Delphi Tutorial

zu den Themen:

DLL-Funktionen importieren;
DLLs schreiben;
Aufrufkonventionen;
API/C-Header konvertieren;
Spezielle Delphi-Strukturen;
Import- & Export-Tabelle

geschrieben von:

-=ASSARBAD=-

Kontaktmöglichkeiten:

http://assarbad.net

DLL-Tutorial@assarbad.net

Alle Ausführungen beziehen sich auf Win32-Systeme ab Windows 95, NT4 respektive – anderenfalls sind entsprechende Stellen im Text extra für eine entsprechende Windows-Version ausgewiesen.
Die Beispiele können mit Delphi ab Version 3 und teils erst ab Version 4 nachvollzogen werden.

May the source be with you, stranger ...
Снижок это не только кефир, снижок это стиль жизни.

Version 1.10a [2003-10-05]

© 2001 – 2003 by -=Assarbad=-
Korrekturlesung (bis 1.08b): Mathias Simmack (http://www.simmack.de)

Dieses Tutorial darf ausdrücklich inhaltlich und layouttechnisch ungeändert in Form der Orginal-PDF-Datei zusammen mit den anderen im Archiv enthaltenen Dateien (Quelltexte) weitergegeben, sowie in elektronischer (z.B. Internet) und physischer Form (z.B. Papierdruck) verbreitet werden, solange die Kosten für das physische Medium seitens des Konsumenten den Wert von € 10,- (entsprechend dem Euro-Goldkurs vom 2002-10-22) nicht übersteigen. Im Internet hat die Weitergabe kostenlos zu erfolgen. Ein Downloadangebot hat immer in Form der Orginal-PDF-Datei zu erfolgen, nicht in einer umformatierten Variante.
Es ist eine gute Geste mir im Falle einer Veröffentlichung ein Referenzexemplar zukommen zu lassen bzw. mir die URL mitzuteilen, kontaktieren Sie mich dazu einfach per eMail um ggf. meine Postadresse zu erhalten.
Ausnahmen von obigen Regeln (Layoutveränderungen, Inhaltsänderungen) erteile ich gern nach Absprache, auch dazu kontaktieren Sie mich bitte per eMail mit Angabe des geplanten Veröffentlichungsortes. Das Tutorial steht auch als .sxw (Openoffice.org Writer Format) zur Verfügung und kann bei mir per eMail angefordert werden.

Explizit erteile ich folgenden Personen die Erlaubnis das Layout und Format (auch Dateiformat) dieses Tutorials für eine Onlineveröffentlichung auf ihren Homepages bzw. Community-Seiten sowie Tutorialsammlungen anzupassen:

Philipp Frenzel (www.delphi-treff.de),
Michael Puff (www.luckie-online.de),
Mathias Simmack (www.simmack.de),
Martin Strohal & Johannes Tränkle (www.delphi-source.de),
Ronny Purmann (www.faqsen.de)

Lizenzbedingungen

Das komplette Tutorial inklusive des Quellcodes unterliegt folgender, der BSD-Lizenz abgewandelter, Lizenzvereinbarung (Software und Sourcecode bezieht sich dabei auch auf die Textform des Tutorials):


```
                                    _\\|//_
                                   (` * * ')
_____________________________oooO_(_)_Ooo_____________________________
LEGAL STUFF:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Copyright (c) 1995-2002, -=Assarbad=- ["copyright holder(s)"]
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this
   list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.
3. The name(s) of the copyright holder(s) may not be used to endorse or
   promote products derived from this software without specific prior written
   permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY
DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
                               .oooO     Oooo.
_______________________________(   )_____(   )_____________________________
                                \ (       ) /
                                 \_)     (_/
```


Keine der hier zur Verfügung gestellten Informationen darf zu illegalen Zwecken eingesetzt werden, dies schließt sowohl den Text als auch den Quellcode ein.

Für die Links zu externen Ressourcen übernehme ich keinerlei Verantwortung. Zum Zeitpunkt der Erstellung dieses Schriftstücks, erschienen sie mir nützlich und sinnvoll und enthielten keinerlei erkennbare illegale Inhalte oder Tendenzen.

Die Lizenz ist nur deshalb nicht auf deutsch verfügbar, weil der englische Text als rechtlich verbindlich und korrekt gilt und ich mich nicht in der Lage fühle den Text in dieser Form zu übersetzen. Zumal unsere deutschen Advokaten und Paragraphenakrobaten ja im Sinne des Rechts und der Gleichheit ihre eigene Sprache erfunden haben, die kein anderer mehr zu verstehen hat – aber: „Alle Menschen sind vor dem Gesetz gleich.“ (Artikel 3 Abs. 1), nur die die sich einen Advokaten leisten können, sind gleicher – Danke für Euer Verständnis!

Vorwort

Der Leser sollte mit der Pascal / ObjectPascal[1]-Syntax vertraut sein und als Entwicklungswerkzeug vorzugsweise Delphi 4 oder später zur Verfügung haben. Außerdem wäre ein Ressourcen-Editor (WEDITRES, Visual C Standard)[2] äußerst nützlich und sinnvoll.

Eventuell werde ich dieses Tutorial noch für andere Pascaldialekte als Delphi anpassen.

Wer die Vorgänger dieses Tutorials kennt, der weiß: es hat sich stark verändert. Die alten Versionen waren zeitweise als CHM und hernach als HTML verfügbar. Im Sinne der Druckbarkeit des Dokuments, habe ich mich jedoch für das PDF-Format entschieden. Dies sollte auch denjenigen, die eigentlich nicht überall Internet verfügbar haben, die Möglichkeit geben, das Tutorial auch offline jederzeit lesen und drucken zu können. Für mich persönlich ist dies eine wichtige Eigenschaft eines jeden Dokumentes ;)

Format:
Zur Erstellung habe ich OpenOffice.org 1.1 verwendet. Die Konvertierung zu 2/1-PDF erfolgte mittels pdf-Factory Pro 2.0. Beide Programme kann ich nur empfehlen, auch wenn letzteres nur als Shareware und damit nicht-kostenlos verfügbar ist.

Danksagung:
Dank möchte ich zumal denen sagen, die mich bei der Entwicklung meiner Programme und auch meiner Tutorials durch Feedback und zum Teil auch Korrekturen so freizügig unterstützt und zu deren Verbesserung beigetragen haben.

Mein **spezieller Dank** geht dabei an folgende Personen:

Eugen Honeker,
Mathias Simmack,
Michael Puff,
Nico Bendlin,
Ronny Purmann,
„Steffer“,
Thomas Mueller (http://www.dummzeuch.de)

Nicht unerwähnt bleiben, soll der Einsatz von Mathias Simmack als Korrekturleser. Da er der bessere Rhetoriker ist, ist es schön sich seiner Hilfe gewiß sein zu dürfen.

Dank auch an all jene Künstler die einen bei solch kreativen Prozessen wie dem Programmieren durch ihre Musik immer wieder von Neuem inspirieren:

Wolfgang Amadeus Mozart, Antonio Vivaldi, Ludwig van Beethoven, Poeta Magica, Kurtzweyl, Krless, Sarbande, Vogelfrey, Nightwish, Manowar, Blind Guardian, Weltenbrand, In Extremo, Wolfsheim, Carl Orff, Veljanov, Lacrimosa, Finisterra, Enigma, Beautiful World, Adiemus, Земфира, Би-2

1 ObjectPascal ist der Oberbegriff für OOP-fähiges Pascal. Es gibt neben Delphi noch weitere.
2 Siehe Referenzen.

Was sind DLLs?

DLL steht für Dynamic Linked Library (dynamisch gelinkte Bibliothek).

Gemeinhin bezeichnet man in der IT als Bibliothek das, was eine Ansammlung wiederverwendbarer Funktionen, Objekten oder Variablen enthält. DLLs sind da keine Ausnahme - sie werden meist genutzt um Variablen und Funktionen zu exportieren, ActiveX-Kontrollelemente verfügbar zu machen[3] und nicht zuletzt auch um globale Windows-Hooks und prozeßübergreifendes API-Hooking zu implementieren.

Da DLLs stark davon abhängig sind, wie Windows den Speicher innerhalb der Prozesse verwaltet, folgt hier erst einmal ein kleiner Ausflug in die Interna des Windows Speichermanagers[4].

Die Windows Speicherverwaltung (NT-Plattform)

Unter Windows gibt es das, wovon viele Programmierer zu DOS-Zeiten noch geträumt haben – einen zusammenhängenden Speicherbereich von 4 GB Größe. Unter DOS gab es noch die Segmentierung in 64 kB-Stücke und teilweise Einschränkungen bezüglich der Verteilung von Daten und Code in diesen Segmenten.
Windows hat mit den ersten 32bit-Versionen den „Flat Memory Space“ (engl. für flacher/linearer Speicherbereich) eingeführt, in dem die Anordnung von Daten und Code praktisch irrelevant ist, und der mit 32bit Zeigern adressiert wird (= 2^{32}-1 Byte).
Außerdem steht diese Größe theoretisch jedem Prozeß zur Verfügung. Will heißen, jeder Prozeß „sieht“ insgesamt 4 GB Speicher die er nutzen kann[5].
Prozesse können nur mit bestimmten Berechtigungen, APIs und Methoden auf den Speicherbereich anderer Prozesse zugreifen. Einzig eine DLL kann in mehreren Prozessen quasi gleichzeitig laufen. Dies ist auch der Grund, warum ein globaler Windows-Hook immer in eine DLL ausgelagert werden muß – schließlich muß er ja in mehreren Prozessen quasi gleichzeitig laufen können.

DLLs sind ganz normale „ausführbare“ Dateien im PE-Format[6]. Einziger Unterschied ist genau genommen, daß sie nicht wirklich direkt ausgeführt werden können (z.B. durch Doppelklick im Windows Explorer) – stattdessen exportieren sie bestimmte Funktionen, von denen der Aufrufer (engl. „caller“) die Syntax kennen muß.
Da die Syntax irgendwoher bekannt sein muß, muß man sie also z.B. in Form einer Dokumentation o.ä. haben[7] oder selbst herausbekommen – zum Beispiel mit einem Disassembler oder Debugger.
Dies ist aber nicht die einzige Hürde. Wird eine DLL in den Speicherbereich eines Prozesses geladen, so existiert sie danach als Abbild innerhalb dieses Speicherbereiches und man kann vom eigenen Code aus zum Code der DLL „springen“ - gemeinhin wird dieser Prozeß als *Funktionsaufruf* bezeichnet.
Nun ist es aber auch so, daß man wissen muß wohin man springen muß um eine bestimmte Funktion aufzurufen. Dazu kann der Aufrufer die Export-Tabelle der entsprechenden aufgerufenen DLL (engl. „callee“) auswerten. Oder der Aufrufer kennt die entsprechenden Adressen schon anhand seiner Import-Tabelle, welche der Image Loader des Betriebssytems füllt, oder aber er holt sich letztlich mit Hilfe der angebotenen Kernel32-Funktion[8] `GetProcAddress()` die Adresse der aufzurufenden Funktion[9].

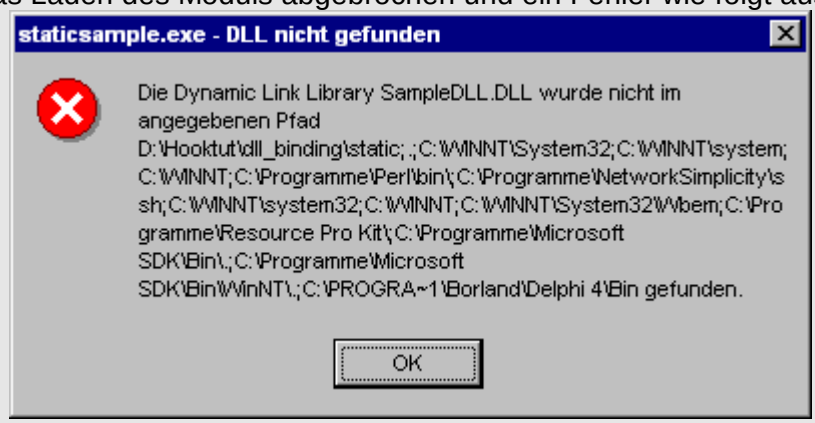
3 DLLs welche ActiveX-Kontrollelemente enthalten, werden gemeinhin mit der Endung OCX anzutreffen sein.
4 Es wird hier nur Windows NT bzw. die NT-Plattform behandelt. Erstens, da die Consumer-Windows-Versionen 95, 98 und Me längst veraltete Technologien sind, welche noch auf DOS aufsetzen. Zweitens, da dort doch einiges ziemlich anders ist und drittens, da die NT-Plattform die zukünftigen Windows-Versionen formen wird.
5 Obwohl es da auf NT noch Einschränkungen gibt. Normalerweise sind 2 GB davon für den Kernel reserviert und der Rest für den Benutzermodus. Will heißen ein einzelner Prozeß „sieht“ 2 GB. PAE (Physical Address Extension, ein Intel-Feature) wird hier nicht besprochen oder beachtet!
6 PE steht für Portable Executable (engl. für „portierbare ausführbare Datei“).
7 Microsoft bietet dazu das Platform SDK (PSDK) mit C-Header-Dateien und HTML-Hilfe.
8 Kernel32.dll ist eine Systembibliothek die unter allen 32bit Varianten zur Verfügung steht.
9 Mehr dazu (Import-, Export-Tabelle und GetProcAddress) auf den nächsten Seiten.

Betrachten wir die Möglichkeiten des Imports von DLL-Funktionen einmal näher:

1. Auswertung der Export-Tabelle
   Hierzu muß der Aufrufer die Struktur einer PE-Datei kennen und auswerten können. Dies ist die am seltensten benutzte Variante. Sie wird bevorzugt von Assembler-Programmierern, und unter diesen besonders gern von Viren-Programmierern, benutzt.
   Wenn Du nähere Informationen dazu suchst, empfehle ich z.B. die Seite von Iczelion[10].
2. Import-Tabelle der eigenen Moduldatei (EXE)
   Werden DLLs *statisch* eingebunden („linked"), so enthält das entstandene Modul eine sogenannte Importtabelle. Diese Importtabelle enthält Sprünge zu noch nicht festgelegten Adressen von Funktionen, deren Name allein bekannt ist. Der Image Loader[11] des Betriebssystems füllt diese Tabelle beim Laden des Moduls mit den entsprechenden Adressen. Dazu bedient er sich eines ähnlichen Mechanismus' wie er auch Programmierern mit der Funktion `GetProcAddress()` zur Verfügung steht.
3. Ermitteln der Adressen mithilfe von `GetProcAddress()`
   Lädt man eine DLL dynamisch („runtime dynamic linking"), so muß man der Funktion `GetProcAddress()` das Handle zum geladenen Modul übergeben, sowie den Namen[12] der gewünschten Funktion angeben. Gibt es die gewünschte Funktion nicht, so wird NIL (aka „Null") zurückgegeben, und das Programm kann Schritte zur Fehlerbehandlung einleiten.

Was macht der Image Loader genau

Der Image Loader mappt (mappen: eingedeutscht von engl. abbilden) das Modul in den Speicher und initialisiert verschiedene Strukturen (TLS, PEB usw.). Unter anderem auch die Import-Tabelle, welche, wie schon gesagt, anfangs nur leere Adressen enthält. Dabei wird anhand des Namens jeder Funktion deren Adresse in der entsprechenden DLL ermittelt. Gibt es einmal keine solche Funktion, wird das Laden des Moduls abgebrochen und ein Fehler wie folgt ausgegeben:



Es handelt sich dabei um eine System-Fehlermeldung (Kann anhand des Fensterhandles nachgewiesen werden). Ansonsten springt der Loader danach an den Einsprungspunkt (entry point) der EXE-Datei oder im Falle einer DLL zu deren `DLLMain()`-Funktion.

Hier sehen wir nun auch schon den ersten großen Nachteil des statischen Ladens einer DLL: es kann keine Fehlerbehandlung seitens des zu ladenden Moduls erfolgen. Gerade bei Anwendungen, welche APIs (ToolHelp API[13]) verwenden, die auf einem System verfügbar sind (Windows 2000 aufwärts und ab Windows 95), aber auf einem anderen System nicht existieren (Windows NT 4.0), *verbietet sich die Benutzung der statischen Einbindung*! Hier muß auf dynamische Einbindung zurückgegriffen werden.

Alle, die sich für die Auswertung der Import-Tabelle, und der Export-Tabelle einer DLL interessieren, möchte ich auf Appendix E verweisen.

10 Siehe Referenzen.
11 Die Instanz, welche EXE-Dateien und andere Modultypen initialisieren und laden kann. Unter NT beginnen die Funktionen, welcher sich der Loader bedient mit „Ldr".
12 Der Name kann auch eine in PChar gecastete Zahl zwischen 0 und $FFFF sein (high order word := 0). Ich benutze der Anschaulichkeit halber erst einmal nur Funktionen, die per Namen exportiert werden.
13 Ermöglicht u.a. das Auflisten von Prozessen und so weiter, existiert aber nicht für Windows NT 4.0!

Kommen wir nun zu zwei kleinen Beispielanwendungen, die eine Beispiel-DLL verwenden. Alle drei Quellen werden entwickelt und dabei kurz erklärt. Es empfiehlt sich an dieser Stelle in das Projektverzeichnis `.\SOURCE\01_DLL` zu wechseln und am besten mit jeweils einer Instanz von Delphi jedes Projekt zu öffnen.

Beispiel: Simple DLL ohne VCL und ohne DLLMain()

Als erstes wenden wir uns einmal der Beispiel-DLL zu, da sie ja von beiden Beispielanwendungen importiert (gelinkt) werden soll. Als Anmerkung: Diese DLL ist noch nicht abhängig von irgendwelchen delphi-spezifischen Komponenten. Dazu später mehr.

In Delphi deklarieren wir eine DLL (und auch ActiveX-Module) über das Schlüsselwort `library`.

```
library SampleDLL;
uses
  windows;

{$INCLUDE ..\dlgres\compilerswitches.pas}
{$INCLUDE ..\dlgres\dlg_consts.pas}

var hwnd: Cardinal = 0;

.
.
.

exports
  OneFunction,
  OneFunction_CDECL index 2,
  OneFunction_STDCALL_ index 3 name 'OneFunction_STDCALL',
  initDLL;

end.
```

Anhand des Quelltextes ist schon ersichtlich, daß der Aufbau sich nicht großartig von dem eines normalen Projektes (für eine EXE) unterscheidet. Am wichtigsten ist sicherlich die `exports`-Direktive, welche den Compiler anweist die entsprechenden Funktionen in der Exporttabelle aufzuführen. In unserem Beispiel, werden die Funktionen wie folgt exportiert:

1. '`OneFunction`' und '`initDLL`' behalten ihren Namen, der Index ist noch nicht bekannt.
2. '`OneFunction_CDECL`' behält den Namen und bekommt den Index 2.
3. '`OneFunction_STDCALL_`' wird zu '`OneFunction_STDCALL`' und mit dem Index 3.

Es gibt aber noch eine weitere Möglichkeit, sowie deren Kombination mit ersterer:

Wie kann man DLL-Funktionen exportieren (anhand von Delphi)

Wie schon erwähnt, kann eine Funktion einen Namen oder eine Zahl von 0 bis $FFFF haben (Index), mit denen man sie ansprechen kann. Üblicherweise haben diejenigen mit Namen auch einen Index, aber umgekehrt ist dies nicht immer der Fall. Da der Name oft Aufschluß über die Arbeitsweise oder Funktionalität einer Funktion geben kann, verwenden manche Entwickler die Methode des Exports allein über den Index als eine Art des Cracking-Schutzes. In Delphi ist dies auch möglich indem man einen Index und einen leeren Namen zuweist.
Die Delphi-Hilfe gibt folgendes Beispiel:

```
  DoSomethingABC  index 1 name 'DoSomething';
```

Wie man sehen kann, wird die Funktion, welche innerhalb des DLL-Projektes als `DoSomethingABC()` angesprochen wird, unter dem anderen Namen `DoSomething()` exportiert und erhält den Index 1.
Verteilt man keine Indexnummern, so wird in der Export-Tabelle die zuletzt aufgeführte Funktion als erstes auftauchen (delphispezifisch und eigentlich irrelevant). Zum Anschauen der Export-Tabelle und allgemein der Abhängigkeiten von Modulen, empfehle ich den Dependency Walker[14]. Pumi erwähnte noch die Möglichkeit TDUMP zu nutzen: `tdump sampledll.dll c:\dmpout.txt`

Schauen wir uns unsere DLL nun im Dependency Walker an, bekommen wir obige Annahmen bestätigt:

E	Ordinal ^	Hint	Function	Entry Point
C	1 (0x0001)	3 (0x0003)	initDLL	0x00003D30
C	2 (0x0002)	1 (0x0001)	OneFunction_CDECL	0x00003CB4
C	3 (0x0003)	2 (0x0002)	OneFunction_STDCALL	0x00003CF0
C	4 (0x0004)	0 (0x0000)	OneFunction	0x00003C88

Beispiel: Statischer Import von DLL-Funktionen

Da der statische Import häufiger verwendet wird, und auch leichter zu implementieren ist, soll er hier als erstes Beispiel gebracht werden.
Der statische Import muß also, wie wir schon wissen, noch vor der eigentlichen Ausführung der EXE-Datei (bzw. des Moduls) geschehen. Der Compiler muß dabei schon die Import-Tabelle anlegen und die Namen der Funktionen, sowie den Namen der exportierenden DLL kennen.
Dies erreicht man durch die `external`-Direktive. Sie bindet eine Funktion ein. Auch dabei gibt es wieder verschiedene Möglichkeiten – aber schauen wir uns zuallererst einmal den entsprechenden Quelltext an:

```
function OneFunction(param1, param2, param3: Cardinal): integer;
  external 'SampleDLL.DLL';
function OneFunction_CDECL(param1, param2, param3: Cardinal): integer; cdecl;
  external 'SampleDLL.DLL' index 2;
function OneFunction_STDCALL_(param1, param2, param3: Cardinal): integer;
  stdcall; external 'SampleDLL.DLL' name 'OneFunction_STDCALL';
```

Wir importieren die Funktionen erst einmal so wie sie exportiert wurden – inklusive der korrekten Aufrufkonvention[15] und Indizes bzw. Namen. Selbst '`OneFunction_STDCALL`' wird nun wieder so importiert, daß wir sie im Quellcode wieder als '`OneFunction_STDCALL_`' ansprechen können. Es ist also möglich innerhalb des Hauptprogrammes (im Quelltext) einen anderen symbolischen Namen für Funktionen zu vergeben als die exportierte Funktion hat. Dies ist z.B. in den Win32 API Headern wichtig um die Unicode- und Ansiversionen von Funktionen standardmäßig zuzuweisen. So existiert bspw. von "MessageBox" eigentlich die Ansivariante "MessageBoxA" und die Unicodevariante (Wide) "MessageBoxW". Standardmäßig wird bei Delphi die Ansiversion dem Namen "MessageBox" zugewiesen. Mit den meisten API-Funktionen passiert dies so. Für Beispiele schau Dir bitte die `Windows.PAS` an. In C(++) wird üblicherweise ein Makro („UNICODE“) benutzt, um dem symbolischen Namen der Funktion entweder die Unicode- oder die Ansi-Variante zuzuweisen.
Man sieht sehr schön, daß die Funktionen wie gewohnt deklariert werden, nur daß statt des Funktionskörpers (`begin end;`) eine `external`-Deklaration folgt - ganz ähnlich den wahrscheinlich schon bekannten `forward`-Deklarationen[16]. `external` bekommt als „Parameter“ den Namen der exportierenden DLL übergeben, sowie optional den Namen oder Index der zu importierenden Funktion. Davor steht noch (jedoch nach der Parameterdeklaration) die Aufrufkonvention. Standardmäßig wird die Konvention `register` verwendet, also immer dann wenn keine andere Konvention angegeben wird (Anm.: Heißt bei mir im Beispielprogrammen auf den Buttons „PASCAL“!).

14 Siehe Referenzen.
15 Weiter unten folgt noch eine ausführlichere Besprechung von Aufrufkonventionen.
16 Siehe Delphi-Hilfe und ObjectPascal-Dokumentation.

Grundsätzlich gibt es keine Unterschiede zwischen der Deklaration von Funktion und Prozedur beim Import oder auch Export. Einzig die Adresse ist später relevant.
Aus C(++) kennt man ja die Tatsache, daß `VOID`, also nichts, „zurückgegeben“ wird, statt eine Funktion ausdrücklich als Prozedur zu deklarieren. Das ist zwar nicht rein delphispezifisch – denn andere Sprachen, wie zum Beispiel Visual Basic kennen auch diesen Unterschied. Gerechtfertigt ist die Unterscheidung hingegen nicht wirklich.

Da ich schon kurz von Aufrufkonventionen gesprochen habe und dieser Begriff ja auch im Zusammenhang mit DLLs nicht ganz unwichtig ist, hier eine kleine Erklärung:

Aufrufdeklarationen in Delphi

Aufrufkonvention	Beschreibung
`register`	Standardkonvention für Funktionen in Delphi, wenn nicht explizit anders deklariert. Dabei werden die Parameter wie bei `pascal` in der Reihenfolge ihrer Deklaration zuerst in die Register EAX, EDX, ECX und eventuell verbleibende auf den Stack geschoben. Parameter, die nicht in Register geschoben werden, sind Real-Typen und Methoden-Zeiger.
`pascal`	Existiert eigentlich nur für Abwärtskompatibilität und wurde meines Wissens nach von Windows 3.x (16bit) standardmäßig benutzt. Hier wird nichts in Register sondern alles auf den Stack geschoben.
`cdecl`	Diese Deklaration ist eigentlich für den Import von C(++) Funktionen aus DLLs da. Das besondere ist, daß eine solche Funktion eigentlich eine beliebige Anzahl von Parametern entgegennehmen kann, da der Aufrufer (i.e. Compiler/Linker) verantwortlich ist, die Parameter auf dem Stack zu platzieren. Unter Delphi ist es mit der variablen Parameteranzahl dann aber auch schon vorbei[17]. Es soll nur als Hinweis dienen. Parameter werden entgegengesetzt ihrer Deklaration auf den Stack geschoben.
`stdcall`	Diese Aufrufkonvention wird sowohl von (fast?) allen Win32-API-Funktionen, als auch von der NT-Native API verwendet. Auch VB/VBA kennt diese Aufrufkonvention (und zwar als einzige der hier aufgeführten!). Sie ist also durchaus für eigene DLLs zu empfehlen. Die Parameterübergabe erfolgt ebenfalls entgegengesetzt der Reihenfolge ihrer Deklaration auf den Stack.
`safecall`	Diese Konvention wird fast ausschließlich von Interface-Methoden bei der ActiveX-Programmierung verwandt. Im Großen und Ganzen entspricht sie am ehesten `stdcall`. Näheres ist mir nicht bekannt.

Der große Unterschied zwischen den Aufrufkonventionen liegt im Endeffekt darin, daß sie zum Beispiel String-Parameter verschieden behandeln. Gleiches gilt für `out`, `var` und `const` Parameter[18], die ebenfalls von den verschiedenen Aufrufkonventionen verschieden behandelt werden.

Das enthaltene Beispielprogramm zeigt einmal kurz, wie es sich auswirken kann, wenn man die Aufrufkonventionen nicht beachtet. Einfach mal etwas rumspielen, aber bitte beachten, daß unter Windows 9x durchaus ein Systemabsturz anstehen könnte – also bitte alle Daten vorher sichern. Unter Windows NT sollte ein Absturz des Systems so gut wie ausgeschlossen sein - außer dem Programm selbst sollte dort nichts anderes abstürzen.

17 Ich weiß, daß ab Delphi 6 oder Delpi 7 der so genannte Ellipsis-Operator (wie unter C/C++) zur Verfügung steht, um eine variable Anzahl von Parametern zu unterstützen.
18 Sehr empfehlenswert zum Thema ist die Delphi-Hilfe ;)

Beispiel: Dynamischer Import von DLL-Funktionen

Der dynamische Import funktioniert, wie schon gesagt, über das Holen der Adresse der zu importierenden Funktion durch den Aufrufer.
Zuallererst muß der Aufrufer aber auch die Syntax der Funktion bekannt gemacht bekommen. Auch wenn ein Funktionsaufruf „untendrunter“ quasi nur ein Sprung an eine bestimmte Adresse ist, kann man ja in seinem Programm nicht wild an irgendeine Adresse springen lassen – wie sollen da die Parameter übergeben werden? Es muß also ein sogenannter *Funktionsprototyp* her. In C(++) sind die Funktionsprototypen meist in einer Header-Datei[19] aufgeführt. Dies ist auch der Grund warum diese vorher nach Delphi (ObjectPascal) übersetzt werden müssen (siehe Appendix A). Ähnlich wie in C(++) ist die Funktionsprototypen-Deklaration in Delphi auch nur eine normale Typendeklaration:

```
type
  TFNOneFunction = function(param1, param2, param3: Cardinal): integer;
  TFNOneFunction_CDECL = function(param1, param2, param3: Cardinal): integer;
    cdecl;
  TFNOneFunction_STDCALL = function(param1, param2, param3: Cardinal): integer;
    stdcall;
```

TFN ist übrigens in Delphi eine offiziell anerkannte Notation[20] für Funktionstypen.
Da wir die Typen deklariert haben, brauchen wir nun noch die entsprechenden Variablen:

```
var
  OneFunction: TFNOneFunction = nil;
  OneFunction_CDECL: TFNOneFunction_CDECL = nil;
  OneFunction_STDCALL_: TFNOneFunction_STDCALL = nil;
```

Die Variablen müssen global deklariert sein um sie mit `nil` initialisieren zu können, denn Delphi erlaubt das Vorinitialisieren von Variablen nur im globalen Kontext. Nun müssen wir uns in einer, ich schlage vor getrennten Funktion, die Adressen oder sogenannten Eintrittspunkte der Funktionen holen:

```
Procedure GetEntryPoints;
var
  lib:THandle;
begin
  lib := LoadLibrary(@szNameDLL[1]);
  case lib = 0 of
    TRUE:
      begin
        @OneFunction_CDECL := @whatifnoentry;
        @OneFunction := @whatifnoentry;
        @OneFunction_STDCALL_ := @whatifnoentry;
        messagebox(0, @dll_notloaded[1], nil, 0);
      end;
    else
    begin
      @OneFunction := GetProcAddress(lib, @szNameOneFunction[1]);
      if not Assigned(OneFunction) then @OneFunction := @whatifnoentry;
      @OneFunction_CDECL := GetProcAddress(lib, @szNameOneFunction_CDECL[1]);
      if not Assigned(OneFunction_CDECL) then @OneFunction_CDECL :=
        @whatifnoentry;
      @OneFunction_STDCALL_ := GetProcAddress(lib, @szNameOneFunction_STDCALL[1]);
      if not Assigned(OneFunction_STDCALL_) then @OneFunction_STDCALL_ :=
        @whatifnoentry;
    end;
  end;
```

Man kann das Handle durch einen Aufruf von `FreeLibrary()` freigeben. Dies wird aber auch von Windows erledigt, wenn der Prozeß beendet wird.

19 Headerdateien deklarieren Typen- und Funktionsprototypen.
20 In C wird meist PF oder PFN als Präfix benutzt. Sie ist keinesfalls verpflichtend!

Was passiert, wenn eine DLL dynamisch geladen wird?

Wird eine DLL geladen, so wird als erstes überprüft, ob sie schon einmal in den Prozeß geladen wurde. Ist dies der Fall, wird ein sogenannter Referenzzähler um den Wert 1 erhöht. Ist die DLL noch nicht geladen, so wird sie geladen. Dabei werden die Abhängigkeiten zu anderen DLLs aufgelöst und diese gegebenenfalls auch geladen. Danach wird der Referenzzähler auf 1 gesetzt.
Was wir daraus lernen ist, dass es zu jedem `LoadLibrary()` bzw `LoadLibraryEx()` auch einen Aufruf von FreeLibrary geben muß.

Unter Win16 (Windows 3.1 etc) hatte das sogenannte Modulhandle, welches nun von `LoadLibrary()` zurückgegeben wird, noch eine andere Bedeutung, es war ein echtes Instanzenhandle welches auch Aussagen über eine schon laufende Instanz der gleichen Anwendung zuließ. Dies ist unter Win32 nicht mehr so, da jeder Prozeß seinen eigenen Speicherraum hat. Der Begriff Instanzenhandle hat sich bis heute gehalten. Bei EXE-Dateien (im Portable Executable Format) ist die Adresse, an der die EXE geladen wurde, durch das Modul-/Instanzenhandle ersichtlich. Meist handelt es sich um die Adresse $400000 im Speicherbereich des Prozesses. Hinzugeladene DLLs bekommen dann jeweils andere Adressen, aber das Modul-/Instanzenhandle dieser DLLs zeigt auch deren Adresse an. Dies ermöglicht z.B. auch den Trick, mit welchem man die Export-Tabelle auswertet (Methode 1 auf Seite 5). Um niemanden hier hängenzulassen, will ich eine kleine Kostprobe geben, obwohl solche Cracks wie Iczelion dies in Assembler noch eleganter hinbekommen.

```
uses
  ImageHlp;

procedure LoadedDLLExportsFunc(aFileName: String; aList: TStrings);
type
  PDWORDArray = ^TDWORDArray;
  TDWORDArray = array[0..0] of DWORD;
var
  imageinfo: LoadedImage;
  pExportDirectory: PImageExportDirectory;
  dirsize: Cardinal;
  pDummy: PImageSectionHeader;
  i: Cardinal;
  pNameRVAs: PDWORDArray;
  Name: string;
begin
  imageinfo.MappedAddress := PChar(GetModuleHandle(@aFileName[1]));
  if Assigned(imageinfo.MappedAddress) then
  try
    imageinfo.FileHeader := ImageNtHeader(imageinfo.MappedAddress);
    pExportDirectory := ImageDirectoryEntryToData(imageinfo.MappedAddress,
      True, IMAGE_DIRECTORY_ENTRY_EXPORT, dirsize);
    if (pExportDirectory <> nil) then
    begin
      try
        pNameRVAs := PDWORDArray(PChar(imageinfo.MappedAddress) +       DWORD
(pExportDirectory^.AddressOfNames));
      except
        aList.Add('ERROR: #' + IntToStr(GetLastError));
      end;
      for i := 0 to pExportDirectory^.NumberOfNames - 1 do
        aList.Add(PChar(imageinfo.MappedAddress) + pNameRVAs^[i]);
    end;
  finally
  end;
end;
```

In der gelb markierten Zeile sehen wir sehr schön, wie das Modulhandle in einen Pointertyp gecastet wird. Der Rest danach ist nur noch Kenntnis der Strukturen. Hier möchte ich dann wirklich nochmals an Iczelion und das Platform SDK verweisen. Nicht die Unit einzubinden vergessen (grün markiert). Das Beispiel habe ich nur von Tino's (Delphi-Forum) Beispiel abgewandelt.

Weiter im Text. Wie wir schon wissen, werden die Namen der Funktionen und der Name der DLL (ggf. mit Pfad) benötigt. Wir müssen sie also vorher wie folgt deklariert haben:

```
const
  szNameOneFunction = 'OneFunction';
  szNameOneFunction_CDECL = 'OneFunction_CDECL';
  szNameOneFunction_STDCALL = 'OneFunction_STDCALL';
  szNameDLL = 'SampleDLL.DLL';
```

Außerdem fällt bei genauerem Hinschauen auf, daß die Adresse einer Funktion `WhatIfNoEntry` auftaucht. Dies ist eine einfach konzipierte Dummy-Funktion, die dann einspringt, wenn die Adresse der Funktion in der DLL nicht gefunden wurden:

```
Function WhatIfNoEntry: Integer;
begin
  Messagebox(hdlg, @noentry[1], nil, 0);
  Halt;
end;
```

Das Programm wird damit beendet, da aufgrund verschiedener Aufrufkonventionen und Parameteranzahl nicht garantiert werden kann, dass der Stack danach noch sauber ist.

Nun schauen wir uns an, was unsere `GetEntryPoints`-Funktion genau macht:

- Zuallererst wird die DLL mit `LoadLibrary()` geladen.
- Im Erfolgsfall (`lib<>0`) geht es weiter, indem wir mit `GetProcAddress()` die einzelnen Eintrittspunkte für die Funktionen ermitteln. Wird dabei eine Adresse nicht gefunden:
  `not Assigned(fn)=True`,
  dann setzen wir stattdessen unsere Dummy-Funktion (`WhatIfNoEntry`).
- Schlug schon das Laden der DLL fehl, so setzen wir für alle importierten Funktionen die Adresse unserer Dummy-Funktion und geben eine Fehlermeldung aus (`Messagebox`).

Wie erwähnt macht es der Image-Loader des Betriebssystems nicht viel anders, aber er bricht den Ladevorgang komplett ab, sobald ein Fehler auftritt. Wir machen hingegen mit der Dummy-Funktion weiter und geben nebenbei eine Fehlermeldung aus, sollte schon das Laden der DLL fehlgeschlagen sein.
So funktioniert das Programm notfalls auch ohne die DLL – nur eben zum Beispiel mit eingeschränkten Funktionen. Der statische Import hätte schon das Laden vereitelt.

Um ein entsprechendes Programm zu schreiben, welches z.B. auf Windows NT 4.0 Prozesse mit Hilfe der NT Native API auflistet, aber das gleiche auch auf Windows 95 leisten soll (wo es natürlich keine NT Native API gibt), ist der dynamische Import der relevanten Funktionen unabdingbar. Allerdings muß – oder besser sollte – das Programm den Code, der sich unterscheidet, in einer eigenen programminternen Funktions-Sammlung zusammenfassen. Diese würde dann immer direkt vom Hauptprogramm aufgerufen, und das Hauptprogramm muß sich nicht mehr um Dinge wie Kompatibilitätsprüfung und Codeverzweigung kümmern (i.e. Transparenz).
Solche generischen Funktionen, die andere Funktionen generischer kapseln, nennt man Wrapper[21].

Ein Beispiel, welches bei Delphi in Form einer Unit vorliegt, ist die konvertierte ToolHelp API in der `tlhelp32.pas`. Alle dort implementierten Funktionen überprüfen vor dem Aufruf der eigentlichen API-Funktion, ob diese erfolgreich importiert werden konnte.

Ein weiteres, ganz konkretes Beispiel ist die Verwendung der so beliebten API RegisterServiceProcess(), die nur auf Consumer-Windows-Systemen zu Verfügung steht, und das Programm so unter NT/2000 unweigerlich abschmieren läßt.
In Fragen wie „Wie kann ich das Programm vor dem Taskmanager (Strg+Alt+Entf) verstecken?“, taucht diese Funktion immer wieder auf – jedoch meist ohne den geringsten Hinweis auf die Tatsache, daß sie nur auf Windows 95/98/Me verfügbar und zudem noch undokumentiert ist[22].

21 Wrapper, Funktionswrapper. Engl. „to wrap“ bedeutet einschlagen, einwickeln.

22 Unbekannt ist den meisten auch die Tatsache, daß der Trick mit dem Verhindern von Strg+Alt+Entf auch auf Consumer-Windows beschränkt ist. Auch wenn der Aufruf von `SystemParametersInfo()` auf Windows NT und 2000 zumindest keinen Schaden anrichtet.

Nachbemerkungen und Schlußfolgerungen zu den Beispielen

Es ist wichtig anzumerken, daß die Beispiele ohne Verwendung der VCL entstanden. Bei den DLLs wäre dies ohnehin nicht notwendig gewesen. Wie auch immer, ich habe der Einfachheit halber ein VCL-Beispiel zum Selbst-Kompilieren beigelegt, welches später mit den VCL-typischen Eigenheiten etwas ausführlicher besprochen werden soll.

Zuallererst rufen wir uns aber einmal wieder ins Gedächtnis, was wir weiter oben gelernt haben:
1. DLLs sind da um Funktionen, Variablen, Objekte wiederzuverwenden. Wiederverwenden schließt andere Programmiersprachen und Programme an sich ein.
2. Aufrufkonventionen können über Absturz oder Ablauf entscheiden.
3. Parameter und Syntax allgemein müssen dem Aufrufer bekannt sein.

Der erste und zweite Punkt machen schon einmal klar, daß wir uns zuallererst Gedanken machen müssen, was die DLL leisten können soll. Um eine DLL zum Beispiel mit anderen Programmiersprachen zu verwenden, müssen wir sogar die Strukturen, welche wir in Delphi als Typen vorgegeben haben, bis aufs genaueste kennen. Gehen wir das einmal kurz in einem Vergleich von Delphi mit C und VB durch[23]:

Delphi	C (am Beispiel Visual C 6.0)	Visual Basic (6.0)
`String`, `AnsiString`[24], `WideString`	Kann deklariert werden wenn Struktur bekannt. Eigentlich heißt es aber, die Unit `ShareMem.PAS` sollte eingebunden werden. Unter C existiert die natürlich nicht.	Keine Chance. VB ist zwar nicht typensicher, aber mit dem Referenzzähler und dem Längendoppelwort kann es nichts anfangen.
`PChar`, `Char`	Bekannt als Cstring. Andere API-typische Deklarationen wären `LPCSTR`, `LPCTSTR` (Ansi). `CHAR` existiert auch.	Das versteht sogar VB.
`PWideChar`, `WideChar`	`WideChar` existiert als `WCHAR` und `PWideChar` gibt es unter verschiedenen Namen. Z.B. `LPCWSTR`, `LPCTSTR` (Unicode)	Da `PWideChar` dem OLE String entspricht, kennt VB diesen natürlich ebenfalls.
Diverse API-Strukturen	Die meisten sind bekannt.	Viele bekannt oder übersetzbar, bei vielen allerdings unmöglich, denn Pointer sind VB nicht bekannt und es ist nicht immer möglich einen Workaround hinzubekommen.
`String[..]`, `ShortString` (Pascal-Strings)	Kann als Array von `CHAR` angesprochen werden.	Wenn als Pointer übergeben und mit Null abgeschlossen, sollte auch das gehen ;)
`vari:Array[2..5] of Byte`	`BYTE vari[4]` (beginnt aber bei Index 0!!!)	Möglich.
`stdcall`	Bekannt: `WINAPI` bzw. `PASCAL`	Einzige bekannte Aufrufkonvention.
`cdecl`	Bekannt. Angabe nicht explizit nötig.	Keine Chance. Ein Workaround mit Einbindung über eine DLL oder Ähnlichem ist denkbar!
`register`	Nicht vorhanden.	
Andere Aufrufkonventionen	Ich kenne nur fastcall als weitere Konvention[25].	
Nicht bekannt. Aber mit Tricks möglich (`asm`).	`fastcall`	

23 Eine weitere Tabelle findet sich in Appendix A. Dort sind verschiedene API-Typen (aus C/C++) kurz mit Delphi-Entsprechung aufgeführt.

24 Stringtypen in Delphi werden intern verwaltet und basieren auf enger Zusammenarbeit der Systemunits mit dem Compiler. Eigentlich stehen vor dem String noch ein Referenzzähler und ein Längendoppelwort.

25 Das will aber nichts heißen. C ist nicht meine „Muttersprache“ bei den Programmiersprachen. `naked` würde ich kaum als Aufrufkonvention bezeichnen wollen.

Wie wir in der Tabelle sehen, ist es vielfach nicht möglich ObjectPascal-Strukturen/-typen nach VB oder VBA zu übertragen. Und gerade die Aufrufkonventionen stellen ein großes Hindernis in Bezug auf VB/VBA-Kompatibilität dar. Konventionen wie z.B. `register` verbieten sich schon von selbst. Bei Delphi nach C und umgekehrt sieht das ganze schon weniger kritisch aus. Dennoch sind auch hier folgende Dinge zu beachten:

- `String`, `AnsiString` und `WideString` sollten generell vermieden werden! Stattdessen solltest Du für `String` und `AnsiString` ein `PChar` (`PAnsiChar`) und für `WideString` ein `PWideChar` verwenden. In der Standardeinstellung entsprechen sich `String` und `AnsiString`.
  Die Delphi-Stringtypen können durch Typecasten in einen Pointertyp verwandelt und durch die eingebaute „Compiler-Magic“[26] einfach an Funktionen übergeben werden, welche eigentlich einen Pointer auf ein entsprechend breites Zeichen (`Char`/`WideChar`) verlangen.
- `stdcall` sollte (unter Windows) den anderen Aufrufkonventionen immer vorgezogen werden um weitestgehende Kompatibilität zu gewährleisten.

Der dritte Punkt besagt nichts anderes, als daß wir die Prototypen in einer Unit oder einer Headerdatei bekannt machen müssen. Eine Unit reicht vollkommen aus. Ein Programmierer einer beliebigen anderen Pascal- oder C-nahen Sprache sollte so in der Lage sein, die Syntax für sich zu nutzen.

Der Unterschied zwischen beiden Projekttypen (statisch/dynamisch) ist in meinem Beispiel mit der Compilerdirektive `{$DEFINE STATIC}` festgelegt. Dies ist eine selbstdefinierte Direktive!.

Die Besprechung über den Import von DLL-Funktionen ist hiermit abgeschlossen.

Strings als Parameter / ShareMem / DLLMain()

Die Schlüsselworte `library` und `export` haben wir schon weiter oben besprochen, weshalb ich hier an dieser Stelle nur darauf verweise.

Strings und ShareMem

Erstellt man mit dem Assistenten[27] aus Delphi 3 eine neue DLL, so findet man folgendes vor:

```
library Project1;

{ Wichtiger Hinweis zur DLL-Speicherverwaltung: ShareMem muß  die
  erste Unit im Uses-Anweisungsteil des Interface-Abschnitts Ihrer
  Unit sein, wenn Ihre DLL Prozeduren oder Funktionen exportiert, die
  String-Parameter oder Funktionsergebnisse übergeben. Dies gilt für
  alle Strings die an und von Ihrer DLL übergeben werden --  selbst
  für diese, die in Records oder Klassen verschachtelt sind. ShareMem
  ist die Schnittstellen-Unit zur DELPHIMM.DLL, welche Sie mit Ihrer
  DLL weitergeben müssen. Um die Verwendung von DELPHIMM.DLL zu
  vermeiden, übergeben Sie String-Parameter unter Verwendung von
  PChar- oder ShortString-Parametern. }

uses
  SysUtils,
  Classes;

begin
end.
```

26 Compiler-Magic wird das Ganze von Borland selbst bezeichnet. Gemeint sind verschiedene Tricks die z.B. String und PChar „kompatibel“ machen, ohne daß man sie typecasten müßte.

27 Gemeint ist das, was erscheint wenn man Datei/Neu auswählt.

Aha, wir brauchen also eine gewisse `ShareMem.PAS` in der `Uses`-Klausel. Hmm, aber warum an erster Stelle? Nun, das ist relativ einfach zu erklären. Die `ShareMem.PAS` implementiert einen eigenen Delphi-Speichermanager, der sich von dem standardmäßig verwendeten unterscheidet. Da dieser Speichermanager aber in der Unit-Initialisierung aufgerufen wird, muß die Initialisierung von `ShareMem.PAS` vor allen anderen Units erfolgen. Deshalb hat sie an erster Stelle zu stehen.

Nachdem wir geklärt haben, was `Sharemem.PAS` grob gesehen macht, schauen wir uns das Ganze mal näher und anhand der Quellen von ShareMem an.
ShareMem verwendet eine DLL, die Du Deinem Programm, wenn es ShareMem verwendet, auch beilegen mußt[28]:

```
(* Delphi 3 *)
const
  DelphiMM = 'delphimm.dll';

(* Delphi 4 *)
const
  DelphiMM = 'borlndmm.dll';
```

Um eine DLL zu „verwenden“, muß man üblicherweise zumindest einige ihrer Funktionen importieren:

```
function SysGetMem(Size: Integer): Pointer; external DelphiMM;
function SysFreeMem(P: Pointer): Integer; external DelphiMM;
function SysReallocMem(P: Pointer; Size: Integer): Pointer; external DelphiMM;
function GetHeapStatus: THeapStatus; external DelphiMM;
function GetAllocMemCount: Integer; external DelphiMM;
function GetAllocMemSize: Integer; external DelphiMM;
```

Da hier statisches Einbinden der DLL verwendet wird, kann unsere eigene DLL natürlich nicht auf das Nichtvorhandensein der mit `DelphiMM` bezeichneten DLL reagieren.

> Es heißt also nicht vergessen - die DLL, welche in Deiner Delphi-Version von der `ShareMem.PAS` verwendet wird, muß der Installation beigelegt werden. Das vergißt man leicht, wenn man Delphi sowieso installiert hat und damit die DLL auf dem System vorhanden ist.

Hier ein weiterer Ausschnitt[29] aus der `Sharemem.PAS`:

```
const
  SharedMemoryManager: TMemoryManager = (
    GetMem: SysGetMem;
    FreeMem: SysFreeMem;
    ReallocMem: SysReallocMem);

initialization
  if not IsMemoryManagerSet then
    SetMemoryManager(SharedMemoryManager);
end.
```

Hier sehen wir, daß die Adressen der statisch importierten Funktionen den einzelnen Mitgliedern einer Struktur vom Typ TMemoryManager übergeben werden. Um mehr darüber zu erfahren, solltest Du Dir ein gutes Delphibuch[30] kaufen und im Internet bei der Homepage der KOL und von Optimal Code[31] vorbeischauen. Auf jeden Fall wird die Struktur dann innerhalb der Initialisierung an `SetMemoryManager()` übergeben, wenn noch kein anderer Speichermanager festgelegt wurde. Wäre diese Bedingung nicht enthalten, so könnte `ShareMem.PAS` theoretisch auch an jeder beliebigen anderen Stelle in der `Uses`-Klausel stehen.

28 Die Namen der DLLs unterscheiden sich zwischen Delphi 3 und 4 – wie es mit darauffolgenden Versionen aussieht, kann ich leider nicht sagen.
29 Die Ausschnitte stammen aus Delphi 3 Professional. Nur die Konstantendeklaration stammt aus Delphi 4.
30 Empfehlen kann ich die Delphi-Titel von Andreas Kosch. Anzutreffen ist er im Entwickler-Forum.
31 Siehe Referenzen.

ShareMem.PAS wird aber nur benötigt, wenn Funktionen exportiert werden, die Stringparameter verwenden. An dieser Stelle kann man also nochmals nur davon abraten. Da es aber möglicherweise Szenarien gibt, die Strings als Parameter voraussetzen – z.B. ein altes Plugin-Interface eines Delphi-Programms – wurde die Möglichkeit mit ShareMem eingeräumt.

DllMain() Funktion

Jede DLL kann eigene Strukturen und Objekte initialisieren, wenn sie weiß, wann sie in den Prozeß geladen wird. Woher weiß sie nun aber genau das? In allen möglichen Büchern finden sich Varianten, die es vermeiden Initialisierungscode zwischen das **begin end.** zu schreiben, welches in jeder DLL genau wie in jedem Programmprojekt vorhanden ist.

Stattdessen wird angeregt die DLLMain-Funktion, deren Adresse in der Variablen DLLProc aus einer der System-Units steht, durch eine eigene Funktion[32] zu ersetzen.
Hier ein Beispiel in dem die Adresse unserer eigenen DLLMain-Funktion in der Pointervariablen DLLProc gesetzt wird. Dabei wird der alte Wert von DLLProc zuvor in DLLProcNext gesichert und dann die Adresse unserer DLLMain() gesetzt. Unsere DLLMain() ruft dann die Funktion an der vorigen Adresse auf, wenn diese ungleich nil ist. Dadurch wird Kompatibilität mit Units gewährleistet, die noch vor unserem Initialisierungscode den Wert von DLLProc verbogen haben.

```
library VCL_SampleDLL;

uses
  ShareMem,
  Windows;

var
  DLLProcNext: procedure(Reason: Integer); stdcall = nil;

procedure DLLMain(Reason: Integer); stdcall;
begin
  case Reason of
    DLL_PROCESS_ATTACH:
      DisableThreadLibraryCalls(hInstance);
    DLL_THREAD_ATTACH:
      ;
    DLL_THREAD_DETACH:
      ;
    DLL_PROCESS_DETACH:
      ;
  end;
  if Assigned(DLLProcNext) then DLLProcNext(Reason);
end;

begin
  DLLProcNext := Pointer(InterlockedExchange(Integer(DLLProc), Integer(@DLLMain)));
  DLLMain(DLL_PROCESS_ATTACH);
end.
```

Dies ist auch die Methode, die ich verwende und empfehle.
Schauen wir uns das nochmal näher an. Die DLLMain-Funktion überprüft also den Parameter namens Reason[33] um den Grund des Aufrufs herauszufinden und entsprechend zu reagieren. Vier davon sind im Platform SDK genannt und bisher auch die einzigen vorhandenen.

> Um die Parameter auszuwerten benutze ich eine case-Schleife. Dazu sollte man bemerken, daß bei Mehrfachauswertungen (`if x then y else if z then ...`) eine case-Anweisung vom Compiler effektiver übersetzt wird. Es lohnt sich also durchaus solche Monsterkonstrukte mit x-mal if-else durch eine schön durchdachte case-Schleife zu ersetzen.

32 Funktion und Prozedur unterscheiden sich ja nicht wirklich. Deshalb nenne ich alles beides Funktion.
33 „Reason“ (Aussprache wie „riesn“ ;-) ist englisch für „Grund“

Es gibt dabei folgende Reasons für den Aufruf von `DLLMain()`:

Reason / Grund	Bedeutung
`DLL_PROCESS_ATTACH`	Die DLL wurde in den Speicherbereich des aktuellen Prozesses geladen, während der Prozeß lud – oder aufgrund eines Aufrufs von `LoadLibrary()` oder `LoadLibraryEx()`. An dieser Stelle kann die DLL eigene Strukturen (z.B. den TLS – Thread Local Storage) und Objekte initialisieren.
`DLL_PROCESS_DETACH`	Die DLL wird wieder aus dem Speicherbereich des aktuellen Prozesses entfernt. Dies kann entweder geschehen, wenn der Prozeß beendet wird oder wenn die DLL durch einen Aufruf von `FreeLibrary()` entladen wird. Initialisierte Strukturen und Objekte können an dieser Stelle wieder von der DLL freigegeben werden.
`DLL_THREAD_ATTACH`	Der aktuelle Prozeß erzeugt einen neuen Thread. Wenn dies passiert, ruft das System im Kontext des neuen Threads die `DLLMain()`-Funktion auf und ermöglicht der DLL so z.B. den TLS zu initialisieren. Der Thread, der die `DLLMain()`-Funktion schon mit `DLL_PROCESS_ATTACH` aufgerufen hat, tut dies kein zweites mal mit `DLL_THREAD_ATTACH`. Dieser Grund tritt nur dann auf, wenn die DLL bereits in den Prozeß geladen ist.
`DLL_THREAD_DETACH`	Ein Thread wird beendet. Hat die DLL einen Zeiger auf eine Struktur (z.B. eine TLS) alloziert, muß sie den entsprechenden Speicher wieder freigeben. Dazu wird die `DLLMain()`-Funktion vom System mit diesem Parameter im Kontext des endenden Threads aufgerufen.

Statt die Variable `ExitProc` zu manipulieren, um beim Entladen der DLL eigenen Code auszuführen, sollte man seinen Code in der `DLLMain()` an der Stelle einfügen, wo `DLL_PROCESS_ATTACH` ausgewertet wird.

Wird die DLL dynamisch geladen, so kann man rein theoretisch noch die `DLLMain()` abfangen[34]. Beim statischen Laden passiert alles noch bevor der Prozeß selbst geladen ist. Und das Entladen findet entsprechend beim bzw. kurz nach dem Beenden des Prozesses statt.

Zusammenfassung:

- ShareMem nur verwenden, wenn wirklich nötig, da man sich damit die Abhängigkeit an eine weitere DLL aufhalst.
- `DLLMain()` benutzen, indem die Variable `DLLProc` die Adresse der Funktion `DLLMain()` zugewiesen bekommt.
- Für die Initialisierung von DLLs statt des Codes zwischen `begin end.` lieber den Code in der case-Schleife der DLLMain() unter `DLL_PROCESS_ATTACH` und entsprechend den Deinitialisierungscode unter `DLL_PROCESS_DETACH` einfügen.
- Ein Delphi 5 spezifisches Problem mit `DLLMain()` wird in **Appendix D** besprochen!

DLL und VCL – Formular in DLL auslagern

Oft sieht man in Foren und Boards die Frage: „Wie kann ich ein Formular in eine DLL auslagern“. Nun, nichts leichter als das. Überlegen wir doch mal was passiert wenn ein Formular angezeigt wird?

1. Das Formular wird geladen, also eine Instanz des TFormX erstellt.
2. Das Formular wird angezeigt (modal oder nicht-modal)
3. Das Formular wird am Ende des Programms entladen/zerstört.

Stärker zu empfehlen ist, statt einer DLL eine BPL (ein Package) zu erzeugen in dem das Formular ausgelagert wird. Dies hat mehrere Vorteile: 1. oben genannte (und unten gezeigte) Kopfstände sind nicht mehr notwendig und 2. bietet Delphi die ideale Compiler-Unterstützung für diese Variante.

34 Auf der Seite von EliCZ finden sich dazu nähere Informationen (siehe ApiHooks / DllMain-Hooks).

Auf eine DLL umgelegt, sollte der Schritt 3 irgendwie schon integriert sein. Mir fiele dazu ein, das Formular immer beim Laden der DLL zu erzeugen und beim Entladen der DLL zu zerstören. Eine zweite Methode ist, das Zerstören gleich in die OnClose-Methode des Formulars einzubetten. In beiden Fällen muß die DLL Funktionen exportieren, die das Anzeigen erledigen.

Schauen wir uns Variante 2 einmal anhand von etwas Quelltext näher an. Zuerst die DLL:

```
procedure TFormDLL.FormClose(Sender: TObject; var Action: TCloseAction);
begin
// Form beim Schließen freigeben
  Action := caFree;
end;

procedure TFormDLL.FormCreate(Sender: TObject);
var
  pc: PChar;
begin
// Modul anzeigen
  GetMem(pc, MAX_PATH);
  if Assigned(pc) then
  try
    ZeroMemory(pc, MAX_PATH);
    GetModuleFileName(hInstance, pc, MAX_PATH);
    Label2.Caption := string(pc);
  finally
    FreeMem(pc);
  end else
    Label2.Caption := 'Konnte Modulnamen nicht ermitteln.';
end;

procedure FormShowModal(parent: Pointer); stdcall;
begin
  FormDLL := TFormDLL.Create(nil);
  if Assigned(parent) then
    FormDLL.SetParent(parent);
  FormDLL.Caption := FormDLL.Caption + ' Modal';
  FormDLL.ShowModal;
end;

function FormShowNormal(parent: Pointer): Pointer; stdcall;
begin
  FormDLL := TFormDLL.Create(nil);
  if Assigned(parent) then
    FormDLL.SetParent(parent);
  FormDLL.Caption := FormDLL.Caption + ' Normal';
  FormDLL.Show;
  result := FormDLL;
end;
```

Die wichtige Zeile ist gelb hinterlegt. Sie gewährleistet, daß das Formular beim Schließen freigegeben wird. In der `OnClose`-Methode ist ein `FormX.Free` nämlich nicht explizit möglich. Da wir uns bei dieser DLL für das automatische Entladen beim Schließen entschieden haben, ist es nicht nötig noch irgendetwas freizugeben oder zu bereinigen wenn das Formular aufgerufen wurde.

Wir sehen auch die beiden zu exportierenden Funktionen `FormShowModal()` und `FormShowNormal()`, welche das Formular mithilfe der `Create`-Methode erzeugen und es dann einmal Modal und einmal Normal anzeigen. Der Aufruf erfolgt von den `OnClick`-Methoden der beiden Buttons im Formular aus. Da wir sie im `Interface`-Teil der Unit deklariert haben, sind beide zu exportierende Funktionen auch im Hauptprojekt (i.e. DPR-Datei der DLL) verfügbar, von wo aus wir sie exportieren:

```
exports
  FormShowModal,
  FormShowNormal;
```

Nun noch die Anwendung in der das Formular aufgerufen wird und fertig ist die ganze Geschichte.

```
program VCL_call1;
{$APPTYPE CONSOLE}
uses
Forms,
Windows;

const
  dllname = 'VCL_SampleDLL.dll';

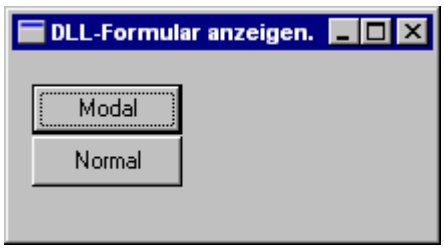
procedure FormShowModal(parent: Pointer); stdcall; external dllname;
function FormShowNormal(parent: Pointer): Pointer; stdcall; external dllname;

begin
  Writeln('Zeige modales Formular');
  FormShowModal(nil); ;
  Writeln('Zeige normales Formular. Das Fenster wird nicht reagieren, da keine');
  Writeln('Nachrichtenschleife existiert.');
  Writeln('Zum Beenden der Anwendung ENTER druecken.');
  FormShowNormal(nil);
  Application.ProcessMessages ;
  Readln;
end.
```

Was auffallen sollte, daß die Anwendung eine Konsolenanwendung ist und deshalb leider keine Hauptschleife für Fensternachrichten besitzt. Da das modale Anzeigen die Funktion erst zurückkehren läßt, wenn das Formular geschlossen wird, gibt es dort keinerlei Probleme. Aber wenn das nicht-modale Formular angezeigt wird, kehrt die Funktion sofort zurück und es gibt quasi keine Zeit mehr die Messages zu verarbeiten – Resultat ist ein meist weißes oder auch anderweitig nicht reagierendes Fenster. Um die Anwendung zu Beenden, im Konsolenfenster ENTER drücken.

Ich habe auch eine zweite Anwendung beigelegt, welche ein Formular mit zwei Buttons öffnet. Einer öffnet das Formular aus der DLL modal, der andere normal. Außerdem ist die DLL nicht wie in der Konsolenanwendung statisch, sondern dynamisch eingebunden. Damit ist es möglich noch eine Fehlerausgabe zu liefern - die oben erklärten Vorteile von dynamischem Einbinden werden damit nochmals verdeutlicht.

Hier das Fenster der GUI-Anwendung und das DLL Formular:



Nico hatte noch etwas zu ergänzen. Er sagte (und das ist vollkommen richtig), daß eine Unit, die vor unserem Code (zwischen `begin end.`) aufgerufen wird, schon längst `DLLProc` überschrieben haben könnte – womit wir dann blind diesen neuen Wert überschreiben würden. Daraufhin habe ich das Projekt nochmals nach seinen Vorschlägen korrigiert. Die alte Version findet sich noch in der Datei `.\SOURCE\01_DLL\!VCL\VCL_SampleDLL_01.dpr`.

Anmerkungen:
Um diese Beispiele nachzuvollziehen, mußt Du jeweils die Projektdateien aus dem Verzeichnis `.\SOURCE\01_DLL\!VCL` öffnen und kompilieren. Danach kannst Du das ganze austesten.
Viel Spaß und Erfolg.

[Die Projekte funktionieren ab Delphi 3!!!]

DLLs Huckepack – DLLs als binäre Ressource

Ich habe eine Methode entwickelt, mit der ich DLLs direkt in die EXE-Datei einfüge. Dieses Verfahren eignet sich insbesondere für Anwendungen welche einen globalen Hook benötigen. Bevor wir aber zum Verfahren selbst kommen, möchte ich kurz auf Vor- und Nachteile eingehen.

Vorteile:
- Kompakte Installation
- DLL kann niemals zum Laden fehlen

Nachteile:
- EXE wird größer, was auch den Speicherbedarf für das Speicherabbild der EXE betrifft.
- Das Programm kann die DLL auch nur dynamisch Laden.
- Probleme können auftreten wenn die DLL schon existiert (z.B. 2te Instanz) und in Benutzung ist.

Dieses Verfahren benutze ich bei diversen nonVCL-Anwendungen welche eine Hook-DLL benötigen. Es ermöglicht auf einfache Weise nur eine EXE weiterzugeben und sowohl Probleme mit fehlenden DLLs, als auch mit falschen DLL-Versionen zu umgehen. Die Nachteile solltest Du aber nicht außer acht lassen. Diese Nachteile sind möglicherweise die besten Gründe gegen das Verfahren. Bei meinen kleinen nonVCL-Anwendungen (zB 40 kB Anwendung und 20 kB DLL) ist der Unterschied in der Imagegröße (40 kB zu 60 kB) marginal. Wenn aber die Anwendung, sagen wir mal ein gewisses Datenbankprojekt, 1.0 MB oder auch nur 500 kB groß, mit einer DLL von 300 kB wäre, und ich müßte mich für oder gegen mein Verfahren entscheiden - ich würde mich dagegen entscheiden! Warum? Weil man da langsam von Größenordnungen spricht, die bei einem Anwender-PC (der ja meist doch spartanischer als ein Entwickler-PC ausgerüstet ist) den RAM ausreizen könnten – auch wenn mein Rechner mit 1 GB RAM nur drüber lachen würde ;) ... Noch sinnloser wird es mein Verfahren zu benutzen, wenn die Anwendung sowieso mit DLLs ausgeliefert werden würde. Da kann man es sich wirklich sparen. Hier die Kernroutine[35]:

```
function ExtractResTo(Instance: hInst; BinResName, NewPath, ResType: string):
  boolean;
var
  ResSize,
    HG,
    HI,
    SizeWritten,
    hFileWrite: Cardinal;
begin
  result := false;
  HI := FindResource(Instance, @binresname[1], @ResType[1]);
  if HI <> 0 then begin
    HG := LoadResource(Instance, HI);
    if HG <> 0 then
    try
      ResSize := SizeOfResource(Instance, HI);
      hFileWrite := CreateFile(@newpath[1], GENERIC_READ or GENERIC_WRITE,
        FILE_SHARE_READ or FILE_SHARE_WRITE, nil, CREATE_ALWAYS,
        FILE_ATTRIBUTE_ARCHIVE, 0);
      if hFileWrite <> INVALID_HANDLE_VALUE then
      try
        result := (WriteFile(hFileWrite, LockResource(HG)^, ResSize,
          SizeWritten, nil) and (SizeWritten = ResSize));
      finally
        CloseHandle(hFileWrite);
      end;
    except;
    end;
  end;
end;
```

Das Projekt zum Verfahren befindet sich in im Verzeichnis `.\SOURCE\01_DLL\!BinRes`.

Es folgen ein paar kurze Erläuterungen zur Routine:

35 Die Routine hat sich so schon seit ca. einem Jahr bewährt.

- Als Parameter müssen übergeben werden:
  - Das Instanzenhandle des Moduls mit der zu extrahierenden Ressource
  - Der Name der Ressource (kann hier kein numerischer Wert sein!)
  - Der Pfad in den die Ressource extrahiert werden soll.
  - Der Typ der Ressource. Bspw: „RCDATA“[36].
- Die Ressource wird gesucht und ein Handle geholt.
- Die Ressource wird geladen und ein Pointer auf die Ressource geholt.
- Eine Datei wird erzeugt und die Ressource wird in ihrer gesamten Größe hineingeschrieben.

In jedem Fall wird die Datei neu erstellt (wenn es bzgl. der Attribute möglich ist). Es wird also nicht nachgefragt, ob die Datei überschrieben werden soll.
Sollte einer der Schritte fehlschlagen, kann es möglich sein, daß die Funktion fehlschlägt. Zumindest gibt die Funktion bei Erfolg `TRUE` zurück.

Mein Beispielprojekt nimmt statt einer DLL eine Batchdatei zur Demonstration der Extraktion. Dadurch wird das ganze Projekt weitaus kleiner, und der Demonstrationseffekt (Starten der Batchdatei) ist mit weit kleinerem Aufwand verbunden.

DLLs Huckepack die Zweite – Im Rucksack mit PEBundle

PEBundle ist ein Programm von Jeremy Collake[37], welches es ermöglicht in zwei verschiedenen Modi eine DLL oder ein anderes Modul an eine EXE-Datei anzufügen. Selbst statisch eingebundene Dateien können so aufgelöst werden.
Betrachtet man sich die Methodik von PEBundle, dann ist mein obiges Verfahren einfach nur noch lächerlich. Da PEBundle aber gerade so ausgeklügelt ist, habe ich mich entschieden es hier mit vorzustellen.

Modus 1:
Der eine Modus schreibt die DLL, welche an die EXE angefügt wurde, temporär auf die Festplatte. Dieser Modus funktioniert in jedem Fall. Ein Loader[38] extrahiert die angefügten Module und lädt danach die EXE-Datei (welche auch quasi an den Loader angefügt ist). Dadurch wird es möglich, daß die DLLs, schon bevor der Image Loader des Betriebssystems sie zu laden versucht, zur Verfügung stehen.

Modus 2 (erweiterter Modus):
Dieser Modus ist wirklich faszinierend. Es wird dabei nichts extra auf die Festplatte geschrieben. Die ganzen Adressen der Funktionen in der DLL werden vom PEBundle-Loader im Speicher aufgelöst und vermutlich durch API-Hooking modifiziert.
Dieser Modus ist wirklich respektabel, und auch wenn man es nach etwas Probieren sicher hinbekommen könnte, soetwas nachzubauen, die Tatsache, daß es mit einem generischen Loader und mit einem generischen Programm welches den Code implantiert funktioniert, ist wirklich großartig. Ich kann das Programm wirklich empfehlen – zumal man bei Jeremy schnellen und persönlichen Support bekommt.

36 Es gibt vordefinierte Konstanten in der `Windows.PAS`. Die Konstanten beginnen mit dem Präfix „`RT_`“ z.B. `RT_CURSOR`, `RT_ICON` oder `RT_RCDATA`.
37 Siehe Referenzen.
38 Von PEBundle implantiert.

Glossar

- **9x**
  sogenanntes „Consumer Windows“ alle Windowssysteme von Windows 95 bis Windows Me die nicht auf NT basieren. XP Home Edition basiert zwar auf NT, zählt aber dennoch zu der Consumer-Reihe.
- **ActiveX**
  Objektorientierte Technologie welche das Verteilen eigener Objekte (z.B. Komponenten) ermöglicht. Außerdem werden Interfaces zum Ansprechen der Objekte zur Verfügung gestellt, wodurch das ganze unabhängig von der verwendeten Programmiersprache wird.
- **API-Hooking**
  So nennt man das Umleiten eines fremden Funktionsaufrufes (meist in einer DLL) an eine eigene Routine, welche dann (meist) den alten Einsprungspunkt wieder aufruft. Interessant wird es dabei eigentlich erst, sobald man damit andere Prozesse manipuliert. Unter Windows NT kommt dabei der sogenannte Copy-On-Write Mechanismus zum tragen.
- **Copy-On-Write**
  Dies ist ein Modus für Speicherseiten. Folgendes gilt für Windows NT/2000/XP:
  Bei DLLs wird dieser Modus unter Windows NT benutzt. Es ist z.B. sehr einfach möglich eine DLL-Funktion per API-Hooking umzuleiten (einfach zumindest innerhalb des eigenen Prozesses). Nun denkt man sich natürlich, dies gälte für das ganze System, also alle Prozesse - dies ist zum Glück nicht der Fall. Solange niemand am Speicherbereich einer DLL rumpfuscht, teilen sich alle Prozesse im System das gemappte DLL-Abbild (MMF) an jeweils verschiedenen Adressen ihres eigenen privaten Speicherraumes. Schreibt hingegen ein Prozeß in ein solches Abbild, werden im Copy-On-Write Modus die Speicherseiten kopiert, dann beschrieben und beeinflussen so nur den aktuellen Prozeß. Mit entsprechenden Berechtigungen kann dies unter NT auch in fremden Prozessen geschehen. Meist durch den Einsatz von APIs wie: CreateRemoteThread() und den Virtual-Memory-Funktionen.
- **MMF – Memory Mapped Files**
  Memory Mapped Files sind ein Mechanismus, mit dem erstens IPC sehr elegant ermöglicht wird und zweitens Dateien in den Speicherbereich eines Prozesses abzubilden. Es ist sowas wie die Umkehrung des Prinzips der Paging-Datei.
- **Modul, Moduldatei**
  Module nennt man generell erstmal alle PE-Dateien. Module können geladen und zum Teil auch ausgeführt werden.
- **NT – New Technology (?)**
  Es gibt Leute die behaupten es hieße eigentliche Neanderthaler-Technologie – dazu kann man meinen was man will. Wie auch immer mit NT bezeichne ich die auf NT basierenden Windowssysteme: Windows NT/2000/XP/.NET. Sie gehören zu den stabilsten Windowssystemen die es gibt.
- **OOP – Object Oriented Programming**
  Objektorientierte Programmierung
- **OCX**
  Spezielle DLLs, welche ActiveX-Kontrollelemente enthalten.
- **ToolHelp API**
  Die Funktionen der Toolhelp API erleichtern die Beschaffung von Informationen über Prozesse. Die API wurde insbesondere für Debugger geschaffen.
- **Typecasting**
  Typecasting bedeutet die On-The-Fly-Konvertierung von (kompatiblen) Typen im Quellcode. Nehmen wir zum Beispiel an, wir hätten einen `Pointer` P und wissen, wir können 3 `DWORD`s weiter eine Struktur vom Typ PMyType finden. Allerdings kann man weder mit PMyType noch mit Pointer ordinal rechnen. PChars und PBytes hingegen sind ordinale Typen und erlauben dies:
  `TMyType:=PMyType(PChar(P)+3*sizeof(DWORD))^;`
- **Windows-Hooks**
  Dies sind die Hooks welche von Windows selbst mit entsprechenden APIs unterstützt werden, sie ermöglichen es prozeßlokal verschiedene Aktionen abzufangen, oder systemglobal selbiges zu tun. Die Windows 9x/Me implementierung ist weder vollständig noch gut nutzbar!!!

Referenzen

Name	Kontaktadresse, ISBN oder URL
Delphi-Forum / AUQ	http://www.delphi-forum.de
Delphi-Groups-Forum	http://www.delphi-groups.de/YaBBSe
Delphi-Source.de	http://www.delphi-source.de
Delphi-Praxis	http://www.delphipraxis.net
Dependency Walker	Enthalten im Platform SDK von Microsoft oder http://www.dependencywalker.com
Entwickler-Forum	http://www.entwickler-forum.de
Iczelion	http://win32asm.cjb.net
Jeremy Collake	http://collakesoftware.com
KOL & MCK	http://xcl.cjb.net
LCC & WEDITRES	http://www.cs.virginia.edu/~lcc-win32 Enthalten in LCC für Win32
OpenOffice.org	http://www.openoffice.org
Optimal Code	http://www.optimalcode.com
Pdf-Factory (Pro)	http://www.fineprint.com
Platform SDK	http://www.microsoft.com/msdownload/platformsdk/sdkupdate
Programming Applications for Microsoft Windows	ISBN 1-57231-996-8 Jeffrey Richter / Microsoft Press
Spotlight Delphi Forum	http://spotlight.de/nzforen/dlp/t/forum_dlp_1.html
Windows NT/2000 Native API Reference	ISBN 1-57870-199-6 Gary Nebbett / Macmillan Technical Publishing USA

Appendix A: Konvertierung von API-Headern nach Delphi

Zum Thema *Konvertierung von Win32 API-Funktionen* ist hier als erstes zu sagen, (fast) alle Win32 API-Funktionen sind als `stdcall` zu deklarieren (Eine Ausnahme ist `wsprintf`, welche `cdecl` verlangt!). Außerdem gilt folgendes:

- Typen die mit „H“ beginnen sind meist Handles und können mit `THandle` übersetzt werden, aber es gibt auch Übersetzungsvarianten wo z.B. der Typ `HICON` auch in Delphi als `HICON` deklariert wird. Typen welche mit „PF“ und „PFN“ beginnen, sind meist Funktionspointer (i.e. Funktionsprototypen). Typen die mit „LP“ beginnen sind Pointertypen, genauer gesagt Long Pointer (also 32bit). Bspw. `LPCSTR` wird zu `PChar`, `LPCTSTR` ist ein `String` aus `TCHARS` und `TCHARS` werden in C über Makros/Präprozessor entweder zu `WCHAR`, in Delphi heißt es explizit `WideChar` (Unicode), oder `Char` (Ansi).
  Dies ist so unter Delphi nicht möglich und Borland behandelt Unicode-Strings immernoch sehr stiefmütterlich. Dies merkt man spätestens bei der Kompatibilität zwischen `PWideChar` und `WideString`, mit der es nicht weit her ist. Bei `PChar` und `AnsiString` gibt es ja bekanntlich eher keine Probleme. Auch empfiehlt es sich die Unit `Windows.PAS` einzubinden um viele der Typen aus den API-Headern ohne Umbenennung verfügbar zu haben. Eine Auflistung findet sich im Appendix B.

- Die meisten Strukturen aus den API-Headern müssen als `packed`[39] deklariert werden. Also:
  ```
  TMyType=packed record end;
  ```
  Der Unterschied zwischen `packed` und normal tritt erst dann zutage, wenn Typen als Member verwendet werden, welche nicht an der 32bit-Grenze ausgerichtet sind. Ein Beispiel wäre eine Struktur mit Bytes (1 Byte = 8 Bit):
  ```
  TMyType = {packed} record
    r,g,b:Byte;
    x:Word;
  end;
  ```
  Ungepackt nimmt diese Struktur
  `1*sizeof(DWORD)+sizeof(Word)` also 48bit = 6 Byte in Anspruch,
  die gepackte Variante ist schon mit
  `3*sizeof(Byte)+1*sizeof(Word)` also 40bit = 5 Byte dabei.
  `Packed` ist auch ein wenig langsamer, da mehr Arbeit (i.e. mehr Code) nötig ist an die einzelnen Member heranzukommen! Nur wenn die Member direkt über der Ausrichtungsgrenze liegen, dann wird eine unterschiedliche Größe des Records zu bemerken sein.

- In vielen Fällen gibt es auch ein Problem mit der Übersetzung der Parameterübergabe unabhängig von Aufrufkonventionen und so weiter. Vergleichen wir einmal eine Deklaration aus Delphi mit einer aus dem PSDK[40]. Nehmen wir an, sowohl im C-Header[41], wie auch in Delphi haben wir eine Struktur wie oben (`TMyType`) deklariert und es sei folgender abgewandelter Typ deklariert: `PmyType = ^TMyType;`.
  Dann würde die folgende C-Deklaration
  ```
  DWORD ApiFunc(CONST TMYTYPE* lpmt);
  ```
  zu folgender Delphi-Deklaration übersetzt:
  ```
  function ApiFunc(var mt:TMyType):DWORD;
  ```.
  Dies ist immer der Fall wenn in C ein Pointer auf eine Struktur übergeben wird. Ein Beispiel wäre die Übersetzung von `CreateFontIndirect()`. Weitere gültige Übersetzungen wären:
  ```
  function ApiFunc(lpmt:PMyType):DWORD;
  function ApiFunc(const mt:TMyType):DWORD;
  function ApiFunc(out mt:TMyType):DWORD;
  ```
  Aber gemeinhin wird es einfacher empfunden eine Variable direkt und nicht über einen Pointer zu übergeben. Vom Compiler werden `var`, `const` und `out` vor Parametern meist als Pointer übersetzt[42]. Im Quelltext sind sie natürlich nicht immer adäquat verwendbar.

---

39 Packed ist englisch für „gepackt“.
40 Siehe Referenzen.
41 Natürlich auch in C usw. Also angenommen alles ist korrekt deklariert.
42 Meist deshalb, weil bei `const` zum Teil einfach der Wert übergeben wird, statt seiner Referenz.

- Ein weiteres Problem taucht oft mit Unions auf, da diese in der Delphihilfe zwar beschrieben sind, jedoch nicht an der Stelle, wo man sie vermutet, nämlich nicht unter der Bezeichnung Unions.
  Da diese Elemente so selten auftreten, hier eine kurze Beschreibung – mehr findet sich in Appendix C.
  Unions sind Record-Member[43], die am gleichen Offset[44] vom Start des Records aus auftreten können. Dabei müssen die Typen der Member die gleiche Größe[45] haben, insofern nicht als Unter-Record definiert werden. Zum Beispiel wäre eine Konstruktion möglich, um entweder einmal die Länge des Strings und eine Checksumme und ein andermal ein Pointer auf dessen Vorgänger und Nachfolger in eine einzige Struktur zu packen[46]. Welche Bedeutung das entsprechende Member hat, hängt allein vom Kontxt ab, in dem es dann steht.
  `PMyType` ist als Pointertyp genauso groß wie ein `DWORD` oder ein normaler `Pointer` und deswegen vertragen sich beide von der Größe her auch in einer Union und darauffolgende Elemente haben im Endeffekt das gleiche Offset. Es finden sich jede Menge Beispiele, allerdings für nichtverschachtelte Unions. Verschachteln ist bei Unions nämlich tatsächlich möglich. Mehr zu alledem, wie schon gesagt, in Appendix C.

- C-Kenntnisse sind immer gut. Insbesondere ist darauf zu achten, daß Delphi keinen Präprozessor hat und deshalb z.B. weder „normale“ Makros noch diverse `#define` Statements kennt.

- Wie schon in der `Windows.PAS` ersichtlich, müssen folgende Compilerschalter benutzt werden um korrekte Ausrichtung bestimmter Typen (Aufzählungstypen, Records) zu gewährleisten:
  ```
  {$ALIGN ON}
  {$MINENUMSIZE 4}
  ```

43 Record-Member sind die Unterelement eines Records (Struktur als Typ). Member ist englisch für Mitglied.
44 „Abstand“ von einer Startposition aus.
45 Mit `sizeof()` zu ermitteln.
46 Ein nachfolgendes Element muß immer in die Klammer des letzten Union-Elements! Schachteln möglich.

Appendix B: API-Typen und ihre Delphi-Entsprechungen

Nicht vordeklarierte Typen können meist durch Basistypen von Delphi, oder durch jene die in der `Windows.PAS` vordeklariert sind, ersetzt werden. Bei jenen wo ich weiß, daß sie in einer anderen Unit als der `Windows.PAS` deklariert sind, habe ich dies vermerkt.

Pointertypen (`P*, LP*`) entsprechen meist folgendem Schema (* steht für Typenbezeichnung):

```
type P* = ^*; type LP* = ^*;
```

Die Übersetzung der Handletypen (`H*`) in der `Ddeml.PAS` finde ich äußerst zweifelhaft. Sie werden dort als `Longint` übersetzt!!! Handletypen entsprechen üblicherweise folgendem Schema:

```
type H* = LongWord; type H* = THandle;
```

Nicht vordeklarierte Typen (nicht in Systemunits und `Windows.PAS`) sind in der Tabelle hellgrau hinterlegt. Weiß impliziert, daß der Typ in den Systemunits oder der `Windows.PAS` unter gleichem Namen bereits vordeklariert ist! Alles was nicht grau hinterlegt ist sollte ohne große Kopfstände nutzbar sein. Blau bedeutet, daß es den Typen oder das Element so in Delphi nicht gibt. Die Angaben stammen aus Delphi 4.

Name	Beschreibung und Entsprechung(en)
ATOM	Beschreibung siehe Platform SDK. Es handelt sich um ein 16bit Handle und kann deshalb als `Word` deklariert werden, ist aber vordeklariert.
BOOL	Ist ein Boolean-Typ welcher 32bit breit ist und bei dem `BOOL = 0 -> FALSE` und `BOOL <> 0 -> TRUE` ist. Es muß TRUE oder FALSE zugewiesen werden.
BOOLEAN	Ist ein 8bit Boolean-Typ und entspricht dem `Boolean` aus Delphi.
BYTE	8bit vorzeichenloser Integertyp. Entspricht dem `Byte` aus Delphi.
CALLBACK	Keine Entsprechung in Delphi. Callback-Funktionen werden entsprechend der Vorgabe als `stdcall` deklariert und das wars.
CHAR	Entspricht dem Delphityp `Char`.
COLORREF	Ist ein 32bit vorzeichenloser Integertyp der die 3 Bytes für R, G und B eines Farbwertes hält, das oberste Byte bleibt meist ungenutzt.
CONST	Entspricht dem `const` Schlüsselwort aus Delphi. Ist aber zumindest in Delphi kein Typ! Man sollte aber echte Konstanten verwenden. Delphi ermöglicht es die Regeln bei Konstanten durch Compilerdirektiven zu lockern, wodurch sich einer Konstante auch Werte zuweisen lassen.
CRITICAL_SECTION	Ausschlußobjekt welches z.B. im Zusammenhang mit Threads Verwendung findet. Ist in `Windows.PAS` als `TRTLCriticalSection` vordeklariert.
DWORD	32bit vorzeichenloser Integertyp. Entspricht `LongWord`.
DWORD_PTR	32bit Zeiger. Kann mit `Pointer` übersetzt werden, oder man deklariert: `type DWORD_PTR = Pointer;`
DWORD32	32bit vorzeichenloser Integertyp. Kann als DWORD übersetzt werden, oder man deklariert (`Windows.PAS` erforderlich): `type DWORD32 = DWORD;`
DWORD64	64bit vorzeichenloser Integertyp. Kann als `ULARGE_INTEGER` übersetzt werden (`Windows.PAS` erforderlich).
FLOAT	Gleitkommatyp von einfacher Präzision. Wird als `Single` übersetzt.
HACCEL	Handle auf eine Accelerator-Tabelle.
HANDLE	In der `Windows.PAS` als `THandle` deklariert. Beliebiges Handle.
HBITMAP	Handle auf GDI-Bitmap
HBRUSH	Handle auf Brush.

Name	Beschreibung und Entsprechung(en)
HCONV	Nicht deklariert (bzw. Ddeml.PAS). Stattdessen THandle benutzen oder wie folgt deklarieren: `type HCONV = LongWord;`
HCONVLIST	Nicht deklariert (bzw. Ddeml.PAS). Stattdessen THandle benutzen oder wie folgt deklarieren: `type HCONVLIST = LongWord;`
HCURSOR	Handle auf GDI Cursor-Objekt.
HDC	Handle auf Gerätekontext (Device Context).
HDDEDATA	Nicht deklariert (bzw. Ddeml.PAS). Stattdessen THandle benutzen oder wie folgt deklarieren: `type HDDEDATA = LongWord;`
HDESK	Handle auf einen Desktop.
HDROP	Nicht deklariert. Stattdessen THandle benutzen oder wie folgt deklarieren: `type HDROP = LongWord;`
HDWP	Handle auf eine Deferred Window Position Struktur.
HENHMETAFILE	Handle auf eine verbesserte Metadatei (Bildformat).
HFILE	Handle auf eine Datei.
HFONT	Handle auf einen GDI Font Objekt.
HGDIOBJ	Handle auf ein beliebiges GDI Objekt.
HGLOBAL	Handle auf einen mit GlobalAlloc() oder GlobalReAlloc() allozierten Speicherbereich.
HHOOK	Handle auf ein Hookobjekt.
HICON	Handle auf ein GDI Icon.
HIMAGELIST	In der Commctrl.PAS als HIMAGELIST deklariert. Handle auf eine Imagelist.
HIMC	Nicht deklariert. Stattdessen THandle benutzen oder wie folgt deklarieren: `type HIMC = LongWord;`
HINSTANCE	In der Windows.PAS als HINST deklariert. Instanzenhandle. Stattdessen kann auch HMODULE verwendet werden. Beide entsprechen sich vollständig. Bitte niemals hInstance verwenden, denn dies ist der Variablenname welcher in der Systemunit für die aktuelle Instanz verwendet wird!!!
HKEY	Handle auf einen Registryschlüssel.
HKL	Handle auf eine Input Locale (Stichworte: Lokalisierte Eingabe, IME)
HLOCAL	Handle auf einen mit LocalAlloc() oder LocalReAlloc() allozierten Speicherbereich.
HMENU	Handle auf ein Menü.
HMETAFILE	Handle auf eine Metadatei (Bildformat).
HMODULE	Handle auf eine Modulinstanz. Kann synonym zu HINST verwendet werden.
HMONITOR	Nicht deklariert. Stattdessen THandle benutzen oder wie folgt deklarieren: `type HMONITOR = LongWord;`
HPALETTE	Handle auf eine GDI Palette.
HPEN	Handle auf einen GDI Pen.
HRGN	Handle auf eine Region (Wird z.B. benutzt um runde Fenster zu erzeugen).
HRSRC	Handle auf eine Ressource (in einer geladenen EXE, DLL oder anderen PE).
HSZ	Nicht deklariert (bzw. Ddeml.PAS). Stattdessen THandle benutzen oder wie folgt deklarieren: `type HSZ = LongWord;`
HWINSTA	Handle auf eine Window Station.
HWND	Handle auf ein Fenster.

Name	Beschreibung und Entsprechung(en)
INT	32bit vorzeichenbehafteter Integertyp. Entspricht dem LongInt und Integer aus Delphi (32bit!!!).
INT_PTR	32bit Zeiger. Kann mit Pointer übersetzt werden, oder man deklariert: `type INT_PTR = Pointer;`
INT32	32bit vorzeichenbehafteter Integertyp. Entspricht dem LongInt und Integer aus Delphi (32bit!!!). Kann wie folgt deklariert werden: `type INT32 = Integer;`
INT64	64bit vorzeichenbehafteter Integertyp. Entspricht dem Delphityp Int64.
LANGID	16bit Sprachen-ID (Language Identifier). Kann als Word deklariert werden.
LCID	32bit Locale Identifier. Kann als DWORD deklariert werden.
LCTYPE	32bit Locale Type Konstante. Kann als DWORD deklariert werden.
LONG	Siehe INT.
LONG_PTR	Siehe INT_PTR.
LONG32	Siehe INT32.
LONG64	Siehe INT64.
LONGLONG	Siehe auch INT64
LPARAM	Parameter von Fensternachrichten.
LPBOOL	Pointer auf BOOL. Siehe dort. `type LPBOOL = ^BOOL;`
LPBYTE	Pointer auf BYTE. Siehe dort. Entspricht dem Delphi PByte.
LPCOLORREF	Pointer auf COLORREF. Siehe dort. `type LPCOLORREF = ^COLORREF;`
LPCRITICAL_SECTION	In der Windows.PAS als PRTLCriticalSection deklariert. Pointer auf TRTLCriticalSection. Siehe dort.
LPCSTR	Nullterminierter String aus Ansichars. Wird mit PChar übersetzt.
LPCTSTR	Kommt drauf an, ob als Unicode oder Ansi deklariert. Wird wenn als Ansi, meist mit PChar übersetzt, sonst als PWideChar.
LPCVOID	Pointer auf ein Zeichen. Am besten wäre wohl Pointer oder PChar zu verwenden.
LPCWSTR	Nullterminierter String aus Unicodezeichen (2Byte pro Zeichen). Wird als PWideChar übersetzt.
LPDWORD	Pointer auf ein DWORD. Siehe dort.
LPHANDLE	Pointer auf ein Handle. Nicht deklariert. Man verwende stattdessen PHandle.
LPINT	Pointer auf INT. Siehe dort.
LPLONG	Pointer auf LONG. Siehe dort.
LPSTR	Siehe LPCSTR.
LPTSTR	Siehe LPCTSTR.
LPVOID	Pointer auf nichts. Übersetzt mit Pointer.
LPWORD	Pointer auf Word. Siehe dort.
LPWSTR	Siehe LPCWSTR.
LRESULT	Parameter bei Fenster-Funktionen.
LUID	Nicht deklariert. Kann mit TLargeInteger übersetzt werden.
PBOOL	Siehe LPBOOL. In der Windows.PAS deklariert.
PBOOLEAN	In der Windows.PAS als PBOOLEAN deklariert. Pointer auf Boolean.
PBYTE	Ordinaltyp! In der Windows.PAS als PBYTE deklariert.

Name	Beschreibung und Entsprechung(en)
PCHAR	Ordinaltyp! Pointer auf ein Char. Ist in den Systemunits vordeklariert.
PCRITICAL_SECTION	Siehe LPCRITICAL_SECTION.
PCSTR	Siehe LPCSTR. Nicht deklariert.
PCTSTR	Siehe LPCTSTR. Nicht deklariert.
PCWCH	Siehe LPCWSTR. Nicht deklariert.
PCWSTR	Siehe LPCWSTR. Nicht deklariert.
PDWORD	Siehe LPDWORD.
PFLOAT	In der Windows.PAS als PSingle deklariert.
PHANDLE	Pointer auf HANDLE.
PHKEY	Pointer auf HKEY. Siehe dort.
PINT	Pointer auf INT. Siehe dort.
PLCID	Pointer auf LCID. Siehe dort.
PLONG	Pointer auf LONG. Siehe dort.
PLUID	Pointer auf LUID. Siehe dort.
POINTER_32	32bit Pointer. Entspricht Pointer.
POINTER_64	Existiert so nicht. Ich verwende ULARGE_INTEGER.
PSHORT	Pointer auf SHORT. Siehe dort.
PSTR	Siehe LPSTR.
PTBYTE	Pointer auf TBYTE. Siehe dort.
PTCHAR	Pointer auf TCHAR. Siehe dort.
PTSTR	Siehe LPTSTR.
PUCHAR	Pointer auf UCHAR. Siehe dort.
PUINT	Pointer auf UINT. Siehe dort.
PULONG	Pointer auf ULONG. Siehe dort.
PUSHORT	Pointer auf USHORT. Siehe dort.
PVOID	Einfacher Pointer. Siehe LPVOID.
PWCHAR	Pointer auf WCHAR. Siehe dort.
PWORD	Pointer auf WORD. Siehe dort.
PWSTR	Siehe LPWSTR. Siehe dort.
REGSAM	32bit vorzeichenloser Integertyp. Ist in der Windows.PAS als vom Typ ACCESS_MASK deklariert.
SC_HANDLE	Deklariert in der WinSvc.PAS. Einfaches Handle. Entspricht THandle.
SC_LOCK	Deklariert in der WinSvc.PAS. 32Bit Pointertyp. Entspricht Pointer.
SERVICE_STATUS_HANDLE	Deklariert in der WinSvc.PAS. Einfaches Handle. Entspricht DWORD.
SHORT	16bit vorzeichenloser Integertyp. Entspricht dem Delphityp Word.
SIZE_T	Einfacher vorzeichenloser Integertyp. Kann als LongWord deklariert werden.
SSIZE_T	Einfacher Integertyp. Kann als LongInt deklariert werden.
TBYTE	Wenn Unicode deklariert, entspricht es Word ansonsten entspricht es Byte.
TCHAR	Wenn als Unicode deklariert dann entspricht es WideChar, ansonsten entspricht es AnsiChar.
UCHAR	Entspricht dem Delphityp Char.
UINT	32Bit vorzeichenloser Integertyp. Entspricht DWORD.

Name	Beschreibung und Entsprechung(en)
UINT_PTR	32bit Pointertyp. Entspricht Pointer.
UINT32	Siehe UINT.
UINT64	In der Windows.PAS als ULARGE_INTEGER deklariert. 64Bit vorzeichenloser Integertyp.
ULONG	32Bit vorzeichenloser Integertyp.
ULONG_PTR	Siehe DWORD_PTR.
ULONG32	Siehe ULONG.
ULONG64	In der Windows.PAS als TLargeInteger deklariert. 64Bit vorzeichenloser Integertyp.
ULONGLONG	In der Windows.PAS als TLargeInteger deklariert. 64Bit vorzeichenloser Integertyp.
UNSIGNED	Keine Entsprechung. Taucht als Schlüsselwort vor Typenbezeichnungen auf um anzugeben, daß diese nicht vorzeichenbehaftet sind. Meist kann man es ignorieren, da zum Beispiel Zeichen in Delphi gar nicht vorzeichenbehaftet sein können.
USHORT	Siehe WORD.
VOID	Nichts. Ist meist der Rückgabewert einer Funktion. Diese muß dann als procedure übersetzt werden. Hat also keinen Rückgabewert. Sollte es tatsächlich als Parameter auftauchen können, so handelt es sich mit einiger Sicherheit um einen Pointer. Ich bezweifele aber, daß dieser Fall existiert.
WCHAR	Entspricht WideChar.
WINAPI	Kein Typ. Entspricht der Aufrufkonvention stdcall.
WORD	16bit vorzeichenloser Integertyp. Entspricht dem gleichnamigen Delphitypen.
WPARAM	Siehe LPARAM.

Im Archiv zum Tutorial ist auch eine Include-Datei enthalten, welche (wenn sie direkt hinter der USES-Sektion steht) gute Dienste beim Übersetzen leisten dürfte. Einfach mit folgender Zeile einbinden:

```
{$INCLUDE .\WindowsTypes.inc}
```

Bitte beachten: Die .inc-Datei muß im selben Verzeichnis sein wie die Datei in der diese Zeile steht. Außerdem sollten für die nun noch fehlenden Typen die entsprechenden Units eingebunden werden.

> Die in der Tabelle aufgelisteten Typen werden im Platform SDK als „Windows Data Types“ genannt. Die Liste findet sich im Platform SDK unter:
> `Development Guides\Windows API\Windows API Reference\Windows Data Types`

Hinweis: Bevor man sich hier abmüht irgendwelche Windows APIs nach ObjectPascal zu konvertieren, sollte man auf den einschlägigen Seiten nachschauen, insbesondere bei:

http://www.delphi-jedi.org

Appendix C: Unions in Delphi

Unions, das hatte ich ja schon in Appendix A kurz angeführt, machen oftmals Probleme in Delphi. Es gibt nun aber leider API-Strukturen die als Unions deklariert sind. Der Vollständigkeit halber will man diese natürlich auch authentisch übersetzen. Dies ist mit Delphi, obwohl viele Programmierer davon nichts wissen, tatsächlich möglich. Durch die Verwendung von Unions kann man:

- Unter Umständen das Typecasten sparen.
- Oftmals die Deklaration verschiedener Records sparen.
- Komplizierte Strukturen vereinfachen (vereinfachen ist dabei relativ zu sehen)

Es finden sich einige Beispiele für Unions in dem Beispielprojekt unter `.\SOURCE\C_Unions`. Darin kann man auch gut sehen, wieviele eigentlich verschiedenen Strukturen in einer Union stecken.

Es gibt aber etwas zu beachten: In C sind Unions auch innerhalb eines Records (`struct`) abgeschlossen und deshalb deren Reihenfolge sehr wohl relevant. Eigentlich aneinander hängende Unions muß man in Delphi aber verschachteln. Dies sieht so aus, daß weitere Unions und auch weitere normale Member des Records (nicht der Union) an das letzte Member der übergeordneten Union herangehangen werden. Während eine einfache Union also so aussieht wie oben im Appendix A schon gezeigt, sähe eine verschachtelte Union beispielsweise aus wie folgt:

```
TMyStringV2 = packed record
  case Integer of
    0: (Len, Checksum: DWORD);
    1: (Prev, Next: PMyString;
      szText: PChar;
      case Integer of
        0: (file1: THandle);
        1: (
          case Integer of
            0: (icon1: HICON);
            1: (
              cursor1: THandle;
              OtherElements: array[0..1] of DWORD;
              )
            )
        )
end;
```

Vereinfachen läßt sich das ganze, indem jede Union in einen eigenen Record geschachtelt wird. Also etwa so - die Funktionalität ist hierbei die gleiche wie im Beispiel oben:

```
TMyStringV2R = packed record
  union1: record // start 1st union record
    case Integer of
      0: (Len, Checksum: DWORD);
      1: (Prev, Next: PMyString);
  end; // 1st record ends here
  szText: PChar;
  union2: record // start 2nd union record
    case Integer of
      0: (file1: THandle);
      1: (union3: record // start 3rd union record
          case Integer of
            0: (icon1: HICON);
            1: (cursor1: THandle);
        end; ) // 3rd union record ends here
  end; // 2nd union record ends here
  OtherElements: array[0..1] of DWORD;
end;
```

Um da hineinzukommen und den Trick zu finden, muß man selbst mal die ein oder andere Union basteln. Also auf auf und probieren, probieren, probieren ... wie Lenin schon sagte ... oder so ähnlich

Appendix D: Delphi 5 ruft DLLProc, bitte kommen ...

Diese Ergänzung wurde ebenfalls von Nico Bendlin angeregt.

Es handelt sich um einen Artikel aus dem Borland Developer Support – genauer gesagt um Artikel „How to make sure the function DLLPROC is called when your DLL is unloaded.“ (Artikel-ID 27753). Es geht dabei um die eigenständige Korrektur (Patch) eines Fehlers. Dies ist nur eine freie Übersetzung ins Deutsche. Der Orginalartikel findet sich unter:

http://community.borland.com/article/0,1410,27753,00.html

Wie stellen Sie sicher, daß DLLPROC aufgerufen wird, wenn Ihre DLL entladen wird.

(Autor: Personal der Borland Entwicklerunterstützung)

FRAGE:
Warum wird die DLLPROC nicht aufgerufen, wenn meine DLL entladen wird?

ANTWORT:
Aufgrund einiger Änderungen in der Laufzeitbibliothek wurde die DLLProc nicht korrekt aufgerufen. Diese nun folgende Änderung wird dieses neue Verhalten korrigieren, welches Sie festgestellt haben. Sobald Sie die Korrektur angewandt haben, müssen Die die Laufzeitbibliothek neu kompilieren und die neue System.dcu in das Projektverzeichnis Ihres Projektes kopieren. Ihr Projekt wird dann die neu erstellte System.dcu mit dem korrigierten Code verwenden. In der System.pas, suchen Sie bitte nach "`@@skipTLSproc`" (und ändern Sie die Stelle wie folgt):

```
@@skipTLSproc:
{ Call any DllProc }
  PUSH ECX
  MOV ECX,[ESP+8] //Changed from 4 to 8.
  TEST ECX,ECX
  JE @@noDllProc
  MOV EAX,[EBP+12]
  MOV EDX,[EBP+16]
  CALL ECX
```

Produkte:
Delphi 5.x

Plattform:
Windows 2000 1.0

Artikel-ID: 27753
Herausgabedatum: September 11, 2001
Zuletzt geändert: September 16, 2001

Appendix E: Weitere Begrifflichkeiten und deren Erklärung, aber ohne explizite Delphi-Unterstützung als FAQ

Q: Was ist „Delay Loading“?

A: Delay Loading ist cool. Man bezeichnet als Delay Loading die Methode, bei der DLLs erst dann in den Prozess geladen werden, wenn sie auch benötigt würden. Vielleicht ist noch in Erinnerung, dass ich weiter oben sagte, dass man bei der ToolHelp-API nur auf dynamische Einbindung zurückgreifen kann. Dies ist nicht ganz richtig ;) ... es geht auch mit Delay Loading. Würde man beispielsweise eine Versionsüberprüfung (bzgl. Betriebssystemversion) in das Programm einbauen und auf dem jeweiligen System nur die jeweils vorhandenen Funktionen benutzen, so kann man quasi den Image-Loader austricksen und eine Meldung, nach der die DLL nicht geladen werden konnte, vermeiden. Eine weitere Anwendung findet diese Methode bei Anwendungen die sehr viele DLLs laden müssen. Der Startvorgang kann so stark beschleunigt werden, da benötigte Dateien erst on-demand (also bei Bedarf) geladen werden. Soweit ich weiß, bietet Delphi leider keine Unterstützung für Delay Loading.

Q: Was sind „Function Forwarder“ / „Function Export Forwarder“?

A: Die Antwort ist sehr einfach. Es sind Funktionen die von einer DLL exportiert werden, die aber eigentlich auf eine ganz andere DLL verweisen. Ein Beispiel wäre die Funktion `GetLastError()` aus der Kernel32.dll, welche eigentlich auf die Funktion `RtlGetLastWin32Error()` in der Ntdll.dll verweist. Ich habe dazu auch ein kleines Programm geschrieben, welches die Function Forwarder in einer gegebenen DLL auflistet: `/stuff/!export/export_forw.zip`

Q: Wozu braucht man DisableThreadLibraryCalls()?

A: Diese Funktion dient dazu, die Benachrichtigungen einer DLL auszuschalten, wenn ein neuer Thread im zugehörigen Prozeß erstellt wird. Dies dient einfach der Steigerung der Performance bei bestimmten Anwendungen mit mehreren Threads. Verfügbar in Delphi.

Q: Was macht FreeLibraryAndExitThread()?

A: Diese Funktion verringert den Wert des Referenzzählers um 1 und beendet den aufrufenden Thread. Es ist einfach eine nützliche Kombination zweier Funktionen mit dem Zusatz, daß es ausgeschlossen ist, daß `ExitThread()` aufgerufen wird nachdem die DLL freigegeben wurde. Dies würde u.U. zu einem Fehler führen.

Q: Wie exportiert man Variablen?

A: Tja, in Delphi habe ich das leider noch nicht geschafft. Hier müßte man den Umweg über eine Funktion gehen. Aber beispielsweise C bietet es an. Soweit ich weiß, ist es in Delphi auch nicht möglich solche Symbole zu importieren. Ein Beispiel ist `NtGlobalFlag` aus `ntoskrnl.exe`.

Q: Können auch EXE-Dateien etwas exportieren?

A: Wie man oben am Beispiel von ntoskrnl.exe sieht, ist es möglich (auch in Delphi). Ich habe allerdings noch keinerlei Verwendung dafür gefunden. Ich hatte beispielsweise schon versucht eine EXE so zu gestalten, dass sie auch gleichzeitig als Hook-DLL benutzt werden kann. Leider half es nichts, das Betriebssystem ließ sich nicht austricksen. Sobald ich etwas finde, wird man es hier lesen können.

Viel Spaß mit
der Programmierung in Delphi
und viel Erfolg,

wünscht Dir

Oliver aka Assarbad