschied, der zum Absturz führen kann.
Strings dürfen nie direkt bei einem Aufruf angegeben werden!

Als Beispiel:

```
@Funktion(Zahl&, 1120, Bereich1#, Ber2#)
```

nicht erlaubt ist:

```
@Funktion(Zahl&, 1120, "ABCDE", Ber2#)
```

Nun zurück zur DLL, zu den Details.

Da es durchaus sinnvoll sein kann in der DLL auch API-Aufrufe zu verwenden ist es dazu notwendig einige Includedateien, welche im Masm32-Verzeichnis sind, in die Source einzubinden:

```
.386
.model flat, stdcall
option casemap :none   ; case sensitive
include    D:\masm32\include\windows.inc
include    D:\masm32\include\user32.inc
include    D:\masm32\include\kernel32.inc
include    D:\masm32\include\gdi32.inc
include    D:\masm32\include\winmm.inc
includelib D:\masm32\lib\user32.lib
includelib D:\masm32\lib\kernel32.lib
includelib D:\masm32\lib\gdi32.lib
includelib D:\masm32\lib\winmm.lib
```

Es müssen nicht alle Includes vorhanden sein – zumindest die, die von den benutzten Aufrufen gebraucht werden. Werden keine API-Aufrufe benutzt, müssen diese Includes nicht eingebaut werden.

Wie ist das mit den Parametern und welche Daten bekommen wir und wie?

Alle Parameterwerte bekommen wir in der selben Reihenfolge wie beim Aufruf als LongInteger (hier im Assembler werden sie als DWord (oder DD = 4Bytes) bezeichnet) auf dem Stack aufgereiht in die DLL.

Dabei sind Unterschiede zu beachten.

Ein Aufruf aus Profan könnte so aussehen:

```
Let Var% = 25
Let Var& = 98
Long Ber#,0 = 67
String Ber#,4 = "ABCDEFGH"

@Funktion(1234,Var%,Var&,Ber#)     ' hier wird für Ber# die Adresse der ersten Stelle von Ber# übergeben,
                                     ' somit also die Adresse zum LongInt-Wert 67
```

die Entgegennahme dieser Werte ist in der DLL so zu lösen:

```
          .data
NamFeld   db    50 dup(?)  ; muß für den String groß genug sein

.code

. . . . . .

Funktion  proc Nummer1, Varia2, Zahl3, Adr4:DWord
          mov   eax,Nummer1  ; Parameter 1: eax bekommt den Wert 1234
          mov   ebx,Varia2   ; Parameter 2: ebx bekommt den Wert 25
          mov   edx,Zahl3    ; Parameter 3: edx bekommt den Wert 98
          mov   esi,Adr4     ; Parameter 4: esi bekommt die Adresse von Ber#,0

          mov   eax,dword ptr[esi] ; eax bekommt jetzt den Wert 67

          add   esi,4        ; zeige auf den Beginn des Strings (= Ber#,4)
          lea   edi,NamFeld  ; NamFeld ist im Datenbereich ein Puffer
          mov   ecx,...      ; hier muß noch die Stringlänge eingetragen werden!
          cld                ; directionFlag löschen
          rep   movsb        ;   und den String nach NamFeld übertragen
```

Merke: Numerische Daten kommen immer als DWord (DD) direkt am Stack daher, während bei Bereichsvariablen immer die Adresse als DWord kommt (unabhängig davon ob in der Bereichsvariablen numerische Werte stehen).

Damit die Funktion ein Ergebnis an das aufrufende Profanprogramm zurückgeben kann, ist der Wert vor dem Beenden der Funktion in das Register eax zu stellen (Das Ergebnis kann natürlich nur ein DWord bzw. LongInt sein).

```
          mov   eax,1       ; als Ergebnis wird der Wert 1 an Profan zurückgegeben
          ret
Funktion  endp
```

In dieser Form kann nur ein numerisches Ergebnis an Profan zurückgegeben werden. Soll das (oder ein weiteres) Ergebnis ein String sein, ist das über eine Bereichsvariable zu lösen. So wie der String eingelesen wurde kann in die Bereichsvariable (oder eine eigene Bereichsvariable, deren Adresse als Parameter übermittelt wurde) geschrieben werden.

Das obere Beispiel stimmt im Prinzip. Aber: Grundsätzlich sollte auch in der Dll ein String, so wie in Profan, mit einem 0h Byte abgeschlossen sein. Das erleichtert die Verarbeitung und bietet darüber hinaus die Möglichkeit mit API-Aufrufen effizienter zu programmieren.

Wenn wir obiges Beispiel

```
          lea   edi,NamFeld  ; NamFeld ist im Datenbereich ein Puffer
          mov   ecx,...      ; hier muß noch die Stringlänge eingetragen werden!
          cld                ; directionFlag löschen
          rep   movsb        ;   und den String nach NamFeld übertragen
```

durch die folgende Anweisung ersetzen:

```
          invoke lstrcpy, addr NamFeld, esi
```

ersparen wir uns einige Anweisungen und brauchen uns um die Länge des Strings auch nicht zu kümmern. Mit diesem einzigen API-Aufruf wird der String aus der Bereichsvariablen in den Puffer `NamFeld` kopiert und mit einem 0h-Byte abgeschlossen. Für `lstrcpy` brauchen wir zwei Parameter. Die Adressen des empfangenden und des sendender Bereichs.

Aber falls wir doch die Stringlänge brauchen so könnten wir mit

```
invoke lstrlen, esi
```

die Länge des Strings feststellen (`esi` zeigt zur ersten Stelle des Strings es kann aber auch jedes andere Register verwendet werden). Die Länge des Strings (ohne 0h-Byte) wird in `eax` übergeben. Für den API-Aufruf ist als einziger Parameter die Adresse mitzugeben an der der zu prüfende String beginnt.

Bei der Angabe einer Adresse für API-Aufrufe muß man genau unterscheiden: ist es die direkte Adresse des Stringbeginnes oder ist es der Label (zur Adresse) des Puffers!

```
mov   esi,Adr4; Parameter 4: esi bekommt die Adresse von Ber#,0
```

hier wird vom Profanprogramm die **Adresse** von `Ber#,0` übergeben und somit ist der Inhalt des Registers `esi` auch die Adresse.

Daher ist

```
invoke lstrlen, esi
```

korrekt.

Befindet sich ein String im Datenbereich (der DLL) ab dem Label NamFeld:

```
NamFeld   db   "A1B2C3DOL",0
```

Und wir wollen die Länge des Strings in `NamFeld` ermitteln muß man mit der Adressierung aufpassen!

Hier

```
invoke lstrlen, NamFeld
```

zu verwenden **wäre falsch**!
Für `NamFeld` würde nicht die Adresse wo der String beginnt, sondern es würden die ersten 4 Bytes des Strings (ist der Inhalt des Labels) als Adresse verwendet was natürlich in einen völlig falschen Speicherbereich zeigt.

Hier muß

```
invoke lstrlen, addr NamFeld
```

geschrieben werden. `addr` (in TAsm heißt es `offset`) weist den Assembler an nicht den Inhalt sondern die Adresse des Labels zu verwenden.

In `invoke` Anweisungen muß `addr` verwendet werden, `offset` ist hier ungültig!

Es wäre aber auch möglich:

```
lea    eax,NamFeld
invoke lstrlen, eax
```

oder

```
mov    eax,offset NamFeld ; hier ist offset anzuwenden
invoke lstrlen, NamFeld
```

zu schreiben.

Unter TAsm werden Prozeduren (ein API-Aufruf ist im Prinzip auch ein Prozedurenaufruf) mit `CALL` aufgerufen. Im Masm32 wird anstelle von `CALL` der Aufruf `INVOKE` benutzt.
Anschließend an `INVOKE` kommt der Name der Prozedur, ein Komma und danach, durch Kommata getrennt, kommen die für diese Prozedur vorgesehenen Parameter (im Prinzip wie bei Profan).

`addr – offset` ?

Das scheint ja im Prinzip das selbe zu sein. In MAsm32 ist bei `invoke` immer `addr` zu benutzen. Für normale Assemblerinstruktionen gilt weiterhin `offset`.

`lstrlen` und `lstrcpy` sind API Aufrufe. Diese sind in der Hilfedatei Win32.hlp beschrieben (leider in englisch und in der Form für Delphi –oder C++?). Nähere Hinweise siehe Anhang unter Win32.hlp und API-Workshop.

Auch selbstgeschriebene Prozeduren in der Dll werden mit `invoke` aufgerufen. Auch hier können Parameter mitgegeben werden. Es ist aber zu beachten, daß sich solche aufgerufenen Prozeduren in der Source immer vor deren Aufrufen befinden müssen (ist in Profan ebenso).

Es ist zwar anders auch möglich, dann muß mit PROTO jede dieser Prozeduren am Anfang deklariert werden. Dann können aufgerufene Prozeduren auch hinter der aufrufenden Prozedur sein.

Beschreibung des DLL Beispieles.

```
.386
.model flat, stdcall
option casemap :none   ; case sensitive
include    D:\masm32\include\windows.inc
include    D:\masm32\include\user32.inc
include    D:\masm32\include\kernel32.inc
include    D:\masm32\include\gdi32.inc
include    D:\masm32\include\winmm.inc
includelib D:\masm32\lib\user32.lib
includelib D:\masm32\lib\kernel32.lib
includelib D:\masm32\lib\gdi32.lib
includelib D:\masm32\lib\winmm.lib
```

Bei den Datenbereichen gibt es drei Arten.

Dieser hier ist fix im Programm, hier können auch Werte zugeordnet sein und die Inhalte können verändert werden.

```
.data
EinByte    db    0   ; Byte
EinWort    DW    25  ; Wort (Profan Integer% = 2-Bytes)
Doppelwort DD    ?   ; Doppelwort (Profan LongInt& = 4-Bytes)
QuadWort   DQ    0   ; 4-Wort (8-Bytes, müßte Profan Float! Entsprechen)
```

Dieser Datenbereich kann nur im Ablauf geändert werden. Er wird erst beim Programmaufruf zugeordnet!
Hier können keine Werte in der Source angegeben werden. Dieser Datenbereich muß nicht vorhanden sein.

```
.data?
Nochwas    db    ?
```

; Dieser Konstantenbereich kann nicht im Ablauf geändert werden. Dieser Bereich muß nicht vorhanden sein. Diese Textzuordnungen könnten sich genausogut auch in `.data` befinden.

```
.const
CopyRight  db  "DLL1.DLL.",13,10,"Version 3.0, vom 30.August 2003   "
           db  "Autor g.putschalka@web.de",13,10,10
           db  "Für Schäden durch eine Fehlfunktion "
           db  "der DLL kann keine",13,10,"Haftung übernommen werden.",0
CopyRig02  db  "Info",0
```

Der Codebereich. Hier werden die Befehle zum Programmablauf angegeben.

```
.code
LibMain   proc hInstDLL:DWORD, reason:DWORD, unused:DWORD
                        ;Start-Einsprungspunkt der DLL (siehe End – Anweisung)
          mov  eax,1
          ret
LibMain   Endp

addieren  proc  Wert1, Wert2, OpCod:DWord
          mov   eax,Wert1
          .if   OpCod==1
            add   eax,Wert2
          .else
            sub   eax,Wert2
          .endif
; Der Wert in eax ist beim Beenden der Funktion immer das Ergebnis für Profan
; hier ist es das Ergebnis der Addition
          ret
addieren  endp
```

Zuerst wird der Schlüssel (von `OpCod`) geholt ob zu addieren oder zu subtrahieren ist.

Zuerst wird der erste Wert (`Wert1`)in das Register eax geladen. Danach wird mit `.if`, `.else`, (`.elseif`) und `.endif` entschieden ob der zweite Wert (`Wert2`) addiert oder subtrahiert wird. Die Operatoren sind:

==	prüfen auf equal
<	prüfen auf kleiner als
>	prüfen auf größer als
>=	prüfen auf nicht kleiner als
<=	prüfen auf nicht größer als

Natürlich kann, so wie bisher, auch mit `cmp` und `je` (`jne` ...) die Entscheidung für addieren oder subtrahieren getroffen werden.

```
SubReal4  proc   Wert1, Wert2:DWord
          LOCAL  IntVar:DWord
          fninit
          fld    dword ptr Wert1
          fld    dword ptr Wert2
          fsub
          fstp   IntVar                      ; Ergebnis zwischenspeichern
          mov    eax,IntVar
          ret
SubReal4  endp
```

Mit dieser Funktion werden 2 Realzahlen (Floatzahlen) verarbeitet. `Wert1` minus `Wert2` ist `Ergebnis`. Für diese Funktion werden die Realzahlen im Profanprogramm in Bitmuster in LongInts umgewandelt und diese Bitmuster an die DLL übergeben. Das Resultat wird ebenfalls als Bitmuster an Profan zurückgegeben und dort wieder in eine Realzahl umgewandelt.
Zur Verarbeitung wird der Matheprozessor benutzt.
Die Variable `IntVar` wird hier lokal in der Funktion definiert (könnte aber ebenso auch global in `.data` angegeben sein). Es sollte für Floatzahlen aber besser das Muster in der nächsten Funktion angewendet werden wo echte 64-Bit Floatwerte verarbeitet werden, weil gelegentlich durch die Um-/Rückwandlung Float-Bitmuster falsche Dezimalstellen entstehen können.

```
MulReal8  proc  Wert1, Wert2, Ergebnis:DWord

          mov     eax,Wert1         ; Adresse Variable 1
          mov     edx,Wert2         ; Adresse Variable 2
          fninit
          fld     qword ptr [eax]
          fld     qword ptr [edx]
          fmul
          mov     edi,Ergebnis      ; Adresse Variable 3
          fstp    qword ptr [edi]   ; Ergebnis direkt in Variable 3 stellen

          mov     eax, 1   ; irgendein Wert, = Returnwert zu Profan
          ret
MulReal8  endp
```

Mit dieser Funktion werden 2 Realzahlen (Floatzahlen) miteinander multipliziert. Zum Unterschied von der Funktion **SubReal4** werden hier die Floatzahlen in der originalen 8-Byte Größe verarbeitet. Da diese Zahlen nicht direkt an die Funktion übergeben werden können (zumindest nicht auf einfachem Wege) werden die Adressen der Variablen von Profan an die DLL übergeben. Das Ergebnis kommt in die 3. Floatvariable, deren Adresse ebenfalls mit dem Aufruf mitgeliefert wird.

```
Ergebnis& = @MulReal8(@Addr(Wert1!),@Addr(Wert2!),@Addr(Ergebnis!))
```

Zur Verarbeitung wird der Matheprozessor benutzt.

```
umdrehen  proc  DatenAdr:DWord
          mov   esi,DatenAdr       ; Adresse 1 Stelle im String
          invoke lstrlen, DatenAdr ; ermittle Stringlänge
          mov   edi,esi
          add   edi,eax
          dec   edi                ; Adresse letzte Stelle im String
          shr   eax,1              ; divid. Länge/2  ohne Rundung
          mov   ecx,eax            ; Anzahl Durchläufe
          .while ecx>0
            ; tausche Bytes in [esi] <--> [edi]
            mov  al,byte ptr [esi]
            mov  ah,byte ptr [edi]
            mov  byte ptr [esi],ah
            mov  byte ptr [edi],al
            inc  esi
            dec  edi
```

```
          dec  ecx
        .endw

        mov   eax,DatenAdr       ; Rückparameter an Profan übergeben (eax)
        ret
umdrehen  endp
```

Mit dieser Funktion wird ein String in der Reihenfolge der Zeichen umgekehrt.
Es wird hier `esi` auf die erste Stelle und `edi` durch Ermittlung der Stringlänge auf die letzte Stelle des Strings positioniert.
Die Stringlänge geteilt durch 2 ohne Aufrundung, (mit `shift` rechts um ein Bit) ergibt die Anzahl der Schleifendurchläufe. Dieser Wert wird in das Register `ecx` gestellt.
In der Schleife werden die Bytes in `[esi]` und `[edi]` vertauscht, die beiden Register auf die nächste Stelle positioniert (`esi` aufwärts, `edi` abwärts) und `ecx` um 1 vermindert. Hat `ecx` den Wert 0 werden die Durchläufe beendet.
Da beim Tauschen der Bytes hier nicht mit `lodsb` und `stosb` gearbeitet wird, muß die Adressierung indirekt erfolgen. Deshalb werden die Register in Klammern gesetzt. Das bedeutet: hole/setze das Byte an die Adresse die im Register steht.

```
mov   al,byte ptr [esi]
```

dagegen würde

```
mov   al,byte ptr esi
```

den Wert des niederwertigsten Bytes des Registers `esi` in das Register `al` laden (was ein Teil der Adresse wäre).
Zum Schluß wird noch die Adresse der Bereichsvariablen als Ergebnis an Profan zurückgegeben. Ist zwar überflüssig, soll aber zeigen was in Profan damit gemacht werden kann (folgt später).

```
concat  proc  Adress1, Adress2, Adress3:DWord
        invoke lstrcpy, Adress3, Adress1
        invoke lstrcat, Adress3, Adress2
        ret
concat  endp
```

Mit concat werden zwei Strings zusammengehängt und an die Adresse im 3. Parameter übertragen.
Mit den API-Aufrufen wird zuerst der String von der Adresse in `Adress1` mit `lstrcpy` an die Adresse in `Adress3` kopiert. Danach wird der String von der Adresse in `Adress2` mit `lstrcat` an den String an der Adresse in `Adress3` angehängt.

```
Dll_Info  proc
          invoke  MessageBoxA, 0,addr CopyRight,addr CopyRig02,MB_OK
          ret
Dll_Info  endp
```

Hier wird nur eine Messagebox angezeigt. Dient hier zur Anzeige der Version der DLL.

```
          End       LibMain
```

Bei der (letzten) Anweisung `End` muß noch der Einsprungspunkt zur DLL angegeben werden (`LibMain`).

Wichtig bei API-Aufrufen ist, daß Register nach dem Aufruf verändert sein können. Daher wichtige Inhalte von Registern vor dem Aufruf sichern.

Nun noch eine Ergänzung zur Ausführung der Funktion **`umdrehen`**

In Profan erfolgt der Aufruf bzw. die Anzeige danach mit:

```
Let F1$ = "ABCDFGH"
String A#,0 = F1$
@umdrehen(A#)
Print "umdrehen:",F1$,"=",@String$(A#,0)
```

In der Funktion wird auch die Adresse zum umgedrehten String als Ergebnis zurückgeliefert. Daher könnte die Auswertung aber auch so erfolgen (die Variable `Abc&` muß natürlich deklariert sein):

```
Let Abc& = @umdrehen(A#)
Print "umdrehen:",F1$,"=",String$(Abc&,0)
```

Weil in der Funktion am Ende die Adresse der Bereichsvariablen `A#` in das Register `eax` gestellt wurde kommt diese Adresse an Profan als Ergebnis zurück und kann bei der Auswertung auch statt der Bereichsvariablen benutzt werden. Kann interessant sein, wenn das Ergebnis nicht an den Beginn sondern in den String zeigt.

Hilfestellung beim Testen.

Selten wird ein Programm (auch eine DLL) gleich auf Anhieb richtig funktionieren. Es gibt bei Assembler auch eine Debug Möglichkeit. Doch nicht immer muß die Fehlersuche aufwändig sein.
Oft genügt es festzustellen wie an einem bestimmten Punkt im Programm der Wert in einem Register ist. Oder ob ein String den richtigen Inhalt hat.

Dafür gibt es in diesem Workshop eine Includedatei Support.inc.

Sie beinhaltet 3 Prozeduren:

`DspReg` Inhalte der Register eax, ebx,ecx,edx,edi und esi anzeigen. Der Aufruf hat keine Parameter.
Aufruf:
```
invoke DspReg
```

`DspString` einen String anzeigen. Der String muß mit 0h abgeschlossen sein! Der Aufruf hat 2 Parameter.
Aufruf:
```
invoke DspString, addr String, addr Titel
```

`DspHString` einen String in hexadezimaler Form anzeigen (ein 0h beendet nicht den String). Der Aufruf hat 3 Parameter.
Aufruf:
```
invoke DspString, addr String, Länge, addr Titel
```

Während bei `DspString` der String mit 0h abgeschlossen sein muß, werden bei `DspHString` 0h-Bytes nicht als Stringende betrachtet. Daher muß im Parameter 2 die Länge des anzuzeigenden Strings angegeben werden. Es werden jedoch nur maximal 50 Zeichen angezeigt.

Bei `Titel` kann noch die Adresse zu einem Text angegeben werden, welcher in der Kopfzeile der Messagebox angezeigt wird. Wird hier statt `addr Titel` eine Null angegeben, wird ein Standardtitel angezeigt.

Um die Prozeduren anwenden zu können muß vor der ersten Funktion die Zeile

```
include    Support.inc
```

durch entfernen des Semikolon aktiviert werden (ggf. mit dem Pfad, wenn sich die Includedatei nicht im selben Verzeichnis befindet).

Hier noch eine Beschreibung der Instruktionen die in der `Proc LibMain` eingesetzt werden können.

Einen Dank an Frank Abbing der mir diese Infos lieferte.

Die LibMain wird immer dann aufgerufen, wenn die Dll gestartet und beendet wird, der Programmierer kann also beim Start und Ende eigenen Programmcode aufrufen (z.B. Infofenster, Registrierabfragen, Zusatzdll's laden usw.).

Das ginge dann so:

```
LibMain proc hInstDLL:DWORD, reason:DWORD, unused:DWORD

 ; wenn hier code benutzt wird, müssen alle benutzten Register gesichert werden (z.B. push ebx)

      .if reason == DLL_PROCESS_ATTACH

  ; hier kann Code stehen, der ausgeführt wird, wenn die Dll gestartet wird...

      .elseif reason == DLL_PROCESS_DETACH

  ; hier kann Code stehen, der ausgeführt wird, wenn die Dll beendet wird...

      .elseif reason == DLL_THREAD_ATTACH

  ; hier kann Code stehen, der ausgeführt wird, wenn das Hauptprogramm der Dll einen neuen Thread gestartet hat.
  ; Da das mit Profan aber nicht geht, ist es für uns uninteressant

      .elseif reason == DLL_THREAD_DETACH

  ; hier kann Code stehen, der ausgeführt wird, wenn das Hauptprogramm der Dll einen Thread beendet hat.
  ; Da das mit Profan aber nicht geht, ist es für uns uninteressant

      .endif

 ; alle eventuell gesicherten Register müssen jetzt wieder restauriert werden, (z.B. pop ebx)

      mov eax,1    ;Rückgabe an Hauptprogramm, 1 = keine Fehler.
      ret

LibMain   Endp
```

Damit endet hier die Beschreibung über die Grundlagen.

Im Anhang folgen noch einige Informationen.

Anhang.

Download Masm32:

Unter http://www.codingcrew.de/masm32/index.php gibt es eine nützliche Seite, von der auch der Compiler (derzeit V8) kostenlos heruntergeladen werden kann.
Alternativ gibt es auch von http://spiff.tripnet.se~iczelion/files/masm32v8.zip die Downloadmöglichkeit.

Download Win32.hlp:

Kann von ftp://ftp.borland.com/pub/delphi/techpubs/delphi2/win32.zip kostenlos heruntergeladen werden. Sprache: englisch.

Literatur:

Assembler Referenz von Oliver Müller. Franzis Verlag. ISBN 3-7723-7505-7 Sprache: deutsch.
Preis ca. Euro 45,-

Ergänzend zu APIs gibt es

„Das Win32API“ 5 Bände.

Die Bücher wurden von Wolfgang Soltendick herausgegeben und erschienen im Verlag C&L. Sprache: deutsch, Preis pro Band ca. Euro 50,-

Band 1 beschreibt „LZ32“, „ComCtl32“ und „Kernel32“. ISBN 3-932311-05-1
Band 2 beschreibt „SHELL32“, „COMDLG32“ und „OLE32“. ISBN 3-932311-36-1
Band 3 beschreibt „User32“ und „AdvAPI32“. ISBN 3-932311-37-X
Band 4 beschreibt „DiskCopy“, „GDI32“ und „MultiMedia“. ISBN 3-932311-52-2
Band 5 beschreibt ???

WIN32 Programmierung“ (API Bible) Win32 Programmierung (API Bible) von Richard J. Simon, deutsche Übersetzung von Judith Muhr. Komplette Referenz zu Windows 95 und NT. Alle APIs und Kontrollfunktionen mit Beispielen. Mit Programmbeispielen (nicht in Profan!) auf CD. Die Beispiele dürften in C++ geschrieben sein.. Sprache: deutsch. Ca. Euro 50,-
SAMS (Haar bei München) ISBN 3-8272-4502-8

Im 5-bändigen **Win32 API** werden zu allen Funktionen Beispiele in C++ und Delphi angeführt. Doch sehe ich diese Bücher nur als Referenz; und sehr umfangreich beschrieben.

Zum Erlernen ist aber die **API Bible** wahrscheinlich eher geeignet (und billiger).

Im **Win32 API** sind die Strukturen z.T. genauer beschrieben und oft sind nicht nur Labels sondern auch die zugehörigen Werte angeführt.

In der **Win32.hlp** sind anzugebende Werte nur als Label (symbolische Bezeichnung) aber nie mit dem anzugebenden Wert angeführt.

Mehr Informationen zu Windows-APIs gibt es auf meiner H.P.:

- Ein API Workshop. Grundlagen wie die Win32.hlp zu verstehen ist.

- API_Hilfe. Ein Programm das die in Win32.hlp angezeigten Labels und deren Werte anzeigt. Ist aber für Assembler nicht so wichtig, da in der Assemblersource auch die Labels eingetragen werden können.

Die hier erstellten Unterlagen sind sicher nicht vollständig, sie wurden aus meinen Erfahrungen (und Interpretationen der Beschreibungen) erstellt. Falls Fehler gefunden wurden bitte ich um Verständigung damit ich diese Unterlagen berichtigen kann.

Viel Erfog wünscht Gerhard Putschalka.