Dan Appleman

Dan Applemans Win32 Puzzle-Buch

Puzzles und Tutorials
für Visual Basic-Profis

Galileo Press

Galileo Computing

Dan Appleman
Dan Applemans Win32 Puzzle-Buch
Puzzles und Tutorials für Visual Basic-Profis
490 S., 2000, CD, 79,90 DM, ISBN 3-934358-21-7

William R. Vaughn
ADO 2.5
VB-Datenbankprogrammierung für Profis
400 S., 2000, CD, 89,90 DM, ISBN 3-934358-78-0

Christian Gross
Windows DNA
E-Business-Applikationen mit
Windows 2000, COM+, Visual Studio
700 S., 2000, CD, 119,90 DM, ISBN 3-934358-75-6

Dave Baum
Dave Baums
LEGO® MINDSTORMS™ Roboter
Der Profi-Guide
372 S., 2000, CD, 69,90 DM, ISBN 3-934358-39-X

Ulrich Kaiser
C/C++
Von den Grundlagen zur
professionellen Programmierung
1050 S., 2000, CD, 79,90 DM, ISBN 3-934358-03-9

André Willms
C++ STL
Verstehen, anwenden, erweitern
327 S., 2000, CD, 69,90 DM, ISBN 3-934358-20-9

Michael Hyman, Phani Vaddadi
Effektive C++-Techniken
300 S., 2000, CD, 59,90 DM, ISBN 3-934358-28-4

P. Grässle, H. Baumann, Ph. Baumann
UML projektorientiert
Geschäftsprozessmodellierung,
Systemintegration, WebEDI
300 S., ca. 69,90 DM, ISBN 3-934358-58-6
ab 09/2000

Thomas Theis
PHP 4
Webserver-Programmierung
für Um- und Einsteiger
360 S., 2000, CD, 69,90 DM, ISBN 3-934358-63-2

Ray Rischpater
WAP und WML
Wireless Web – Das neue Internet
432 S., 2000, 79,90 DM, ISBN 3-934358-76-4

Christian Wenz
JavaScript
Browserübergreifende Lösungen
528 S., 2000, CD, 79,90 DM, ISBN 3-934358-94-2

Westy Rockwell, Elmar Geese
XML und Java
XML und Java: Das Team der Zukunft!
350 S., CD, ca. 79,90 DM, ISBN 3-934358-57-8
ab 09/2000

Friedrich Esser
Java 2
450 S., CD, ca. 59,90 DM, ISBN 3-934358-66-7
ab 10/2000

Ch. Voecks, S. Gungl, S. Weiß-Kubat
Datenbanken im Web
Pragmatische und kostengünstige Lösungen
300 S., ca. 69,90 DM, ISBN 3-934358-70-5
ab 10/2000

Günter Born
Dateiformate – eine Referenz
Datenbanken, Tabellenkalkulation,
Textdarstellung, Grafik, Multimedia,
Sound, Internet
1000 S., ca. 119,90 DM, ISBN 3-934358-83-7
ab 11/2000

Die Deutsche Bibliothek – CIP-Einheitsaufnahme
Ein Titeldatensatz für diese Publikation ist bei
der Deutschen Bibliothek erhältlich

ISBN 3-934358-21-7

Die amerikanische Originalausgabe trägt den Titel: Dan Appleman's Win32 API Puzzle Book and Tutorial for Visual Basic Programmers
© 1999 by Daniel Appleman
(ISBN 1-893115-0-11, APress™)

Der Name Galileo Press geht auf den italienischen Mathematiker und Philosophen Galileo Galilei (1564–1642) zurück. Er gilt als Gründungsfigur der neuzeitlichen Wissenschaft und wurde berühmt als Verfechter des modernen, heliozentrischen Weltbilds. Legendär ist sein Ausspruch **Eppur se muove** (Und sie bewegt sich doch). Das Emblem von Galileo Press ist der Jupiter, umkreist von den vier Galileischen Monden. Galilei entdeckte die nach ihm benannten Monde 1610.

Lektorat Judith Stevens
Fachliche Beratung Michael Kofler, Graz
Einbandgestaltung Barbara Thoben, Köln
Übersetzung Lemoine International
Herstellung Petra Strauch, Bonn
Satz Reemers EDV-Satz, Krefeld
Druck und Bindung Bercker Graphischer Betrieb, Kevelaer

Inhalt

Vorwort

Als Gary Cornell und ich APress gründeten, wollten wir beide ein Buch für die neue Firma schreiben. Mein Buch sollte etwas Besonderes werden. Ich wollte die mir überlassene Entscheidungsfreiheit voll nutzen, sowohl im Hinblick auf den Inhalt als auch auf den Stil des Buches. Es sollte ein Buch werden, mit dem jeder Visual Basic-Programmierer alle Fragen im Hinblick auf die Win32-API selbst beantworten können sollte. Außerdem sollte es ein Buch werden, mit dem VB-Programmierer die Verwendung selbst kompliziertester API-Aufrufe wirklich verstehen würden. Und es sollte ein Buch werden, das Spaß macht: Es zu lesen und zu schreiben.

Die meisten Bücher (einschließlich meines letzten Buches) ähneln einem Vortrag. Sie beginnen damit, dass Informationen präsentiert werden. Es wird die Durchführung verschiedener Arbeitsschritte beschrieben, die Erläuterungen zu Beispielcodes enthalten. Manchmal sind einige Fragen enthalten, oder es werden Vorschläge zur weiteren Vorgehensweise gemacht. Und viele dieser Bücher eignen sich als optimales Mittel gegen Schlaflosigkeit.

Verstehen Sie mich nicht falsch. Ich bin sehr stolz auf meine bisher veröffentlichten Bücher. Tatsache ist: Sie werden wahrscheinlich mein API-Buch **(Dan Appleman's Visual Basic Programmer's Guide to the Win32 API)** als Referenz benötigen, um das vorliegende Buch optimal nutzen zu können. Als verantwortungsvoller Dozent habe ich versucht, genaue und auch nützliche Informationen in einer möglichst lockeren und eingängigen Form bereitzustellen. Vorträge stellen eine effiziente Methode der Informationsweitergabe dar und können demnach als wertvolle Wissensquelle dienen. Dennoch sind es Vorträge, und Vorträge stellen nun mal nicht unbedingt die beste Lernmethode dar.

Das vorliegende Buch ist kein Vortrag.

Um was es sich bei diesem Buch handelt, ist mir selbst nicht ganz klar. Vielleicht ist es ein Spiel. Oder vielleicht auch ein Spielzeug. Es soll auf jeden Fall eine Herausforderung sein. Ich hoffe, dass Sie mein Buch faszinierend, bestechend, unterhaltsam, frustrierend und erstaunlich finden. Und ich hoffe, dass Sie mit diesem Buch eine Menge lernen – nicht bloß reine Informationen und Verfahren, sondern auch Lösungsansätze und eigene Ansichten. Vor allem aber wünsche ich Ihnen viel Spaß.

Danksagung

Es ist jedesmal interessant, das erste Buch für einen Verleger zu schreiben. Gelegentlich ist es frustrierend. Dann wieder ist es eine Erfahrung, die man »durchlebt« (eine andere Art zu sagen, dass man diese Erfahrung nie wieder machen möchte). Als Gary, Bill und ich APress gründeten, wollten wir eine autorenfreundliche Firma entstehen lassen. Dies bedeutete für uns, dass der Produktionsprozess möglichst genau den Anforderungen eines Autors entsprechen sollte. Und da ich der erste Autor von APress war, wurde ich zum Versuchskaninchen.

Ich kann nur sagen, dass Bill Pollock, der Verleger, ein tolles Team zusammenbrachte. Ich bin recht wählerisch geworden, was meine Erwartungen an einen Lektor angeht – jemand, der meine Grammatik, meine Rechtschreibung und meine Sprache verbessert, dies jedoch auf eine Weise, die nicht mit dem Stil kollidiert, den ich für ein bestimmtes Kapitel vorgesehen habe. Lunaea Weatherstone hat in dieser Hinsicht ihren Job mehr als gut erledigt und wurde dabei durch Chris Sabooni, den Lektor, tatkräftig unterstützt. Danke auch an Lori Ash, die an diesem Projekt von Beginn an mitgearbeitet hat. Carel Lombardi leitete dieses Projekt und war für mich die primäre Ansprechpartnerin. Ihr Gespür für Qualität und ihre stets freundliche Art, mit der sie mich daran erinnerte, den vorgegebenen Zeitplan einzuhalten, machten Sie zur perfekten Projektmanagerin. Derek Yee war für Satz und Grafiken verantwortlich und hat es geschafft, die Grafiken nicht nur optisch ansprechend, sondern auch mit der nötigen Genauigkeit zu gestalten.

Eine der Prioritäten, die wir uns bei APress gesetzt hatten, war die, höchstmögliche Qualität zu liefern. Die Produktionsqualität wurde durch Bill und sein Team gesichert, doch die Qualität des Inhalts ist ein noch wichtigerer Punkt. Franky Wong, ein guter Freund und Kollege bei Desaware, fungierte für dieses Projekt als technischer Redakteur. Wenn ich sagen würde, dass er tolle Arbeit geleistet hat, wäre dies eine Untertreibung.

Abschließend möchte ich all denen danken, die dieses Buch gerade lesen bzw. bereits eines meiner früheren Bücher gelesen oder aber ein Desaware-Produkt erworben haben. Ohne Ihre Unterstützung wäre dieses Buch nicht geschrieben worden.

Dan Appleman

Januar 1999

dan@desaware.com

Haben Sie Lust auf ein Spiel?

Haben Sie jemals Kindern beim Spielen zugesehen?

Es ist erstaunlich, wie sich Kinder in ein Spiel vertiefen können. Spaß ist eine ernste Sache, wenn man ein Kind ist.

Bemerkenswert ist die Aussage der meisten Kinderpsychologen, dass Kinder gerade beim Spielen eine Menge lernen.

Es scheint, dass uns diese Form des Lernens verloren gegangen ist. Das Lernen wird zu einer Aufgabe – zunächst, weil wir gute Noten bekommen möchten, später, weil wir auf dem Laufenden bleiben müssen, wenn wir in der heutigen Zeit mithalten wollen. Und so geht der Spaß am Lernen nach und nach verloren. Das Spielen wird ersetzt durch endlose, langweilige Vorträge und trockene, theoretische Lehrbücher. Wer sagt, dass Informationen in dieser Form vermittelt werden müssen? Wer sagt, dass technische Handbücher ernst und trocken sein müssen? Ich bin dagegen!

Und ich bin auch dagegen, dass Lernen harte Arbeit sein muss!

In gewisser Weise haben wir Programmierer Glück. Die meisten von uns erhalten von Zeit zu Zeit die Gelegenheit, neue Technologien auszuprobieren, auch wenn sich die so investierte Zeit bei einem vorgegebenen Projekt nicht immer rechtfertigen lässt bzw. dem Kunden nicht in Rechnung gestellt werden kann. Wir verbringen Stunden damit, herauszufinden, wie etwas funktioniert oder finden neue, tolle Methoden heraus, wie man etwas erledigen kann, auch wenn wir dieses Wissen später nie anwenden. Man kann dies »Lernen«, »Forschen« oder »Testen« nennen, aber, unter uns gesagt, handelt es sich doch eigentlich um eine Spielerei. Es macht Spaß. Und genau auf diese Weise lernt es sich am besten.

Also, lassen Sie uns gemeinsam lernen.

Lassen Sie uns spielen.

Aufbau dieses Buches

Der Aufbau dieses Buches unterscheidet sich von den Büchern, die ich bisher geschrieben habe. Ich kann mich ehrlich gesagt nicht daran erinnern, vorher bereits etwas Ähnliches gesehen zu haben. Das Buch besteht aus drei Teilen mit unterschiedlichen Zielen. Darüber hinaus wurde dieses Buch **nicht** so geschrieben, dass es von Anfang bis Ende gelesen werden müsste.

Teil I: Die Puzzle[1]

Jedes »Kapitel« in Teil I umfasst ein kleines Puzzle. Die ersten Puzzle (im ersten Buchabschnitt) sollten zu einer nur mittelschweren Frustration führen, selbst für solche VB-Programmierer, die sich im Anfangsstadium befinden und nur über begrenzte Erfahrung bei der API-Programmierung verfügen. Der Schwierigkeitsgrad der Puzzle erhöht sich, sodass die Puzzle am Ende des Buches – Abschnitte 4 und 5 – selbst für Experten eine Herausforderung darstellen sollten. Obwohl die meisten der Puzzle ein Codebeispiel umfassen, das nicht funktioniert und berichtigt werden muss, sollten Sie sich dennoch auf einige Überraschungen gefasst machen.

Wenn Sie Probleme haben, finden Sie Hilfe in Anhang A, »Hinweise« (siehe auch die nachstehenden Hinweisinformationen).

Teil II: Die Lösungen

Hier finden Sie die Lösungen für die Puzzle in Teil I. Jede Lösung umfasst nicht nur den richtigen Code, sondern auch eine detaillierte Erläuterung des vorliegenden Problems sowie des entsprechenden Lösungsansatzes. Obwohl ich verstehen kann, dass die Versuchung, direkt zur jeweiligen Lösung zu wechseln, sehr groß sein kann, sollten Sie dennoch versuchen, einige Zeit über jedes Puzzle nachzudenken, bevor Sie sich die Lösung ansehen. Auf diese Weise ist der Lerneffekt höher, und Sie werden das Erlernte besser behalten.

Teil III: Die Tutorien

Diese detaillierten Tutorien umfassen Themen, die sich auf den API-Funktionsaufruf aus Visual Basic beziehen. Einige dieser Themen wurden speziell ausgesucht, um einige Wissenslücken zu schließen, die bei der Ausbildung zum Visual Basic-Programmierer erfahrungsgemäß entstehen. Andere Themengebiete beinhalten Beispiele für fortgeschrittene Programmierer, die sich für den Puzzle-Ansatz nicht eignen. Wieder andere der Tutorien enthalten Informationen oder Verfahrensweisen, von denen ich nicht wusste, wo ich sie denn einordnen sollte.

Die Anhänge

Anhang A: Hinweise

Die Lösungen enthalten sämtliche Informationen, die zur Lösung der Puzzle in diesem Buch benötigt werden. Da Sie dieses Buch lesen, ist Ihnen sehr wahrscheinlich bewusst, dass es viele Fälle gibt, in denen ein Programmierproblem nicht einfach

1. Anmerkung des Verlags: Engl. »Puzzles« wurde mit »Puzzle« übersetzt.

mit Hilfe eines Buches gelöst werden kann. Sie werden auch andere Informationsquellen heranziehen müssen, um die Antwort auf eine Frage zu erhalten. Dieses Buch wurde beispielsweise **nicht** konzipiert, um als Referenz bei der Beantwortung von Fragen im Hinblick auf spezielle API-Funktionen zu dienen.

Wenn Sie die Lösung zu einem Problem nicht wissen, verwenden Sie Anhang A, »Hinweise«. Dort finden Sie Hinweise auf weitere Informationsquellen.

Im Folgenden werden einige Standardinformationsquellen genannt, die Sie stets zu Rate ziehen sollten:

- **The Win32 SDK or MSDN (Microsoft Developer's Network Library oder http://www.microsoft.com):**
  Hier finden Sie die aktuelle C/C++-Dokumentation für die fragliche API bzw. die aktuellste Bugliste. Visual Basic enthält eine eingeschränkte Version der MSDN-Bibliothek, die wiederum die Win32 SDK-Dokumentation umfasst; Sie verfügen also bereits über die benötigten Informationen.
- **Die C-Headerdateien:**
  Hierbei handelt es sich um jene C-Headerdateien, die von C++ Windows-Entwicklern verwendet werden und gleichzeitig als wichtigste Referenz in Bezug auf die richtigen Funktionsbeschreibungen dienen. Der vollständige Satz der Win32 SDK-Headerdateien ist, dank freundlicher Unterstützung von Microsoft, auf der Begleit-CD-ROM zu diesem Buch enthalten.
- **Die Visual Basic-Dokumentation:**
  In dieser Dokumentation sind einige Informationen zum API-Funktionsaufruf enthalten.
- **Kapitel 1 bis 4 meines Buches »Dan Appleman's Visual Basic Programmer's Guide to the Win32 API«** (SAMS, ISBN 0-672-31590-4)**:**
  Diese Kapitel enthalten grundlegende Informationen zu Windows und dem API-Funktionsaufruf, deren Kenntnis zur Lösung der Puzzle in diesem Buch von höchster Wichtigkeit sind.

Zusätzlich enthalten die Hinweise in Anhang A Verweise auf eine oder mehrere der folgenden Informationsquellen, die bei der Lösung der Puzzle ebenfalls nützlich sein können:

Mein Api-Buch (siehe oben), Anhang D:
Ich habe in vielen Fällen auf dieses spezielle Kapitel in meinem API-Buch verwiesen, da es sich auf die im Puzzle verwendete API bezieht. Obwohl Ihnen das Kapitel auch nur zur Puzzlelösung dienen mag, stellt es doch darüber hinaus nützliche Hintergrundinformationen zu den fraglichen Funktionen bereit und dient als weitergehende Informationsquelle.

- **Verweise auf die »Zehn API-Gebote«**
  Hierbei handelt es sich um einen Verweis auf die »Zehn Gebote für die sichere API-Programmierung« (nachfolgend aufgeführt). Diese können als Richtlinien bei der Lösungsfindung für nahezu jedes API-Problem herangezogen werden.
- **Tutorien**
  Hierbei handelt es sich um einen Verweis auf ein entsprechendes Tutorium (in Teil III dieses Buches), in dem die der Puzzlelösung zugrundeliegenden Konzepte erläutert werden. Diese Hinweise sind besonders hilfreich für Programmierer mit geringerer Erfahrung und eignen sich gerade zur Lösung der leichteren Puzzle.
- **Mein anderes Buch »Dan Appleman's Developing COM/ActiveX Components with Visual Basic 6.0: A Guide to the Perplexed«** (SAMS, ISBN 1-56276-576-0)**:** Was hat ein Buch über ActiveX mit der Win32-API zu tun? Eine ganze Menge, wenn Sie bedenken, dass mit der OLE- oder ActiveX-Technologie von Microsoft mit Hilfe von DLLs implementiert wird, die praktisch über Hunderte von Funktionen verfügen.

Anhang B: Häufig gestellte Fragen

In diesem Buch werden zahlreiche Methoden für den API-Funktionsaufruf aus Visual Basic vorgestellt. Aufgrund des Buchaufbaus sind diese Techniken über die verschiedenen Puzzle und Tutorien verteilt. Daher habe ich mich entschieden, den Lesern eine Liste mit Querverweisen zur Verfügung zu stellen, aus der zu entnehmen ist, an welcher Stelle des Buches eine bestimmte Methode behandelt wird.

Wie die meisten technischen Bücher verfügt auch dieses über einen Index, damit Informationen zu bestimmten Themen schnell aufgefunden werden können. Da es sich bei jedem Index jedoch um nichts weiter als ein Suchsystem nach dem Schlüsselwortprinzip handelt, weiß jeder, der bereits mit einer Internetsuchmaschine gearbeitet hat, dass diese Suchsysteme nur begrenzte Funktionen besitzen.

Dieser Anhang enthält neben dem Index eine Liste mit häufig gestellten Fragen. Hierbei handelt es sich de facto um ein Verzeichnis der API-Techniken, die in Kategorien unterteilt wurden.

Anhang C: Die APIGID32.DLL-Bibliothek

Dieser Anhang enthält eine Dokumentation für die DLL **apigid32.dll**, eine Dynamic Link Library, die über nützliche Funktionen zur Arbeit mit API-Funktionen von Visual Basic aus verfügt. Für diejenigen, die daran interessiert sind, C++-DLLs für die Verwendung mit Visual Basic zu schreiben, wurde der Quellcode dieser DLLs beigefügt.

Verwendung der CD-ROM

Die wichtigste Datei auf der Begleit-CD-ROM zu diesem Buch stellt die Datei **pzl1.hlp** im Stammverzeichnis dar. Doppelklicken Sie einfach auf die Datei, um den Hilfebildschirm zu starten. Die Hilfe umfasst Folgendes:

- Die aktuellsten Installationsanmerkungen und Berichtigungen, die in letzter Minute vorgenommen wurden, also das Äquivalent zur traditionellen Readme-Datei
- Links zum Installationsprogramm, mit denen benötigte Steuerelemente und DLLs zur Ausführung des Beispielprogramms installiert werden
- Eine halbstündige Videopräsentation
- Eine Liste der CD-ROM-Hauptverzeichnisse sowie deren Inhalt
- Den Onlinekatalog von Desaware

Der Beispielcode befindet sich in den Verzeichnissen **SourceVB6** und **SourceVB5**. Das Verzeichnis **SourceVB6** enthält den VB6-Quellcode, der als Referenz für den Quellcode in diesem Buch dient. Das Verzeichnis **VB5** enthält die VB5-Versionen des Beispielprogramms. Das (in den Puzzlen und Lösungen verwendete) Codeverhalten basiert jedoch auf VB6. Die VB5-Versionen wurden als zusätzliche Hilfe beigefügt und sollten das gleiche Verhalten aufweisen, eine Garantie hierfür kann ich jedoch nicht übernehmen.

Im Verzeichnis **VB6** befinden sich drei Unterverzeichnisse, **Puzzle**, **Lösungen** und **Tutorien**. Jedes dieser Verzeichnisse enthält weitere Unterverzeichnisse für die einzelnen Puzzle oder Tutorien. Das Verzeichnis **Puzzle** enthält den Beispielcode, der das im Puzzle beschriebene Problem zeigt. Im Verzeichnis **Lösung** befinden sich die Lösungen für jedes Puzzle. Im Verzeichnis **Tutorien** ist der Beispielcode für die Tutorien gespeichert.

Das Verzeichnis **Cheaders** enthält, dank der freundlichen Unterstützung von Microsoft, die C-Headerdateien von Microsoft, die unter dem Programm **Open Tools** zur Verfügung gestellt werden. In diesen Headerdateien finden Sie die Werte von Konstanten sowie Datentypinformationen.

Weitere Informationen zum Inhalt der CD-ROM finden Sie in der Datei **pzl1.hlp**.

Info zu Betriebssystemen

Ich weiß nicht, welches Betriebssystem Sie verwenden. Selbst wenn Sie es mir sagen würden, würde ich immer noch nicht wissen, welches Betriebssystem Sie einsetzen.

Als dieses Buch in Druck ging, haben Sie wahrscheinlich eines der folgenden Betriebssysteme verwendet: Windows 95, Windows 98, Windows NT 3.51, Windows NT 4.0 oder die Betaversion von Windows 2000.

Dies bedeutet, dass ich jedes der Puzzle unter einem dieser fünf Betriebssysteme hätte testen müssen, um sicher sagen zu können, dass die beschriebenen Lösungen tatsächlich mit den Ergebnissen übereinstimmen, die Sie erhalten. Aber selbst diese Tests würden aus verschiedenen Gründen nicht ausreichen:

- Sie verfügen vielleicht über die Originalversion von Windows 95, oder Sie haben ein Service Pack installiert, oder es handelt sich um die OSR1- oder OSR2-Version.
- Ihre NT 3.51-Installation kann über eines von fünf verschiedenen Service Packs verfügen.
- Ihre NT 4.0-Installation kann über eines von sechs verschiedenen Service Packs verfügen.

Selbst wenn ich wüsste, welche Windows 95-Version Sie verwenden oder welche NT-Service Packs Sie installiert haben, auch dies würde noch nicht ausreichen, da bei jeder Installation eines Microsoft-Pakets – sei es nun Visual Studio (in sämtlichen Versionen und einschließlich der jeweiligen Service Packs), Internet Explorer (mit monatlichen Änderungen) oder Microsoft Office – unter Umständen neue Systemkomponenten installiert werden.

Was sollte ein Autor also tun?

Wenn wir bei Desaware Software veröffentlichen, testen wir alle gängigen Betriebssystemversionen mit jeweils einer Grundinstallation und dem aktuellsten Service Pack. Anschließend starten wir unsere Anwendungen oder Komponenten mit Hilfe unseres eigenen Produkts (VersionStamper) mit Selbstdiagnosefähigkeiten aus, um so nachvollziehen zu können, welche Fehler auf einem Kundensystem vorliegen (damit in bestimmten Fällen automatisch Komponenten von unserer FTP-Site heruntergeladen werden können).

Ein Buch kann sich jedoch nicht selbst aktualisieren, und eine gedruckte Seite ist nicht in der Lage, Ihr System zu scannen.

Deshalb wird in diesem Buch angenommen, dass Sie Windows NT 4.0 verwenden. Windows NT stellt keine schlechte Wahl dar, wenn Sie genauer darüber nachdenken – die meisten professionellen Entwickler arbeiten statt mit Windows 95/98 mit Windows NT, da es sich bei Windows NT um ein stabiles Betriebssystem mit exzellenter Prozessisolation handelt, eine Notwendigkeit für die Arbeit auf API-Ebene –, und Windows 95 und Windows 98 sind ... nun ja, es sind Betriebssysteme. Windows 2000 wird im Hinblick auf die in diesem Buch angeführten Anwendungen wahrscheinlich ein ähnliches Verhalten wie NT 4.0 aufweisen – vorausgesetzt natürlich, es wird jemals ausgeliefert.

Unterschiede zu Windows 95/98 werden nur dann aufgeführt, wenn sich diese auf die Ergebnisse beziehen, da ein professioneller Entwickler nicht auf die Systemunterschiede hingewiesen werden muss, die sich bei der Anwendungsentwicklung ergeben. Bei einer Durchsicht der Puzzle wird Ihnen auffallen, dass diese Art der Entwicklung immer unter Windows NT durchgeführt werden sollte, da unter NT gegenüber Windows 95/98 eine bessere Fehlerermittlung sowie detailliertere Fehlerberichte verfügbar sind.

Was Sie in diesem Buch nicht finden

In diesem Buch wird der Aufruf von API- und DLL-Funktionen von Visual Basic aus behandelt.

Wenn Sie noch ein Anfänger sind, fragen Sie sich vielleicht »Was ist eine API-Funktion?« oder »Was ist eine DLL-Funktion?«. Vielleicht machen Sie sich auch darüber Sorgen, dass Sie nicht wissen, wie Windows funktioniert. Sie kennen unter Umständen die Begriffe **Fensterzugriffsnummer** oder **Gerätekontext** nicht.

Tja, ich werde Sie Ihnen nicht erklären.

Dieses Buch behandelt nicht die Win32-API selbst. Dieses Buch habe ich bereits geschrieben. Es heißt **Dan Appleman's Visual Basic Programmer's Guide to the Win32 API**. Dieses Buch ist ein Bestseller unter den Büchern für Visual Basic-Programmierer und kann in jedem Buchhandel mit gut sortierter Computerliteratur erworben werden (oder Sie bestellen es direkt bei Desaware, Inc. unter **http://www.desaware.com/** oder bei einer anderen Mailorderfirma). Die ersten vier Kapitel dieses Buches enthalten Informationen, mit denen selbst ungeübte VB-Programmierer die Grundlagen der API-Programmierung von Visual Basic aus erlernen können. Der verbleibende Teil des Buches beschäftigt sich mit dem Kern der Win32-API, der Funktionsweise von Windows sowie der API-Funktionen. Das Buch umfasst 1500 Seiten (plus einige weitere Kapitel auf CD-ROM). Wenn Sie denken, dass ich in der Lage wäre, die Informationen dieses Buches in diesem viel weniger umfangreichen Buch zusammenzufassen, haben Sie sich leider getäuscht. Und nicht nur das, warum sollte ich mir deswegen Sorgen machen?

Schließlich handelt es sich bei diesem Puzzlebuch nicht um eine umfassende Referenz. In diesem Buch soll Ihnen nicht die Funktionsweise der Win32-API beigebracht werden. Statt dessen lernen Sie, wie Sie die C/C++-Dokumentation interpretieren, um eigene Deklarationen zu erstellen und komplexe API-Probleme selbst zu lösen. Dieses Buch soll Ihnen dabei helfen, die Mechanismen eines API-Aufrufs zu verstehen, damit Sie die Informationen in meinem API-Buch effektiv anwenden können, auch wenn keine Beispiele vorhanden sind, oder damit Sie eine Funktion auf eine Weise anwenden können, die in keinem der Beispiele erläutert wird.

Warum dies? Weil die Win32-API ständig größer wird – es werden täglich neue Funktionen hinzugefügt – und ich kann meine Zeit nicht darauf verwenden, Mammutbücher zu schreiben, in denen all diese Funktionen beschrieben werden. Darüber hinaus scheint sich außer mir keiner für diesen Job zu interessieren.

Wenn Sie nicht in der Lage sind, sämtliche Puzzle in diesem Buch zu lösen, heißt dies nicht, dass Sie ein weiteres dieser Mammutbücher zur Win32-API lesen müssen. Sie werden fähig ein, die C/C++-Dokumentation zu verwenden und schnell und genau eigene Deklarationen zu erstellen. Nicht nur dies, Sie werden qualifiziert sein, Ihr eigenes VB-Programmiererhandbuch zu einer beliebigen der Win32-Erweiterungsbibliotheken zu schreiben.

Und nach all der Werbung bleibt noch zu sagen, dass mir ein paar Zeilen ihrerseits jederzeit willkommen sind – wenn Sie denn wirklich wollen.

Die zehn Gebote der sicheren API-Programmierung (überarbeitete Fassung)

Ich habe die ursprünglichen »Zehn Gebote der sicheren API-Programmierung« um 1992 geschrieben, und seither sind verschiedene Versionen dieser Gebote in Publikationen, Konferenzen, Websites und Büchern erschienen. Ich bin der Meinung, dass durch die Einhaltung dieser Gebote nahezu jedes Problem bei der API-Programmierung gelöst werden kann, aus mir nicht bekannten Gründen scheinen einige Leute dies jedoch nicht ohne weiteres glauben zu wollen. Vielleicht kann ich diese Leute mit dem vorliegenden Buch davon überzeugen, da viele der Puzzle Hinweise enthalten, die auf eines der zehn Gebote verweisen.

Machen Sie sich keine Sorgen, wenn Ihnen einige (oder alle) Konzepte in den Geboten im Moment unklar erscheinen. Wenn Sie sich durch die Puzzle, Lösungen und Tutorien arbeiten, werden sie Ihnen schnell in Fleisch und Blut übergehen.

1 Gedenke der ByVal-Anweisung und halte Sie in Ehren!

Die richtige Verwendung der `ByVal`-Anweisung ist der wichtigste Faktor bei der erfolgreichen API-Programmierung über Visual Basic. Denken Sie daran, dass das `ByVal`-Schlüsselwort manchmal nicht nur in der Deklaration, sondern im Funktionsaufruf selbst verwendet wird. Zur Anwendung weitergehender Techniken ist es zwingend erforderlich, die Funktionsweise von `ByVal` genau zu kennen und zu wissen, welche Auswirkungen diese Anweisung auf den API-Funktionsaufruf von Visual Basic aus hat.

2 Du sollst deine Parametertypen überprüfen!

Das Verwenden der richtigen Parametertypen ist unter Win32 noch wichtiger geworden als unter Win16. Unter Win32 werden die meisten Parameter, unabhängig von ihrer tatsächlichen Größe, als 32-Bit-Werte übergeben. Daher werden über die traditionelle Fehlermeldung »bad DLL calling convention« weniger Fehler ermittelt als unter Win16. Das Ergebnis sind gut versteckte, datenabhängige Bugs, die nur sehr schwer ausfindig zu machen sind. Denken Sie stets daran, dass bei vielen API-Funktionen für einen Parameter mehrere Datentypen verwendet werden können, was es überaus wichtig macht, die Unterschiede bei der Übergabe der verschiedenen Parametertypen an die Funktionen genau zu kennen. Denken Sie außerdem daran, dass Visual Basic eine tückische Konvertierungseigenschaft besitzt, durch die ohne Vorwarnung eine automatische Wertetypenumwandlung vorgenommen wird. Dies kann dazu führen, dass Sie einen Ihrer Meinung nach richtigen Parametertyp übergeben, der tatsächlich jedoch einen ungültigen Wert enthält.

3 Du sollst deine Rückgabetypen überprüfen!

Durch die meisten API-Funktionen werden 32-Bit-Werte zurückgegeben. Probleme bei der Werterückgabe treten häufig dann auf, wenn von einer Deklaration kein Wert vom Typ `Long` zurückgegeben wird. Der häufigste Fehler hierbei besteht darin, dass für die Funktionsdeklaration kein Rückgabetyp angegeben wurde. In diesem Fall wird von Visual Basic der Standardrückgabetyp `Variant` verwendet, der mit Sicherheit falsch ist und den Fehler »bad DLL calling convention« oder einen Speicherausnahmefehler auslöst.

4 Du sollst deine Zeichenfolgen initialisieren, sonst wirst du untergehen!

API-Funktionen, die Zeichenfolgenparameter verwenden, erkennen diese üblicherweise als Adressen zu Speicherstellen, die eine auf einen Nullwert endende Zeichenfolge enthalten. Viele Funktionen können Zeichenfolgenwerte wieder an die aufrufende VB-Anwendung zurückgeben, indem diese Speicherstelle mit den Daten geladen wird. Die API-Funktion kann jedoch nicht ermitteln, wieviel Platz im verfügbaren Puffer vorhanden ist. Bei einigen Funktionen kann die Puffergröße als separater Parameter übergeben werden. Bei anderen Funktionen dagegen ist einfach eine bestimmte Puffergröße erforderlich. Wenn Sie eine noch nicht initialisierte oder einen leere Zeichenfolge an eine Funktion übergeben, weist der Speicherpuffer entweder keinen Platz oder einen einzelnen Zeichenwert auf (der auf NULL endende Zeichenwert). Wenn die API-Funktion versucht, Daten an die bereitgestellte Speicheradresse zu laden, werden mit Sicherheit wichtige Daten in Ihrem Anwendungsspeicher überschrieben. Dies kann zu sofortigen Speicherausnahmefehlern bis hin zu schwer auffindbaren Bugs führen, die nur dann auftreten, sobald Sie bereits Tausende von Kopien Ihrer Anwendung ausgeliefert haben. Stellen Sie deshalb immer sicher, dass Sie die Zeichenfolgenpuffer initialisieren, falls auch nur im Entferntesten die Möglichkeit besteht, dass der Puffer durch die API-Funktion bearbeitet wird.

5 Verwende nicht As Any, denn es ist böse!

Wenn Sie einen Parametertyp als `As Any` deklarieren, sind Sie allein dafür verantwortlich, den richtigen Datentyp an den Code zu übergeben, durch den die Funktion aufgerufen wird. In Visual Basic wird keinerlei Typenprüfung vorgenommen. Die `As Any`-Deklarationen werden am häufigsten bei API-Funktionsparametern verwendet, die mehr als einen Datentyp verwenden können. Glücklicherweise können dank der Option `Alias` in der `Declare`-Anweisung in Visual Basic mehrere Deklarationen für dieselbe Funktion erstellt werden.

Dennoch gibt es Fälle, in denen es sehr viel bequemer ist, den Parametertyp `As Any` beizubehalten. So kann beispielsweise eine Funktion so viele verschiedene Datentypen verwenden, dass es zu umständlich wäre, für jeden Datentyp eine andere Deklaration zu definieren – und sich diese zu merken. Oder die Funktion verfügt über mehr als einen Parameter, der verschiedene Datentypen annehmen kann, wodurch für jeden neuen Typ mehrere Deklarationen erstellt werden müssen.

Ganz gleich, welchen Ansatz Sie wählen, es ist äußerst wichtig, den `As Any`-Parameter und dessen Funktion zu verstehen. Außerdem kann man nie wissen, wann man vielleicht zufällig an den Code eines anderen Entwicklers gerät, der diese Regeln leider nicht eingehalten hat.

6 Du sollst Option Explicit angeben!

Bei der Arbeit mit Visual Basic sollte immer die Editoroption **Variablendeklaration erforderlich** gesetzt werden. Wenn Sie Code überarbeiten, der erstellt wurde, ohne dass diese Option gesetzt wurde, sollten Sie sicherstellen, dass Sie den Befehl **Option Explicit** zu Beginn eines jeden Codemoduls hinzufügen (einschließlich **Formular, Benutzersteuerelement, Klassenmodul** usw.).

Setzen Sie diese Option nicht, wird durch jeden Verweis auf eine Variable automatisch eine leere Instanz dieser Variablen erstellt. Hierzu gehören auch Fälle, in denen versehentlich eine neue Variable erstellt wird, da der Name einer vorhandenen Variablen falsch geschrieben wurde. Schlimmer noch ist, aus der Sicht eines API-Programmiers, dass die neu erstellte Variable sehr wahrscheinlich den Typ `Variant` erhält. Unabhängig von der Sichtweise entsprechen die Ergebnisse jedoch höchstwahrscheinlich nicht denen, die Sie erzielen wollten. Der zusätzliche Aufwand, der durch die Deklaration jeder Variablen vor deren Verwendung entsteht, ist jedoch nichts im Vergleich zu der mühseligen Arbeit, die bei der Suche nach versteckten Bugs entsteht, die durch typografische Fehler verursacht werden.

7 Ehre deine VB- und Win32-Ganzzahlen, denn sie stimmen nicht überein!

Eine Ganzzahl umfasst 16 Bits. Oder nicht? Wenn man in der C- oder C++-Dokumentation nachschlägt (der Standardsprache für die gesamte Windows-Dokumentation), beziehen sich unter Win32 alle `Int`-Werte und sämtliche Verweise auf Ganzzahlen auf einen 32-Bit-Wert. Wenn eine Person sich also auf »`Integer`-Werte« (Ganzzahlen) bezieht oder auf diese in einem Buch verwiesen wird, sollte man immer den jeweiligen Kontext berücksichtigen.

8 Überprüfe stets deine Funktionsnamen, denn es wird nun die Groß- und Kleinschreibung berücksichtigt, und die Funktionsnamen können Suffixe aufweisen!

Unter Win32 wird bei sämtlichen API-Funktionen die Groß-/Kleinschreibung beachtet. Im Gegensatz dazu spielte die Groß- oder Kleinschreibung der Funktionsnamen bei der Win16-API keine Rolle. Achten Sie in Situationen, in denen die Funktion einen Zeichenfolgenwert verwendet, außerdem auf Suffixe. Es kann oft

vorkommen, dass der Name der Funktion in der DLL über die angehängten Buchstaben »A« oder »W« verfügt, durch den der ANSI- oder Unicode-Einsprungpunkt der Funktion angegeben wird.

9 Du sollst deine Parameter und Rückgabewerte prüfen!

Eine der schönsten Eigenschaften von Visual Basic ist die, dass es sich um eine interpretierte Sprache handelt. Dies bedeutet, dass Sie Ihren Code bei der Ausführung an einem beliebigen Punkt anhalten können, um die Werte der einzelnen Parameter zu überprüfen. Wenn ein API-Aufruf nicht wie erwartet funktioniert, stoppen Sie an dem Punkt und untersuchen die im Aufruf verwendeten Parameterwerte. Suchen Sie nach Werten, die keinen Sinn ergeben, und achten Sie dabei besonders auf Parameter, die gültige Daten enthalten sollten, statt dessen jedoch den Wert 0 aufweisen (Hinweis auf einen Fehler in einem vorangegangenen API-Aufruf). Überprüfen Sie die Rückgabewerte der Funktionen, und rufen Sie mit Hilfe der Methode `Err.LastDllError` zusätzliche Fehlerinformationen ab.

10 Speichere deine Arbeit so oft wie möglich!

Das Gute an der Verwendung von API-Funktionsaufrufen ist, dass diese nach der richtigen Deklaration und Programmierung extrem zuverlässig sind. Weniger gut bei der Verwendung von API-Funktionen ist es, dass, bis Sie diese richtig deklariert und programmiert haben, die erzeugten Fehler häufig zu einer Speicherausnahme und somit zu einem Programmabsturz, zu einem Hängenbleiben von Visual Basic und in einigen Fällen sogar zu einem Systemabsturz führen. Nichts ist frustrierender als nach dem Hinzufügen dutzender neuer Codezeilen auf die Schaltfläche **Starten** zu klicken, nur um mitzuerleben, wie dieser Code durch einen Speicherausnahmefehler im Nichts verschwindet. Zu meinem eigenen Besten speichere ich deshalb meinen Code, bevor ich ihn ausführe. Und das möchte ich auch Ihnen dringend empfehlen.

Teil I
Die Puzzle

Es gibt verschiedene Möglichkeiten, eine Reise zu planen. Manche Menschen treffen Vorbereitungen, packen sorgfältig ihre Koffer, lernen ein paar Sätze in der jeweiligen Sprache des Urlaubslandes und planen jedes Detail der Reise so genau wie möglich. Andere werfen lediglich eine Unterhose und eine Zahnbürste in ihren Rucksack, besorgen sich das nächstbeste Flugticket und los geht's ...

Man kann jedes Buch als eine Art Reise betrachten. Vergessen wir hierbei für einen Moment die schlechten Bücher, die Reisen ähneln, die Sie bei einem fragwürdigen Reiseveranstalter gebucht haben, der sich nach Antritt der Reise aus jeder Verantwortung stiehlt und Sie allein im tiefsten Dschungel zurücklässt, wo Sie plötzlich einem Löwen gegenüberstehen, der Sie ansieht, als sei Ihr Name »Abendessen«.

Selbst die guten Bücher (zu denen dieses hoffentlich zählt) können sich vom Stil her stark unterscheiden. Mit einigen Büchern reisen Sie in der ersten Klasse. Diese Bücher sind freundlich aufgemacht und bereiten Sie in einer luxuriösen Umgebung auf jeden Ihrer nächsten Reiseschritte vor.

Das vorliegende Buch ähnelt eher einer Abenteuerreise. Es soll für Sie eine Herausforderung darstellen und selbst die erfahrensten Programmierer ins Schlingern bringen. Dem liegt die Idee zugrunde, Ihr Wissen und Ihre Erfahrung zu erweitern, damit Sie Vertrauen in Ihre eigenen Fähigkeiten erlangen und jede Herausforderung annehmen, die sich Ihnen entgegenstellt.

Wenn Sie eher der Typ Reisender sind, der vorausplant, sollten Sie zu Teil III wechseln und die Tutorien lesen, um sich mit den Zielen vertraut zu machen, zu denen Sie demnächst vorstoßen werden.

Aber wenn Sie zu dem Typ Reisenden gehören, der sich einfach in das nächste Abenteuer stürzt, dann lesen Sie einfach weiter. Die Tutorien können Sie immer noch lesen, wenn (falls) Sie auf Probleme stoßen.

Abschnitt 1: Der Anfang

Haben Sie jemals ein Programmierhandbuch für Fortgeschrittene gekauft? Dann haben Sie sich bestimmt auch gefragt, warum das erste Kapitel mit einer ausführlichen Erläuterung von Variablen, einfachen Operationen oder Aufgaben wie z. B. dem Hinzufügen von Steuerelementen oder Formularen beginnen muss. Ich fand diese Art von Büchern schon immer sehr frustrierend. Nicht, weil etwas dagegen spräche, diese Art von Informationen zur Verfügung zu stellen, sondern weil sich die weiterführenden Informationen, nach denen ich üblicherweise suche, auf einige wenige Kapitel am Ende des Buches beschränken, wobei diese dann häufig auch noch wenig hilfreich sind.

Auf der anderen Seite müssen Sie jedoch auch das Dilemma verstehen, in dem sich jeder Autor befindet: Wenn Sie ein Buch auf einer zu weit fortgeschrittenen Ebene beginnen, riskieren Sie, eine Menge Leser zu verlieren und weitaus weniger Bücher zu verkaufen[1]. Wenn das Buch auf einer zu einfachen Ebene beginnt, frustrieren Sie den professionellen Entwickler, der es hasst, Aufgüsse von dem zu lesen, was er längst weiß. Es handelt sich also um eine Gratwanderung, die es zu absolvieren gilt.

Daher richtet sich dieses Buch an Visual Basic-Programmierer mit fortgeschrittenen Kenntnissen, die sich zu Experten mausern möchten. Die Puzzle setzen voraus, dass der Leser über eine gewisse Erfahrung im Umgang mit API-Funktionsaufrufen verfügt. Alle Leser, die diese Kenntnisse nicht besitzen, sollten unbedingt die Tutorien lesen, bevor Sie mit der Bearbeitung der Puzzle beginnen. Aufgrund dieser Annahmen habe ich mir die Freiheit genommen, auch die ersten Puzzle in diesem Abschnitt etwas schwieriger zu gestalten[2], als man vielleicht bei einem ersten Kapitel erwarten würde. Ich denke, selbst der erfahrene Programmierer wird bei diesen Puzzlen noch einmal genauer hinsehen müssen, auch wenn sich die Lösung im Nachhinein als sehr einfach herausstellt.

1. Ich weiß, dass dies ein bisschen kapitalistisch klingt, aber ich denke, es ist wichtig, dass die Leser ein wenig über die wirtschaftliche Seite der Verlagsbranche erfahren sollten, denn erst dadurch wird die Veröffentlichung vieler lausiger Bücher erklärbar. Ich habe vor kurzem ein Essay mit dem Titel »Are you learning Visual Basic backwards?« geschrieben, den Sie sich auf unserer Website unter **www.desaware.com** ansehen können.
2. OK, vielleicht in einigen Fällen auch noch etwas schwieriger.

Puzzle 1

Wo steckt denn jetzt der API-Aufruf?

Wir leben in einer kleinen Welt, und Sie können nie wissen, wann Sie Ihre Software anpassen müssen, sodass Sie auch in anderen Teilen dieser Welt ausgeführt werden kann. Obwohl es vielleicht merkwürdig erscheint, bestehen die anderen Länder weiterhin darauf, eigene Sprachen, Währungs- und Satzzeichen zu verwenden. Nicht nur das, sie verwenden auch noch andere Datums- und Zeitformate.[3]

Windows verwendet den Begriff »Ländereinstellung«, um einen Standort und dessen spezifische Merkmale zu definieren. Jede Ländereinstellung verfügt über eine eindeutige Nummer. Glücklicherweise ist es in Windows sehr einfach, die Merkmale der Ländereinstellung zu ermitteln, unter der eine Anwendung ausgeführt wird.

Nachfolgend ein einfaches Programm, mit dem Sie die Nummer der aktuellen Ländereinstellung ermitteln können:

```
Private Declare Function GetUserDefaultLcid Lib "User32" () As Long
Private Sub Command1_Click()
   Dim lcid&
   Dim info$
   lcid = GetUserDefaultLcid()
   MsgBox lcid, vbOKOnly, "User Default LCID"
End Sub
```

Ergebnisse

Laufzeitfehler 453:

```
DLL-Einsprungspunkt GetUserDefaultLcid in User32 nicht gefunden
```

Wo steckt denn nun der API-Aufruf? Berichtigen Sie die Deklaration, damit das Programm funktioniert.

3. Obwohl mir häufig vorgeworfen wird, dass ich sehr USA-bezogen eingestellt sei, möchte ich doch betonen, dass der vorangegangene Absatz sich auf jede Sprache beziehen lässt, in der Sie entwickeln.

Puzzle 2

Der letzte Fehler

Eine faszinierende Aufgabe, der sich Windows-Programmierer gelegentlich gegenübersehen, ist die Verwendung von Programmen zur Steuerung des Systems oder einer anderen Anwendung. Der erste Schritt bei solchen Programmen besteht darin, die Fensterzugriffsnummer für die andere Anwendung herauszufinden. Die API-Funktion `FindWindow` kann, bei vorhandenem Titel bzw. vorhandener Klasse, dazu verwendet werden, diese Fensterzugriffsnummer zu ermitteln. Die C-Deklaration für `FindWindow` lautet folgendermaßen:

```
HWND FindWindow(
   LPCTSTR lpClassName, // Zeiger auf den Klassennamen
   LPCTSTR lpWindowName // Zeiger auf den Fensternamen
);
```

Diese Funktion wurde so konzipiert, dass entweder die Fensterklasse (der Fenstertyp) oder deren Name (der Titel) angegeben werden kann. Sie können auch beides angeben, was allerdings nicht notwendig ist. Da im vorliegenden Beispiel der Fenstertitel verwendet werden soll, setzen Sie den Klassennamen auf NULL.

Sie könnten die Deklaration für `FindWindow` in der Datei **api32.txt** nachsehen (diese befindet sich auf der Begleit-CD-ROM zu diesem Buch), oder Sie schauen in die Datei **win32api.txt**, die im Lieferumfang von Visual Basic enthalten ist, aber warum versuchen Sie nicht, sie selbst herauszufinden?

Da `FindWindow` Bestandteil des Fenstersubsystems ist, wird `User32` verwendet (`kernel32` enthält OS-Kernfunktionen, `GDI32` enthält Grafikfunktionen; Beschreibungen hierzu finden Sie in Teil III dieses Buches unter Tutorium 1, »Auffinden von Funktionen«). Da diese Funktion Zeichenfolgenparameter verwendet, wissen Sie, dass sowohl ANSI- als auch Unicode-Einsprungpunkte vorhanden sind, d.h., Sie müssen einen Alias verwenden, um auf den ANSI-Einsprungpunkt zugreifen zu können.[4] Ein HWND ist eine Zugriffsnummer, die unter Win32 den Wertetyp `Long` aufweist, folglich ist der Rückgabewert vom Typ `Long`. Beide Parameter sind Zeichenfolgen, die `ByVal` übergeben werden müssen, damit Sie als auf NULL endende Zeichenfolgen übertragen werden. Die Deklaration lautet also:

4. Wenn Sie nicht wissen, dass API-Funktionen mit Zeichenfolgenparametern separate Einsprungpunkte aufweisen, oder keine Ahnung haben, was mit den Begriffen »ANSI«, »Unicode« oder »Einsprungpunkt« gemeint ist, dann sollten Sie dieses Puzzle unterbrechen und Tutorium 5, »Das `ByVal`-Schlüsselwort – die Lösung für 90 % aller API-Probleme«, in Teil III dieses Buches lesen.

```
Declare Function FindWindow Lib "User32" Alias "FindWindowA" _
(ByVal lpClassName As String, ByVal lpWindowName As Long) As Long
```

Das Projekt **LastErr.vbp** (auf der Begleit-CD-ROM) beinhaltet ein Formular mit einem Textfeld `txtCaption`. In dieses können Sie den Titel des Hauptfensters für ein beliebiges Programm eingeben. Über die Funktion `FindWindow` wird die Zugriffsnummer für dieses Fenster abgerufen. Anschließend verwendet das Programm die Funktion `PostMessage`, um eine WM_CLOSE-Nachricht in die Nachrichtenwarteschlange für das Fenster zu stellen – die gleiche Nachricht, die gesendet wird, wenn Sie das Feld **Schließen** oder das Systemmenü zum Schließen eines Fensters verwenden.

```
Private Declare Function PostMessage Lib "User32" Alias _
"PostMessageA" (ByVal hWnd As Long, ByVal Message As Long, ByVal _
wParam As Long, ByVal lParam As Long) As Long
Private Const WM_CLOSE = &H10
Private Sub cmdClose_Click()
   Dim WindowHandle As Long
   WindowHandle = FindWindow("", txtCaption.Text)
   If WindowHandle = 0 Then
     MsgBox "No window: Last Error was " & Err.LastDllError
     Exit Sub
   Else
     Call PostMessage(WindowHandle, WM_CLOSE, 0, 0)
   End If
End Sub
```

Ergebnisse

Die standardmäßige Bezeichnung lautet **Untitled – Notepad**, dem Namen des Anwendungsfensters eines Standardeditors. Das Programm schlägt fehl. Im Meldungsfeld wird der Fehlercode `LastError` angezeigt, ein Wert, der vom Betriebsystem ausgegeben wird und über den Fehlerinformationen bereitgestellt werden.[5] Ihre Aufgabe besteht aus zwei Arbeitsschritten:

1. Finden Sie die Bedeutung des letzten Fehlercodes heraus.
2. Ändern Sie das Programm so ab, dass es funktioniert.

5. Der Fehlercode `LastDllError` trägt unter Windows NT die Nummer 123, unter Windows 95/98 lautet die Fehlernummer 0.

Puzzle 3

Möchte Poly einen Keks?

Die GDI-Schnittstelle (Graphical Device Interface) von Windows verfügt über weitaus mehr Grafikfunktionen als in Visual Basic zur Verfügung stehen. Eine dieser Funktionen erlaubt Ihnen das Zeichnen eines Polygons mit beliebig vielen Seiten in nur einem Arbeitsschritt. Diese Polygon-Funktion wird in der C-Dokumentation folgendermaßen definiert:

```
BOOL Polygon(
HDC hdc                 // Zugriffsnummer für den Gerätekontext
CONST POINT *lpPoints,  // Zeiger auf die Scheitelpunkte des Polygons
int nCount              // Zählung der Scheitelpunkte des Polygons
);
```

Der `hdc`-Parameter stellt den Gerätekontext für das Fenster dar, in dem das Polygon gezeichnet wird. Bei dem `lpPoints`-Parameter handelt es sich um einen Zeiger auf die erste von mehreren Koordinaten, mit denen die zu verbindenden Punkte beschrieben werden. Der Parameter `nCount` bezeichnet die Anzahl der Punkte.

Können Sie das folgende Programm so abändern, dass die Figur in Abbildung P3-1 angezeigt wird?

```
' Poly Example Program
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten
Option Explicit
Private Type POINTAPI
   X As Long
   Y As Long
End Type
Private Declare Function Polygon Lib "gdi32" (hdc As Long, _
lpPoint As POINTAPI, nCount As Long) As Long
' Funktion lädt ein Array von Punkten
Private Sub LoadPointArray(ByVal Width As Long, ByVal Height _
As Long, ByVal Increment As Integer, PointArray() As POINTAPI)
   Dim curidx As Integer
   ReDim PointArray((Height \  Increment) + 2)
   Do
     curidx = curidx + 1
     PointArray(curidx).X = Width
     PointArray(curidx).Y = Height - curidx * Increment
```

```
      curidx = curidx + 1
      PointArray(curidx).X = 0
      PointArray(curidx).Y = curidx * Increment
   Loop While curidx * Increment < Height

End Sub

Private Sub Form_Paint()
   Dim points() As POINTAPI
   LoadPointArray Width, Height, 5, points()
   Call Polygon(hWnd, points(0), UBound(points) + 1)
End Sub

Private Sub Form_Resize()
   Refresh
End Sub
```

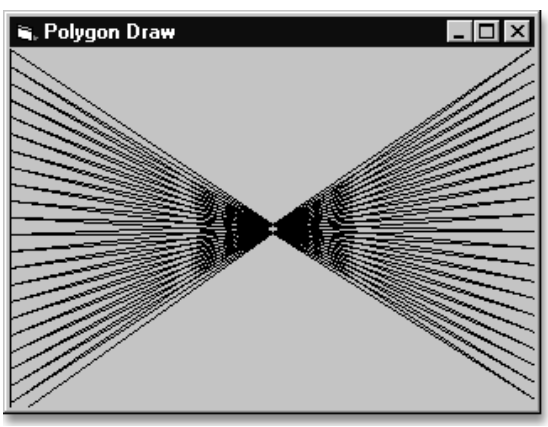


Abbildung P3-1 Richtige Anzeige für das Poly-Programm

Ergebnisse

Nach der Ausführung des oben aufgeführten Codes wird ein leeres Formular angezeigt.

Können Sie den Code berichtigen? Die Aufgabe kann schwieriger sein, als sie zunächst aussieht.

Puzzle 4

Nomen est Omen

Wir geben unseren Computern gerne Namen. Nein, diese Tatsache rührt nicht daher, dass wir einem Computer etwas menschlichere Züge verleihen möchten. Es ist halt so, dass bei der Installation von Windows ein Computername angegeben werden muss.

Glücklicherweise ist es mit Hilfe der Win32-API sehr einfach, über ein Programm den Namen des Computers zu ermitteln, auf dem dieses Programm ausgeführt wird.

In C wird die Funktion `GetComputerName` folgendermaßen deklariert:

```
BOOL GetComputerName(
LPTSTR lpBuffer,    // Adresse des Namenspuffers
LPDWORD nSize       // Adresse für die Größe des Namenspuffers
);
```

Sie könnten sich das folgende einfache Programm ausdenken, mit dem der Computername abgerufen und anschließend in einem Namensfeld angezeigt wird:

```
' Computer Name
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten

Option Explicit

Private Declare Function GetComputerName Lib "kernel32" (ByVal _
ComputerName As String, ByVal BufferSize As Long) As Long

Private Const MAX_COMPUTERNAME_LENGTH = 15

Private Sub Form_Load()
   Dim s$
   Call GetComputerName(s$, MAX_COMPUTERNAME_LENGTH + 1)
   lblName.Caption = s$
End Sub
```

Ergebnisse

Das Ergebnis mag als Kommentar zu einem Code genügen, den Sie sich einfach ausdenken. Wie wäre es, wenn Sie nun Hand anlegen und das Programm so abändern, dass es funktioniert?

Puzzle 5

Finden Sie den Namen des ausführenden Programms!

Sie kennen den Namen Ihres Programms. Schließlich haben Sie es kompiliert und sehr wahrscheinlich auch installiert. Wenn Sie jedoch Softwarekomponenten erstellen, können diese Komponenten von vielen anderen Anwendungen aufgerufen werden, und Sie wissen nicht unbedingt, welche Anwendung Ihre Komponente verwendet.

Warum sollten Sie dies wissen wollen? Der einfachste Grund dafür wäre, dass Sie die Komponente lizenzieren möchten. Sie möchten in diesem Fall ermitteln können, ob die Komponente innerhalb der Visual Basic-Laufzeitumgebung oder innerhalb eines kompilierten ausführbaren Programms verwendet wird. Wird die Komponente in der VB-Umgebung ausgeführt, können Sie überprüfen, ob eine Lizenz vorliegt, bevor Sie die Komponente zur Ausführung freigeben.

Über die Win32-API ist es sehr einfach, den Namen des Programms zu ermitteln, durch den ein Prozess ursprünglich gestartet wurde. Dies gelingt mit der Funktion `GetModuleFileName`. Diese Funktion wird im Win32 Software Development Kit (SDK) folgendermaßen definiert:

```
DWORD GetModuleFileName(
   HMODULE hModule,   // Zugriffsnummer des Moduls, nach dessen Dateiname
                      // gesucht wird
   LPTSTR lpFilename, // Zeiger auf Puffer, der den Modulpfad empfangen
                      // soll
   DWORD nSize        // Puffergröße, in Zeichenwerten
   );
```

Die Konvertierung dieser Parametertypen in Visual Basic sollte nicht schwer fallen. Bei Parameter `hModule` handelt es sich um eine Zugriffsnummer, daher muss es sich um einen 32-Bit-Wert vom Typ `Long` handeln. Der `lpFilename`-Parameter ist eine Zeichenfolge, die mit dem vollständigen Pfad sowie dem Namen der ausführbaren Datei geladen wird. Dieser Parameter muss `ByVal` als Zeichenfolge deklariert werden. Bei dem Parameter `nSize` handelt es sich um einen weiteren 32-Bit-Wert des Typs `Long`, der die Länge des Zeichenfolgenpuffers enthält. Und da die Funktion im Rahmen der Prozessverwaltung eingesetzt wird, befindet sie sich in der DLL (Dynamic Link Library) `kernel32`. Die Funktion kann also folgendermaßen deklariert werden:

```
Private Declare Function GetModuleFileName Lib "kernel32" Alias _
"GetModuleFileNameA" (ByVal hModule As Long, ByVal lpFileName As _
String, ByVal nSize As Long) As Long
```

Der Parameter `hModule` bedarf einer weiteren Erläuterung. Eine Modulzugriffsnummer unter Windows enthält die Adresse, an der das angegebene Modul in den Speicher geladen wird. Beim Start einer Anwendung wird die ausführbare Datei im Speicher zugeordnet, beginnend mit der Moduladresse für die ausführbare Datei. Jede von der Anwendung verwendete DLL wird einer anderen Adresse zugeordnet, die zur Moduladresse für diese DLL wird.[6] Mit der Funktion `GetModuleFileName` kann der Dateiname für ein beliebiges von einem Prozess verwendetes Modul abgerufen werden.

Im vorliegenden Fall wenden wir einen Trick an: Wenn Sie den Parameter `hModule` auf 0 setzen, wird über die Funktion der Name der Anwendung abgerufen, die ursprünglich den Prozess gestartet hat.

Das Projekt `ExecutableFinder` ist eine einfache ActiveX-DLL-Komponente, die ein einklassiges Objekt mit dem Namen `Server` enthält. Dieses Objekt stellt eine einzige Methode bereit, mit der das aufrufende Element ermitteln kann, ob es sich bei der Art der ausgeführten Anwendung um Visual Basic handelt. Wenn über eine Anwendung ein Serverobjekt erstellt wird, wird die `ExecutableFinder`-DLL in den Prozess geladen. Die `IsThisVB`-Methode verwendet die `GetModuleFileName`-Funktion, um zu ermitteln, ob es sich bei dem ausgeführten Programm um ein Visual Basic-Programm handelt. Wenn Sie ein Programm innerhalb der Visual Basic-Entwicklungsumgebung ausführen, gibt `GetModuleFileName` an, dass es sich, je nachdem, ob Sie VB6, VB5 oder VB4 verwenden, um das Programm **vb6.exe**, **vb5.exe** oder **vb4.exe** handelt. Wenn Sie ein kompiliertes ausführbares Programm aufrufen, wird der Pfad dieses Programms abgerufen.

Die `Server`-Klasse enthält den folgenden Code:

```
' What's the Executable?
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten
Option Explicit

Private Declare Function GetModuleFileName Lib "kernel32" Alias _
"GetModuleFileNameA" (ByVal hModule As Long, ByVal lpFileName As _
String, ByVal nSize As Long) As Long
Private Const MAX_PATH = 260
```

6. In Anhang B, »Häufig gestellte Fragen«, finden Sie weitere Informationen zu Modulzugriffsnummern.

```
Public Function IsThisVB() As Boolean
   Dim ExecName As String
   Dim LastBackslashPos As Long
   ExecName = String$(MAX_PATH + 1, 0)
   Call GetModuleFileName(0, ExecName, MAX_PATH)
   ' Jetzt den Pfad abschneiden
   LastBackslashPos = InStrRev(ExecName, "\ ")
   If LastBackslashPos = 0 Then
     LastBackslashPos = InStrRev(ExecName, ":")
   End If
   ExecName = Mid$(ExecName, LastBackslashPos + 1)
   If LCase$(ExecName) = "vb6.exe" Or _
     LCase$(ExecName) = "vb5.exe" Or _
     LCase$(ExecName) = "vb32.exe" Then
     IsThisVB = True
   End If

End Function
```

Der Parameter `ExecName` ist eine Zeichenfolge, die durch MAX_PATH + 1 Bytes initialisiert wird. Dieser Wert ist lang genug, um den längsten vom System unterstützten Pfad aufzunehmen. Die `GetModuleFileName`-Funktion lädt die Zeichenfolge mit dem vollständigen Pfad der ausführbaren Datei, durch die der Prozess gestartet wurde.

In diesem Fall ist der vollständige Pfad unwichtig – wichtig ist nur der zuletzt aufgeführte Name der ausführbaren Datei. Sie können den Pfad entfernen, indem Sie nach dem letzten Backslash suchen (oder nach dem Doppelpunkt, falls das Programm vom Stammverzeichnis aus ausgeführt wird). Mit Hilfe der Funktion `Mid$` werden alle Zeichen vor dem Backslash aus der Zeichenfolge entfernt. Abschließend wird unter Berücksichtigung der Groß- und Kleinschreibung ein Vergleich mit den drei Namen der Visual Basic-Programme durchgeführt. Die Funktion gibt den Wert `True` zurück, wenn eine Übereinstimmung mit einem der drei Programmnamen konstatiert wird.

Ergebnisse

Fügen Sie der Visual Basic-Umgebung zu Testzwecken ein einfaches Projekt hinzu, wie in der `IsThisVB`-Gruppe gezeigt.

```
' IsThisVBTest Sample Program
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten
```

```
Option Explicit

Private Sub Form_Load()
   Dim serverobject As New Server
   If serverobject.IsThisVB Then
     Label1.Caption = "Yes - it's VB"
   Else
     Label1.Caption = "No - it's not VB"
   End If
End Sub
```

Aus irgendeinem Grund kann mittels dieses Programms nicht ermittelt werden, ob die Ausführung innerhalb der Visual Basic-Umgebung erfolgt.

Ihre Aufgabe besteht darin, den Grund dafür herauszufinden.

Puzzle 6

Wo bleibt das Icon?

Die meisten Programmierer kennen sich mit Icons aus – diesen kleinen niedlichen rechteckigen Icons, die in Windows verwendet werden.[7] Icons können in speziellen Icondateien (üblicherweise mit der Erweiterung **.ico**), als Ressourcen innerhalb von Windows oder in DLLs gespeichert werden.

Windows stellt verschiedene integrierte Icons bereit, die mit Hilfe der Funktion `LoadIcon` geladen werden können. In der Win32-Dokumentation wird diese Funktion folgendermaßen definiert:

```
HICON LoadIcon(
   HINSTANCE hInstance, // Zugriffsnummer für Anwendungsinstanz
   LPCTSTR lpIconName   // Iconnamenzeichenfolge oder Ressourcenbezeichner
   );
```

Diese Beschreibung erscheint sehr einfach. Wie jede Zugriffsnummer wird der `hInstance`-Parameter als Typ `Long` definiert und als Wert übergeben. Bei dem Parameter `IpIconName` handelt es sich um eine Zeichenfolge. Der Typ lässt sich folgendermaßen beschreiben:

LP (longfar): Zeiger.

C Konstant (Constant): Dies bedeutet, dass die API-Funktion den Inhalt der Zeichenfolge nicht verändert.

T Variiert in Abhängigkeit vom Einsprungpunkt. Als Format für die Zeichenfolge wird für den ANSI-Einsprungpunkt ANSI, für den Unicode-Einsprungpunkt Unicode verwendet.

STR Auf NULL endende C-Zeichenfolge.

Die Deklaration lautet also:

```
Private Declare Function LoadIcon Lib "user32.dll" Alias "LoadIconA" _
   ( ByVal hInstance As Long, _
   ByVal lpIconName As String) As Long
```

7. Wieso rechteckig? Icons sind immer rechteckig – es ist nur so, dass Teile eines Icons transparent angezeigt werden können, so dass der Eindruck entsteht, Icons könnten viele verschiedene Formen aufweisen.

Wenn Sie eine Iconressource laden, enthält der Parameter `hInstance` die Modulzugriffsnummer der geladenen DLL. Ist der `hInstance`-Parameter NULL, wird ein SystemIcon geladen. Der `IpIconName`-Parameter gibt das zu ladende Icon an und verfügt außerdem über die folgende interessante Beschreibung:

Er zeigt auf eine in NULL endende Zeichenfolge, die den Namen der zu ladenden Iconressource enthält. Alternativ kann dieser Parameter den Ressourcenbezeichner in **low-order word** und o in **high-order word** enthalten. Verwenden Sie zum Erstellen dieses Wertes das Makro `MAKEINTRESOURCE`.

Die Funktion `MAKEINTRESOURCE` wird in der Headerdatei `winuser` folgendermaßen definiert:

```
#define MAKEINTRESOURCEA(i) (LPSTR)((DWORD)((WORD)(i)))
#define MAKEINTRESOURCEW(i) (LPWSTR)((DWORD)((WORD)(i)))
#ifdef UNICODE
#define MAKEINTRESOURCE  MAKEINTRESOURCEW
#else
#define MAKEINTRESOURCE  MAKEINTRESOURCEA
#endif // !UNICODE
```

Wenn der Parameterwert für `hInstance` NULL lautet, kann es sich bei dem `IpIconName`-Parameter um eine der folgenden vordefinierten Konstanten handeln:

```
#ifdef RC_INVOKED
#define IDI_APPLICATION     32512
#define IDI_HAND            32513
#define IDI_QUESTION        32514
#define IDI_EXCLAMATION     32515
#define IDI_ASTERISK        32516
#if(WINVER >= 0x0400)
#define IDI_WINLOGO         32517
#endif /* WINVER >= 0x0400 */
#else
#define IDI_APPLICATION     MAKEINTRESOURCE(32512)
#define IDI_HAND            MAKEINTRESOURCE(32513)
#define IDI_QUESTION        MAKEINTRESOURCE(32514)
#define IDI_EXCLAMATION     MAKEINTRESOURCE(32515)
#define IDI_ASTERISK        MAKEINTRESOURCE(32516)
#if(WINVER >= 0x0400)
#define IDI_WINLOGO         MAKEINTRESOURCE(32517)
#endif /* WINVER >= 0x0400 */
#endif /* RC_INVOKED */
```

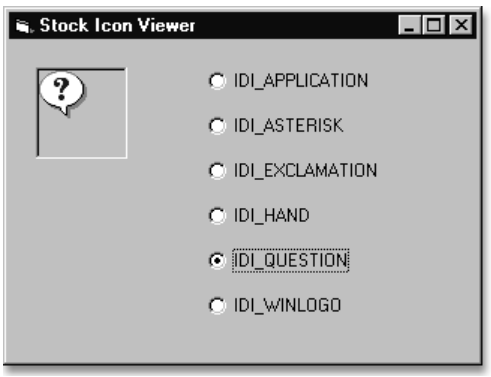


Abbildung P6-1 Das Hauptformular der Anwendung **IconView**

Abbildung P6-1 zeigt das Hauptformular der Anwendung **IconView** so, wie es normalerweise angezeigt werden sollte. Über jede Optionsschaltfläche wird ein anderes vordefiniertes Icon ausgewählt.

Die Variable `m_IconID` im **IconView**-Projekt (auf der Begleit-CD-ROM) enthält den dem jeweiligen Icon entsprechenden Konstantenwert. Während des Ereignisses `picture box Paint` wird die Funktion `LoadIcon` dazu verwendet, die aktuelle Iconressource zu laden. Die `DrawIcon`-Funktion wird zum Zeichnen der Icons im Bildfeld verwendet. Die `DrawIcon`-Funktion wird in der Win32-Dokumentation folgendermaßen definiert:

```
BOOL DrawIcon(
   HDC hdc,                  // Zugriffsnummer für Gerätekontext
   int X,                    // x-Koordinate des linken oberen Bildpunktes
   int Y,                    // y-Koordinate des linken oberen Bildpunktes
   HICON hIcon               // Zugriffsnummer des zu zeichnenden Icons
   );
```

Der Parameter `hDC` stellt den Gerätekontext des Fensters dar, in dem Sie zeichnen werden, über die Parameter `X` und `Y` wird der Iconstandort angegeben, und durch den `hIcon`-Parameter wird die Zugriffsnummer für das Icon angegeben – sämtliche 32 Bits, alle als Wert.

Hier das Beispielprogramm:

```
' IconView Puzzle
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten

Option Explicit

Private Declare Function LoadIcon Lib "user32.dll" Alias _
```

```
   "LoadIconA" ( _
   ByVal hInstance As Long, _
   ByVal lpIconName As String) As Long

Private Declare Function DrawIcon Lib "user32" (ByVal hdc As Long, _
ByVal x As Long, ByVal y As Long, ByVal hIcon As Long) As Long

Private Const IDI_APPLICATION = 32512&
Private Const IDI_HAND = 32513&
Private Const IDI_QUESTION = 32514&
Private Const IDI_EXCLAMATION = 32515&
Private Const IDI_ASTERISK = 32516&
Private Const IDI_WINLOGO = 32517&

Private m_IconID As Long

Private Sub Form_Load()
   m_IconID = IDI_APPLICATION
End Sub

Private Sub optIcon_Click(Index As Integer)
   Select Case Index
Case 0
       m_IconID = IDI_APPLICATION
     Case 1
       m_IconID = IDI_ASTERISK
     Case 2
       m_IconID = IDI_EXCLAMATION
     Case 3
       m_IconID = IDI_HAND
     Case 4
       m_IconID = IDI_QUESTION
     Case 5
       m_IconID = IDI_WINLOGO   End Select
   Picture1.Refresh
End Sub


Private Sub Picture1_Paint()
   Dim iconhandle As Long
   iconhandle = LoadIcon(0, m_IconID)
```

```
    If iconhandle = 0 Then
      MsgBox GetErrorString(Err.LastDllError)
    Else
      Call DrawIcon(Picture1.hdc, 0, 0, iconhandle)
    End If
End Sub
```

Ergebnisse

Es werden keine Icons angezeigt. Woran liegt das?

Puzzle 7

Überladene Grafiken

Zu den nützlichsten Win32-API-Funktionen gehören verschiedene Grafikfunktionen, die die Funktionen von Visual Basic mehr als in den Schatten stellen. Viele dieser Win32-Grafikfunktionen stellen nicht nur komplexe Zeichen- und Füllfunktionen bereit, sondern verfügen außerdem über eine enorme Ausführungsgeschwindigkeit. Dies ist ein Grund, weshalb in Windows viele Zeichenoperationen an die Grafikkarte oder Druckerengine weitergeleitet werden können, anstatt die Grafiken auf Ihrem Computer zu verarbeiten.

In diesem Beispiel wenden wir uns einer der einfacheren Grafikfunktionen zu, nämlich der Polyline-Funktion. In der Win32 SDK wird diese Funktion folgendermaßen definiert:

```
BOOL Polyline(
   HDC hdc,                        // Zugriffsnummer für Gerätekontext
   CONST POINT *lppt,              // Adresse des Arrays mit den Endpunkten
   int cPoints                     // Anzahl der Punkte im Array
   );
```

Daraus ergibt sich Folgendes:

- Der `hdc`-Parameter ist die Zugriffsnummer eines Gerätekontextes für Zeichenoperationen. Wie bei die meisten Zugriffsnummern handelt es sich um einen Wert vom Typ `Long`.
- Der `Ippt`-Parameter ist ein Array mit `POINT`-STRUKTUREN. Das `CONST`-Schlüsselwort gibt an, dass die Werte in diesem Array nicht durch die API-Funktion verändert werden können.
- Der `nCount`-Parameter stellt die Anzahl der Arrayeinträge dar, also die Anzahl der Linienpunkte. Durch die Funktion werden sämtliche Punkte des Arrays durch Linien in der aktuellen Farbe miteinander verbunden.

Eine `POINT`-Struktur wird in C folgendermaßen definiert:

```
typedef struct  tagPOINT
   {
   LONG x;
   LONG y;
   }  POINT;
```

Die POINT-Struktur sollte in Visual Basic wie folgt deklariert werden:

```
Type POINTAPI
     x As Long
     y As Long
End Type
```

Warum wird der Name von POINT in POINTAPI geändert? Da es sich bei POINT um ein Wort handelt, das für Visual Basic reserviert ist, kann es bei einer Ersetzung dieses Wortes durch eine benutzerdefinierte Struktur zu Missverständnissen kommen, nicht nur bei den Personen, die Ihr Programm verstehen möchten, sondern auch auf der Seite von Visual Basic selbst.

Das Polyline-Programm ist so konzipiert, dass ein Kreis in eine bestimmte Anzahl von Punkten unterteilt wird, die nach dem Zufallsprinzip durch Linien miteinander verbunden werden, wie in Abbildung P7-1 dargestellt wird.

Das Programm verfügt über ein Textfeld, in das Sie die Anzahl der Punkte eingeben können, die Schaltfläche **Löschen**, mit der Sie das Formular löschen können, die Schaltfläche **Draw Shape**, mit der Sie die Zeichenoperation ausführen, und das Kontrollkästchen **Auto**, über welches das Programm nach dem Zufallsprinzip die Punkteanzahl auswählt und die Linien basierend auf diesen Punkten einzeichnet. Auf diese Weise werden Spezialeffekte erzielt, die eine fast hypnotische Wirkung haben.

Das nachstehende Beispiel zeigt das Polyline-Programm. Das Nachvollziehen der Funktion SetupPoints kann sich als etwas schwierig erweisen, wenn Sie sich nicht einige trigonometrische Berechnungen aus Ihrer Schulzeit ins Gedächtnis zurückrufen. Es ist zur Lösung dieses Puzzles zwar nicht unbedingt erforderlich, die mathematischen Berechnungen zu verstehen, die sich hinter der Zeichenoperation verbergen, nachfolgend werde ich jedoch all denen eine kurze Erläuterung geben, die diesen Versuch wagen möchten.

Abbildung P7-2 wird Ihnen dabei helfen, die folgende Logik zu verstehen.

Im nachstehenden Beispiel weist die Form eine größere Breite als Höhe auf. Das bedeutet, dass der Radius des Kreises der halben Höhe entspricht, also ScaleHeight/2. Der vertikale Abstand zwischen dem Kreismittelpunkt und einem beliebigen Punkt auf der Kreislinie entspricht dem Radius multipliziert mit sin(A). Der Kreismittelpunkt ist (ScaleWidth/2, ScaleHeight/2), d. h. diese Werte müssen zum Abstand vom Kreismittelpunkt addiert werden, um die Position des Punktes im Formular zu berechnen. Der ScaleMode-Parameter für das Formular wird auf Pixel eingestellt, da die API-Funktion Polyline die Angabe der Koordinatenwerte in Pixeln erwartet.

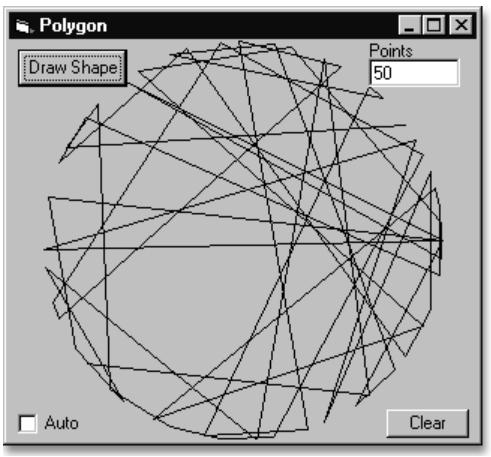


Abbildung P7-1 Das Polyline-Programm verbindet die Punkte eines Kreises nach dem Zufallsprinzip durch Linien miteinander

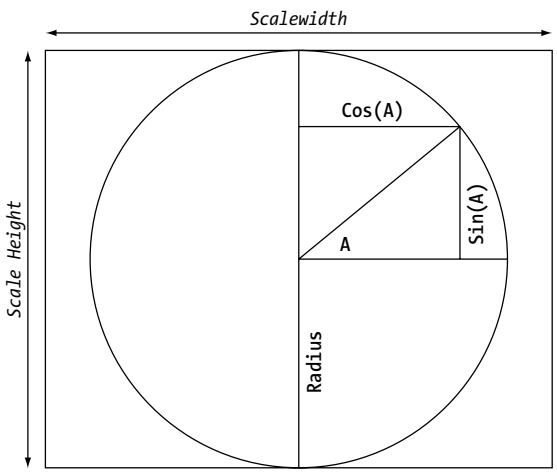


Abbildung P7-2 Darstellung der Kreisberechnung

```
' Polyline Example
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten

Option Explicit

Private Type POINTAPI
     X As Long
     y As Long
End Type
```

```
Private Declare Function Polyline Lib "gdi32" (ByVal hdc As Long, _
lppt() As POINTAPI, ByVal nCount As Long) As Long

Private PointArray() As POINTAPI

Private CurrentColorIndex As Integer

' Wir benötigen die Arccos-Funktion - arccos(0) ist Pi
Private Function Arccos(X As Double)
   Arccos = Atn(-X / Sqr(-X * X + 1)) + 2 * Atn(1)
End Function
' Diese Funktion unterteilt den Kreis in Punkte und verteilt diese beliebig
Private Sub SetupPoints(PointCount As Long, Radius As Long, Xoffset As _
Long, Yoffset As Long)
   Dim AngleIncrement As Double
   Dim PointNumber As Long
   Dim SwapPoint As Long
   Dim TempPoint As POINTAPI

   ' Arccos(0) ist Pi. 2 * Pi stellt die Anzahl der Radianten im Kreis dar.
   ' 2 * Pi /
 PointCount stellt den Winkel (in Radianten) zwischen den Punkten dar.
   AngleIncrement = 2 * Arccos(0) / PointCount
   ReDim PointArray(PointCount)
   ' Die Cos- und Sin-Funktionen erhalten die x- und y-
Werte für die Punktpositionen entlang des Kreises
   ' bei einem vorgegebenen Winkel.
   For PointNumber = 0 To PointCount - 1
     PointArray(PointNumber).X = Cos(PointNumber * PointCount) * _
     Radius + Xoffset
     PointArray(PointNumber).y = Sin(PointNumber * PointCount) * _
     Radius + Yoffset
   Next PointNumber
   ' Zurückkehren an den ursprünglichen Punkt
   LSet PointArray(PointCount) = PointArray(0)

   ' Ersten oder letzten Punkt nicht verschieben
   For PointNumber = 1 To PointCount
     ' Jeden Punkt durch einen zufällig ermittelten Punkt ersetzen
     SwapPoint = Int(Rnd() * PointCount)
     LSet TempPoint = PointArray(SwapPoint)
```

```
      LSet PointArray(SwapPoint) = PointArray(PointNumber)
      LSet PointArray(PointNumber) = TempPoint
   Next PointNumber
End Sub


Private Sub chkAuto_Click()
   Timer1.Enabled = chkAuto.Value
End Sub

Private Sub cmdClear_Click()
   ' Form löschen
   Me.Cls
End Sub

Private Sub cmdDraw_Click()
   Dim points As Long
   Dim Radius As Long

   ' Abrufen der Punkteanzahl vom Textfeld
   points = txtPoints.Text
   If points = 0 Then points = 2
   ' Der Radius ist kleiner als die halbe Höhe oder Breite
   If ScaleWidth < ScaleHeight Then
      Radius = ScaleWidth / 2
   Else
      Radius = ScaleHeight / 2
   End If

   SetupPoints points, Radius, ScaleWidth / 2, ScaleHeight / 2
   Call Polyline(hdc, PointArray(), points)
   ' Zur nächsten der 16 Standardfarben wechseln
   CurrentColorIndex = (CurrentColorIndex + 1) Mod 16
   Me.ForeColor = QBColor(CurrentColorIndex)
End Sub

Private Sub Form_Load()
   Randomize
End Sub

Private Sub Timer1_Timer()
```

```
    ' Zeichenoperation mit den aktuellen Einstellungen durchführen
    cmdDraw_Click
    ' Neuen Zählerstand für nächsten Durchlauf einstellen
    txtPoints.Text = Int(Rnd() * 50) + 5
End Sub
```

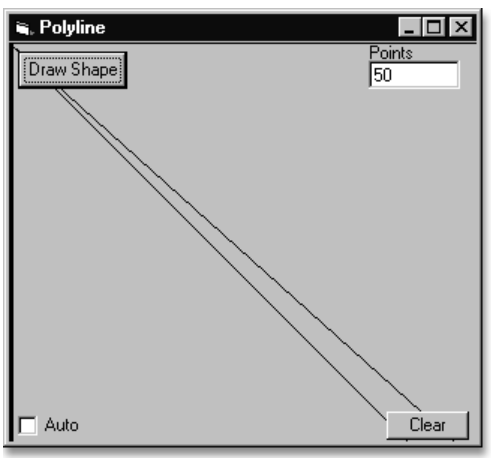


Abbildung P7-3 Das aktuell mit dem Programm gezeichnete Bild entspricht nicht dem gewünschten Ergebnis

Ergebnisse

Die `Polyline`-Funktion arbeitet sehr schnell. Sie erkennen dies, wenn Sie auf die Schaltfläche **Auto** klicken und beobachten, wie die Zeichnung automatisch aktualisiert wird.

Dies sollte eigentlich passieren, wenn Sie auf die Schaltfläche **Draw Shape** klicken. Die tatsächlichen Ergebnisse variieren allerdings von einem leeren Formular bis zu einer Zeichnung, die der in Abbildung P7-3 gezeigten ähnelt.

Können Sie das Programm reparieren?

Puzzle 8

Bockspringen

Windows enthält die API-Funktion `Getversionex`, mit der Sie über Ihre Anwendung die Version des derzeit laufenden Betriebssystems ermitteln können. Die Information über die Betriebssystemversion mag unwichtig erscheinen und ist auch unerheblich für jene Entwickler, die ausschließlich Visual Basic verwenden. Die Systemversionsangabe kann jedoch für Programmierer, die die Win32-API verwenden, äußerst wichtig sein. Warum? Weil Microsoft dem Betriebssystem ständig neue API-Funktionen hinzufügt. Über die Versionsinformation können Sie vor dem Aufruf einer Funktion sicherstellen, dass diese auch tatsächlich verfügbar ist.

Die Leistung einer Win32-Anwendung wird unter einem älteren Betriebssystem üblicherweise erheblich eingeschränkt. Mit anderen Worten: Wenn Sie die Funktionalität neuerer Systeme wie z.B. Windows 2000 voll nutzen möchten, sollten Sie über Ihre Anwendung keine Funktionen aufrufen, die von einem älteren System nicht unterstützt werden. Statt dessen sollten Sie diesen Teil der Anwendung deaktivieren, einen alternativen Ansatz zur Bereitstellung der jeweiligen Funktionalität verwenden oder dem Benutzer den Vorschlag machen, sein System aufzurüsten.

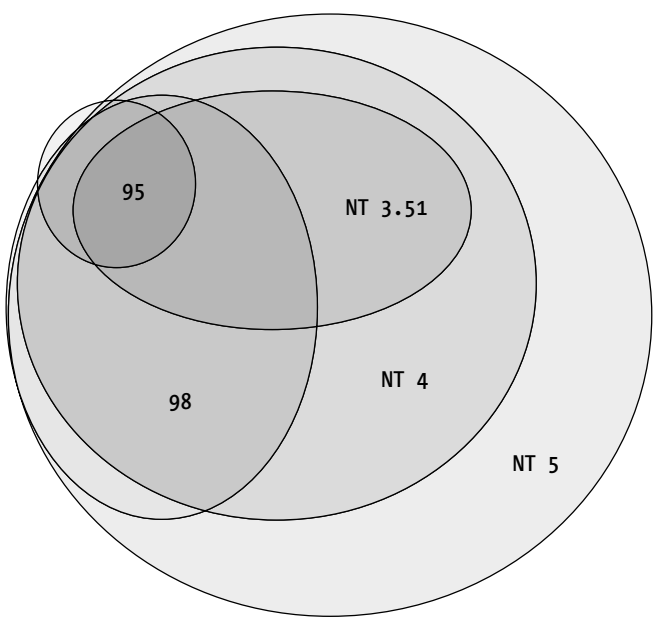


Abbildung P8-1 Jede Betriebssystemversion unterstützt unterschiedliche Win32-API-Funktionen

Unter den derzeit verwendeten Betriebssystemen (nein, wir werden die 16-Bit-Systeme nicht länger berücksichtigen, auch wenn diese z. T. noch verwendet werden) stellen Windows 95 und NT 3.51 die Systeme mit der niedrigsten Funktionalität dar. Bei beiden Systemen können verschiedene unter NT 4, und Windows 98 verfügbare Funktionen nicht verwendet werden bzw. werden nicht unterstützt. In Abbildung P8-1 sind die sich überschneidenen Funktionen zu sehen. Wie Sie erkennen können, baut jede Windows NT-Version auf dem API-Satz einer früheren Version auf. Windows NT, Version 4.0, verfügt über die meisten der Funktionen von Windows 95 und Windows 98, umgekehrt lässt sich dies leider nicht behaupten. Windows 98 unterstützt mehr NT-Funktionen als Windows 95, bei weitem jedoch nicht alle.

In Win32-Anwendungen wird die Funktion `GetVersionEx` dazu verwendet, detaillierte Informationen über das Betriebssystem abzurufen.

In der Windows-Dokumentation wird diese Funktion folgendermaßen definiert:

Mit der `GetVersionEx`-Funktion können erweiterte Informationen zur derzeit ausgeführten Betriebssystemversion abgerufen werden.

```
BOOL GetVersionEx(
  LPOSVERSIONINFO lpVersionInformation   // Zeiger auf Version
                                        // Informationsstruktur
);
```

Die Funktion verwendet als Parameter einen Zeiger auf eine `OSVERSIONINFO`-Struktur, die folgendermaßen definiert wird:

```
typedef struct _OSVERSIONINFO{
DWORD dwOSVersionInfoSize;         ' Größe der Struktur
DWORD dwMajorVersion;              ' Hauptversionsnummer
DWORD dwMinorVersion;              ' Unterversionsnummer
DWORD dwBuildNumber;               ' Build-Nummer
DWORD dwPlatformId;                ' Plattformbezeichner
TCHAR szCSDVersion[ 128 ];         ' Zeichenfolge mit zusätzlichen
                                   ' Informationen
}  OSVERSIONINFO;
```

Bei der Plattform-ID handelt es sich um eine der folgenden Konstanten:

`VER_PLATFORM_WIN32_WINDOWS`	Win32 unter Windows 95 oder Windows 98
`VER_PLATFORM_WIN32_NT`	Win32 unter Windows NT
`VER_PLATFORM_WIN32_CE`	Win32 unter Windows CE

Der Unterversionsnummer lautet bei Windows 95 0, bei Windows 98 lautet sie 10. Die Hauptversionsnummer lautet sowohl für Windows 95 als auch für Windows 98 4.

Sie können zunächst versuchsweise den folgenden Code verwenden:

```
' WinVer Project
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten

Private Type OSVERSIONINFO
   dwOSVersionInfoSize As Long
   dwMajorVersion As Long
   dwMinorVersion As Long
   dwBuildNumber As Long
   dwPlatformId As Long
   szCSDVersion As String * 128
End Type

Private Declare Function GetVersionEx Lib "kernel32" _
(os As OSVERSIONINFO) As Long

Private Const VER_PLATFORM_WIN32s = 0
Private Const VER_PLATFORM_WIN32_WINDOWS = 1
Private Const VER_PLATFORM_WIN32_NT = 2


Option Explicit

Private Sub Form_Load()
   Dim os As OSVERSIONINFO
   Dim res As Long
   Dim s$
   Dim nullpos&
   res = GetVersionEx(os)
   If res <> 0 Then ' success
     Select Case os.dwPlatformId
       Case VER_PLATFORM_WIN32_NT
         s$ = "Windows NT version " & os.dwMajorVersion _
         & os.dwMinorVersion
       Case VER_PLATFORM_WIN32_WINDOWS
         s$ = "Windows "
         If os.dwMinorVersion = 10 Then s$ = s$ & "98" Else _
```

```
          s$ = s$ & "95"
        Case Else
          s$ = "Unknown OS"
      End Select
      s$ = s$ & vbCrLf
      s$ = s$ & "Build: " & os.dwBuildNumber & vbCrLf
      nullpos = InStr(os.szCSDVersion, Chr$(0))
      If nullpos > 1 Then s$ = s$ & Left$(os.szCSDVersion, nullpos - 1)
    Else
      s$ = "GetVersionInfoEx failed"
    End If
    Label1.Caption = s$
End Sub
```

Ergebnisse

Laufzeitfehler 453: DLL-Einsprungpunkt `GetVersionEx` in `kernel32` nicht gefunden.

Können Sie das Programm reparieren?

Abschnitt 2: Alles o.k.?

Auch wenn die meisten der Puzzle in Abschnitt 1 vielleicht etwas schwierig wirken, so enthalten sie doch alle relativ übliche und einfache Fehler. Jetzt, da Sie sich ein wenig an den Debuggingprozess gewöhnt haben, können Sie sich an etwas komplexere Probleme wagen, die auch sehr häufig auftreten, deren Lösung jedoch ein besseres Verständnis derjenigen Abläufe erforderlich macht, die sich beim Aufruf einer API-Funktion im Hintergrund abspielen.

Auch jetzt können Sie sich zunächst die Tutorien ansehen, wenn Sie möchten.

Viel Glück.

Puzzle 9

Übersetzen von DEVMODE

Einer der Vorteile von Windows ist die Möglichkeit, Code zu schreiben, der nur geringfügig von den zugrunde liegenden Systemgeräten abhängt. Es ist noch nicht lange her, da musste ein Programmierer, der seine Anwendung mit Druckfunktionen ausstatten wollte, für jeden unterstützten Drucker eigene Treiber erstellen und verteilen. Heutzutage werden die Geräteeigenschaften von Windows so gut verborgen, dass die meisten Visual Basic-Programmierer sich nicht darum zu kümmern brauchen, welchen Drucker- oder Anzeigetyp das Programm verwendet.

Die Windows-API umfasst viele Funktionen, die es dem Programmierer nicht nur erlauben, die verwendeten Geräte zu ermitteln, sondern auch deren Konfiguration vorzunehmen. So haben Sie bestimmt schon einmal die **Systemsteuerung** dazu verwendet, die Einstellungen der Grafikkarte zu ändern und beispielsweise eine neue Auflösung oder Farbtiefe einzustellen. Ihnen ist wahrscheinlich aber nicht bewusst, dass Sie diese Operation auch unter Verwendung der API-Funktion `ChangeDisplaySettings` relativ einfach von jedem Programm aus durchführen können.

Diese API-Funktion verwendet als Parameter eine Struktur mit dem Namen `DEVMODE`. Die `DEVMODE`-Struktur ist eine Art Multifunktionsstruktur, die zur Beschreibung vieler Gerätetypen unter Windows eingesetzt wird, insbesondere für Drucker und Anzeigegeräte.

Die Funktion `EnumDisplaySettings` ermöglicht Ihnen auf einfache Weise, die `DEVMODE`-Struktur für jede Anzeigeeinstellung abzurufen, die von Ihrer Grafikkarte unterstützt wird. Diese Struktur bildet gleichzeitig den Kernpunkt dieses Puzzles.

In C wird die `DEVMODE`-Struktur folgendermaßen deklariert:

```
/* Größe einer Gerätenamenzeichenfolge */
#define CCHDEVICENAME 32

/* Größe einer Formularnamenzeichenfolge */
#define CCHFORMNAME 32

typedef struct _devicemodeA {
   BYTE   dmDeviceName[CCHDEVICENAME];
   WORD dmSpecVersion;
   WORD dmDriverVersion;
   WORD dmSize;
   WORD dmDriverExtra;
```

```
    DWORD dmFields;
    short dmOrientation;
    short dmPaperSize;
    short dmPaperLength;
    short dmPaperWidth;
    short dmScale;
    short dmCopies;
    short dmDefaultSource;
    short dmPrintQuality;
    short dmColor;
    short dmDuplex;
    short dmYResolution;
    short dmTTOption;
    short dmCollate;
    BYTE   dmFormName[CCHFORMNAME];
    WORD   dmLogPixels;
    DWORD  dmBitsPerPel;
    DWORD  dmPelsWidth;
    DWORD  dmPelsHeight;
    DWORD  dmDisplayFlags;
    DWORD  dmDisplayFrequency;
    DWORD  dmICMMethod;
    DWORD  dmICMIntent;
    DWORD  dmMediaType;
    DWORD  dmDitherType;
    DWORD  dmICCManufacturer;
    DWORD  dmICCModel;
    DWORD  dmPanningWidth;
    DWORD  dmPanningHeight;
}  DEVMODEA, *PDEVMODEA, *NPDEVMODEA, *LPDEVMODEA;

typedef struct _devicemodeW {
    WCHAR  dmDeviceName[CCHDEVICENAME];
    WORD dmSpecVersion;
    WORD dmDriverVersion;
    WORD dmSize;
    WORD dmDriverExtra;
    DWORD dmFields;
    short dmOrientation;
    short dmPaperSize;
    short dmPaperLength;
```

```
    short dmPaperWidth;
    short dmScale;
    short dmCopies;
    short dmDefaultSource;
    short dmPrintQuality;
    short dmColor;
    short dmDuplex;
    short dmYResolution;
    short dmTTOption;
    short dmCollate;
    WCHAR  dmFormName[CCHFORMNAME];
    WORD   dmLogPixels;
    DWORD  dmBitsPerPel;
    DWORD  dmPelsWidth;
    DWORD  dmPelsHeight;
    DWORD  dmDisplayFlags;
    DWORD  dmDisplayFrequency;
    DWORD  dmICMMethod;
    DWORD  dmICMIntent;
    DWORD  dmMediaType;
    DWORD  dmDitherType;
    DWORD  dmICCManufacturer;
    DWORD  dmICCModel;
    DWORD  dmPanningWidth;
    DWORD  dmPanningHeight;
} DEVMODEW, *PDEVMODEW, *NPDEVMODEW, *LPDEVMODEW;
```

Es werden zwei Versionen der Struktur definiert, eine zur Verwendung mit dem ANSI-Einsprungpunkt, die andere zur Verwendung mit dem Unicode-Einsprungpunkt. Diese beiden Versionen unterscheiden sich nur dadurch voneinander, dass die Zeichenfolgen innerhalb der Strukturen unter der ANSI-Version als Bytearrays und unter der Unicode-Version als WCHARs (16-Bit-Ganzzahlen) erscheinen. Die `EnumDisplaySettings`-Funktion wird in der Win32-Dokumentation folgendermaßen definiert:

```
BOOL EnumDisplaySettings(
    LPCTSTR lpszDeviceName,         // Gibt das Anzeigegerät an
    DWORD iModeNum,                 // Gibt den Anzeigemodus an
    LPDEVMODE lpDevMode             // Zeigt auf die Struktur, die
                                    // die Einstellungen erhalten soll
    );
```

Der Parameter `lpszDeviceName` sollte 0 sein, damit das aktuelle Anzeigegerät verwendet wird.

Der `iModeNum`-Parameter bezieht sich auf die Anzahl der zu untersuchenden Anzeigemodi. Der Trick bei der Aufzählung aller Anzeigeeinstellungen besteht darin, mit 0 zu beginnen und diesen Parameter so lange zu erhöhen, bis die Funktion den Wert 0 zurückgibt und damit anzeigt, dass es sich um eine ungültige Einstellung handelt.

Bei dem Parameter `lpDevMode` handelt es sich um einen Zeiger auf die `DEVMODE`-Struktur, die geladen wird, wenn die Anzeigeeinstellungen dem `iModeNum`-Parameter entsprechen.

Das Beispielprogramm ist relativ verständlich. Die `DEVMODE`-Struktur wird durch eine einfache Übersetzung der Werte von `Short` in VB `Integer`, `DWORD` in VB `Long` und `BYTE` in VB `Byte` konvertiert.

```
' DisplayModes Puzzle
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten
Option Explicit

' Größe einer Gerätenamenzeichenfolge
Private Const CCHDEVICENAME = 32

' Größe einer Formularnamenzeichenfolge
Private Const CCHFORMNAME = 32

Private Type DEVMODE
      dmDeviceName(CCHDEVICENAME) As Byte
      dmSpecVersion As Integer
      dmDriverVersion As Integer
      dmSize As Integer
      dmDriverExtra As Integer
      dmFields As Long
      dmOrientation As Integer
      dmPaperSize As Integer
      dmPaperLength As Integer
      dmPaperWidth As Integer
      dmScale As Integer
      dmCopies As Integer
      dmDefaultSource As Integer
      dmPrintQuality As Integer
      dmColor As Integer
```

```
      dmDuplex As Integer
      dmYResolution As Integer
      dmTTOption As Integer
      dmCollate As Integer
      dmFormName(CCHFORMNAME) As Byte
      dmLogPixels As Integer
      dmBitsPerPel As Long
      dmPelsWidth As Long
      dmPelsHeight As Long
      dmDisplayFlags As Long
      dmDisplayFrequency As Long
      dmICMMethod As Long
      dmICMIntent As Long
      dmMediaType As Long
      dmDitherType As Long
      dmReserved1 As Long
      dmReserved2 As Long
End Type

Private Declare Function EnumDisplaySettings Lib "user32" _
   Alias "EnumDisplaySettingsA" ( _
   ByVal lpszDeviceName As String, _
   ByVal iModeNum As Long, _
   lpDevMode As DEVMODE) As Long

Private Sub AddDisplayMode(lpDevMode As DEVMODE)
   Dim s As String
   With lpDevMode
     s = .dmPelsWidth & " x " & .dmPelsHeight & " "
     s = s & .dmBitsPerPel & " Bits/Pixel "
     If .dmDisplayFrequency > 1 Then s = s & .dmDisplayFrequency & " Hz"
   End With
   List1.AddItem s
End Sub

Private Sub Form_Load()
   Dim dm As DEVMODE
   Dim ModeNumber As Long
   Dim res As Long
   Do
     res = EnumDisplaySettings(vbNullString, ModeNumber, dm)
```

```
      If res Then
        AddDisplayMode dm
      End If
      ModeNumber = ModeNumber + 1
    Loop While res
End Sub
```

Ergebnisse

Die erzielten Ergebnisse werden in Abbildung P9-1 dargestellt.

Offensichtlich stimmt etwas nicht. Können Sie das Problem lokalisieren?

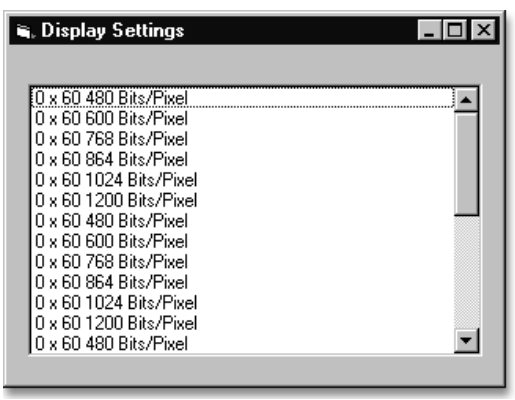


Abbildung P9-1 Ergebnisse des Puzzles **DisplayModes.vbp**

Puzzle 10

Ein umweltbezogenes Problem

Hin und wieder werden Sie auf DLL-Funktionen stoßen, die eine Zeichenfolge vom Typ `LPSTR` oder `LPTSTR` zurückgeben. Es gibt nicht allzu viele Zeichenfolgen dieser Art im Kern der Win32-API, aber einige der Standard-DLLs und Erweiterungsbibliotheken machen von diesen Zeichenfolgen Gebrauch. Hier sei als Beispiel die `GetEnvironmentStrings`-Funktion der Win32-API genannt. Computerprogramme existieren wie auch alle lebendigen Wesen im Kontext einer bestimmten Umwelt, nämlich in ihrer **Umgebung**. Die Funktion `GetEnvironmentStrings` ermöglicht es Ihnen, die Umgebungsvariablen einer Anwendung zu untersuchen. In der C-Dokumentation wird diese Funktion folgendermaßen beschrieben:

> *Jeder Prozess verfügt über einen verknüpften Umgebungsblock. Dieser Umgebungsblock besteht aus einem auf NULL endenden Block von Zeichenfolgen, die mit einem Nullwert enden (d.h., am Ende des Blocks liegen zwei NULL-Bytes vor). Jede Zeichenfolge sieht daher wie folgt aus:*
>
> *`name=value`*
>
> *`LPVOID GetEnvironmentStrings(VOID)`*
>
> *Wenn `GetEnvironmentSettings` aufgerufen wird, wird einem Block von Umgebungszeichenfolgen ein Speicherbereich zugeordnet. Wird der Block nicht länger benötigt, sollte er durch das Aufrufen von `FreeEnvironmentSettings` wieder freigegeben werden.*

Obwohl es sich bei der Deklaration um LPVOID handelt, macht die Erklärung deutlich, dass tatsächlich ein Zeiger auf eine Zeichenfolge zurückgegeben wird. In der Datei **Win32api.txt**, die im Lieferumfang von Visual Basic enthalten ist, wird die Funktion folgendermaßen deklariert:

```
Declare Function GetEnvironmentStrings Lib "kernel32" Alias "GetEnvironment
StringsA" () As String
```

Das Projekt **EnvStr.vbp** (auf der Begleit-CD-ROM zu diesem Buch) verwendet diese Funktion, um die Umgebungszeichenfolgen einer Anwendung abzurufen:

```
Private Declare Function GetEnvironmentStrings Lib "kernel32" Alias
"GetEnvironmentStringsA" () As String
Private Sub cmdGetStrings_Click()
   Dim Environment As String
   Dim CurrentPosition As Long
   Dim NewPosition
```

```
  Environment = GetEnvironmentStrings()
  Do
    ' Nach dem nächsten Nullwert suchen, der den Einzelwert trennt
    ' Umgebungsvariablen
    NewPosition = InStr(CurrentPosition + 1, Environment, Chr$(0))
    If NewPosition > CurrentPosition + 1 Then
      ' Falls eine Umgebungsvariable gefunden wird, nächste hinzufügen,
      ' bis Nullwert zurückgegeben wird
      List1.AddItem Mid$(Environment, CurrentPosition + 1, _
      NewPosition - CurrentPosition - 1)
      ' Suche beim nächsten Zeichen starten
      CurrentPosition = NewPosition
    Else
      Exit Do
    End If
  Loop While True

End Sub
```

Ergebnisse

Wir leben in einer modernen Welt, d.h., wir machen uns um unsere Umwelt (und um unsere Umgebung) Gedanken! Dieser Code macht jedoch mehr Probleme als ein vergiftetes Hühnerei. Wenn es schon nicht möglich ist, die Welt zu retten, können Sie es dann wenigstens mit diesem Programm versuchen?

Puzzle 11

Registrierungsspiele, Teil 1

Das Lesen in der Systemregistrierung ist nicht nur eine der Aufgaben, die ein Visual Basic-Programmierer häufig bewältigen, sie ist gleichzeitig die frustrierendste. Die Registrierung ist eine hierarchisch aufgebaute Datenbank, die in gewisser Weise einem Dateisystem ähnelt. Die Verzeichnisse eines Dateisystems entsprechen hierbei den Registrierungsschlüsseln. Die Dateien eines Dateisystems wiederum entsprechen den Werten, die ein bestimmter Schlüssel aufweist. In Abbildung P11-1 wird eine typische Anzeige des Windows-Registrierungs-Editors dargestellt. Unter Windows NT 4.0 verfügt die Registrierung über fünf sichtbare Stämme, bei denen es sich ausnahmslos um konstante Werte handelt. Der hervorgehobene Eintrag verfügt über den folgenden Registrierungspfad:

HKEY_LOCAL_MACHINE\ SYSTEM\ CurrentControlSet\ Control

Der rechte Fensterausschnitt in der Abbildung zeigt, dass dieser Schlüssel über drei benannte Werte verfügt – `CurrentUser`, `SystemStartOptions` und `WaitToKillServiceTimeout`. Ein Schlüssel kann auch über einen Standardwert verfügen und weist dann keinen Namen auf. Der Wert des Datentyps erscheint direkt hinter dem Namen, im vorliegenden Fall gehören alle Werte dem Typ `REG_SZ` (Text-daten) an. Auf den Datenwert folgt der Datentyp.

Ihre Aufgabe (wenn Sie sie annehmen) besteht nun darin, ein kurzes Programm zu schreiben, mit dem die Teilschlüssel unter diesem Schlüssel aufgezählt und die Werte eines jeden Schlüssel angezeigt werden. Damit die Aufgabe nicht zu schwierig wird, beschränkt sich das Beispiel nur auf diejenigen Schlüssel, die unter dem `Control`-Schlüssel aufgeführt sind, es soll also nicht weiter in die Hierarchie vorgedrungen werden. Diese Erfahrung selbst zu machen, bleibe dem Leser überlassen.

Da Sie beim Lesen der Registrierung in viele Fallen geraten können, wird diese Aufgabe in verschiedene Puzzle unterteilt. Denken Sie jedoch nicht, dass ich dies tue, um Ihnen die Sache so leicht wie möglich zu machen. Es ist lediglich so, dass ich bei der Lösung API-basierter Probleme diese Strategie oft selbst anwende: Es ist immer besser, kleine Teile der Anwendung zu programmieren, zu testen und auftretende Bugs sofort zu entfernen, wenn man sie denn findet. Auf diese Weise gewinnen Sie einen Einblick in den Entwicklungsablauf für das API-Subsytem und können so verhindern, dass sich Fehler im Code fortpflanzen.

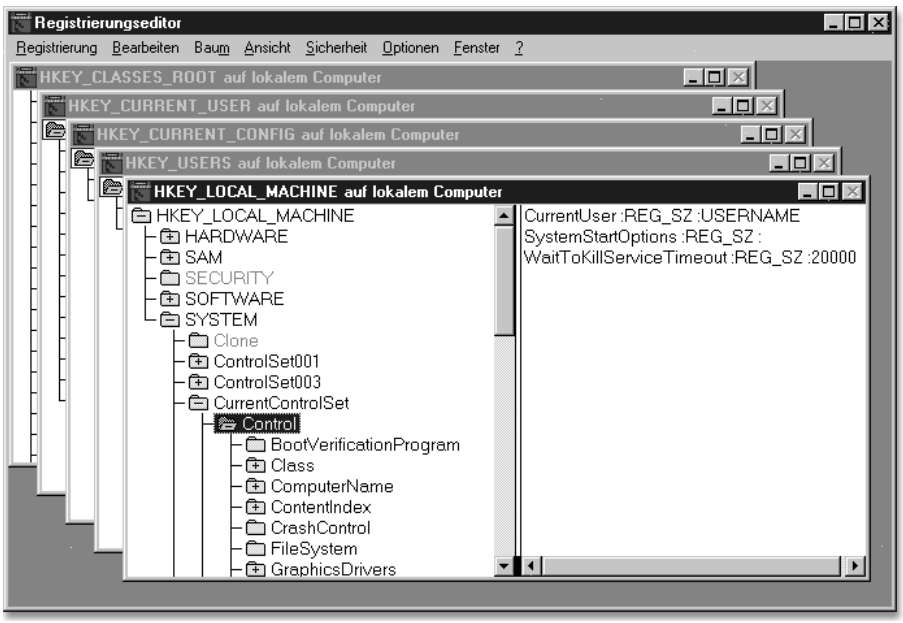


Abbildung P11-1 Eine typische Anzeige des Windows-Registrierungs-Editors

Zum Öffnen eines Registrierungsschlüssels verwenden Sie die Funktion `RegOpenKeyEx`. In der API-Dokumentation wird diese Funktion folgendermaßen definiert:

```
LONG RegOpenKeyEx(
   HKEY hKey,                    // Zugriffsnummer des
                                 // geöffneten Schlüssels
   LPCTSTR lpSubKey,             // Adresse des zu öffnenden
                                 // Teilschlüsselnamens
   DWORD ulOptions,              // reserviert
   REGSAM samDesired,            // Maske für Sicherheitszugriff
   PHKEY phkResult               // Zugriffsnummeradresse des
                                 // geöffneten Schlüssels
   );
```

Bei dem Parameter `HKEY` handelt es sich um eine Zugriffsnummer, also einen 32-Bit-Wert vom Typ `Long`. Sie finden die folgenden Stammkonstanten in der Datei **api32.txt** auf der Begleit-CD-ROM zu diesem Buch:

```
Public Const HKEY_CLASSES_ROOT = &H80000000
Public Const HKEY_CURRENT_USER = &H80000001
Public Const HKEY_LOCAL_MACHINE = &H80000002
Public Const HKEY_USERS = &H80000003
```

Public Const HKEY_PERFORMANCE_DATA = &H80000004
Public Const HKEY_CURRENT_CONFIG = &H80000005

Der `lpSubKey`-Parameter ist eine Zeichenfolge, über die der zu öffnende Teilschlüssel angegeben wird. In diesem Fall handelt es sich hierbei um `\ SYSTEM\ CurrentControlSet\ Control`.

Der Parameter `ulOptions` ist reserviert und sollte auf 0 gesetzt werden.

Bei dem `samDesired`-Parameter handelt es sich um einen `Long`-Wert, mit dem die gewünschten Zugriffsrechte definiert werden. Sie können beispielsweise die Konstante `KEY_QUERY_VALUE` angeben, wenn Sie die Werte eines Schlüssels lesen möchten. Ihre Anforderung wird mit den Zugriffsrechten verglichen, die dem Konto zugewiesen wurden, unter dem das Programm ausgeführt wird. Wenn die eingeräumten Rechte für die Anforderung ausreichen, wird die Anforderung akzeptiert, und Sie können sie ausführen.

Der Parameter `phkResult` ist eine `Long`-Variable, die mit dem geöffneten Schlüssel geladen wird.

Wenn der Schlüssel nicht länger benötigt wird, wird dieser über die separate Funktion `RegCloseKey` wieder geschlossen.

Im folgenden Programm wird zunächst versucht, den Schlüssel `Control` zu öffnen. Anschließend wird ein Bericht über den Erfolg oder Misserfolg der Operation ausgegeben:

```
' Reading Registry Keys
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten

Option Explicit
Private Declare Function RegCloseKey Lib "advapi32.dll" _
(ByVal hKey As Long) As Long
Private Declare Function RegOpenKeyEx Lib "advapi32.dll" _
Alias "RegOpenKeyExA" (ByVal hKey As Long, ByVal lpSubKey As String, _
ByVal ulOptions As Long, ByVal samDesired As Long, phkResult As Long) _
As Long
Private Const KEY_QUERY_VALUE = &H1
Private Const KEY_SET_VALUE = &H2
Private Const KEY_CREATE_SUB_KEY = &H4
Private Const KEY_ENUMERATE_SUB_KEYS = &H8
Private Const KEY_NOTIFY = &H10
Private Const SYNCHRONIZE = &H100000
```

```
Private Const STANDARD_RIGHTS_READ = &H20000
Private Const KEY_READ = ((STANDARD_RIGHTS_READ Or _
KEY_QUERY_VALUE Or KEY_ENUMERATE_SUB_KEYS Or KEY_NOTIFY) And _
(Not SYNCHRONIZE))

Dim controlkey As String
Private Const HKEY_LOCAL_MACHINE = &H80000002

Private Sub Form_Load()
   Dim rootkey As Long
   Dim res As Long

   controlkey = "SYSTEM/CurrentControlSet/Control"
   res = RegOpenKeyEx(HKEY_LOCAL_MACHINE, controlkey, KEY_READ, 0, rootkey)
   If res <> 0 Then
     MsgBox "Unable to open registry key: " & GetErrorString(res)
     Exit Sub
   Else
     MsgBox "Key opened successfully"
   End If
   Call RegCloseKey(rootkey)
End Sub
```

Sie finden die `GetErrorString`-Funktion in der Datei **ErrString.bas**. Diese Funktion verwendet die Funktion `FormatMessage`, um einen Fehlercode in eine Beschreibung einer Systemfehlermeldung zu konvertieren.

Ergebnisse

Im Meldungsfeld wird angezeigt, dass der Registrierungsschlüssel nicht geöffnet werden kann. Das System konnte die angegebene Datei nicht finden. Können Sie das Puzzle lösen?

Puzzle 12

Registrierungsspiele, Teil 2

Nun, da Sie wissen, wie man einen Registrierungsschlüssel öffnet, soll uns der nächste Schritt dahin führen, die Teilschlüssel und deren Werte aufzuzählen. Die Hauptfunktionen bei der Aufzählung von Schlüsselinformationen sind die Funktionen `RegEnumKey` und `RegEnumValue`. Win32 unterstützt darüber hinaus die Funktion `RegEnumKeyEx`, mit der Sie zusätzliche Schlüsselinformationen abrufen können wie beispielsweise den Zeitpunkt der letzten Schlüsseländerung. Für unsere Zwecke reicht jedoch die Funktion `RegEnumKey` aus. In C wird die `RegEnumKey`-Funktion wie folgt deklariert:

```
LONG RegEnumKey(
   HKEY hKey,              // Zugriffsnummer des abzufragenden Schlüssels
   DWORD dwIndex,          // Index des abzufragenden Teilschlüssels
   LPTSTR lpName,          // Pufferadresse für den Teilschlüsselnamen
   DWORD cbName            // Größe des Teilschlüsselpuffers
   );
```

Wir werden uns in Puzzle 13 die VB-Deklaration für diese Funktion ansehen. Im vorliegenden Puzzle werden wir uns der Funktionsweise dieser Funktion zuwenden. Die Funktion des `hKey`-Parameters ist offensichtlich: Es handelt sich um eine Zugriffsnummer zu dem Schlüssel, dessen Teilschlüssel aufgezählt werden sollen. Der Parameter `dwIndex` nimmt den Index des Teilschlüssels auf, dessen Namen Sie abrufen möchten. Der `lpName`-Parameter ist ein Puffer, der mit dem Teilschlüsselnamen geladen wird. Es handelt sich hierbei um einen Zeichenfolgenparameter. Weil aber über die API-Funktion die Zeichenfolge unter Einschluss von Daten geladen wird, muss die Zeichenfolgenvariable mit einer Länge initialisiert werden, die für den Schlüsselnamen ausreichend ist.

Man könnte nun – als eine von mehreren Methoden – die Funktion in einer Schleife aufrufen und dabei den `dwIndex`-Parameter so lange ansteigen lassen, bis ein Fehler auftritt. Der `lpName`-Parameter könnte mit einem sehr großen Wert initialisiert werden (laut Win32-API-Dokumentation ist eine Parametergröße von mehr als `MAX_PATH` + 1 Zeichen oder 261 Byte nie erforderlich).

Ein anderer Ansatz bei der Verwendung dieser Funktion ginge davon aus, die maximale Schlüsselnamenlänge und die Anzahl der vorhandenen Teilschlüssel vorauszusagen. Dies kann mit Hilfe der Funktion `RegQueryInfoKey` erreicht werden. Diese Funktion wird folgendermaßen deklariert:

```
LONG RegQueryInfoKey (
   HKEY hKey,                      // Zugriffsnummer des abzufragenden
                                   // Schlüssels
   LPTSTR lpClass,                 // Pufferadresse für die
                                   // Klassenzeichenfolge
   LPDWORD lpcbClass,              // Adresse für den
                                   // Klassenzeichenfolgenpuffer
   LPDWORD lpReserved,             // reserviert - muss gleich NULL sein
   LPDWORD lpcSubKeys,             // Pufferadresse für
                                   // Teilschlüsselanzahl
LPDWORD lpcbMaxSubKeyLen,          // Pufferadresse für maximale
                                   // Teilschlüsselnamenlänge
   LPDWORD lpcbMaxClassLen,        // Pufferadresse für maximale
                                   // Klassenzeichenfolgenlänge
   LPDWORD lpcValues,              // Pufferadresse für die Anzahl
                                   // der Werteeinträge
   LPDWORD lpcbMaxValueNameLen,    // Pufferadresse für maximalen Wert
                                   // der Namenslänge
   LPDWORD lpcbMaxValueLen,        // Pufferadresse für maximalen Wert
                                   // der Datenlänge
   LPDWORD lpcbSecurityDescriptor, // Pufferadresse für die Länge der
                                   // Sicherheitsdeskriptorlänge
   PFILETIME lpftLastWriteTime     // Pufferadresse für letzten
                                   // Schreibvorgang
   );
```

Sieht etwas furchteinflößend aus, oder? Lassen Sie sich nicht von der Unmenge von Parametern verunsichern. Der erste Parameter dient als Zugriffsnummer für den zu untersuchenden Schlüssel. Die übrigen Parameter sind sämtlich Zeiger auf Variablen, die zusammen mit den Informationen bezüglich des Schlüssels geladen werden. Alle Parameter verfügen über das Suffix `LP` oder `P` (Hinweis auf einen Zeiger). Die meisten dieser Zeiger verweisen auf `DWORD` (32-Bit-Variablen), die Visual Basic `Long`-Variablen verwenden. Da es sich bei allen Parametern um Zeiger handelt, werden Sie als Verweis übergeben (mit Ausnahme des `IpClass`-Parameters, der wie die meisten Zeichenfolgen als Wert übergeben wird).

Die Visual Basic-Deklaration für diese Funktion lautet deshalb:

```
Private Declare Function RegQueryInfoKey Lib "advapi32.dll" Alias _
"RegQueryInfoKeyA" (ByVal hKey As Long, ByVal lpClass As String, _
lpcbClass As Long, lpReserved As Long, lpcSubKeys As Long, _
lpcbMaxSubKeyLen As Long, lpcbMaxClassLen As Long, lpcValues As _
```

```
Long, lpcbMaxValueNameLen As Long, lpcbMaxValueLen As Long, _
lpcbSecurityDescriptor As Long, lpftLastWriteTime As FILETIME) _
As Long
```

Bei der `FILETIME`-Struktur handelt es sich um eine 64-Bit-Struktur, die Systemdaten enthält. (Die Win32-API stellt verschiedene Funktionen bereit, die zusammen mit der `FILETIME`-Struktur verwendet werden können, aber im Rahmen dieses Buches jedoch nicht näher erläutert werden sollen.)

```
Private Type FILETIME
      dwLowDateTime As Long
      dwHighDateTime As Long
End Type
```

Bevor Sie die Teilschlüssel aufzählen, sollten Sie die `RegQueryInfoKey`-Funktion dazu verwenden, um die Anzahl der Schlüssel sowie mit Hilfe der folgenden Funktion die maximale Länge der Teilschlüsselnamen zu ermitteln:

```
Private Function GetKeyInfo(ByVal hKey As Long, NumberOfKeys _
As Long, MaxKeyNameLength As Long) As Long
   Dim cbClass As Long
   Dim cSubKeys As Long
   Dim cbMaxSubKeyLen As Long
   Dim cbMaxClassLen As Long
   Dim cValues As Long
   Dim cbMaxValueNameLen As Long
   Dim cbMaxValueLen As Long
   Dim cbSecurityDescriptor As Long
   Dim ftLastWriteTime As FILETIME
   Dim res As Long

   res = RegQueryInfoKey(hKey, vbNullString, cbClass, _
         0, cSubKeys, cbMaxSubKeyLen, cbMaxClassLen, _
         cValues, cbMaxValueNameLen, cbMaxValueLen, _
         cbSecurityDescriptor, ftLastWriteTime)
   If res <> 0 Then
     MsgBox "RegQueryInfoKey error: " & GetErrorString(res)
   Else
     NumberOfKeys = cSubKeys
     MaxKeyNameLength = cbMaxSubKeyLen
   End If
   GetKeyInfo = res
End Function
```

Beachten Sie, wie `vbNullString` als Parameter für den `IpClass`-Parameter übergeben wird. Auf diese Weise wird der API-Funktion ein `NULL`-Zeiger übermittelt (ein Zeiger mit dem Wert 0). Die Funktion `RegQueryInfoKey` wurde so konzipiert, dass Sie einfach NULL-Werte für die Parameter übergeben können, zu denen Sie Informationen benötigen. Erwähnenswert wäre zudem, daß die Systemleistung verbessert wird, da Windows keine Werte berechnet, die Sie nicht angefordert haben.

Die Herausforderung

Die oben gezeigte `GetKeyInfo`-Funktion ist sehr umfangreich und ineffizient. Windows wird gezwungen, Informationen zu berechnen, die von der Funktion eigentlich nicht benötigt werden. Die Herausforderung für Sie besteht darin, eine effizientere Lösung zu finden, bei der ebenfalls die Funktion `RegQueryInfoKey` eingesetzt wird. Und wenn Sie schon einmal dabei sind, könnten Sie eigentlich auch gleich zwei Lösungen erarbeiten, die beide effizienter sind als die zuvor genannte!

Oh, ich hatte ganz vergessen zu sagen, dass das obige Beispiel bei Verwendung von Windows NT überhaupt nicht funktioniert.

Puzzle 13

Registrierungsspiele, Teil 3

Bevor Sie dazu übergehen, Die Werte aufzuzählen, lassen Sie uns kurz wiederholen, wie man bei der Aufzählung von Schlüsseln vorgeht. Die bereits zuvor genannte Funktion `RegEnumKey` wird in der Win32-API-Dokumentation folgendermaßen beschrieben:

```
LONG RegEnumKey(
   HKEY hKey,              // Zugriffsnummer des abzufragenden Schlüssels
   DWORD dwIndex,          // Index des abzufragenden Teilschlüssels
   LPTSTR lpName,          // Pufferadresse für den Teilschlüsselnamen
   DWORD cbName            // Größe des Teilschlüsselpuffers
   );
```

Die Visual Basic-Deklaration ist leicht verständlich:

```
Private Declare Function RegEnumKey Lib "advapi32.dll" Alias _
"RegEnumKeyA" (ByVal hKey As Long, ByVal dwIndex As Long, ByVal _
lpName As String, ByVal cbName As Long) As Long
```

Die `EnumerateKeys`-Funktion wurde erweitert, um eine vollständige Aufzählung durchzuführen. Hierbei wird die Funktion `GetKeyInfo3`-Funktion dazu verwendet, die Anzahl der Schlüssel sowie die benötigte Pufferlänge zu ermitteln. Die `keybuffer`-Variable wird für diese Länge vorinitialisiert (plus ein Extrabyte für den abschließenden NULL-Wert), um ein mögliches Überschreiben von Daten im Speicher auszuschließen. Die `RegEnumKey`-Funktion wird für jeden Teilschlüssel einmal aufgerufen. Bei einem erfolgreichen Aufruf wird das `clsKeyValues`-Objekt erstellt. Dieses Objekt wird zunächst mit dem Namen des Teilschlüssels geladen. Anschließend wird die Funktion `RegOpenKeyEx` dazu verwendet, eine Zugriffsnummer für den Teilschlüssel zu öffnen und abzurufen. Diese Zugriffsnummer wird der `EnumerateValues`-Funktion übergeben, die eine weitere Auflistung mit einer Liste der Teilschlüsselwerte zurückgibt. Anschließend wird der Teilschlüssel geschlossen und das Objekt `clsKeyValues`-Objekt der *`KeyCollection`*-Auflistung hinzugefügt. Diese Auflistung wird als Ergebnis der `EnumerateKeys`-Funktion wie folgt zurückgegeben:

```
' Aufzählen aller Teilschlüssel für diesen Schlüssel, Zurückgeben einer Auf
listung
' mit den clsKeyValues -Objekten
Private Function EnumerateKeys(ByVal hKey As Long) As Collection
   Dim NumberOfSubKeys As Long
   Dim MaxSubKeyLength As Long
```

```
    Dim res As Long
    Dim KeyIndex As Long
    Dim keybuffer As String
    Dim KeyCollection As New Collection
    Dim currentkey As clsKeyValues
    Dim nulloffset As Long
    Dim NewKey As Long

    res = GetKeyInfo3(hKey, NumberOfSubKeys, MaxSubKeyLength)
    If res <> 0 Then
      Exit Function
    End If

    keybuffer = String$(MaxSubKeyLength + 1, 0)

    For KeyIndex = 0 To NumberOfSubKeys - 1
      res = RegEnumKey(hKey, KeyIndex, keybuffer, MaxSubKeyLength + 1)
      If res = 0 Then
        Set currentkey = New clsKeyValues
        nulloffset = InStr(keybuffer, Chr$(0))
        currentkey.KeyName = Left$(keybuffer, nulloffset - 1)
        ' Werte hier aufzählen
        ' Teilschlüssel öffnen
        res = RegOpenKeyEx(hKey, currentkey.KeyName, 0, KEY_READ, NewKey)
        ' Anfordern einer Auflistung mit den Teilschlüsselwerten
        Set currentkey.ValueNames = EnumerateValues(NewKey)
        ' Teilschlüssel schließen
        Call RegCloseKey(NewKey)
        KeyCollection.Add currentkey
      End If
    Next KeyIndex
    Set EnumerateKeys = KeyCollection
End Function
```

Aufzählen der Werte

Die `RegEnumValue`-Funktion ist in gewisser Hinsicht flexibler als die `RegEnumKey`-Funktion, da mit ihr die Daten für jeden Wert während der Aufzählung abgerufen werden können. Wir werden die Vorteile dieser Funktion im Moment jedoch nicht einsetzen, um die Aufgabe möglichst einfach zu halten. Wie bei der `RegQueryInfoKey`-Funktion können Sie für die Parameter, die Sie nicht verwenden möch-

ten, einfach einen NULL-Zeiger an diese Funktion übergeben. Im vorliegenden Fall werden die Parameter IpType, IpData und IpcbData auf NULL gesetzt. Nachdem Sie diese drei Parameter sowie den IpReserved-Parameter eliminiert haben, stimmt die Funktion fast mit der RegEnumKey-Funktion überein, wie nachstehend deutlich wird:

```
LONG RegEnumValue(
   HKEY hKey,                    // Zugriffsnummer des
                                 // abzufragenden Schlüssels
   DWORD dwIndex,                // Index des abzufragenden Wertes
   LPTSTR lpValueName,           // Pufferadresse für die
                                 // Wertezeichenfolge
   LPDWORD lpcbValueName,        // Adresse für die
                                 // Größe des Wertpuffers
   LPDWORD lpReserved,           // reserviert - auf NULL gesetzt
   LPDWORD lpType,               // Pufferadresse für Typcode
   LPBYTE lpData,                // Pufferadresse für die Wertedaten
   LPDWORD lpcbData              // Adresse für die Größe
                                 // des Datenpuffers
   );
```

Die Visual Basic-Deklaration lautet folgendermaßen:

```
Private Declare Function RegEnumValue Lib "advapi32.dll" _
Alias "RegEnumValueA" (ByVal hKey As Long, ByVal dwIndex As Long, _
ByVal lpValueName As String, ByVal lpcbValueName As Long, _
ByVal lpReserved As Long, ByVal lpType As Long, ByVal lpData As Long, _
ByVal lpcbData As Long) As Long
```

Lassen Sie uns nun mit dem Code weitermachen, der die Werteaufzählung veranlasst. Dieser Code basiert auf dem Code, der zur Aufzählung von Schlüsseln verwendet ist und ist mit diesem nahezu identisch. Die GetValueInfo-Funktion basiert auf der GetKeyInfo3-Funktion und ruft die Anzahl der Teilwerte sowie die Größe des zur Speicherung der Wertenamen benötigten Puffers ab (siehe unten).

```
' Abrufen der Werteanzahl sowie der maximalen Wertelänge
Private Function GetValueInfo(ByVal hKey As Long, NumberOfValues, _
MaxValueNameLength) As Long
   GetValueInfo = RegQueryInfoKeyV3(hKey, vbNullString, 0, 0, 0, 0, _
   0, VarPtr(NumberOfValues), VarPtr(MaxValueNameLength), 0, 0, 0)
End Function
```

Die EnumerateValues-Funktion stimmt nahezu vollständig mit der EnumerateKeys-Funktion überein. Der einzige Unterschied zwischen diesen Funktionen besteht darin, dass anstelle der Erstellung einer Auflistung von cIsKeyValues-Objekten eine Auflistung der Wertenamen erzeugt wird, wie Sie nachstehend erkennen können.

```
Private Function EnumerateValues(ByVal hKey As Long) As Collection
   Dim NumberOfSubValues As Long
   Dim MaxSubValueLength As Long
   Dim res As Long
   Dim ValueIndex As Long
   Dim Valuebuffer As String
   Dim ValueCollection As New Collection
   Dim currentValue As clsKeyValues
   Dim nulloffset As Long

   res = GetValueInfo(hKey, NumberOfSubValues, MaxSubValueLength)
   If res <> 0 Then
     Exit Function
   End If

   Valuebuffer = String$(MaxSubValueLength + 1, 0)

   For ValueIndex = 0 To NumberOfSubValues - 1
     res = RegEnumValue(hKey, ValueIndex, Valuebuffer, _
     MaxSubValueLength, 0, 0, 0, 0)
     If res = 0 Then
       nulloffset = InStr(Valuebuffer, Chr$(0))
       If nulloffset<=1 Then
          ValueCollection.Add "Default"
       Else
         ValueCollection.Add Left$(Valuebuffer, nulloffset-1)
       End If
     End If
   Next ValueIndex
   Set EnumerateValues = ValueCollection
End Function
```

Das Programm wurde um ein zweites Listenfeld erweitert. Das `Form_Load`-Ereignis wurde so ergänzt, dass das erste Listenfeld mit der Liste der Teilschlüsselnamen geladen wird, die unter Verwendung des folgenden Codes durch die `Enumerate Keys`-Funktion zurückgegeben wird:

```
Set keylist = EnumerateKeys(rootkey)

For Each currentkey In keylist
   List1.AddItem currentkey.KeyName
Next
Set GlobalKeyCollection = keylist
```

Wenn Sie auf das Listenfeld klicken, wird die `List1_Click`-Funktion aufgerufen. Über diese Funktion wird in der `GlobalKeyCollection`-Auflistung nach dem entsprechenden `cIsKeyValues`-Objekt gesucht. Wurde das richtige Objekt gefunden, wird das zweite Listenfeld gelöscht, und es werden mit Hilfe des folgenden Codes die Wertenamen für den jeweiligen Teilschlüssel geladen:

```
Private Sub List1_Click()
   Dim thiskey As String
   Dim thisvalue As Variant
   Dim currentkey As clsKeyValues
   thiskey = List1.Text

   For Each currentkey In GlobalKeyCollection
     If currentkey.KeyName = thiskey Then
       List2.Clear
       For Each thisvalue In currentkey.ValueNames
         List2.AddItem thisvalue
       Next
     End If
   Next currentkey

End Sub
```

Ergebnisse

Natürlich funktioniert der hier gezeigte Code nicht. Wenn er das täte, wäre es ja schließlich kein Puzzle. Die Herausforderung für Sie besteht darin, das Programm zu reparieren.

Puzzle 14

Registrierungsspiele, Teil 4

Jetzt bleibt nur noch eine Aufgabe in unserem Registrierungsprojekt übrig: das Lesen von Daten aus der Registrierung. Das Programm wurde dahingehend verändert, dass die Daten in einem separaten Feld angezeigt werden, wenn Sie auf das Listenfeld mit den Registrierungswertenamen klicken, ▲Abbildung P14-1.

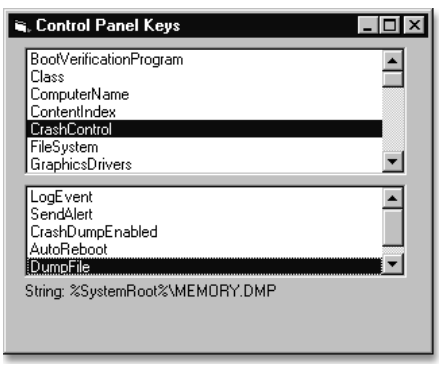


Abbildung P14-1 Die Anwendung **Reg4.vbp** während der Ausführung

Durch das `List2_Click`-Ereignis wird unter Verwendung eines Schlüsselnamens aus Listenfeld `List1` ein Schlüssel geöffnet, und anschließend wird die Funktion `DisplayValue` aufgerufen, welche die Datenwerte anzeigen soll:

```
Private Sub List2_Click()
   Dim thiskey As String
   Dim KeyToCheck As Long
   Dim res As Long

   lblValue.Caption = ""   ' Bezeichnung löschen

   res = RegOpenKeyEx(HKEY_LOCAL_MACHINE, _
   "SYSTEM\ CurrentControlSet\ Control\ " & List1.Text, _
   0, KEY_READ, KeyToCheck)
   ' Dies sollte eigentlich funktionieren, aber Vorsicht ist besser als
   ' Nachsicht
   If res = 0 Then
     Call DisplayValue(KeyToCheck, List2.Text)
     Call RegCloseKey(KeyToCheck)
   End If
End Sub
```

Die DisplayValue-Funktion verwendet zum Lesen der Registrierungswerte die Funktion RegQueryValueEx. In der Microsoft-Dokumentation wird diese Funktion folgendermaßen beschreiben:

```
LONG RegQueryValueEx(
   HKEY hKey,                    // Zugriffsnummer des
                                 // abzufragenden Schlüssels
   LPTSTR lpValueName,           // Adresse des abzufragenden
                                 // Wertenamens
   LPDWORD lpReserved,           // reserviert - muss gleich NULL sein
   LPDWORD lpType,               // Pufferadresse für Wertetyp
   LPBYTE lpData,                // Adresse des Datenpuffers
   LPDWORD lpcbData              // Adresse für Datenpuffergröße
   );
```

Bei dem Parameter hKey handelt es sich um eine Zugriffsnummer für den zu öffnenden Schlüssel, wie immer ein Wert vom Typ Long. Der IpValueName-Parameter stellt den Wertenamen dar. Der Parameter IpReserved sollte auf NULL gesetzt werden. Der IpType-Parameter ist ein Zeiger auf eine Long-Variable. Diese Variable wird mit einer Konstanten geladen, durch die der Datentyp in diesem Registrierungseintrag beschrieben wird. Eine Teilliste der Variablentypen listet Tabelle P14-1 auf.

REG_NONE = 0	Kein Wertetyp
REG_SZ = 1	Zeichenfolge, die auf NULL endet
EG_EXPAND_SZ = 2	Auf NULL endende Zeichenfolge mit Verweisen auf die noch nicht erweiterten Umgebungsvariablen
REG_BINARY = 3	Binäre Daten
REG_DWORD = 4	32-Bit-Long-Wert
REG_MULTI_SZ = 7	Eine Reihe von Zeichenfolgen, die durch NULL-Zeichenwerte getrennt werden und auf zwei NULL-Werte enden

Tabelle S14-1 Teilliste der Registrierungsdatentypen

Die Zeichenfolgendaten werden in der Registrierung als Unicode-Zeichenfolgen gespeichert, die ANSI-Version der RegQueryValueEx-Funktion ruft jedoch ANSI-Zeichenfolgen ab.

Bei dem IpData-Parameter handelt es sich um die Adresse eines Puffers, in den die Registrierungsdaten geladen werden. Der IpcbData-Parameter ist ein Zeiger auf eine Long-Variable, die die Größe des Datenpuffers enthält. Wenn der Puffer nicht groß genug ist, um die Registrierungsdaten zu speichern, gibt die Funktion den Fehler Nummer 234 zurück (ERROR_MORE_DATA), der darauf hinweist, dass ein grö-

ßerer Puffer erforderlich ist. Anschließend wird die `lpcbData`-Variable mit der erforderlichen Puffergröße geladen. Lautet der `lpData`-Parameter NULL, wird über die Funktion `RegQueryValueEx` die Variable `lpcbData` mit der benötigten Pufferlänge geladen (Fehler 234 wird in diesem Fall nicht zurückgegeben). Wenn es sich bei dem Registrierungswert um einen Zeichenfolgendatentyp handelt, umfasst die Pufferlänge den abschließenden NULL-Wert.

Die `DisplayValue`-Funktion verwendet die API-Funktion `RegQueryValueEx` zum Lesen und Anzeigen der Daten. Diese Funktionsversion verwendet nur die Daten vom Typ Zeichenfolge, `Long` oder binär, da diese Datentypen am häufigsten eingesetzt werden. Nachfolgend wird die Funktion veranschaulicht:

```
Private Sub DisplayValue(ByVal hKey As Long, ByVal ValueName As String)
   ' Laden des lblValue-Datenfeldes mit den Daten
   ' Können Sie Zeichenfolgen, binäre Datentypen und Long-Werte verwenden?
End Sub
```

Ergebnisse

Wie Sie sehen, funktioniert die Funktion einwandfrei.

- Technische Revision an Autor
  Dan, du hast vergessen, die VB-Deklaration und den Code für dieses Beispiel einzufügen. Bitte hol' das nach.
- Verleger an Technische Revision
  Kann Dan nicht erreichen. Er ist, glaube ich, in Disney World oder so. Kannst du das mit dem Code in Ordnung bringen?
- Technische Revision an Verleger
  Sorry, aber bei meinem Gehalt kann ich das Buch leider nicht komplett umschreiben. Wenn Dan sein E-Mail-Konto nicht durchsieht, während er in Urlaub ist, müssen wir die Seite wohl so übernehmen, wie sie ist. Es wird allerdings so aussehen, als sei Dan da ein ganz schön dicker Fehler unterlaufen.
- Verleger an Technische Revision
  Ich habe eine Idee. Erzähl' den Lesern einfach, dass Sie die Deklaration und den Code schreiben müssen. Dan wird schon noch einen brauchbaren Code einfügen, bevor das Buch gedruckt wird. Keiner wird merken, dass wir geschlampt haben.
- Technische Revision an Verleger
  Dies Buch ist sowieso schon reichlich merkwürdig – wahrscheinlich fällt's keinem auf. Denk' nur daran, die Bemerkungen löschen zu lassen, bevor das Buch in Druck geht.

Puzzle 15

Welche Zeitzone ist es denn nun?

Haben Sie jemals Windows 95/98 oder NT 4.0 oder höher installiert? Dann erinnern Sie sich vielleicht daran, dass Sie bei einer der letzten Installationsschritte die Zeitzone angeben müssen, innerhalb der die Installation stattfindet.[1] Vielleicht haben Sie auch schon mal in die Situation geraten, dass Sie Ihren Computer gestartet haben und Ihnen mitgeteilt wurde, dass die Uhrzeit auf die Sommer- oder Winterzeit umgestellt wurde. Windows verfügt eben über ein geniales Zeiterkennungssystem.

Die Win32-API verfügt über eine Reihe von Funktionen, die es Ihnen ermöglichen, mit Ihren Anwendungen die Vorteile der Uhrzeit- und Datumsunterstützung des Betriebssystems zu nutzen. In diesem Puzzle widmen wir uns einer einfachen Aufgabe, nämlich der Ermittlung der Zeitzone und der Zeitdifferenz zur UTC[2]. Des weiteren untersuchen wir, ob momentan die Sommer- oder die Winterzeit in Kraft ist. Die Funktion, mit der diese Information abgerufen werden kann, lautet `GetTimeZoneInformation` und wird in der Win32-API-Dokumentation folgendermaßen deklariert:

```
DWORD GetTimeZoneInformation(
   LPTIME_ZONE_INFORMATION lpTimeZoneInformation  // Adresse der
                                                  // Zeitzoneneinstellungen
   );
```

Worum handelt es sich bei `LPTIME_ZONE_INFORMATION`?

Den ersten Hinweis erhalten wir durch das Präfix `LP`. `LP` gibt an, dass es sich bei dem Parameter um einen Zeiger handelt, der auf die `TIME_ZONE_INFORMATION`-Struktur zeigt. In der Win32-API-Dokumentation wird diese Struktur wie folgt beschrieben:

```
typedef struct _TIME_ZONE_INFORMATION {  // tzi
   LONG       Bias;
   WCHAR      StandardName[ 32 ];
   SYSTEMTIME StandardDate;
   LONG       StandardBias;
   WCHAR      DaylightName[ 32 ];
```

1. O.k., wenn ich ehrlich bin, ist das eine reichlich überflüssige Frage, denn jeder, der dieses Buch liest, hat diese Betriebssysteme bestimmt schon öfter installiert als ihm lieb war.
2. UTC bedeutet Universal Time Coordinated, häufig auch Coordinated Universal Time – es sei denn, Sie ziehen die alte Variante vor, dann nennen Sie das Ganze Greenwich Mean Time oder GMT. UTC bezeichnet die Zeit bei 0° Länge, den Längengrad, der sich in Greenwich, England befindet.

```
   SYSTEMTIME DaylightDate;
   LONG       DaylightBias;
}  TIME_ZONE_INFORMATION;
```

Das `Bias`-Feld gibt eine Zeitdifferenz in Minuten von der UTC an, sodass gilt: UTC = Ihre Ortszeit + `Bias`. Über das `StandardBias`-Feld wird die zusätzliche Abweichung angegeben, die sich ergibt, wenn die Standardzeit (Winterzeit) in Kraft ist. Das `DaylightBias`-Feld enthält die zusätzliche Abweichung, die sich ergibt, wenn die Sommerzeit in Kraft ist.

Während das `StandardDate`-Feld das Datum enthält, an dem das System von der Sommerzeit auf die Winterzeit umstellen sollte, nimmt das `DaylightDate`-Feld enthält das Datum auf, an dem das System von der Winterzeit zur Sommerzeit wechseln sollte. Die Felder `StandardName` und `DaylightName` geben den Namen der Zeitzone für Sommer- bzw. Winterzeit an.

Bei `SYSTEMTIME` handelt es sich um eine weitere Struktur, die wie folgt definiert ist:

```
typedef struct _SYSTEMTIME {   // st
   WORD wYear;
   WORD wMonth;
   WORD wDayOfWeek;
   WORD wDay;
   WORD wHour;
   WORD wMinute;
   WORD wSecond;
   WORD wMilliseconds;
}  SYSTEMTIME;
```

Ein `WORD`-Parameter ist ein vorzeichenloser 16-Bit-Wert. Wir werden den Visual Basic `Integer`-Typ verwenden, da dieser eine Länge von 16-Bit aufweist und, falls nötig, jederzeit in einen vorzeichenbehafteten Wert konvertiert werden kann (was hier nicht der Fall sein wird).

Die Anwendung **tz.vbp** verwendet die `GetTimeZoneInformation`-Funktion dazu, Informationen über die aktuelle Zeitzone abzurufen und anzuzeigen.

Das Programm **tz.vbp** (auf der Begleit-CD-ROM) enthält den folgenden Code:

```
' Get Time Zone Information
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten

Option Explicit
```

```
Private Type SYSTEMTIME
      wYear As Integer
      wMonth As Integer
      wDayOfWeek As Integer
      wDay As Integer
      wHour As Integer
      wMinute As Integer
      wSecond As Integer
      wMilliseconds As Integer
End Type

Private Type TIME_ZONE_INFORMATION
      Bias As Long
      StandardName(32) As Integer
      StandardDate As SYSTEMTIME
      StandardBias As Long
      DaylightName(32) As Integer
      DaylightDate As SYSTEMTIME
      DaylightBias As Long
End Type
Private Declare Function GetTimeZoneInformation Lib _
"kernel32" (lpTimeZoneInformation As TIME_ZONE_INFORMATION) As Long

Private Const TIME_ZONE_ID_INVALID = &HFFFFFFFF
Private Const TIME_ZONE_ID_UNKNOWN = 0
Private Const TIME_ZONE_ID_STANDARD = 1
Private Const TIME_ZONE_ID_DAYLIGHT = 2

Private Sub Form_Load()
   Dim tz As TIME_ZONE_INFORMATION
   Dim info As String
   Dim res As Long
   res = GetTimeZoneInformation(tz)
   Select Case res
     Case TIME_ZONE_ID_INVALID
       lblTimeZone.Caption = "GetTimeZoneInformation function failed." _
       & vbCrLf
       lblTimeZone.Caption = lblTimeZone.Caption & "Error: " _
       & GetErrorString(Err.LastDllError)
     Case TIME_ZONE_ID_UNKNOWN
       lblTimeZone.Caption = "Time zone information unavailable."
```

```
    Case TIME_ZONE_ID_STANDARD
      info = "Currently on Standard Time" & vbCrLf
    Case TIME_ZONE_ID_DAYLIGHT
      info = "Currently on Daylight Savings Time" & vbCrLf
  End Select

  info = info & "Local time = UTC + " & tz.Bias / 60 & " hours" & vbCrLf
  info = info & "Standard time = UTC + " & tz.StandardBias / 60 & _
  " hours" & vbCrLf
  info = info & "Daylight time = UTC + " & tz.DaylightBias / 60 & _
  " hours" & vbCrLf
  info = info & "Standard time name: ???" & vbCrLf
  info = info & "Daylight time name: ???" & vbCrLf
  lblTimeZone.Caption = info
End Sub
```

Ergebnisse

In Abbildung P15-1 werden die Ergebnisse dieses Programms dargestellt, wie sie auf meinem System angezeigt werden (das sich an der Westküste der Vereinigten Staaten befindet). Die Ergebnisse für die Ortszeit (basierend auf dem `Bias`-Parameter) sind richtig: Acht Stunden Differenz zur UTC. Die Ergebnisse für `StandardBias` und `DaylightBias` ergeben jedoch keinen Sinn.

Ihre Aufgabe:

Ändern Sie das Programm so ab, dass die Informationen für `StandardBias` und `DaylightBias` richtig gelesen werden.

Finden Sie einen Weg, wie die Felder `StandardName` und `DaylighName` gelesen werden können.

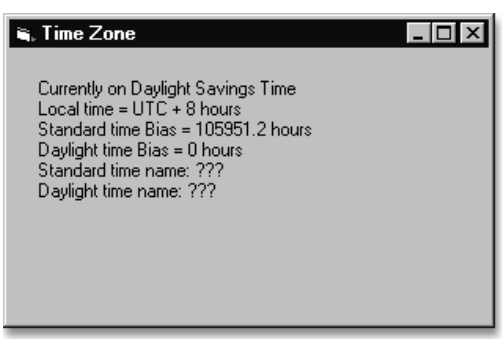


Abbildung P15-1 Typisches Ergebnis bei der Ausführung von **tz.vbp**

Puzzle 16

Seriennummern

Eine der häufigsten Fragen, die ich gestellt bekomme, lautet: Wie kann man die Seriennummer eines Datenträgervolumes abrufen? Sie erreichen dies durch Verwenden der `GetVolumeInformation`-Funktion, die in der Win32 SDK-Dokumentation wie folgt definiert wird:

```
BOOL GetVolumeInformation(
  LPCTSTR lpRootPathName,              // Adresse des Stammverzeichnisses
                                       // des Dateisystems
  LPTSTR lpVolumeNameBuffer,           // Adresse des Volumenamens
  DWORD nVolumeNameSize,               // Länge von
                                       // lpVolumeNameBuffer
  LPDWORD lpVolumeSerialNumber,        // Adresse der Volumeseriennummer
  LPDWORD lpMaximumComponentLength,    // Adresse der maximalen
                                       // Dateinamenlänge für das System
  LPDWORD lpFileSystemFlags,           // Adresse der Dateisystemflags
  LPTSTR lpFileSystemNameBuffer,       // Adresse des Dateisystemnamens
  DWORD nFileSystemNameSize            // Länge von
                                       // lpFileSystemNameBuffer
);
```

Wie viele API-Funktionen erklären sich diese Parameter fast von selbst. Der `lpRootPathName`-Parameter sollte das Stammverzeichnis des Volumes enthalten, zu dem Sie Informationen abrufen möchten. Das folgende Beispielprogramm verfügt über ein Laufwerkssteuerelement, mit dem Sie ein zu prüfendes Laufwerk auswählen können. Anschließend werden Volumename, Dateisystemname, maximale Dateinamenlänge und die Seriennummer in einem Listenfeld angezeigt.

```
' Volume Information
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten

Option Explicit

Private Declare Function GetVolumeInformation Lib "kernel32" _
Alias "GetVolumeInformationA" (ByVal lpRootPathName As String, _
ByVal lpVolumeNameBuffer As String, ByVal nVolumeNameSize As Long, _
ByVal lpVolumeSerialNumber As Long, ByVal lpMaximumComponentLength _
As Long, ByVal lpFileSystemFlags As Long, ByVal _
lpFileSystemNameBuffer As String, ByVal nFileSystemNameSize As Long) As Long
```

```
Private Sub LoadVolumeInfo()
   Dim res&
   Dim VolumeName As String
   Dim FileSystem As String
   Dim SerialNumber As Long
   Dim ComponentLength As Long
   Dim FileSystemFlags As Long
   VolumeName = String$(256, Chr$(0))
   FileSystem = String$(256, Chr$(0))
   res = GetVolumeInformation(Left$(Drive1.Drive, 2) & "\ ", _
   VolumeName, 0, SerialNumber, ComponentLength, _
   FileSystemFlags, FileSystem, 0)
   If res = 0 Then
     MsgBox "GetVolumeInformation Error: " & _
     GetErrorString(Err.LastDllError)
     Exit Sub
   End If
   List1.Clear
   List1.AddItem "Drive " & Drive1.Drive
   List1.AddItem "Name: " & VolumeName
   List1.AddItem "Component Length: " & ComponentLength
   List1.AddItem "Serial #: " & SerialNumber
   List1.AddItem "FileSystem: " & FileSystem

End Sub

Private Sub Drive1_Change()
   LoadVolumeInfo
End Sub

Private Sub Form_Load()
   LoadVolumeInfo
End Sub
```

Ergebnisse

Die von der Funktion zurückgegebenen Informationen sind eher spärlich. Informationen zu der Funktion finden Sie in der Win32 SDK-Dokumentation (oder in meinem Buch »Dan Appleman's Visual Basic Programmer's Guide to the Win32 API«). Tatsache ist jedoch, dass Sie in der Deklaration selbst bereits genügend Informationen finden, die Ihnen bei der Lösung des Problems helfen können.

Nachdem Sie das offensichtliche Problem behoben haben, sollten Ihnen Ergebnisse wie in Abbildung P16-1 angezeigt werden. Notieren Sie sich die Seriennummer, und vergleichen Sie diese mit der Seriennummer, die zurückgegeben wird, wenn Sie zur Anzeige eines Volumeverzeichnisses den Befehl `Dir` eingeben. Warum unterscheiden sich die angezeigten Seriennummern voneinander?

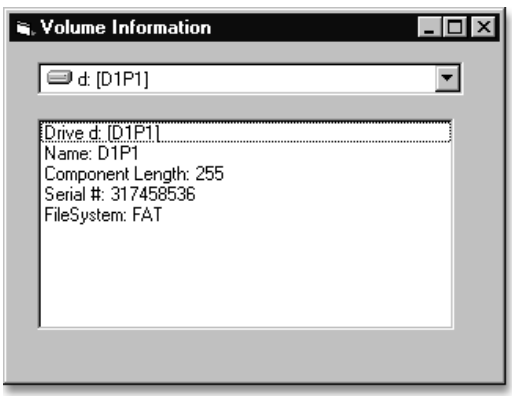


Abbildung P16-1 Ergebnis der Prüfoperation, nachdem der erste Bug behoben wurde

Abschnitt 3: Mittendrin

An dieser Stelle haben Sie bereits die Hälfte der Puzzle gelöst. Ich hoffe, die Puzzle haben Ihnen bisher nicht nur Kopfzerbrechen, sondern auch Vergnügen bereitet. Von nun an bewegt sich der Schwierigkeitsgrad der Puzzle zwischen »mittel« und »hoch«. Was bedeutet das? Es bedeutet, dass Sie selbst mit einiger Erfahrung auf dem Gebiet der API-Programmierung einige der Puzzle als äußerst schwierig beurteilen werden – so schwierig, dass viele der Leser die Puzzle nicht ohne Hilfe werden lösen können. Darüber hinaus erfordert eine Lösung – sollte sie Ihnen denn selbständig gelingen – einen erheblich höheren Zeitaufwand. Die meisten der Puzzle können nicht einfach dadurch gelöst werden, dass Sie sich den Beispielcode ansehen – Nein, höchst wahrscheinlich müssen Sie das Puzzle laden und ein bisschen kämpfen, bis Sie zu einer Lösung kommen.

Auch hier möchte ich Ihnen erneut empfehlen, nicht sofort zu den Lösungen zu wechseln. Selbst wenn Sie das Puzzle nicht lösen können, werden Sie bereits durch den Lösungsversuch zu einem besseren Verständnis der jeweiligen Programmierung gelangen, als Sie sich mit einem Dutzend weiterer Kapitel anlesen könnten.

Puzzle 17

Der DEVMODE im Detail

Die im Lieferumfang von Visual Studio enthaltene Datei **win32api.txt** enthält für die `DocumentProperties`-Funktion die folgende Deklaration:

```
Declare Function DocumentProperties Lib "winspool.drv" Alias _
"DocumentPropertiesA" (ByVal hwnd As Long, ByVal hPrinter As Long, _
ByVal pDeviceName As String, pDevModeOutput As DEVMODE, _
pDevModeInput As DEVMODE, ByVal fMode As Long) As Long
```

In meinem Win32-API-Buch wird diese Funktion wie folgt definiert:

```
Declare Function DocumentProperties Lib "winspool.drv" Alias _
"DocumentPropertiesA" (ByVal hwnd As Long, ByVal hPrinter As Long, _
ByVal pDeviceName As String, ByVal pDevModeOutput As Long, ByVal _
pDevModeInput As Long, ByVal fMode As Long) As Long
```

Hierbei werden sowohl `pDevModeOutput` als auch `pDevModeInput` als Zeiger auf die `DEVMODE`-Datenstrukturen definiert.

Die C-Deklaration lautet folgendermaßen:

```
LONG DocumentProperties(
    HWND hWnd,                  // Zugriffsnummer für Fenster mit
                                // Dialogfeld
    HANDLE hPrinter,            // Zugriffsnummer für Druckerobjekt
    LPTSTR pDeviceName,         // Zeiger auf Gerätename
    PDEVMODE pDevModeOutput,    // Zeiger auf geänderte
                                // Gerätemodusstruktur
    PDEVMODE pDevModeInput,     // Zeiger auf ursprüngliche
                                // Gerätemodusstruktur
    DWORD fMode                 // Modusattribut
   );
```

Wenn Sie Tutorium 4, »Die Arbeitsweise einer DLL: In einem Stackframe«, gelesen haben, wissen Sie, dass die Deklaration des Parameters `As DEVMODE` (wie gezeigt in **win32api.txt**) tatsächlich dazu führt, dass ein Zeiger an eine `DEVMODE`-Struktur übergeben wird. Warum würde die Deklaration in **api32.txt** zu einem komplexeren Ansatz führen und den Abruf der Adresse einer `DEVMODE`-Struktur sowie die explizite Übergabe des Zeigers erforderich machen?

Lässt sich dies auf eine Unachtsamkeit meinerseits zurückführen?

Oder kann es vielleicht sein, dass die Deklaration in der Datei **win32api.txt** falsch ist – so falsch, dass sie möglicherweise zu Speicherausnahmefehlern führt? Und darüber hinaus nicht nur zu Speicherausnahmefehlern, sondern zu Ausnahmefehlern, die periodisch auftreten, je nach verwendeter Systemkonfiguration?

Und falls dies so sein sollte, können Sie dies erklären?

Beispielprogramm

Das Beispielprogramm **DocProp** verwendet die Deklaration aus der Datei **win32api.txt**, um die Druckereinstellungen für den aktuellen Drucker anzuzeigen. Anschließend wird angezeigt, ob für den Drucker das Hoch- oder Querformat ausgewählt wurde. Der Code, mit dem diese Operation ausgeführt wird, lautet folgendermaßen (Deklarationen und Konstanten werden nicht aufgeführt):

```
   Dim hPrinter&
   Dim res&
   Dim dmIn As DEVMODE
   Dim dmOut As DEVMODE

   ' Bei Erfolg ungleich Null
   res = OpenPrinter(Printer.DeviceName, hPrinter, 0)
   If res = 0 Then
      lblStatus.Caption = "Printer could not be opened"
      Exit Sub
   End If

   res = DocumentProperties(hwnd, hPrinter, Printer.DeviceName, _
   dmOut, dmIn, DM_OUT_BUFFER Or DM_IN_PROMPT)
If dmOut.dmOrientation = DMORIENT_LANDSCAPE Then
      lblStatus.Caption = "Printer is in landscape mode"
   Else
      lblStatus.Caption = "Printer is in portrait mode"
   End If
   ' Sie können die Inhalte des dmout-Puffers bearbeiten
   ' und anschließend DocumentProperties mit DM_IN_
BUFFER aufrufen, um die neuen Werte festzulegen
   Call ClosePrinter(hPrinter)
```

Ergebnisse

Je nach System wird dieser Code entweder fehlerfrei ausgeführt, oder es tritt ein Ausnahmefehler auf. Es muß allerdings ein Drucker installiert sein, damit das **DocProp**-Beispiel ausgeführt werden kann.

Puzzle 18

DT muss nach Hause telefonieren

In den letzten Jahren habe ich mich mit einer äußerst merkwürdigen Literaturform beschäftig – dem Genre »Technischer Humor und Mystery«. Hier ein Beispiel:

Viel zu tun habe ich nicht. Ehrlich gesagt befindet sich mein Büro im falschen Viertel. Nein, es sind nicht die Blutflecken auf den zerbrochenen Fensterscheiben, auch wenn dadurch viele attraktive Menschen abgeschreckt werden. Es liegt auch nicht daran, dass ich außerhalb des Silicon Valley arbeite, das immerhin nur eine Flugstunde von Hollywood entfernt liegt. Es liegt einfach daran, dass ich nie zu einem Meeting gehe. Sie wissen schon – ich habe kein Handy, und die Batterie von meinem Pager ist auch immer leer. Nicht, dass es mich interessieren würde, ich gebe sowieso niemandem meine Pagernummer. Sicher, gelegentlich treffe ich mich mit Freunden zum Mittagessen, aber an Meetings nehme ich nicht teil.

Meine Bekannte DT dagegen eilt von einem Meeting zum nächsten. Meistens hängt sie im Juice Club herum, wo Teenager für Hungerlöhne Früchte und Gemüse mit mysteriösen Stärkungsmitteln mischen, die alles andere als ewige Jugend, Vitalität und Gesundheit versprechen. Ich habe mal einen Auftrag von der Lebensmittelbehörde bekommen – sie brauchten jemanden, der Ihre Systeme auf die Jahr-2000-Kompatibilität prüft. Ich darf Ihnen die Ergebnisse nicht verraten, nur soviel: Ab dem ersten Januar 2000 werde ich alle Lebensmittel sterilisieren, bevor ich sie esse. Und Saft werde ich nur von den Früchten trinken, die ich entweder selbst gepflückt habe oder bei denen die Packung auf das Jahr 1998 datiert ist.

Heute war nicht viel los. Mein einziger Klient war ein Typ vom Verteidigungsministerium, der eine vollständige Analyse des anhängigen Verfahrens gegen Microsoft wollte. Ich habe den Starr-Bericht kopiert und ihm die Kopie gratis überlassen. Kann denn überhaupt jemand einen Unterschied zwischen den beiden Angelegenheiten feststellen? Der Nachmittag verging nur schleppend, deshalb beschloss ich, DT zu besuchen.

Als ich sie traf, telefonierte sie gerade. Ich konnte selbst aus den wenigen Bruchstücken, die ich hörte, sofort erkennen, dass es sich um eine geschäftliche Angelegenheit drehte:

»Alles klar. Guter Deal. Sicher. Mein Computer ruft deinen Compi zurück. Ciao. Lass uns mal zusammen Mittagessen gehen.« Klick.

Ich musste lächeln. DT war mit Sicherheit die einzige Person in der Stadt, deren Handy ein Klicken von sich gab, wenn man auflegte.

Das falsche Lächeln auf Ihrem Gesicht verschwand schneller als eine Internetaktie mit schlechten Börsenbilanzen.

»Ich hasse das, aber sie zahlen sicher gut«, sagte sie, nun mit einem echten Lächeln.

»Interessant – aber was sollte die Geschichte mit dem Anruf per Computer? Rufen sich nicht normalerweise Personen gegenseitig an?«

»Das ist der letzte Schrei.« Sie schüttelte mit dem Kopf. »Jetzt, wo alle Welt ganz versessen auf Multimedia ist, möchten sie, dass der Computer einfach alles erledigt. Ich bin schon beschäftigt genug, wenn ich mein Modem für ausgehende Anrufe einrichte, aber hierfür muss ich diese Funktionalität einem der Programme hinzufügen, an denen ich gerade arbeite. Irgendwelche Ideen?«

Ich dachte einen Moment nach. »Wie wär's mit RAS?«

»Ja, daran habe ich auch gedacht. Aber die Remote Access Service-API ist nicht zu gebrauchen. Irgendwie bekomme ich das Ding nicht zum Laufen. Guck mal hier – die `RasEnumEntries`-API. Ich versuche, eine Liste der auf dem System eingerichteten Telefonbucheinträge abzufragen.«

Ich setzte mich neben sie und sah mir den Win32 SDK-Bildschirm an, den sie mir hinhielt:

```
DWORD RasEnumEntries (
    LPTSTR reserved,                // reserviert, muss NULL sein
    LPTSTR lpszPhonebook,           // Zeiger auf vollständigen Pfad
                                    // und Dateinamen
                                    // der Telefonbuchdatei
    LPRASENTRYNAME lprasentryname,  // Puffer, der die
                                    // Telefonbucheinträge empfängt
    LPDWORD lpcb,                   // Puffergröße in Byte
    LPDWORD lpcEntries              // Anzahl der in den Puffer
                                    // geschriebenen Einträge
   );
```

»Sieht doch gar nicht so schlecht aus. Der reservierte Parameter ist NULL, also definieren wir ihn `ByVal As Long` und übergeben 0. Der Beschreibung unten entnehme ich, dass man für `lpszPhonebook` auch NULL setzen kann, damit das Standardtelefonbuch verwendet wird. Hmm ... jetzt wird's schwierig.«

»Kann man wohl sagen«, antwortete sie mit grimmiger Miene. »Der `lprasentryname`-Parameter ist ein Zeiger auf ein Array von `RASENTRYNAME`-Strukturen. Die Struktur sieht so aus:«

```
typedef struct _RASENTRYNAME {
    DWORD  dwSize;
    TCHAR  szEntryName[RAS_MaxEntryName + 1];
} RASENTRYNAME;
```

»Hierbei ist `MaxEntryName` gleich 256. Wir können wahrscheinlich den üblichen Trick anwenden und das erste Element im Array angeben, aber wie groß muss das Array sein?«

»Das ist einfach«, antwortete ich. »Der `Ipcb`-Parameter ist eine `Long`-Variable, die wir mit der Größe des Puffers initialisieren müssen – also mit der Byteanzahl im `Iprasentryname`-Array. Nach der Dokumentation gibt die Funktion Fehlercode 603 zurück, wenn das Array nicht lang genug ist. Fehlercode 603 besagt, dass ein größerer Puffer benötigt wird. Und bei dem `IpcEntries`-Parameter handelt es sich um eine `Long`-Variable, die mit der Anzahl der Einträge im Array geladen wird.«

Sie las sich nochmal die Hinweise in der Dokumentation durch. »Du darfst aber nicht vergessen, dass der `dwSize`-Parameter in der Arraystruktur `Iprasentryname` auf die richtige Arraygröße eingestellt sein muss.«

Wir schrieben kurzerhand das folgende Modul, um die öffentlichen Konstanten und Deklarationen festzuhalten:

```
' RasTest #1
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten.

Option Explicit

Public Const RAS_MaxEntryName = 256
Public Const RAS_MaxEntryNameBuffer = 257

Public Type RASENTRYNAME
    dwSize As Long
    szEntryName As String * RAS_MaxEntryNameBuffer
End Type

Public Declare Function RasEnumEntries Lib "rasapi32.dll" _
Alias "RasEnumEntriesA" (ByVal reserved As Long, ByVal _
lpszPhonebook As String, lprasentryname As RASENTRYNAME, _
lpcb As Long, lpcEntries As Long) As Long
```

Die Zeichenfolgenstruktur RASENTRYNAME umfasst 257 Byte. Die Parameter Ipcb und IpcEntries werden beide als Verweis deklariert, da die Funktion diese als LPDWORD definiert – als Zeiger auf einen Long-Wert. Wir fügten dann noch mit dem folgenden Code ein Formular mit einem Listenfeld hinzu:

```
' RasTest #1
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten

Option Explicit

Private Sub Form_Load()
   Dim BufferSize As Long
   Dim RasEntries As Long
   Dim RasEntryBuffer() As RASENTRYNAME
   Dim RasEntrySize As Long
   Dim Res As Long
   Dim CurrentEntry As Integer
   Dim RasEntryCount As Long

   ReDim RasEntryBuffer(0)
   RasEntrySize = Len(RasEntryBuffer(0))
   RasEntryBuffer(0).dwSize = RasEntrySize

   Res = RasEnumEntries(0, vbNullString, RasEntryBuffer(0), _
   BufferSize, RasEntries)
   If Res = 603 Then
      ' Puffer ist zu klein - Verändern der Größe zur Größenauswahl
      RasEntryCount = BufferSize / RasEntrySize
      ReDim RasEntryBuffer(RasEntryCount - 1)
      Res = RasEnumEntries(0, vbNullString, RasEntryBuffer(0), _
      BufferSize, RasEntries)
   End If

   If Res <> 0 Then
      MsgBox "RAS Error #" & Res, vbOKOnly, "Error"
   End If

   For CurrentEntry = 0 To RasEntries - 1
      lstEntries.AddItem RasEntryBuffer(CurrentEntry).szEntryName
   Next CurrentEntry
End Sub
```

Die Funktion verfügt über eine Variable mit dem Namen `RasEntrySize`, die die Länge der Struktur `RASENTRYNAME` enthält. Wir beginnen damit, dass wir ein Array mit einem einzigen Element erstellen und das zugehörige Feld `dwSize` mit dem `RasEntrySize`-Wert initialisieren. Anschließend rufen wir die Funktion `RasEnumEntries` auf, übergeben das erste Element des Array `RasEntryBuffer`, mit dem ein Zeiger auf das gesamte Array übergeben wird. Wenn als Ergebnis Fehler 603 auftritt, erhöhen wird die Arraygröße, damit diese zur Speicherung der Einträge im Telefonbuch ausreicht. Anschließend rufen wir erneut die `RasEnumEntries`-Funktion auf. Bei einer erfolgreichen Ausführung können wir die Arrayeinträge in einer Schleife durchlaufen und dem Listenfeld die Namen hinzufügen.

DT sah mich mit einem verschlagenen Lächeln an. »Und du glaubst wirklich, das funktioniert? Sieh' dir mal die Ergebnisse an.«

Ergebnisse

Das erste, was ich bemerkte, war die Tatsache, dass von der Funktion `RasEnumEntries` der Fehler 632 zurückgegeben wurde. Dieser RAS-Fehler wird in der Datei **Raserror.h** beschrieben (Teil der Win32 DSK).[1] Die Datei enthält Konstantendeklarationen:

```
#define RASBASE 600

#define ERROR_BUFFER_TOO_SMALL              (RASBASE+3)
/*
 * Aufrufender Puffer zu klein.%0
 */
#define ERROR_BUFFER_INVALID                (RASBASE+10)
/*
 * Puffer ist ungültig.%0
 */

#define ERROR_INVALID_SIZE                  (RASBASE+32)
/*
 * Strukturgröße ist falsch.%0
 */
```

»Strukturgröße falsch? Wie kann denn die Strukturgröße falsch sein?« Das konnte ich nicht begreifen.

1. Unter Windows 95/98 wird durch den ersten Aufruf von `RasEnumEntries` der Fehler 603 erzeugt, ein richtiges Ergebnis im Hinblick darauf, dass ein größerer Puffer benötigt wird. Beim zweiten Aufruf wird jedoch Fehler 610 ausgelöst, mit dem angegeben wird, dass es sich um einen ungültigen Puffer handelt (was auch immer das bedeutet).

»Mir geht's genauso.« Sie runzelte die Stirn. »Der `dwSize`-Parameter umfasst 4 Byte, die Zeichenfolge ist 257 Byte lang, genau wie in der C-Deklaration beschrieben. Zusammen sind das 261 Byte – genau soviel, wie die `RasEntrySize`-Variable enthält. Wie kann denn dann die Strukturgröße falsch sein?«

Ich sah mir noch einmal die Headerdatei **ras.h** an. Das Einzige, was mir ein wenig komisch vorkam, war die `include`-Anweisung am Anfang der Datei:

```
#include <pshpack4.h>
```

Zusammenfassung:

Durch diese Datei wird das Packing in 4-Byte-Strukturen aktiviert.
(D.h., die automatische Ausrichtung der Strukturfelder wird deaktiviert.)

Eine `Include`-Datei wird benötigt, da verschiedene Compiler dies unterschiedlich handhaben. Bei Microsoft-kompatiblen Compilern verwendet diese Datei die push-Option des Packingsystems, sodass die `include`-Datei poppack das letzte Packing zuverlässig wiederherstellen kann.

Demnach verwenden die Strukturen in dieser Datei statt des üblichen 1-Byte-Packings das 4-Byte-Packing. DT und ich sahen einander an und griffen nach unserem Saft, den uns der Kellner in der Zwischenzeit an den Tisch gebracht hatte. Wir nahmen einen tiefen Schluck und sahen uns entsetzt an, als uns beide ein plötzlicher Schwindelanfall überfiel. War es etwa Gift?

Anmerkung des Herausgebers: An diesem Punkt endet die Geschichte. Wenn über DTs Computer ein Anruf erfolgen soll, dann müssen Sie das wohl erledigen. Stellen Sie sicher, dass in Ihrem DFÜ-Telefonbuch mindestens zwei Einträge vorhanden sind, bevor Sie mit der Bearbeitung des Problems beginnen.

Puzzle 19

Die RASDIALPARAMS-Struktur

Das RAS-System von Windows (Remote Access System) ist relativ flexibel und ausgefeilt. Es verfügt über eine Vielzahl von API-Funktionen. Unglücklicherweise kann es relativ schwierig sein, herauszufinden, welche Funktionen in einer bestimmten Umgebung verwendet werden können. Einige der Funktionen stehen unter Windows 95/98 nicht zur Verfügung. Andere setzen unter Windows NT das Vorhandensein eines der aktuellen Service Packs voran. Die `RasGetEntryDialParams`-Funktion kann dazu verwendet werden, Informationen der letzten erfolgreichen DFÜ-Verbindung für einen bestimmten Telefonbucheintrag abzurufen. Auf diese Weise können Sie ermitteln, ob der Benutzer während des letzten Anrufs zwecks Anmeldung einen der Telefonbucheinträge ändern musste. Es gibt also eine Funktion, mit der Telefonbucheinträge gelesen werden können, aber diese funktioniert unter Windows 95/98 nicht, daher müssen Sie selbst eine Lösung finden.

Die `RasGetEntryDialParams`-Funktion wird folgendermaßen definiert:

```
DWORD RasGetEntryDialParams(
    LPTSTR lpszPhonebook,              // Zeiger auf vollständigen
                                       // Pfad und Dateinamen
                                       // der Telefonbuchdatei
    LPRASDIALPARAMS lprasdialparams,   // Zeiger auf eine Struktur,
                                       // die die Verbindungsparameter
                                       // empfängt
    LPBOOL lpfPassword                 // Gibt an, ob das Benutzerkennwort
                                       // abgerufen wurde
   );
```

Diese Funktion ist sehr einfach zu handhaben. Der `IpszPhonebook`-Parameter ist als `ByVal As String` angegeben, aber Sie setzen diesen auf `vbNullString`, da mit einem NULL-Wert das Standardtelefonbuch verwendet wird. Bei dem Parameter `Iprasdialparam` handelt es sich um einen Zeiger auf die `RASDIALPARAMS`-Struktur, daher wird dieser wie die Struktur auch als Verweis deklariert. Der `IpfPassword`-Parameter ist ein `Long`-Wert, der als Verweis deklariert und mit einer Flag geladen wird, die angibt, ob ein gespeichertes Kennwort in die Struktur abgerufen wurde. Über die Funktion wird die `Iprasdialparams`-Struktur mit den verfügbaren Daten für die DFÜ-Verbindung geladen, die sich nicht im Telefonbucheintrag befanden. Dies sollte mit der folgenden Deklaration erreicht werden können:

```
Public Declare Function RasGetEntryDialParams Lib "rasapi32.dll" _
    Alias "RasGetEntryDialParamsA" ( _
    ByVal lpszPhonebook As String, _
    lprasdialparams As RASDIALPARAMS, _
    lpfPassword As Long) As Long
```

Demnach müssen Sie lediglich eine VB-Deklaration für die `RASDIALPARAMS`-Struktur erstellen. Diese wird in der C-Headerdatei folgendermaßen definiert:

```
typedef struct _RASDIALPARAMS {
   DWORD  dwSize;
   TCHAR  szEntryName[RAS_MaxEntryName + 1];
   TCHAR  szPhoneNumber[RAS_MaxPhoneNumber + 1];
   TCHAR  szCallbackNumber[RAS_MaxCallbackNumber + 1];
   TCHAR  szUserName[UNLEN + 1];
   TCHAR  szPassword[PWLEN + 1];
   TCHAR  szDomain[DNLEN + 1] ;
#if (WINVER >= 0x401)
   DWORD  dwSubEntry;
   DWORD  dwCallbackId;
#endif
}  RASDIALPARAMS;
```

Das `dwSize`-Feld muss mit der Strukturgröße geladen werden, bevor die `RasGetEntryDialParams`-Funktion aufgerufen wird.

Das RasTest-Programm aus Puzzle 18 wurde verändert, um die `RasGetEntryDialParams`-Funktion für einen Telefonbucheintrag aufzurufen, der im Listenfeld `lstEntries` der Anwendung angezeigt wird:

```
Private Sub lstEntries_Click()
   Dim rs As RASDIALPARAMS
   Dim res As Long
   Dim Password As Long
   rs.szEntryName = lstEntries.Text
   rs.dwSize = Len(rs)
   lstDetails.Clear
   res = RasGetEntryDialParams(vbNullString, rs, Password)
   If res <> 0 Then
      MsgBox "RAS error #" & res
   Else
      lstDetails.AddItem "Phone: " & rs.szPhoneNumber
      lstDetails.AddItem "User: " & rs.szUserName
```

```
      lstDetails.AddItem "Domain: " & rs.szDomain
   End If
End Sub
```

Ergebnisse

Die Ausführung des Programms **RasTest2.vbp** (auf der Begleit-CD-ROM zu diesem Buch) führt entweder zu Fehler 632 (falsche Strukturgröße) oder, was wahrscheinlicher ist, zu einem Kompilierungszeitfehler. Dies ist nicht überraschend, da die Struktur folgendermaßen definiert wird:

```
Public Type RASDIALPARAMS
   dwSize As Long
   ' Ähm?
End Type
```

Genau: Sie sollen die Deklaration für diese Struktur herausfinden.

Aber ich will Sie nicht mit dem Problem allein lassen. Hier sind einige Auszüge aus der C-Headerdatei, die Ihnen dabei helfen sollen.

Auszug aus der Datei »Ras.h«

```
#include <pshpack4.h>

#define RAS_MaxDeviceType     16
#define RAS_MaxPhoneNumber    128
#define RAS_MaxIpAddress      15
#define RAS_MaxIpxAddress     21

#if (WINVER >= 0x400)
#define RAS_MaxEntryName      256
#define RAS_MaxDeviceName     128
#define RAS_MaxCallbackNumber RAS_MaxPhoneNumber
#else
#define RAS_MaxEntryName      20
#define RAS_MaxDeviceName     32
#define RAS_MaxCallbackNumber 48
#endif
```

Auszug aus der Datei »lmcons.h«

```
//
// Zeichenfolgenlängen für verschiedene LanMan-Namen
//

#define CNLEN       15          // Länge des Computernamens
#define LM20_CNLEN  15          // Länge des LM 2.0-Computernamens
#define DNLEN       CNLEN       // Maximale Länge des Domänennamens
#define LM20_DNLEN  LM20_CNLEN  // Maximale Länge des LM 2.0-Domänennamens

#if (CNLEN != DNLEN)
#error CNLEN and DNLEN are not equal
#endif

                                //
                                // Länge für Benutzer, Gruppe und Kennwort
                                //

#define UNLEN       256         // Maximale Länge des Benutzernamens
#define LM20_UNLEN  20          // Maximale Länge des LM 2.0-Benutzernamens

#define GNLEN       UNLEN       // Gruppenname
#define LM20_GNLEN  LM20_UNLEN  // LM 2.0-Gruppenname

#define PWLEN       256         // Maximale Länge des Kennwortes
#define LM20_PWLEN  14          // Maximale Länge des LM 2.0-Kennwortes
```

Auszug aus der Datei »raserror.h«

```
#define ERROR_CANNOT_FIND_PHONEBOOK_ENTRY(RASBASE+23)
/*
 * Cannot find the phone book entry.%0

#define ERROR_INVALID_SIZE        (RASBASE+32)
/*
 * The structure size is incorrect.%0
 */
```

Wie ich herausgefunden habe, welche Dateien ich verwenden muss? Mit Hilfe der im Windows Explorer verfügbaren Dateisuche ist es leicht, nach Daten zu suchen, die in Dateien enthalten sind. Ein guter Texteditor eignet sich sogar noch besser.

Puzzle 20

Verbindungsherstellung

Abbildung P20-1 zeigt die abschließende Version des **RASTest**-Programms zur Entwicklungszeit.

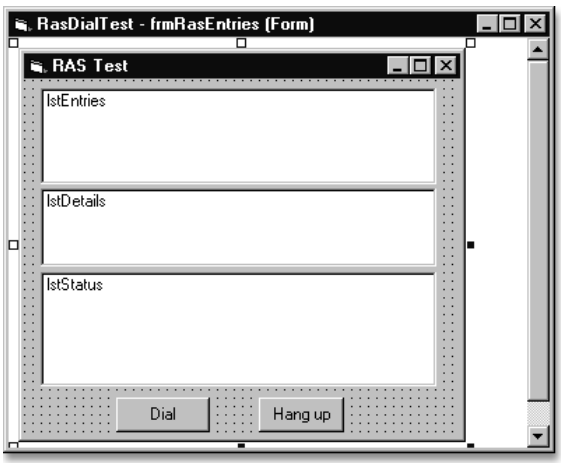


Abbildung P20-1 Das RASTest-Programm zur Entwicklungszeit

In den Puzzle 18 und 19 haben Sie gelernt, wie Sie eine Liste der Telefonbucheinträge abrufen und eine `RASDIALPARAMS`-Struktur mit Informationen zu einem Telefonbucheintrag laden können. All diese Aufgaben dienen letztlich dazu, den Wählvorgang einzuleiten und eine Verbindung herzustellen. Dies kann mit Hilfe der API-Funktion `RasDial` erreicht werden, die in der Win32-Dokumentation folgendermaßen beschrieben wird:

```
DWORD RasDial(
    LPRASDIALEXTENSIONS lpRasDialExtensions, // Zeiger auf Funktion
                                             // Erweiterungsdaten
    LPTSTR lpszPhonebook,                    // Zeiger auf vollständigen
                                             // Pfad und Dateinamen
                                             // der Telefonbuchdatei
    LPRASDIALPARAMS lpRasDialParams,         // Zeiger für den Aufruf
                                             // der Parameterdaten
    DWORD dwNotifierType,                    // Typangabe für RasDial-
                                             // Ereignisbehandlungsroutine
    LPVOID lpvNotifier,                      // Gibt eine Behandlungs-
                                             // routine für RasDial-
                                             // Ereignisse an Zeiger auf
```

```
    LPHRASCONN lphRasConn                    // Variable, die die
                                             // Verbindungszugriffsnummer
                                             // erhält
);
```

Der Parameter `LPRASDIALEXTENSIONS` stellt eine Struktur dar, die Informationen zum Wahltyp enthält. Die Standardwerte eignen sich für nahezu alle Situationen, d.h., Sie können einen NULL-Wert für diesen Parameter oder eine Struktur mit Feldern übergeben, die auf 0 gesetzt sind.

Bei dem Parameter `IpszPhonebook` handelt es sich um den gleichen Parameter, der bereits in den vorangegangenen Beispielen verwendet wurde. Dieser wird üblicherweise `ByVal as String` deklariert, der Wert wird als `vbNullString` übergeben.

Der `IpRasDialParams`-Parameter ist ein Zeiger auf eine `RASDIALPARAMS`-Struktur, die mit Informationen für den gewünschten Telefonbucheintrag geladen wird.

Bei dem Parameter `dwNotifierType` handelt es sich um einen 32-Bit-`Long`-Wert, der `ByVal` deklariert werden sollte. Hieraus ergeben sich vier mögliche Werte. Wenn Sie den `dwNotifierType`-Parameter auf -1 setzen, sollte der `IpvNotifier`-Parameter eine Zugriffsnummer für ein Fenster enthalten, in dem `WM_RASDIALEVENT`-Nachrichten[2] angezeigt werden, mit denen der Wahlvorgang beschrieben wird. Wenn der `dwNotifierType`-Parameter auf einen Wert zwischen 0 und 2 gesetzt wird, sollte der `IpvNotifier`-Parameter die Adresse einer Funktion enthalten, die zusammen mit Benachrichtigungsinformationen während des Wählvorgangs aufgerufen wird. Wenn mit Hilfe einer API-Funktion eine von Ihnen angegebene Funktionsadresse aufgerufen wird, wird die von Ihnen bereitgestellte Funktion als »Rückruffunktion« bezeichnet.

Der Parameter `LPHRASCONN` stellt einen Zeiger auf eine Variable dar, die mit der Zugriffsnummer für die jeweilige Verbindung geladen wird. Wie bei allen Win32-Zugriffsnummern sollte es sich hierbei um einen 32-Bit-`Long`-Wert handeln.

Über die Funktion `RasHangUp` wird die Verbindung für eine vorgegebene Verbindungszugriffsnummer getrennt.

2. Eine Erläuterung des Windows-Nachrichtensystems sowie dessen Unterklassifizierung finden Sie in Dan Applemans »Visual Basic Programmer's Guide to the Win32 API« und in Dan Applemans »Developing COM/ActiveX Components with Visual Basic 6.0«. Weitere Informationen, einschließlich der vollunterstützen VB-authored subclasser, finden Sie im SpyWorks-Paket von Desaware.

Die folgenden Deklarationen definieren die RASDIALEXTENSIONS-Struktur sowie die Funktionen RasDial und RasHangUp:

```
Public Type RASDIALEXTENSIONS
    dwSize As Long
    dwfOptions As Long
    hwndParent As Long
    reserved As Long
End Type

Public Declare Function RasDial Lib "rasapi32.dll" Alias "RasDialA" ( _
    lpRasDialExtensions As RASDIALEXTENSIONS, _
    ByVal lpszPhonebook As String, _
    lpRasDialParams As RASDIALPARAMS, _
    ByVal dwNotifierType As Long, _
    ByVal lpvNotifier As Long, _
    lphRasConn As Long) As Long

Public Declare Function RasHangUp Lib "rasapi32.dll" Alias _
"RasHangUpA" (ByVal hRasConn As Long) As Long

Dim hRasConn As Long
```

Durch das **RasDial**-Beispielprogramm wird ein neues Listenfeld mit Namen lstStatus hinzugefügt, das zusammen mit Benachrichtigungen geladen wird, die von der Rückruffunktion empfangen werden. Die Rückruffunktion ruft eine öffentliche Methode für das Formular UpdateDialStatus auf, wodurch wiederum das Listenfeld mit den Benachrichtigungsinformationen wie folgt geladen wird:

```
Public Sub UpdateDialStatus(ByVal RasConnState As Long, status As String)
   lstStatus.AddItem status
End Sub
```

Der Wählvorgang wird mit Hilfe der hier gezeigten Funktion cmdDial_Click gestartet:

```
Private Sub cmdDial_Click()
   Dim dialext As RASDIALEXTENSIONS
   Dim rs As RASDIALPARAMS
   Dim res As Long
   Dim Password As Long

   If hRasConn <> 0 Then
```

```
      MsgBox "Connection already open"
      Exit Sub
   End If

   lstStatus.Clear
   dialext.dwSize = Len(dialext)

   rs.szEntryName = lstEntries.Text & Chr$(0)
   rs.dwSize = Len(rs)
   res = RasGetEntryDialParams(vbNullString, rs, Password)

   res = RasDial(dialext, vbNullString, rs, 0, AddressOf RasFunc, hRasConn)

End Sub

Private Sub cmdHangup_Click()
   If hRasConn = 0 Then
      MsgBox "No RAS connection open"
      Exit Sub
   End If
   Call RasHangUp(hRasConn)
End Sub
```

Bei `dialext` handelt es sich um eine `RASDIALEXTENSIONS`-Struktur. Hier wird nur das Feld `dwSize` gesetzt, damit die API-Funktionen dieses als gültige Struktur anerkennen. Für alle Strukturfelder wird der Wert 0 beibehalten, damit die Standardwerte verwendet werden.

Anschließend löscht die Funktion das Statuslistenfeld und verwendet die Funktion `RasGetEntryDialParams` dazu, die Informationen bezüglich eines angegebenen Telefonbucheintrags zu laden. Über die `RasDial`-Funktion wird der `dwNotifierType`-Parameter auf 0 gesetzt, um der Funktion mitzuteilen, dass es sich bei dem `IpvNotifier`-Parameter um eine Funktionsadresse handelt, die auf einem Prototyp mit Namen `RasDialFunc` basiert. Der `AddressOf`-Operator kann dazu benutzt werden, die Speicheradresse einer Funktion in einem Standardmodul abzurufen. In der Win32-Dokumentation wird eine `RasDialFunc`-Funktion folgendermaßen beschrieben:

```
VOID WINAPI RasDialFunc(
    UINT unMsg,                  // Aufgetretener Ereignistyp
    RASCONNSTATE rasconnstate,   // Noch einzugebender Verbindungsstatus
    DWORD dwError                // Eventuell aufgetretener Fehler
   );
```

Bei dem unMsg-Parameter dieser Funktion handelt es sich um die Nachricht WM_RASDIALEVENT, die folgendermaßen definiert wird:

```
#define WM_RASDIALEVENT 0xCCCD
```

Sie brauchen sich hierüber jedoch keine Sorgen zu machen, da der Wert von Windows an Ihre Funktion übergeben wird. Wichtiger ist der rasconnstate-Parameter, der einen Wert enthält, mit dem das aktuelle RAS-Ereignis beschrieben wird. Bei dem Typ RASCONNSTATE handelt es sich um eine Aufzählung, also einen 32-Bit-Wert. Die Aufzählung wird folgendermaßen definiert:

```
#define RASCS_PAUSED 0x1000
#define RASCS_DONE   0x2000

#define RASCONNSTATE enum tagRASCONNSTATE
RASCONNSTATE
{
    RASCS_OpenPort = 0,
    RASCS_PortOpened,
    RASCS_ConnectDevice,
    RASCS_DeviceConnected,
    RASCS_AllDevicesConnected,
    RASCS_Authenticate,
    RASCS_AuthNotify,
    RASCS_AuthRetry,
    RASCS_AuthCallback,
    RASCS_AuthChangePassword,
    RASCS_AuthProject,
    RASCS_AuthLinkSpeed,
    RASCS_AuthAck,
    RASCS_ReAuthenticate,
    RASCS_Authenticated,
    RASCS_PrepareForCallback,
    RASCS_WaitForModemReset,
    RASCS_WaitForCallback,
    RASCS_Projected,

#if (WINVER >= 0x400)
    RASCS_StartAuthentication,
    RASCS_CallbackComplete,
    RASCS_LogonNetwork,
#endif
```

```
    RASCS_SubEntryConnected,
    RASCS_SubEntryDisconnected,

    RASCS_Interactive = RASCS_PAUSED,
    RASCS_RetryAuthentication,
    RASCS_CallbackSetByCaller,
    RASCS_PasswordExpired,

    RASCS_Connected = RASCS_DONE,
    RASCS_Disconnected
} ;

#define LPRASCONNSTATE RASCONNSTATE*
```

Der Parameter `dwError` der Funktion `RasDialFunc` ist ein `Long`-Wert, dessen Bedeutung sich nach dem aufgetretenen `RAS`-Ereignis richtet.

Sie benötigen für die Funktion `RasDialFunc` keine `Declare`-Anweisung. Warum? Da es sich bei dieser Funktion nicht um eine der Windows-DLLs handelt. Diese Funktion befindet sich in einem Standardmodul Ihrer eigenen Anwendung. Windows führt einen Aufruf Ihrer Anwendung durch und gibt die Steuerung an Sie ab.

Sie müssen jedoch eine Funktion in einem Standardmodul erstellen, die von Windows aufgerufen werden kann. Hierbei ist es wichtig, dass die Parameter Ihrer Funktion exakt mit den Werten übereinstimmen, die Windows beim Aufruf dieser Funktion verwendet. Das Modul `modRasTest` enthält die folgende Funktion:

```
Public Function RasFunc(unMsg As Long, RasConnState As Long, _
dwError As Long) As Long
   Dim result As String
   Select Case RasConnState
      Case RASCS_OpenPort
         result = "Opening port"
      Case RASCS_PortOpened
         result = "Port opened"
      Case RASCS_ConnectDevice
         result = "Connecting to device"
      Case RASCS_DeviceConnected
         result = "Device connected"
      Case RASCS_AllDevicesConnected
         result = "All devices connected"
      Case RASCS_Authenticate
         result = "Authenticating"
```

```
Case RASCS_AuthNotify
   If dwError = 0 Then
      result = "Authentication Complete"
   Else
      result = "Authentication Failed"
   End If
Case RASCS_AuthRetry
   result = "Authentication Retry Requested"
Case RASCS_AuthCallback
   result = "Remote server requested a callback"
Case RASCS_AuthChangePassword
   result = "Client requested a password change"
Case RASCS_AuthProject
   result = "Projection phase starting"
Case RASCS_AuthLinkSpeed
   result = "Link speed calculation"
Case RASCS_AuthAck
   result = "Authentication phase started"
Case RASCS_ReAuthenticate
   result = "Reauthentication started"
Case RASCS_Authenticated
   result = "Authentication complete"
Case RASCS_PrepareForCallback
   result = "Preparing for callback"
Case RASCS_WaitForModemReset
   result = "Waiting for modem to reset"
Case RASCS_WaitForCallback
   result = "Waiting for callback"
Case RASCS_Projected
   result = "Projection results available"
Case RASCS_StartAuthentication
   result = "Authentication starting"
Case RASCS_CallbackComplete
   result = "Callback complete"
Case RASCS_LogonNetwork
   result = "Logging on to network"
Case RASCS_SubEntryConnected
   result = "Sub entry connected"
Case RASCS_SubEntryDisconnected
   result = "Sub entry disconnected"
Case RASCS_Interactive = RASCS_PAUSED
```

```
            result = "Connection paused"
         Case RASCS_RetryAuthentication
            result = "Retrying authentication"
         Case RASCS_CallbackSetByCaller
            result = "Callback state"
         Case RASCS_PasswordExpired
            result = "Password expired"
         Case RASCS_Connected = RASCS_DONE
            result = "Connected"
         Case RASCS_Disconnected
            result = "Disconnected"
      End Select
      Call frmRasEntries.UpdateDialStatus(RasConnState, result)
End Function
```

Ergebnisse

Oh, ich habe ganz vergessen, Ihnen zu sagen, dass in dem Puzzle noch keine Visual Basic-Deklarationen für die Aufzählung vorhanden sind, d.h., das Programm kann nicht einmal ausgeführt werden. Darin besteht Ihre erste Aufgabe. Wenn Sie über die Deklarationen verfügen, ist die zweite Aufgabe offensichtlich.

Puzzle 21

Welches ist das zugeordnete Laufwerk? Teil 1

Wenn Sie bereits einmal einen Windows-Computer in einem Netzwerk verwendet haben, wissen Sie, dass es möglich ist, Netzwerklaufwerke so zuzuordnen, dass diese als lokale Laufwerke auf Ihrem System angezeigt werden. Dies kann für einen Programmierer von Wichtigkeit sein. Beispielsweise, wenn Sie zur Speicherung von temporären Dateien kein verbundenes Netzlaufwerk verwenden möchten, oder wenn Sie den Netzwerkprovider für das zugeordnete Laufwerk ermitteln möchten. Mit Hilfe der Win32-API kann leicht herausgefunden werden, bei welchen lokalen Laufwerken es sich um zugeordnete Netzwerklaufwerke handelt. Oder etwa nicht?

In der Win32-API-Dokumentation wird die `WNetGetConnection`-Funktion folgendermaßen definiert:

```
DWORD WNetGetConnection(
    LPCTSTR lpLocalName,          // Zeiger auf lokalen Namen
    LPTSTR lpRemoteName,          // Zeiger auf Puffer für Remotenamen
    LPDWORD lpnLength             // Zeiger auf Puffergröße  (in Zeichen)
   );
```

Zusätzlich wird die folgende Beschreibung für den `IpnLength`-Parameter gegeben:

> *lpnLength*
>
> *Zeigt auf eine Variable, mit der die Größe des Puffers (in Zeichen) angegeben wird, auf den über den `IpRemoteName`-Parameter verwiesen wird. Wenn die Funktion fehlschlägt, weil der Puffer zu klein ist, gibt dieser Parameter die erforderliche Puffergröße zurück.*

Bei dem Rückgabewert der Funktion handelt es sich um die Konstante `ERROR_MORE_DATA`, wenn der Puffer nicht groß genug ist, um die Daten zu speichern.

Die API-Deklaration ist recht verständlich:

```
Private Declare Function WNetGetConnection Lib "mpr.dll" Alias _
"WNetGetConnectionA" (ByVal lpszLocalName As String, ByVal _
lpszRemoteName As String, lpnLength As Long) As Long
```

Die DLL **mpr.dll** enthält netzwerkunabhängige API-Funktionen, also die Netzwerkfunktionen, die problemlos ausgeführt werden können, egal, welches Netzwerk bzw. welchen Netzwerkprovider Sie verwenden. Der Parameter `IpszLocalName` wird `ByVal As String` deklariert. Dies gilt auch für `IpszRemoteName`, der darüber hinaus mit der richtigen Länge initialisiert werden muss, da diese Zeichenfolge den Rückgabewert enthalten wird.

Der `lpnLength`-Parameter ist ein Zeiger auf eine `Long`-Variable, die die Pufferlänge enthält. Da die Funktion einen Zeiger erfordert, wird der Parameter als Verweis übergeben.

Im Beispielprogramm wird die Funktion zunächst mit einer leeren Zeichenfolge aufgerufen, die Pufferlänge wird auf 0 gesetzt. Auf diese Weise kann die richtige Pufferlänge für den zweiten Aufruf von Windows eingestellt werden.

Das Projekt **MapInfo1.vbp** enthält für die Datei **MapInfo1.frm** den folgenden Code:

```
' MapInfo1.vbp
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten

Option Explicit

Private Declare Function WNetGetConnection Lib "mpr.dll" Alias _
"WNetGetConnectionA" (ByVal lpszLocalName As String, ByVal _
lpszRemoteName As String, cbRemoteName As Long) As Long
Private Const ERROR_MORE_DATA = 234 '  dderror

   ' Abrufen der Quelle eines zugeordneten Laufwerks
Private Function GetMappedInfo(ByVal Drive As String) As String
   Dim Buffer As String
   Dim BufferLength As Long
   Dim res As Long
   Drive = Left$(Drive, 2)

   res = WNetGetConnection(Drive, Buffer, BufferLength)
   If res = ERROR_MORE_DATA Then
      Buffer = String$(BufferLength, 0)
      res = WNetGetConnection(Drive, Buffer, BufferLength)
   End If

   If res <> 0 Then
      lblSource.Caption = GetErrorString(Err.LastDllError)
   Else
   ' Nullwert am Ende abschneiden
      lblSource.Caption = Left$(Buffer, BufferLength - 1)
      GetMappedInfo = lblSource.Caption
   End If

End Function
```

```
Private Sub Drive1_Change()
   Call GetMappedInfo(Drive1.Drive)
End Sub
```

Die hier aufgeführte Funktion `GetErrorString` verwendet die Funktion `FormatMessage`, um eine Textbeschreibung für den im Falle eines Fehlers empfangenen `Err.LastDIIError`-Wert zu erhalten.

Ergebnisse

Um dieses Beispiel testen zu können, müssen Sie über ein System verfügen, das an ein Netzwerk angeschlossen ist und für das ein Remotenetzlaufwerk einem lokalen Laufwerk zugeordnet wurde.

Wenn mit Hilfe des Laufwerkfeldes ein nicht zugeordnetes Laufwerk ausgewählt wird, wird ordnungsgemäß die Fehlermeldung »Diese Netzwerkverbindung ist nicht vorhanden« angezeigt.

Unter Windows NT wird bei der Auswahl eines nicht zugeordneten Laufwerks die Fehlermeldung »Es sind mehr Daten verfügbar« angezeigt[3].

Moment mal – durch den Code wird bereits beim ersten Aufruf von `WNetGetConnection` die richtige Pufferlänge abgerufen. Es scheint, dass Windows einen Pufferwert zurückgibt, der nicht lang genug ist, um den `IpRemoteName`-Parameter zu speichern. Dies widerspricht der oben zitierten Dokumentation für den `IpnLength`-Parameter.

Handelt es sich um einen Windows-Bug? Wenn nicht, um was dann? Wenn doch, wie umgehen Sie den Bug?

3. Unter Windows 95/98 funktioniert dieses Puzzle. Wenn Sie nicht Windows NT verwenden, empfehle ich Ihnen, das Puzzle zu lesen und direkt zur Lösung zu wechseln. Wiederholen Sie hierbei den folgenden Satz immer und immer wieder: »Ich werde meine Software unter Windows NT testen, ich werde meine Software unter Windows NT testen, ...«

Puzzle 22

Welches ist das zugeordnete Laufwerk? Teil 2

Jetzt, da Sie wissen, wie Sie die Serverinformationen für ein zugeordnetes lokales Laufwerk ermitteln, stellt sich die Frage, wie weitergehende Informationen abgerufen werden können, wie z. B. der Name des Netzwerkproviders.

Die Funktion `WNetGetResourceInformation` kann dazu verwendet werden, Informationen zu einer Netzwerkressource abzurufen, indem die `NETRESOURCE`-Struktur für die Ressource geladen wird. Diese Struktur wird in der API-Dokumentation folgendermaßen definiert:

```
typedef struct _NETRESOURCE {  // nr
DWORD dwScope;
DWORD dwType;
DWORD dwDisplayType;
DWORD dwUsage;
LPTSTR lpLocalName;
LPTSTR lpRemoteName;
LPTSTR lpComment;
LPTSTR lpProvider;
}  NETRESOURCE;
```

Die `WNetGetResourceInformation`-Funktion wird wie folgt beschrieben:

```
DWORD WNetGetResourceInformation (
  LPNETRESOURCE lpNetResource,     // Gibt die Netzwerkressource an,
                                   // zu der Informationen benötigt werden
  LPVOID lpBuffer,                 // Gibt den Puffer an, der
                                   // die Informationen enthalten soll
  LPDWORD lpcbBuffer,              // Gibt die Größe des Puffers an,
                                   // auf den durch IpBuffer verwiesen wird
  LPTSTR *lplpSystem               // Zeiger auf eine Zeichenfolge
                                   // im Ausgabepuffer
  );
```

Bei dem Parameter `IpNetResource` handelt es sich um einen Zeiger auf die `NETRESOURCE`-Struktur, mit der die Netzwerkressource definiert wird, bezüglich der Informationen abgerufen werden sollen. Das Feld `IpRemoteName` in der Struktur muss auf den Namen der Netzwerkressource eingestellt werden (nicht auf den Laufwerksbuchstaben für die lokale Netzwerkzuordnung). Sie können das Feld `dwType` auf die Konstante `RESOURCETYPE_DISK` setzen, um die Leistung zu verbessern. (Die Funktion sucht in diesem Fall nicht nach anderen Ressourcentypen wie beispiels-

weise nach Druckern.) Das Feld `lpProvider` kann auch gesetzt werden, wenn Sie den Netzwerkprovider kennen, dem diese Ressource gehört. Die weiteren Felder in der Struktur werden ignoriert.

Der `lpBuffer`-Parameter enthält einen Zeiger auf einen Speicherpuffer, der zusammen mit einer `NETRESOURCE`-Struktur, gefolgt von benötigten Zeichenfolgeninformationen geladen wird. Der Zeichenfolgenzeiger in der zurückgegebenen `NETRESOURCE`-Struktur zeigt auf die Zeichenfolgen innerhalb dieses Puffers. Der Parameter `lplpSystem` wird ebenfalls gesetzt, um auf eine Zeichenfolge innerhalb dieses Puffers zu verweisen.

Die Größe des Puffers kann mit Hilfe der gleichen Methode ermittelt werden, die bereits in Teil 1 dieses Puzzle verwendet wurde – die erforderliche Puffergröße wird in den Parameter `lpcbBuffer` geladen.

Verwirrt? Hier noch ein abschließender Gedankengang: In der Datei **win32api.txt** werden die Felder `lpLocalName`, `lpRemoteName`, `lpComment` und `lpProvider` der `NETRESOURCE`-Struktur als Zeichenfolgen definiert.

Dies würde zu folgender Deklaration und folgendem Code führen, wie auch gezeigt im Projekt **MapInfo2.vbp** im Unterverzeichnis `Puzzle` auf der Begleit-CD-ROM zu diesem Buch:

```
Private Declare Function WNetGetResourceInformation Lib "mpr.dll" _
Alias "WNetGetResourceInformationA" ( lpNetResource As NETRESOURCE, _
lpBuffer As Byte, lpcbBuffer As Long, lplpSystem As Long) As Long
Private Type NETRESOURCE
        dwScope As Long
        dwType As Long
        dwDisplayType As Long
        dwUsage As Long
        lpLocalName As String
        lpRemoteName As String
        lpComment As String
        lpProvider As String
End Type

' Netzwerkproviderinformationen abrufen
Private Sub GetResourceInfo(resourcename As String)
   Dim nr As NETRESOURCE
   Dim lpSystem As Long
   Dim OutputBufferSize As Long
   Dim OutputBuffer() As Byte
```

```
    Dim res As Long

    lstResource.Clear
    If resourcename = "" Then Exit Sub

    nr.dwType = RESOURCETYPE_DISK
    nr.lpRemoteName = resourcename

    ' Puffer vorbereiten
    OutputBufferSize = 1024
    Do
       ReDim OutputBuffer(OutputBufferSize)
       res = WNetGetResourceInformation(nr, OutputBuffer(0), _
             OutputBufferSize, lpSystem)
    Loop While res = ERROR_MORE_DATA
    If res <> 0 Then
       MsgBox "No Resource Information Available"
       Exit Sub
    End If

    ' Ergebnis-NETRESOURCE abrufen
    Call RtlMoveMemory(nr, OutputBuffer(0), Len(nr))

    lstResource.AddItem "Remote Name: " & nr.lpRemoteName
    lstResource.AddItem "Provider: " & nr.lpProvider

End Sub
```

Ergebnisse

Das Programm funktioniert einwandfrei – bis Sie im Feld für die Laufwerksauswahl ein zugeordnetes Laufwerk angeben. In diesem Fall wird die API-Funktion `GetResourceFunction` aufgerufen und führt zu einem sofortigen Speicherausnahmefehler.

Ihre Aufgabe: Bearbeiten Sie den Code so, dass Sie die Providerinformationen zur Netzwerkressource abrufen können.

Puzzle 23

Fragen gibt es immer wieder

Woher stammen die verwendeten Puzzle?

Eines meiner Ziele bei er Arbeit an diesem Buch war es, realitätsnahe Puzzle zu entwickeln. Was ich hiermit meine? Ich habe in verschiedenen Büchern Beispielcode gefunden, der mir irgendwie künstlich erschien. Beispielsweise wird die objektorientierte Programmierung anhand von Tieren demonstriert – Sie verfügen über eine Klasse mit dem Namen `Tiere`, die Unterklassen wie `Hund` oder `Katze` enthält. Dies kann hilfreich sein, wenn Sie einen Zoo leiten, aber sprechen wir es doch aus: Die meisten Programmierer entwickeln keine Software für Zoos. Vielleicht arbeiten sie in Büros, die in gewisser Weise einem Zoo ähneln – aber ich schweife ab. Ich möchte damit sagen, dass die Beispiele in solchen Büchern häufig etwas realitätsfern sind.

Ich bin froh, dass dieses Buch keine derartigen Beispiele verwendet. Warum ich mir dessen so sicher bin? Statt einer Erklärung möchte ich Ihnen ein Beispiel geben. Betrachten Sie dieses von mir aus als Puzzle zur Erstellung dieser Art von Puzzle.

Zunächst zur Inspiration für die Puzzle.

Im Folgenden eine Anfrage, die ich vor etwa zwei Tagen erhielt:

> *Dan, Sie waren mir häufig eine große Hilfe – ich hoffe, Sie können mir auch bei dieser meiner Meinung nach einfachen Frage helfen.*
>
> *Ich versuche, die Struktur `USER_INFO_2` mit der VBA-Funktion `NetAddUser` einzusetzen. Teil dieser Struktur ist: `PBYTE usri2_logon_hours`; zeigt auf eine 21-Byte-Zeichenfolge (168 Bits), die angibt ...*
>
> *Wie schließe ich dies in meine Typdeklaration ein? Darüber hinaus liegt in der C-Headerdatei eine solche Konstante vor:*
>
> ```
> #define USER_MAXSTORAGE_UNLIMITED ((unsigned long) -1L)
> ```
>
> *Ist dies kein Widerspruch?*
>
> *Sollte ich einfach eine `Long`-Konstante mit dem Wert 1 deklarieren? Vielen Dank im Voraus!*

Ich möchte nicht den Eindruck erwecken, dass ich auf alle Anfragen antworte, die ich täglich erhalte. Aufgrund notorischen Zeitmangels kann ich nur einige der Anfragen beantworten. In diesem Fall erschien mir die Frage jedoch fabelhaft für das vorliegende Buch geeignet zu sein. Ich schrieb eine kurze Antwort und bat darum, den exakten Wortlaut der Frage im Buch verwenden zu dürfen (ohne den Namen des Verfassers zu nennen). Ich erhielt freundlicherweise die Erlaubnis.

Die kurze Antwort bestand in Vorschlägen für das weitere Vorgehen. Ich fügte kein Beispiel zur Verwendung der `NetAddUser`-Funktion bei. Sie können, wenn Sie möchten, selbst versuchen, eine Antwort auf die Frage zu finden. Oder Sie lesen weiter und folgen meinen Erläuterungen zur Erstellung eines Beispielpuzzle. Hierzu werde ich einen Schreibstil verwenden, den ich gelegentlich beim Verfassen von Artikeln verwende und den ich bisher bei keinem anderen Autor entdecken konnte.

Dieses Kapitel ist in Echtzeit geschrieben.

Was ich damit meine? Nun, ich meine damit, dass ich dieses Kapitel schreibe, während ich das Problem zu lösen versuche. Sie schauen mir also sozusagen über die Schulter, während ich versuche, diese Funktion zum Laufen zu bringen. Ich muss während des Schreibens ehrlich sein, also werden Sie auch die Fehler miterleben, die mir bei diesem Prozess unterlaufen. Das Kapitel ist größtenteils in der Vergangenheit geschrieben, um eine konsistente Erzählweise beizubehalten. Seien Sie dennoch versichert, dass ich Text und Code zeitgleich geschrieben habe.

Die Funktion

Mein erster Schritt bestand darin, in der Win32 SDK nach der Funktion `NetAddUser` zu suchen. Hierbei stieß ich auf das erste Problem. Eine derartige Funktion gibt es nicht. Also wandte ich mich der alphabetischen Liste der Funktionen zu und suchte nach allen Funktionen, die mit dem Wort »Net« beginnen. Ich fand schnell heraus, dass es sich bei der gemeinten um die Funktion `NetUserAdd` handeln musste.

In der Win32 SDK wird die Beschreibung dieser Funktion mit einer Warnung eingeleitet, nach der die Funktion nur in Programmen funktioniert, die als Mitglied der Administratorengruppe ausgeführt werden. Nicht allzu überraschend, da wir von einer Funktion sprechen, mit der dem System ein Benutzer hinzugefügt wird. Es handelt sich außerdem um eine Funktion, die nur unter Windows NT verwendet werden kann. Dies ist ebenfalls keine Überraschung, da Windows 95 nicht wirklich dazu tauglich ist, Benutzer zu verwalten. Sie müssen sich unter Verwendung eines Administratorkontos an eine Windows NT-Arbeitsstation oder an einen Windows NT-Server anmelden, um das Programm testen zu können.

In C wird die Funktion folgendermaßen deklariert:

```
NET_API_STATUS NetUserAdd(
    LPWSTR servername,
    DWORD level,
    LPBYTE buf,
    LPDWORD parm_err
   );
```

Was schließen Sie aus dieser Deklaration?

- Der Parameter für den Servernamen lautet `LPWSTR` (`LP` = `Long`-Zeiger, `W` = Wide, `STR` = Zeichenfolge). Es handelt sich um einen Zeiger nach einer `wide`-Zeichenfolge (Unicode). Da über die Funktion explizit eine `wide`-Zeichenfolge angegeben wird, können Sie annehmen, dass für die Funktion nicht zwei separate Unicode- und ANSI-Einsprungpunkte vorhanden sind, sondern dass nur ein einzelner Einsprungpunkt vorliegt. Dieser Parameter kann auch NULL sein, um den Benutzer dem lokalen System hinzuzufügen.
- Der `level`-Parameter wird auf einen Wert von 1, 2 oder 3 gesetzt. Falls der Wert 1 beträgt, zeigt der `buf`-Parameter auf eine `USER_LEVEL_1`-Struktur, bei einem Wert von 2 auf eine `USER_LEVEL_2`-Struktur; falls der Wert auf 3 gesetzt wird, verweist der `buf`-Parameter auf eine `USER_LEVEL_3`-Struktur. Diese Strukturen enthalten Informationen über den hinzuzufügenden Benutzer.
- Der `buf`-Parameter ist ein Zeiger auf die jeweils geeignete Struktur. Warum wird kein spezieller Typ wie beispielsweise `LPUSER_LEVEL_1` verwendet? Da der Parameter auf eine beliebige der verfügbaren Strukturentypen verweisen muss, je nachdem welcher `level`-Parameter verwendet wurde. `LPBYTE` bezeichnet einen Zeiger auf ein Byte und wird häufig eingesetzt, um einen generischen Zeigertyp zu verwenden, wenn der Zeiger auf mehr als einen Typ verweisen muss.
- Der `parm_err`-Parameter ist ein Zeiger auf eine `Long`-Variable, die mit dem Index des Parameters in der Struktur geladen wird, durch die ein Fehler verursacht wurde, und enthält daher die Position des betreffenden Parameters in der Struktur.
- Durch die Funktion wird ein `Long`-Wert zurückgegeben. Gemäß der Dokumentation lautet der Wert `NERR_SUCCESS` wenn die Funktion erfolgreich war. Eine Suche in den Headerdateien zeigt, dass `NERR_SUCCESS` gleich Null ist.

Ich war fast soweit, die Visual Basic-Deklaration zu schreiben. Eine Frage: Wie lautet der Name der DLL? Ich hätte auch im Programm DumpInfo (beschrieben in Tutorium 9, »In einer DLL-Datei: Das Programm **DumpInfo**«, in Teil III dieses Buches) nach der DLL suchen können, die diese Funktion unterstützt, da ich jedoch

bereits die Win32 SDK-Onlinehilfe geöffnet hatte, klickte ich einfach auf die Schaltfläche `QuickInfo`, um die zusammenfassenden Informationen bezüglich dieser Funktion anzuzeigen, ▲Abbildung P23-1.

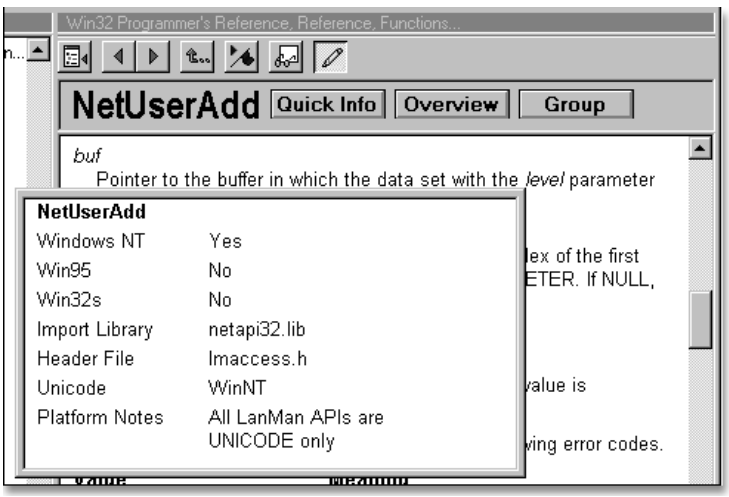


Abbildung P23-1 Das Fenster QuickInfo der Win32 SDK

Die Schlüsselinformation war die Angabe der Importbibliothek, in diesem Fall die **netapi32.lib**. Da der Name der Importbibliothek nicht immer mit dem der DLL übereinstimmt, sah ich kurz im Verzeichnis **system32** auf meinem Windows NT-System nach. Dort fand ich eine DLL mit dem Namen **netapi32.dll**. Ich war mir immer noch nicht absolut sicher, ob es sich um die richtige DLL handelte, aber es war einen Versuch wert. Die Deklaration lautet folgendermaßen:

```
Private Declare Function NetUserAdd Lib "netapi32.dll" ( _
    ByVal servername As Long, ByVal level As Long, _
    ByVal buf As Long, parm_err As Long) As Long
```

Warum traf ich diese Auswahl?

Bei dem `servername`-Parameter für den Servernamen handelt es sich um einen `Long`-Wert. Wenn ich diesen als Zeichenfolge definiert hätte, würde in Visual Basic eine Zeichenfolgenkonvertierung in ANSI erfolgen, wir benötigen jedoch eine Unicode-Zeichenfolge. Durch Angabe des Wertes `ByVal As Long` kann ich eine explizite Adresse an eine Unicode-Zeichenfolge übergeben. Der `level`-Parameter war offensichtlich ein `Long`-Wert. Für den `buf`-Parameter kamen mehrere Möglichkeiten in Frage: Ich könnte diesen als einen der Strukturtypen, als `As Any` oder wieder als `Long` deklarieren und die Adresse der Struktur übergeben. Ich wählte die letzte Möglichkeit. Der `parm_err`-Parameter ist ebenfalls ein `Long`-Wert, dieser

wird jedoch als Verweis übergeben, da es sich bei dem Parameter um einen LPDWORD handelt, also um einen Zeiger auf eine 32-Bit-Variable.

Die Struktur

Da sich die ursprüngliche Frage nur auf die USER_LEVEL_2-Struktur bezog, berücksichtigte ich auch nur diese. Die C-Deklaration für die Struktur lautet folgendermaßen:

```
typedef struct _USER_INFO_2 {
    LPWSTR    usri2_name;
    LPWSTR    usri2_password;
    DWORD     usri2_password_age;
    DWORD     usri2_priv;
    LPWSTR    usri2_home_dir;
    LPWSTR    usri2_comment;
    DWORD     usri2_flags;
    LPWSTR    usri2_script_path;
    DWORD     usri2_auth_flags;
    LPWSTR    usri2_full_name;
    LPWSTR    usri2_usr_comment;
    LPWSTR    usri2_parms;
    LPWSTR    usri2_workstations;
    DWORD     usri2_last_logon;
    DWORD     usri2_last_logoff;
    DWORD     usri2_acct_expires;
    DWORD     usri2_max_storage;
    DWORD     usri2_units_per_week;
    PBYTE     usri2_logon_hours;
    DWORD     usri2_bad_pw_count;
    DWORD     usri2_num_logons;
    LPWSTR    usri2_logon_server;
    DWORD     usri2_country_code;
    DWORD     usri2_code_page;
}
```

Das nenne ich eine Struktur!

So groß die Struktur auch ist, sie enthält nur drei Arten von Feldern, Zeiger auf Unicode-Zeichenfolgen, Long-Werte und einen einzigen PBYTE-Parameter. Zu letzterem kommen wir gleich.

Ich werden nicht einmal versuchen, alle diese Felder zu beschreiben – die jeweiligen Informationen müssen Sie selbst in der Dokumentation nachschlagen. Das Wichtigste hierbei ist, dass Sie in den meisten Fällen einfach NULL oder 0-Werte übergeben können, damit die Standardwerte verwendet werden. Das bedeutet Folgendes: Wenn ich alle Felder als `Long`-Werte definieren könnte, besäße ich die Möglichkeit, für die Parameter den Wert 0 beizubehalten, um die Standardwerte zu verwenden, und ich könnte die Felder explizit auf eine benötigte Adresse setzen. Dies erschien mir so verlockend, dass ich die folgende VB-Deklaration für die Struktur definierte. Hierbei fügte ich Kommentare ein, um erkennen zu können, bei welchen Feldern es sich um Zeichenfolgen handeln musste:

```
Private Type USER_INFO_2
    usri2_name As Long ' Unicode-Zeichenfolge
    usri2_password As Long ' Unicode-Zeichenfolge
    usri2_password_age As Long
    usri2_priv As Long
    usri2_home_dir As Long ' Unicode-Zeichenfolge
    usri2_comment As Long ' Unicode-Zeichenfolge
    usri2_flags As Long
    usri2_script_path As Long ' Unicode-Zeichenfolge
    usri2_auth_flags As Long
    usri2_full_name As Long ' Unicode-Zeichenfolge
    usri2_usr_comment As Long ' Unicode-Zeichenfolge
    usri2_parms As Long ' Unicode-Zeichenfolge
    usri2_workstations As Long ' Unicode-Zeichenfolge
    usri2_last_logon As Long
    usri2_last_logoff As Long
    usri2_acct_expires As Long
    usri2_max_storage As Long
    usri2_units_per_week As Long
    usri2_logon_hours As Long ' Bytezeiger auf einen 168-Bit-Zahlenwert
    usri2_bad_pw_count As Long
    usri2_num_logons As Long
    usri2_logon_server As Long ' Unicode-Zeichenfolge
    usri2_country_code As Long
    usri2_code_page As Long
End Type
```

Bringen Sie's zum Laufen

Es wurde Zeit, ein wenig Code zu schreiben. Ich wollte herausfinden, ob ich auf dem richtigen Weg war oder ob es versteckte Probleme gab, die ich bisher noch nicht gefunden hatte. Ich begann damit, in der Win32 SDK-Dokumentation nachzusehen, welche der Felder unbedingt erforderlich seien.

Sowohl `usri2_Name` als auch `usri2_Password` waren offensichtlich notwendig – machte es doch wenig Sinn, einen Benutzer ohne Namen oder Kennwort zu erstellen.

Ich benötigte außerdem eine Konstante für das Feld `usri2_priv`. Ich fand die Konstanten in der Headerdatei **lmaccess.h**:

```
#define USER_PRIV_GUEST     0
#define USER_PRIV_USER      1
#define USER_PRIV_ADMIN     2
```

Sie wundern sich vielleicht, wie ich eine spezielle Konstante finde, die in einer beliebigen der vielen C-Headerdateien versteckt sein kann. Die Wahrheit ist, ich mogle. Ich verwende den Texteditor Codewright (den ich Ihnen übrigenstens wärmstens empfehlen kann). Dieser Editor unterstützt eine Funktion mit dem Namen `File Grep`, eine dateiübergreifende Textsuche. Ich glaube, Visual Studio verfügt über eine ähnliche Funktion, jedenfalls bei Verwendung der Visual C++-IDE. Sie können sich ein solches Texttool mit Hilfe von Visual Basic auch selbst schreiben. Verwenden Sie die VB-Verzeichnisfunktionen um eine Liste von Dateien zu erstellen, und öffnen Sie diese einzeln im Textmodus. Lesen Sie jede Zeile, und verwenden Sie die `instr`-Funktion dazu, nach dem gewünschten Text zu suchen. Diese Methode ist vielleicht nicht die effizienteste, aber einfach zu programmieren. Natürlich können Sie wie sonst auch die Suchfunktionen des Windows Explorer verwenden. Aber ich schweife schon wieder ab.

Ich fand schnell die folgenden Konstanten heraus:

```
Private Const USER_PRIV_GUEST = 0
Private Const USER_PRIV_USER = 1
Private Const USER_PRIV_ADMIN = 2
```

Darüber hinaus benötigte ich auch die Konstante `UF_NORMAL_ACCOUNT` für den Parameter `usri2_flags`:

```
#define UF_NORMAL_ACCOUNT                0x0200
```

Daraus ergab sich Folgendes:

```
Private Const UF_NORMAL_ACCOUNT = &H200
```

Beachten Sie, dass der Wert hexadezimal angegeben wird.

Ein kurzer Blick in die Dokumentation brachte mich dazu, alle weiteren Parameter auf 0 zu setzen. Bei meinem ersten Test verwendete ich den folgenden Code:

```
Private Sub Command1_Click()
   Dim ul2 As USER_INFO_2
   Dim res As Long
   Dim parm_err As Long

   ul2.usri2_name = StrPtr("puzzle" & chr$(0))
   ul2.usri2_password = StrPtr("pass" & chr$(0))
   ul2.usri2_priv = USER_PRIV_USER

   res = NetUserAdd(0, 2, VarPtr(ul2), parm_err)
   Debug.Print "Result: " & res
   If res <> 0 Then Debug.Print "Error at index: " & parm_err
End Sub
```

Welchen Trick hatte ich für die Parameter `usri2_name` und `usri2_password` angewendet? In der Struktur definierte ich diese als `Long`-Werte, die einen Zeiger auf eine auf NULL endende Unicode-Zeichenfolge erfordern. Der `StrPtr`-Operator gibt einen Zeiger auf die internen Zeichenfolgendaten einer Visual Basic-Zeichenfolge zurück, und wir wissen, dass Visual Basic Daten intern im Unicode-Fomat speichert. Dies ist besonders sicher, da diese Zeichenfolgenparameter schreibgeschützt sind. Über die API-Funktion `NetUserAdd` werden die Daten in diesen Zeichenfolgen also niemals geändert.

Jetzt war es endlich Zeit herauszufinden, was bei der Ausführung des Programms passieren würde.

Das über die `Debug.Print`-Anweisung angezeigte Ergebnis lautete Fehler 87, wobei der Fehler bei Index 8 aufgetreten war.

Ich war neugierig, was Fehler 87 bedeutete. Da es sich um eine API-Funktion handelte, konnte ich zur Anzeige einer textbasierten Beschreibung des Fehlers die `FormatMessage`-Funktion verwenden (wie Sie bereits in der mehrmals verwendeten `GetErrorString`-Funktion sehen konnten). Aber in diesem Fall war ich faul, also suchte ich in der Datei **api32.txt** (auf der Begleit-CD-ROM) einfach nach dem Begriff »`ERROR_`«. Nachdem ich die erste Konstante in der Fehlerliste gefunden hatte, `ERROR_SUCCESS`, suchte ich weiter nach Nummer 87 und fand die Konstante `ERROR_INVALID_PARAMETER`. Ich hätte wissen sollen, dass es sich um diesen Fehler handelte, denn schließlich hatte es in der SDK-Dokumentation geheißen, dass der `parm_err`-Parameter nur gesetzt wird, wenn ein Parameter ungültig ist. Welches

Feld trägt die Nummer 8? Es handelt sich entweder um `usri2_script_path` oder um `usri2_auth_flags`, je nachdem, ob der erste Parameter als 0 oder 1 betrachtet wird. Nichts von beidem ergab einen Sinn. Der `usri2_script_path`-Parameter wird in der Dokumentation eindeutig als Parameter beschrieben, bei dem NULL-Zeichenfolgen verwendet werden können. Über den Parameter `usri2_auth_flags` wird gesagt, dass er für diese Funktion ignoriert wird (die Struktur `USER_INFO_2` wird auch von anderen Funktionen verwendet).

Hoffnungslos verstrickt?

An dieser Stelle hatte ich einen plötzlichen Tiefpunkt. Das ist der Ärger dabei, wenn man Artikel in Echtzeit verfasst. Was tun Sie, wenn man Ihnen eine Deadline gesetzt hat, Sie bereits Stunden investiert, 2000 Wörter für ein Kapitel geschrieben und keine Ahnung haben, ob Sie die Sache hinbekommen? Nach einer kurzen Panikattacke und anschließender Pause zwecks Schokoladenverzehr nahm ich die Sache nochmals genauer unter die Lupe.

In der Dokumentation wird eindeutig beschrieben, dass die Parameter `usri2_script_path` und `usri2_auth_flags` beide NULL sein können. Dennoch weist die Fehlermeldung eindeutig darauf hin, dass das Problem bei einem dieser Parameter liegt.

Wenn ich Zweifel habe, schaue ich mir meistens ein zweites Mal die Dokumentation an, besonders die Dokumentation zu den Parametern, die ich bereits gesetzt habe. Ich bemerkte, dass im Zusammenhang mit dem `usri2_flags`-Parameter die folgende Konstante dokumentiert wurde:

> *`UF_SCRIPT` Das ausgeführte Anmeldeskript. Dieser Wert muss für LAN Manager 2.0 oder Windows NT gesetzt werden.*

Ich hatte bereits vorher festgestellt, dass die Konstante `UF_NORMALACCOUNT` erforderlich war, die `UF_SCRIPT`-Konstante hatte ich jedoch vergessen. Windows stellte also fest, dass die Konstante nicht vorhanden war und entschied fälschlicherweise, dass der Fehler im Parameter `usri2_script_path` begründet lag. Ich weiß nicht, ob es sich hierbei um einen Bug oder um ein fehlgeleitetes Ergebnis handelt, aber eines wird so bewiesen: Wenn Sie mit der Win32-API arbeiten, dann setzen Sie niemals voraus, dass etwas richtig ist, nur weil es in der Dokumentation erwähnt wird. Ich weiß, dass dies ein schrecklicher Gedanke ist, aber lassen Sie sich von dieser Tatsache nicht zu sehr deprimieren. In den meisten Fällen stimmt die Dokumentation und Windows ist, gemessen an seiner Komplexität, ein erstaunlich fehlerfreies System. Die meisten Windows-Systeme stürzen nur ein- bis zweimal pro Tag ab (soviel zur Depressivität).

Bei der Behandlung dieses Problems fiel mir auf, dass ich die Einstellung des `usri2_flags`-Parameters vollständig vergessen hatte. Darum fügte ich den folgenden Code hinzu:

```
Private Const UF_SCRIPT = &H1
ul2.usri2_flags = UF_NORMAL_ACCOUNT Or UF_SCRIPT
```

Der `OR`-Operator wird dazu verwendet, zwei Konstanten miteinander zu verbinden, indem Sie für deren Bitwerte das logische `Oder` verwenden. Dies wird im Einzelnen auch in Tutorium 3, »Ein Boolescher Wert und seine Bitfelder werden bald auseinandergehen«, besprochen.

Als ich nun das Programm ausführte, wurde der Ergebniswert 0 zurückgegeben.

Aber funktionierte das Programm tatsächlich?

Ja und Nein. Ich öffnete den NT-Benutzer-Manager und sah, dass ich einen Benutzer mit dem Namen »Pass« erstellt hatte, jedoch keinen namens »Puzzle«. Wie konnte das passieren?

Ich hatte gemogelt und war dabei erwischt worden.

Sie haben gesehen, dass ich unter Verwendung der `StrPtr`-Funktion mit einer Konstanten Unicode-Zeichenfolgen an die Struktur übergeben hatte:

```
ul2.usri2_name = StrPtr("puzzle" & chr$(0))
```

Was passiert, wenn Sie den `StrPtr`-Operator mit einer Konstanten verwenden? Es wird eine temporäre Variable mit den Daten erstellt, und es wird ein Zeiger auf die Daten übergeben. Sobald aber der Zeiger zurückgegeben wird, wird die temporäre Variable gelöscht. Beim nächsten Aufruf von `StrPtr` wird der Puffer mit einer anderen Zeichenfolge geladen. Ich war froh, dass ich keinen Systemabsturz verursacht hatte. Die Struktur enthält einen Zeiger, und es ist einfach nicht möglich, Zeiger auf temporäre Daten zu verwenden. Ich versuchte, dieses Problem ohne die Erstellung separater Zeichenfolgen zu umgehen und lief in die Falle. Dies ist der Punkt, an dem Erfahrungswerte ins Spiel kommen. Ich hatte vorher noch nie versucht, den `StrPtr`-Operator mit einer Konstanten zu verwenden, weshalb ich sehen wollte, ob dies ein möglicher Lösungsansatz wäre. Ich war überrascht, denn es funktionierte tatsächlich. Glücklicherweise hatte ich bereits einige Erfahrung mit Zeigern auf temporäre Puffer und sich daraus ergebenden ähnlichen Problemen, daher wusste ich sofort, als ich das Benutzerkonto mit dem falschen Namen sah, dass dieser Fehler in irgendeiner Form auf die Verwendung des `StrPtr`-Operators mit Konstanten zurückzuführen sein musste. In der endgültigen Codeversion verwendete ich folglich ich Zeichenfolgenvariablen, deren Addressen für die gesamte Lebensdauer der Funktion gültig bleiben (solange Sie nicht versuchen, die Daten in der Zeichenfolge zu ändern). Hier also die endgültige Fassung der Funktion:

```
Private Sub Command1_Click()
   Dim ul2 As USER_INFO_2
   Dim res As Long
   Dim parm_err As Long
   Dim username As String
   Dim userpass As String
   username = "Puzzle" & Chr$(0)
   userpass = "pass" & Chr$(0)

   ul2.usri2_name = StrPtr(username)
   ul2.usri2_password = StrPtr(userpass)
   ul2.usri2_priv = USER_PRIV_USER
   ul2.usri2_flags = UF_NORMAL_ACCOUNT Or UF_SCRIPT

   res = NetUserAdd(0, 2, VarPtr(ul2), parm_err)
   Debug.Print "Result: " & res
   If res <> 0 Then Debug.Print "Error at index: " & parm_err
End Sub
```

Die Funktion arbeitet sogar dann, wenn Sie das zusätzliche NULL-Zeichen am Ende der Zeichenfolgen nicht anhängen. Ich habe noch nie einen Fall erlebt, in dem kein NULL-Zeichen am Ende einer Zeichenfolge verwendet wurde. Warum fügte ich dann explizit einen NULL-Wert hinzu? Weil weder die Windows- noch die VB-Dokumentation eine Garantie dafür enthalten, dass eine VB-Zeichenfolge nach der Zeichenfolge immer über einen NULL-Wert verfügt. Ohne diese Garantie könnte Microsoft die Arbeitsweise von Visual Basic beliebig ändern und so selbst bereits kompilierte Programme zerstören. Durch das explizite Hinzufügen eines NULL-Zeichens stellen Sie sicher, dass die Zeichenfolge immer auf NULL endet.

Der NT-Benutzer-Manager zeigte nun einen Benutzer mit dem Namen »Puzzle« an. Die Funktion konnte problemlos ausgeführt werden!

Das Puzzle – endlich

Ich hoffe, Ihnen hat das Kapitel in Echtzeit gefallen. Jetzt, da Sie über ein funktionierendes Testprogramm verfügen, lasse ich Sie mit der ursprünglich gestellten Frage allein:

Ich versuche, die Struktur `USER_INFO_2` mit der VBA-Funktion `NetAddUser` einzusetzen. Teil dieser Struktur ist: `PBYTE usri2_logon_hours`; zeigt auf eine 21-Byte-Zeichenfolge (168 Bits), die angibt ...

Wie schließe ich dies in meine Typdeklaration ein? Darüber hinaus liegt in der C-Headerdatei eine solche Konstante vor:

```
#define USER_MAXSTORAGE_UNLIMITED ((unsigned long) -1L)
```

Ist dies kein Widerspruch?

Sollte ich einfach eine `Long`*-Konstante mit dem Wert 1 deklarieren? Vielen Dank im Voraus!*

Können Sie die Frage beantworten?

Puzzle 24

Rufen Sie diese Zeichenfolge zurück!

Von einem Leser:

In Ihrem Buch »VB5 Programmer's Guide to Win32 API« wird die Funktion `EnumSystemLocales` aufgeführt, wenn ich diese jedoch in meiner Anwendung implementiere, kommt es immer zu einem Absturz. Ich verwende den folgenden Code:

Nachfolgend ein Standardmodul:

```
' Locales Puzzle
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten

Option Explicit

Public Declare Function EnumSystemLocales Lib "kernel32" Alias _
"EnumSystemLocalesA" (ByVal lpLocaleEnumProc As Long, ByVal _
dwFlags As Long) As Long

Public Const LCID_INSTALLED = &H1  ' installed Locale ids

Public Function EnumLocalesProc(ByVal Locale As String) As Boolean
    Debug.Print Locale
    EnumLocalesProc = True
End Function
```

Nachfolgend ein Formular mit folgendem Inhalt:

```
' Locale Example
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten

Option Explicit

Private Sub Command1_Click()
   Dim lRet As Long
   lRet = EnumSystemLocales(AddressOf EnumLocalesProc, LCID_INSTALLED)
End Sub
```

Ergebnisse

Mit einer Ländereinstellung werden spezielle Länder- und Sprachkombinationen definiert. Von einem System können gleichzeitig mehrere dieser Ländereinstellungen unterstützt werden. Diese Funktion ermöglicht Ihnen die Anzeige einer Liste sämtlicher Ländereinstellungen auf dem System. Beziehungsweise könnten Sie diese theoretisch anzeigen, denn das vorliegende Beispiel führt, wie der Leser beschreibt, unter Windows NT zu einem Absturz.[4]

4. Unter Windows 95/98 führt die Funktion nicht zu einem Absturz, jedenfalls nicht sofort. Das Ergebnis ist jedoch ebenfalls ungültig.

Abschnitt 4: OLE olé!

Da gibt es die Win32-API mit ihren Tausenden und Abertausenden von Funktionen, die unter praktisch jedem Aspekt für die Steuerung von Windows sorgen. Und dann gibt es da OLE.

OLE, kurz für Object Linking and Embedding, ist die ältere Bezeichnung für ActiveX, einem Modewort der Marketingwelt aus den Tagen, als Microsoft noch mehr Angst vor Netscape hatte als vor dem Justizministerium.[1] Heutzutage werden Sie eher den Begriffen COM oder COM+ begegnen, mit denen die Technologien beschrieben werden, die Teil von OLE sind oder über OLE implementiert werden.

OLE stellt ein großes Subsystem innerhalb von Windows dar und verfügt über eigene DLLs und API-Funktionen.

Es ist äußerst wichtig, den Unterschied zwischen API-Funktionen und den Objektmethoden und -eigenschaften zu verstehen, die Bestandteil von COM sind. Funktionen und COM-Methoden können beide mit Hilfe von DLLs implementiert werden, weitere Gemeinsamkeiten weisen sie jedoch nicht auf.

Bei Verwendung von COM muss über Ihr Programm zunächst ein Objekt erstellt werden. Das Objekt wird anschließend über die DLL (oder EXE) implementiert, wodurch ein Zeiger auf das Objekt, oder, genauer ausgedrückt, ein Zeiger auf die angeforderte Schnittstelle für ein Objekt bereitgestellt wird. Dieser Zeiger verweist tatsächlich auf ein Array mit Funktionszeigern für die Methoden und Eigenschaften der Objektschnittstelle, die Sie angefordert haben. In Visual Basic können Methoden und Eigenschaften für verschiedene Schnittstellentypen aufgerufen werden, aber nicht jede Schnittstelle ist mit Visual Basic kompatibel. Wenn Sie Methoden für eine Schnittstelle aufrufen möchten, die nicht VB-kompatibel ist, müssen Sie eine andere DLL verwenden, um entweder ein `Wrapper`-Objekt zu erstellen, eine benutzerdefinierte Typbibliothek für ein mit VB kompatibles Objekt zu definieren, oder Sie verwenden ein Tool wie Desawares SpyWorks, mit dem ein generischer Zugriff auf beliebige Schnittstellen möglich wird.

Sie können für den Zugriff auf Methoden oder Eigenschaften eines Objekts jedoch nicht die `Declare`-Anweisung verwenden.

Die `Declare`-Anweisung wird ausschließlich für den Zugriff auf Funktionen verwendet, die direkt durch eine DLL bereitgestellt werden. Die DLL enthält eine Liste der Funktionen, die ohne die Erstellung von Objekten aufgerufen werden

1. Ich möchte gar keine Mutmaßungen darüber anstellen, vor wem Microsoft überhaupt noch Angst haben sollte, wenn Sie dies lesen.

können. Hierbei wird ein Mechanismus verwendet, der nicht im Geringsten etwas mit COM zu tun hat. Diese Funktionen werden sozusagen aus der DLL »exportiert«.

Mit Hilfe von Visual Basic-DLLs können COM-Objekte erstellt und unterstützt werden, aber es ist nicht möglich, einen Funktionenexport durchzuführen – es sei denn, Sie verwenden ein Fremdanbietertool (beispielsweise SpyWorks von Desaware) um Visual Basic diese Fähigkeit zu verleihen.

Dieses Buch befasst sich weder mit der Erstellung von COM-Objekten noch mit dem Aufruf von Methoden oder Eigenschaften eines COM-Objekts. (In meinem Buch »Dan Appleman's Developing COM/ActiveX Components with Visual Basic 6.0« wird ausführlich auf dieses Thema eingegangen.) DLLs jedoch, über die COM-Objekte implementiert werden, verfügen sehr wohl über die Unterstützung des Betriebssystems zur Verwendung einer Vielzahl der OLE-Funktionen. Diese Unterstützung manifestiert sich in einer Vielzahl von Funktionen, die von den OLE-System-DLLs selbst exportiert werden – der OLE-API. Einige dieser Funktionen können sehr hilfreich sein, wenn sie direkt durch eine Anwendung aufgerufen werden. In diesem Buch wurde darauf geachtet, sich nicht nur auf eines der Windows-Subsysteme zu konzentrieren, und zwar hauptsächlich deswegen, weil die verwendeten Techniken in den meisten Fällen für alle Subsysteme gelten. Die OLE-API jedoch verdient besondere Aufmerksamkeit. Es ist die einzige API, in der umfangreicher Gebrauch von Objekten und einer Datenstruktur gemacht wird, die als GUID (Global Unique Identifier) bezeichnet wird. Der Umgang mit den Objekten und GUIDs in den API-Funktionen bringt eine Reihe von interessanten Herausforderungen mit sich.

Und um interessante Herausforderungen geht es schließlich in diesem Buch.

Puzzle 25

Universelle Bezeichner, Teil 1

Sie haben bestimmt schon einmal von COM gehört, dem Component Object Model, und das vielleicht im Zusammenhang mit den Begriffen ActiveX oder OLE. Bei COM handelt es sich um ein Objektmodell – eine Methode zur Erstellung und Verwendung bestimmter Objekttypen, die als »Windows-Objekte« bezeichnet werden. (Eine ausführliche Einführung in COM aus der Sichtweise von Visual Basic erhalten Sie in meinem Buch »Dan Appleman's Developing COM/ActiveX Components for Visual Basic 6.0: A Guide to the Perplexed«.)

Die Win32-API umfasst Hunderte von Funktionen, die mit COM zusammenhängen. Diese Funktionen sind Teil des OLE-Subsystems. (OLE steht für Object Linking and Embedding. Es handelt sich hierbei um eine Technologie, mit der COM-Objekte implementiert werden. ActiveX ist ein anderer Name für OLE.)

Jedes COM-Objekt wird durch einen universellen Bezeichner identifiziert, der GUID oder Globally Unique Identifier genannt wird. Dieser Bezeichner ist ein 128-Bit-Zahlenwert, über den das betreffende Objekt eindeutig bestimmt wird. Aufgrund der Art, mit der GUIDs erstellt werden, ist die Möglichkeit, dass ein anderes Objekt dieselbe GUID aufweist, verschwindend gering. Wenn Sie die OLE-Dokumentation lesen, werden Ihnen auch Begriffe wie UUID, CLSID und IID (Universally Unique Identifier, Class Identifier und Interface Identifier) begegnen. Die Unterschiede zwischen diesen Bezeichnern sind lediglich semantischer Natur. Mit anderen Worten: Der einzige Unterschied zwischen einem Klassenbezeichner und einem Schnittstellenbezeichner besteht darin, dass mit dem einen eine Klasse, mit dem anderen eine Schnittstelle bezeichnet wird. Es handelt sich in allen Fällen um einen 128-Bit-Zahlenwert, der aus der Perspektive des Programmierers identisch verwendet wird.

Die Puzzle 25-28 demonstrieren die Verwendung des OLE-Subsystems und der GUIDs. Bevor Sie jedoch mit einer GUID arbeiten können, müssen Sie wissen, wie diese in Visual Basic denn dargestellt wird. Weder Visual Basic noch C++ verfügen ja über einen 128-Bit-Datentyp.

Der folgende Abschnitt aus der Windows-Headerdatei **wtypes.h** enthält die C++-Deklarationen für GUIDs, CLSIDs und IIDs:

```
#ifndef GUID_DEFINED
#define GUID_DEFINED
typedef struct  _GUID
    {
    DWORD Data1;
```

```
    WORD Data2;
    WORD Data3;
    BYTE Data4[ 8 ];
    }    GUID;

#endif // !GUID_DEFINED
#if !defined( __LPGUID_DEFINED__ )
#define __LPGUID_DEFINED__
typedef GUID __RPC_FAR *LPGUID;

#endif // !__LPGUID_DEFINED__
#ifndef __OBJECTID_DEFINED
#define __OBJECTID_DEFINED
#define _OBJECTID_DEFINED
typedef struct  _OBJECTID
    {
    GUID Lineage;
    unsigned long Uniquifier;
    }    OBJECTID;

#endif // !_OBJECTID_DEFINED
#if !defined( __IID_DEFINED__ )
#define __IID_DEFINED__
typedef GUID IID;

typedef IID __RPC_FAR *LPIID;

#define IID_NULL            GUID_NULL
typedef GUID CLSID;
```

Die Herausforderung

Können Sie eine Visual Basic-Deklaration für GUIDs, IIDs und CLSIDs erstellen?

Hinweis

Für dieses Puzzle gibt es keinen entsprechenden Beispielcode auf der Begleit-CD-ROM.

Puzzle 26

Universelle Bezeichner, Teil 2

Der Grund für die Existenz von universellen Bezeichnern ist die Unmöglichkeit, zwei Personen daran zu hindern, Objekte und Schnittstellen mit den gleichen Namen zu versehen. Der Standardname für jedes Projekt in Visual Basic lautet beispielsweise `Project1`. Der Standardname für die erste Klasse in einem solchen Objekt lautet `Class1`. Wenn Sie also zu Testzwecken eine schnelle ActiveX DLL erstellen und nicht daran denken, diese Namen zu ändern, erstellen Sie einen Server mit dem Objekt `Projekt1.Class1`. Wie viele Objekte mit dem Namen `Project1.Class1` befinden sich auf Ihrem System? Ich möchte gar nicht wissen, wie viele dieser Projekte auf meinem System herumschwirren. Und wenn Sie hierzu noch die Objekte zählen, die sich auf allen Systemen der Welt befinden ... eine ziemlich furchtbare Vorstellung.

Die lesbare Form eines wie oben beschriebenen Objekts wird Programm-ID genannt, und Sie sollten immer sicherstellen, dass es sich hierbei um eine eindeutige ID handelt. Die Programm-ID wird in der Registrierung gespeichert, d. h., über die `CreateObject`-Funktion kann der Server ermittelt werden, der das Objekt erstellt hat. Die Entwickler von COM wollten jedoch eine Methode bereitstellen, mit der Objekte selbst dann eindeutig identifiziert werden können, wenn ihnen dieselbe Programm-ID zugewiesen wurde. Daher wird jedes Objekt tatsächlich über seinen `CLSID`-Wert identifiziert. Die Programm-ID in der Registrierung dient lediglich dazu, die `CLSID` für das Objekt aufzufinden. Das Objekt wird tatsächlich unter Verwendung der `CLSID` geladen und nicht mit Hilfe der Programm-ID.

Wenn Sie beispielsweise die `CreateObject`-Funktion dazu verwenden, ein Microsoft Word-Dokument zu erstellen, würden Sie die Programm-ID »Word.Document« verwenden. Windows würde in der Systemregistrierung nach der Zeichenfolge »Word.Document« suchen und ermitteln, dass die CLSID für dieses Objekt {00020906-0000-0000-C000-000000000046} lautet.

Das bedeutet, dass es eine Methode geben muss, mir der die `CLSID` für eine vorgegebene Programm-ID ermittelt werden kann. Sie könnten natürlich in der Registrierung nachsehen, aber es gibt eine OLE-Funktion, die diese Aufgabe für Sie erledigen kann.

Die hier benötigte CLSIDFromProgID-Funktion wird in der Win32-Dokumentation folgendermaßen definiert:

```
HRESULT CLSIDFromProgID(
    LPCOLESTR lpszProgID,          //Zeiger auf die ProgID
    LPCLSID pclsid                 //Zeiger auf die CLSID
   );
```

Der `lpszProgID`-Parameter ist eine Zeichenfolge, die die gesuchte Programm-ID enthält. Bei dem Parameter `pclsid` handelt es sich um eine Struktur, die zusammen mit der `CLSID` für das Objekt geladen wird.

Entwickeln des Testprogramms

Das in diesem Puzzle verwendete Testprogramm ist sehr einfach, besteht jedoch aus vier verschiedenen Dateien. Warum? Weil das Puzzle die `CLSID` für ein Objekt erhält, das Teil des Puzzles ist.

Wie wird dies erreicht?

Zunächst muss es sich bei dem Puzzle um einen ActiveX-Server handeln, da nur ActiveX-Server Objekte bereitstellen können. In unserem Fall ist es ein ActiveX EXE-Server, da ja erreicht werden soll, dass das Programm auch im Standalonemodus ausgeführt werden kann. Aktivieren Sie mit Hilfe der Registerkarte **Komponente** im Dialogfeld **Projekteigenschaften** den Standalonemodus, und stellen Sie auf der Registerkarte **Allgemein** das Startobjekt auf `Sub Main` ein, damit der folgende Code aus dem Modul `modGUIDPuzzle` ausgeführt werden kann:

```
Sub main()
   frmShowGUID.Show
End Sub
```

Das Formular `frmShowGUID` enthält zwei Bezeichnungsfelder. Über das Formular wird der folgende Code implementiert:

```
' GUID Puzzle
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten

Option Explicit

Private Sub Form_Load()
   Dim obj As New GUIDObject
   Dim cid As CLSID
   Dim ciddesc As String
   Dim x As Long
```

```
   cid = obj.GetCLSID()

   ciddesc = Hex$(cid.Data1) & vbCrLf
   ciddesc = ciddesc & Hex$(cid.Data2) & vbCrLf
   ciddesc = ciddesc & Hex$(cid.Data3) & vbCrLf
   For x = 0 To 7
      ciddesc = ciddesc & GetDataString(VarPtr(cid.Data4(x)), 1) & " "
   Next x
   lblGUIDAsStruct.Caption = ciddesc
   lblRawData.Caption = GetDataString(VarPtr(cid), Len(cid))
End Sub
```

Über das Formular wird ein `GUIDObject`-Objekt erstellt, über das die Funktion `GetCLSID` offen gelegt wird. Mit Hilfe dieser Funktion kann die Objekt-`CLSID` abgerufen werden. Das Bezeichnungsfeld `lblGUIDAsStruct` zeigt jedes Feld in der `CLSID`-Struktur unabhängig von den anderen an. Mit Hilfe der Funktion `GetDataString` (die sich im Modul `ErrString` befindet) erscheinen die Daten als hexadezimale Werte.

Über das Bezeichnungsfeld `lblRawData` wird die gesamte Struktur angezeigt, wie sie im Speicher vorliegt. Der Grund für die Anzeige der Strukturfelder und der Speicheranordnung wird später deutlich, ist aber für das Puzzle nicht von besonderer Bedeutung.

Die `GUIDObject`-Klasse legt das `GUIDObject`-Objekt offen und enthält den folgenden Code:

```
' GUID Puzzle Example Class
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten

Public Type CLSID
   Data1 As Long
   Data2 As Integer
   Data3 As Integer
   Data4(7) As Byte
End Type

Private Declare Function CLSIDFromProgID Lib "ole32.dll" _
(ByVal lpszProgId As String, lpclsid As CLSID) As Long

Option Explicit

Public Function GetCLSID() As CLSID
```

```
   Dim cid As CLSID
   Dim res As Long
   Dim MyProgramId As String

   MyProgramId = "GUIDPuzzle.GUIDObject"
   res = CLSIDFromProgID(MyProgramId, cid)
   If res = 0 Then
      GetCLSID = cid
   Else
      ' Was geschieht, wenn Sie hier keinen Kommentar einfügen?
      ' Err.Raise res
   End If
   End If
End Function
```

Diese Klasse nutzt eine neue Funktion von Visual Basic 6, nämlich die Fähigkeit, benutzerdefinierte Typen als Parameter zu verwenden und Werte von Klassenmethoden und -eigenschaften zurückzugeben. Wenn Sie mit Visual Basic 5 arbeiten, werden Sie nicht in der Lage sein, die `CLSID`-Werte, wie hier beschrieben, abzurufen. Sie lösen dieses Problem, indem Sie die Bezeichnungsfelderwerte für das Formular direkt über die Klasse setzen.

Ergebnisse

Wenn Sie das Programm ausführen, wird das in Abbildung P26-1 dargestellte Formular angezeigt.

Offensichtlich stimmt etwas nicht. Ihre Aufgabe: Ändern Sie das Programm so ab, dass die `CLSID` für das Objekt tatsächlich angezeigt wird.

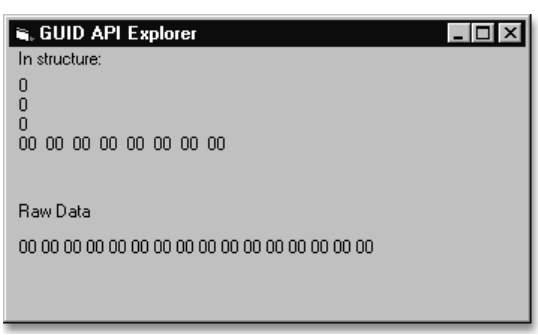


Abbildung P26-1 Die Ergebnisse bei der Ausführung des Programms **GUIDPuzzle**

Puzzle 27

Universelle Bezeichner, Teil 3

In Puzzle 26, »Universelle Bezeichner, Teil 2«, haben Sie gelernt, wie Sie mit der CLSID-Struktur arbeiten und wie die Daten in dieser Struktur dem Benutzer angezeigt werden. Dargestellt wurden einerseits die einzelnen Felder der Struktur, andererseits jedoch die Ursprungsdaten innerhalb derselben. Sie haben darüber hinaus erfahren, dass GUIDs häufig im Zeichenfolgenformat beschrieben werden.

Die Win32-API umfasst die Funktion StringFromCLSID, die dazu verwendet werden kann, eine Zeichenfolgendarstellung der GUID-Daten zu erhalten. In der Win32-Dokumentation wird diese Funktion folgendermaßen definiert:

```
WINOLEAPI StringFromCLSID(
REFCLSID rclsid,                    // Zu konvertierende CLSID
    LPOLESTR * ppsz                 // Indirekter Zeiger auf die
                                    // zurückgegebene Zeichenfolge
   );
```

Parameter

- rclsid
  [in] CLSID, die konvertiert werden soll
- ppsz
  [out] Zeiger auf die Ergebniszeichenfolge

Der Rückgabetyp lautet WINOLEAPI. Dieser Typ wird in der Datei **objbase.h** folgendermaßen definiert:

```
#define WINOLEAPI        STDAPI
```

Der Datentyp STDAPI wird in der Datei **basetyps.h** wie folgt definiert:

```
#define STDAPI           EXTERN_C HRESULT STDAPICALLTYPE
```

Bei EXTERN, C und STDAPICALLTYPE handelt es sich um Begriffe aus der Sprache C, mit der die Aufrufkonvention beschrieben wird. Diese Aufrufkonvention wird in Puzzle 29 erläutert und ist für das vorliegende Puzzle unwichtig. HRESULT gibt an, dass über diese Funktion ein 32-Bit-Long-HRESULT-Wert zurückgegeben wird. Dies bedeutet, dass der Rückgabetyp als Long definiert werden muss. Denken Sie daran, dass das Ergebnis zu einem sofortigen Fehler führen kann, wenn der Wert ungleich 0 ist.

Jetzt zu den Parametern:

Der `rclsid`-Parameter ist ein Zeiger auf die CLSID-Struktur, die in eine Zeichenfolge konvertiert werden soll.

Der Parameter `ppsz` ist ein wenig komplexer. Würde es sich bei dem Parameter um eine LPOLESTR handeln, läge ein Zeiger auf eine OLE-Zeichenfolge oder eine `wide`-Zeichenfolge (Unicode-) vor. Da jedoch ein zusätzliches Sternchen (*) vorhanden ist, handelt es sich hier um einen Zeiger auf einen Zeiger auf eine Zeichenfolge. Ihr erster Gedanke ist vielleicht, eine Zeichenfolge als Verweis zu übergeben, gerade weil Sie wissen, dass in Visual Basic Zeichenfolgen intern als Unicode-Daten gespeichert werden.

Diese Überlegung würde zu folgendem Code führen:

```
' GUID Puzzle Example Class
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten

Public Type CLSID
   Data1 As Long
   Data2 As Integer
   Data3 As Integer
   Data4(7) As Byte
End Type

Private Declare Function CLSIDFromProgID Lib "ole32.dll" _
(ByVal lpszProgId As Long, lpclsid As CLSID) As Long
Private Declare Function StringFromCLSID Lib "ole32.dll" _
(lpclsid As CLSID, lpOlestr As String) As Long

Option Explicit

Public Function GetCLSIDAsString() As String
   Dim cid As CLSID
   Dim MyString As String

   cid = GetCLSID()

   Call StringFromCLSID(cid, MyString)
   GetCLSIDAsString = MyString
End Function
```

Ergebnisse

Natürlich stürzt das Beispiel unter Windows NT mit einem spektakulären Speicherausnahmefehler ab.[2]

Können Sie das Problem lösen?

2. Unter Windows 95/98 stürzt das Programm zwar nicht ab, aber die zurückgegebenen Daten ergeben keinerlei Sinn.

Puzzle 28

Zeichnen von OLE-Objekten

Als ich zum ersten Mal OLE 2.0 sah, dachte ich, es handele sich um die komplexeste Technologie, die Microsoft jemals auf den Markt gebracht habe. Dann plauderte ich mit einem Microsoft-Entwickler. Er erklärte mir, dass es sich wahrscheinlich um die komplexeste Technologie handle, die Microsoft je auf den Markt gebracht habe. Jetzt, nachdem ich mehrere Jahre mit OLE gearbeitet habe und OLE mittlerweile ActiveX und COM heißt, ist mir klar geworden, dass es sich wahrscheinlich um die komplexeste Technologie handelt, die Microsoft jemals auf den Markt gebracht hat.

Deshalb werde ich gar nicht erst versuchen, Ihnen in diesem Buch die OLE-Programmierung von Visual Basic aus zu erläutern. Ich denke, die Puzzle 25, 26 und 27 haben Ihnen bereits einen kleinen Vorgeschmack gegeben. (In meinem Buch »Developing COM/ActiveX Components with Visual Basic 6.0« können Sie eine Menge über die Konzepte von COM und OLE erfahren. Wenn Sie nicht über ein wenig Erfahrung im Umgang mit COM verfügen, werden Sie dieses Puzzle als schwierig empfinden, wenn nicht gar als unverständlich.)

Ich habe an verschiedenen Stellen erwähnt, dass in OLE Array- und VB-Zeichenfolgen (`BSTRS`) sowie in Arrayparametern (`SAFEARRAY`) verwendet werden können. Die Wahrheit ist jedoch, dass Sie wahrscheinlich nie in die Situation kommen werden, diese zu verwenden, da praktisch sämtliche OLE-Funktionen, die diese Parameter verwenden, bereits in Visual Basic selbst verfügbar sind.

Es gibt jedoch einen Parametertyp, der zusammen mit OLE-Funktionen gebraucht werden kann, und den Sie häufig verwenden müssen, wenn Sie die OLE-API einsetzen, nämlich die Objekte.

Anzeigen und Zeichnen von Steuerelementen

Haben Sie jemals versucht, die Inhalte eines einzelnen Steuerelements auszudrucken? Oder haben Sie die Inhalte eines Steuerelements ausgedruckt oder angezeigt, dass nicht tatsächlich sichtbar ist? Visual Basic bietet keine wirklich brauchbare Lösung zur Ausführung dieser Aufgabe. Sie können die Inhalte eines Steuerelements zeichnen oder drucken, indem Sie dieses anzeigen und als Bitmap kopieren. Das Vergrößern und Ausdrucken kleiner Bitmaps führt jedoch üblicherweise zu einer schlechten Bildqualität, insbesondere wenn es sich dabei um Text handelt. Darüber hinaus ist es frustrierend, jedes Steuerelement anzeigen zu müssen, bevor es gedruckt werden kann.

ActiveX-Steuerelemente müssen eine Schnittstelle unterstützen, die als `IViewObject` oder `IViewObject2` bezeichnet wird. Diese Schnittstelle beschreibt einen

Standardsatz von Funktionen, mit denen ein Objekt sich selbst in einen Gerätekontext zeichnen kann (der zu einem Fenster oder einem Drucker gehören kann). Wenn Sie die `IViewObject`-Schnittstelle direkt verwenden und so Steuerelemente dazu zu zwingen könnten, ihre Inhalte in einen Gerätekontext zu zeichnen, liessen Sie nicht sichtbare oder nur teilweise sichtbare Steuerelemente anzeigen. Sie könnten diese auch ausdrucken.

Die OLE-API verfügt über diese Funktion. Die Funktion heißt `OLEDraw` und wird in der Win32 SDK folgendermaßen definiert:

```
WINOLEAPI OleDraw(
    IUnknown * pUnk,      //Zeiger auf das zu zeichnende Objekt
    DWORD dwAspect,       //Darstellungsform für das Objekt
    HDC hdcDraw,          //Gerätekontext, in dem gezeichnet werden soll
    LPCRECT lprcBounds    //Zeiger auf das Rechteck, in dem das Objekt
                          //gezeichnet wird
  );
```

Der `pUnk`-Parameter ist als `IUnknown *` definiert. `IUnknown` ist der Name der Schnittstelle, auf der sämtliche weiteren Schnittstellen basieren. Mit anderen Worten: Wenn Sie über einen Verweis auf ein COM-Objekt verfügen, spielt es keine Rolle, ob dieser in einer Variablen vorliegt, die `As Object`, `As IUnknown` oder als ein beliebiger anderer Objekttyp deklariert ist. Der Verweis enthält stets eine `IUnknown`-Schnittstelle und ist als Parameter gültig, der als Zeiger auf eine `IUnknown`-Schnittstelle deklariert ist.

Die übrigen Parameter sind verständlicher:

Der `dwAspect`-Parameter ermöglicht Ihnen die Angabe eines Darstellungstyps. Für ActiveX-Steuerelemente lautet dieser Wert stets 1 – mit diesem Wert wird das Steuerelement angewiesen, seine Inhalte anzuzeigen.

Bei dem `hdcDraw`-Parameter handelt es sich um eine Zugriffsnummer für einen Gerätekontext. Es kann dabei um einen API- oder Druckergerätekontext gehen.

Der `IprcBounds`-Parameter nimmt die Koordinaten für das Rechteck auf, in dem die Inhalte des Steuerelements gezeichnet werden sollen. LP gibt an, dass es sich um einen `Long-Zeiger (far)` auf eine `RECT`-Struktur handelt. Die Werte für das Rechteck sollten in logischen Koordinaten für das System angegeben werden, das vom Gerätekontext verwendet wird.[3]

3. Kapitel 7 meines Buches »Visual Basic Programmer's Guide to the Win32 API« geht mehr als erschöpfend auf die Verwendung von Gerätekontext, auf die Bedeutung von Koordinatensystemen und die Definition logischer Koordinaten ein. Setzen wir im Moment der Einfachheit halber einfach voraus, dass die Koordinaten in Pixeln angegeben werden.

Das Programm OLEDraw

In Abbildung P28-1 wird das Aussehen des Beispielprogramms **OLEDraw** zur Entwicklungszeit dargestellt. Mit diesem Programm werden zwei Steuerelemente angezeigt: Bei dem ersten Steuerelement handelt es sich um das private Steuerelement `UserControl`, mit dem 16 vertikale farbige Balken angezeigt werden. Das zweite Steuerelement ist das Microsoft-Steuerelement `MonthView`, das Bestandteil der Microsoft Windows Common Controls-2-Bibliothek ist.

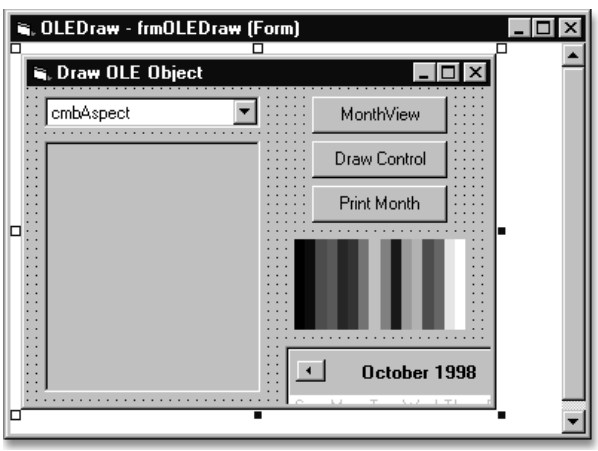


Abbildung P28-1 Aussehen des Beispielprogramms OLEDraw zur Entwicklungszeit

Sie werden bemerken, dass das Steuerelement `MonthView` im Fenster nur teilweise angezeigt wird. Abbildung P28-2 zeigt das Aussehen des Programms mit aktivem Steuerelement, nachdem Sie auf die Befehlsschaltfläche `MonthView` geklickt haben. Besser gesagt, in dieser Abbildung sehen Sie, wie das Programm aussieht, wenn es funktioniert.

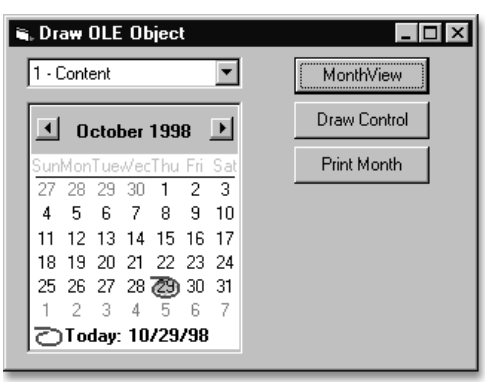


Abbildung P28-2 Das **OLEDraw**-Programm nach dem Klicken auf die Befehlsschaltfläche **MonthView**

Bevor Sie mit der Bearbeitung des Puzzles beginnen, wollen wir uns einen Augenblick der Implementierung des **OLEDraw**-Programms zuwenden.

Das Steuerelement »UserControl«

Das Steuerelement `PrivateCtl` ist ein triviales Steuerelement, mit dem 16 vertikale Balken in unterschiedlichen Farben gezeichnet wird.

```
' OleDraw Sample

' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten
Option Explicit

Private Sub UserControl_Paint()
   Dim x As Long
   Dim bandsize As Long
bandsize = ScaleWidth / 16
   For x = 0 To 15
      Line (bandsize * x, 0)-(bandsize * (x + 1), ScaleHeight), _
      QBColor(x Mod 16), BF
   Next x
End Sub
```

Das Formular

Die `OleDraw`-Funktion wird folgendermaßen deklariert. Die `RECT`-Struktur enthält vier Felder, die den vier Seiten des Rechtecks entsprechen, in dem gezeichnet werden soll. Das `Aspects`-Array enthält die Werte 1, 2, 4 und 8, entsprechend den vier möglichen `Aspect`-Werten. Wie bereits an früherer Stelle erwähnt, ist der einzige Wert, der bei den meisten ActiveX-Steuerelementen nicht sofort zu einem Fehler führt, der Wert 1.

```
' OLE Drawing Puzzle
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten
Option Explicit

Private Declare Function OleDraw Lib "ole32.dll" _
   (pUnk As Object, _
   ByVal dwAspect As Long , _
      ByVal hdcDraw As Long, _
      lprcBounds As RECT) As Long

Private Type RECT
   Left As Long
```

```
   Top As Long
   Right As Long
   Bottom As Long
End Type

Dim Aspects(3) As Long
```

Die Funktionen `DrawTheObject` und `PrintTheObject` sind sich sehr ähnlich. Der einzige Unterschied zwischen diesen beiden Funktionen besteht darin, dass sich das eine auf das Bildsteuerelement, das andere auf das Druckerobjekt bezieht. Die `ScaleMode`-Eigenschaft des Bildsteuerelements ist per Vordefinition auf 3 eingestellt, nämlich auf `vbPixels`, d.h. diese Eigenschaft muss nicht explizit gesetzt werden, wie dies über die Funktion `PrintTheObject` der Fall ist.

```
Private Function DrawTheObject(obj As Object)
   Dim r As RECT
   Dim res As Long
   r.Right = Picture1.ScaleWidth
   r.Bottom = Picture1.ScaleHeight
   res = OLEDraw(obj, Aspects(cmbAspect.ListIndex), Picture1.hdc, r)
   If res <> 0 Then
      Err.Raise res
   End If
End Function


Private Function PrintTheObject(obj As Object)
   Dim r As RECT
   Dim res As Long
   Printer.ScaleMode = vbPixels
   r.Right = Printer.ScaleWidth
   r.Bottom = Printer.ScaleHeight
   res = OLEDraw(obj, Aspects(cmbAspect.ListIndex), Printer.hdc, r)
   If res <> 0 Then
      Err.Raise res
   End If
   Printer.EndDoc
End Function


Private Sub cmdDrawText_Click()
   DrawTheObject Privatectl1
```

```
End Sub

Private Sub cmdDrawWord_Click()
   DrawTheObject MonthView1
End Sub

Private Sub Command1_Click()
   PrintTheObject MonthView1
End Sub

Private Sub Form_Load()
   cmbAspect.ListIndex = 0
   Aspects(0) = 1
   Aspects(1) = 2
   Aspects(2) = 4
   Aspects(3) = 8
End Sub
```

Ergebnisse

Die Ergebnisse nach der Programmausführung richten sich nach dem verwendeten Betriebssystem. In den meisten Fällen verschwindet das Programm kurz, sobald Sie auf eine der Schaltflächen zum Zeichnen klicken. Kurz nachdem das Programm verschwunden ist, passiert nichts mehr. Wir wurden Zeuge von einem der »stillen« Abstürze.

Ihre Aufgabe: Ändern Sie das Programm so ab, dass Sie nicht nur die Steuerelemente im Bildfeld anzeigen, sondern auch die Monatsansicht ausdrucken können.

Abschnitt 5: High Technology

Nun, da wir uns den letzten Puzzlen in diesem Buch nähern, frage ich mich, ob der Schwierigkeitsgrad der Puzzle angemessen ist. Sind sie zu leicht? Sind sie zu schwer? Ich hoffe, dass selbst ungeübte Programmierer mit Hilfe der Tutorien (in Teil III) die meisten Puzzle als Herausforderung, aber auch als machbar betrachten. Ich möchte jedoch ebenso, dass auch die erfahrensten Programmierer nach der Lektüre dieses Buches das Gefühl haben, etwas dazugelernt zu haben.

Die letzten vier Puzzle sind so konzipiert, dass selbst die wirklichen API-Cracks unter Frustrationen leiden werden.

Der Begriff »High Technology« soll hierbei auf eine komplexe Technologie verweisen, die nur von wirklich erfahrenen Experten verstanden wird. Ich denke, nachdem Sie diesen Abschnitt durchgearbeitet haben, werden Sie mit mir übereinstimmen, dass dieser Begriff für die vorliegenden Puzzle angemessen ist.

Puzzle 29

Was tun, wenn's richtig weh tut?

Schnell – wie viele Prozessoren befinden sich auf dem System, auf dem Ihre Anwendung ausgeführt wird? Wie schnell ist die CPU? Was tun Sie, wenn Intel veröffentlicht, dass ein Problem mit einem bestimmten Prozessor vorliegt und Sie den genauen Typ, das Modell und die Stepping Number der CPU ermitteln müssen, mit der Ihre Anwendung ausgeführt wird?

Intel stellt eine DLL zur Verfügung, mit der Sie all diese Informationen erhalten können. Diese DLL heißt **cpuinf32.dll** und kann in der Website von Intel unter **http://developer.intel.com/vtune/cpuid/package.htm** abgerufen werden, auf der Sie auch Informationen zum Herunterladen des Pakets finden.[1]

Bevor Sie jetzt aber ganz wild auf diese coole kleine DLL werden, muss ich Sie warnen: Diese DLL funktioniert nur zuverlässig mit Intel-Prozessoren. Davon einmal abgesehen ergibt sich hierbei ein faszinierendes Problem.

Die DLL umfasst die Funktion `cpuspeed`, mit der Sie die Grundgeschwindigkeit sowie die normalisierten Geschwindigkeiten der CPU ermitteln können.[2] Die C-Deklaration für diese Funktion lautet wie folgt:

```
struct FREQ_INFO cpuspeed(int clocks);
```

Der `clocks`-Parameter sollte auf 0 gesetzt werden, um ihn für den Test die Standardanzahl der Taktzyklen zu verwenden. Die `FREQ_INFO`-Struktur wird folgendermaßen definiert:

```
Type FREQ_INFO
   in_cycles As Long      ' Interne Taktzeiten während des Tests
   ex_ticks As Long       ' Zeit in Mikrosekunden, die während des Tests
                          ' verstreicht
   raw_freq As Long       ' Grundgeschwindigkeit der CPU in MHz
   norm_freq As Long      ' Normalisierte Frequenz der CPU in MHz
End Type
```

Trauen Sie sich zu, die richtigen Deklarationen zu schreiben?

1. Zumindest befanden sich diese Informationen zum Zeitpunkt der Veröffentlichung dieses Buches in der Website. Wenn sich der Standort geändert haben sollte, finden Sie den aktualisierten Standort unter **http://www.desaware.com** – sobald wir den neuen Standort gefunden haben.
2. Als normalisierte Frequenz gilt die an die gemessene Frequenz angenäherte CPU-Standardfrequenz. Praktisch gesehen scheint es sich dabei lediglich um die im Hinblick auf Rundungsfehler angepasste Geschwindigkeit zu handeln.

Puzzle 30

Dateioperationen, Teil 1

Jeder Windows-Benutzer kennt die Dateioperationstypen, die vom Windows Explorer unterstützt werden. Entschuldigung, nicht vom Windows Explorer (obwohl es immer schwieriger wird, den Unterschied zwischen den beiden auszumachen), sondern von der Hauptschnittstelle von Windows. Der Windows Explorer ermöglicht es Ihnen, auf einfache Weise Dateien von einem Verzeichnis in ein anderes zu ziehen, zu verschieben, zu kopieren, zu löschen usw. Wir haben alle schon oft die Dialogfelder gesehen, mit denen wir um eine Bestätigung gebeten werden, ein Feedback zu einer laufenden Operation erhalten oder über aufgetretene Fehler benachrichtigt werden.

Der Windows Explorer verfügt über eine Vielzahl von Dateiverwaltungsfunktionen, und wir alle arbeiten täglich mit diesen Funktionen. Was sich jedoch viele Windows-Programmierer nicht vor Augen führen, ist die Tatsache, dass diese Funktionalität in eine kleine Funktion gepackt wurde, die als `SHFileOperation` bezeichnet wird.

Diese einfache Funktion mit nur einem Strukturparameter wird in der Win32 SDK folgendermaßen definiert:

```
WINSHELLAPI int WINAPI SHFileOperation(
    LPSHFILEOPSTRUCT lpFileOp
   );
```

Die Visual Basic-Deklaration ist leicht verständlich:

```
Private Declare Function SHFileOperation Lib "shell32.dll" _
Alias "SHFileOperationA" (lpFileOp As SHFILEOPSTRUCT) As Long
```

Wie kann nur eine Funktion all diese Dateioperationen ausführen? Die Antwort ist einfach – sämtliche der Parameter werden in der Struktur `SHFILEOPSTRUCT` gebündelt, die in der Win32 SDK wie folgt definiert ist:

```
typedef struct _SHFILEOPSTRUCT { // shfos
    HWND                        hwnd;
    UINT                        wFunc;
    LPCSTR                      pFrom;
    LPCSTR                      pTo;
    FILEOP_FLAGS                fFlags;
    BOOL                        fAnyOperationsAborted;
    LPVOID                      hNameMappings;
    LPCSTR                      lpszProgressTitle;
} SHFILEOPSTRUCT, FAR *LPSHFILEOPSTRUCT;
```

Der einzig unklare Parameter hierbei ist der Datentyp FILEOP_FLAGS, der folgendermaßen definiert ist:

```
typedef WORD FILEOP_FLAGS;
```

Ich bin gerade in Gönnerlaune, deshalb müssen Sie die Strukturdeklaration nicht allein ermitteln. Hier die Deklaration von Microsoft, die in der Visual Basic-Datei **win32api.txt** enthalten ist:

```
Type SHFILEOPSTRUCT
        hwnd As Long
        wFunc As Long
        pFrom As String
        pTo As String
        fFlags As Integer
        fAnyOperationsAborted As Long
        hNameMappings As Long
        lpszProgressTitle As String '  only used if FOF_SIMPLEPROGRESS
End Type
```

Die SHFILEOPSTRUCT-Struktur

Die SHFILEOPSTRUCT-Struktur verwendet zur Definition der auszuführenden Operation viele verschiedene Konstanten.

Das Feld wFunc sollte auf einen der folgenden vier Werte gesetzt werden, mit denen die Hauptoperationen definiert werden:

```
Private Const FO_MOVE = &H1
Private Const FO_COPY = &H2
Private Const FO_DELETE = &H3
Private Const FO_RENAME = &H4
```

Glücklicherweise erklären sich diese Konstanten von selbst.

Das fFlags-Feld sollte, in Kombination mit dem Or-Operator, auf eine der nachstehenden Konstanten gesetzt werden: Mit diesem Feld wird die Funktionsoperation geändert:

```
Private Const FOF_MULTIDESTFILES = &H1
Private Const FOF_CONFIRMMOUSE = &H2
Private Const FOF_SILENT = &H4     ' Keinen Fortschrittsbericht/Bericht
                                   ' erstellen
Private Const FOF_RENAMEONCOLLISION = &H8
Private Const FOF_NOCONFIRMATION = &H10
```

```
Private Const FOF_WANTMAPPINGHANDLE = &H20
Private Const FOF_ALLOWUNDO = &H40
Private Const FOF_FILESONLY = &H80
Private Const FOF_SIMPLEPROGRESS = &H100
Private Const FOF_NOCONFIRMMKDIR = &H200
```

Jetzt beginnt der Spaß. Die Strukturfelder `pForm` und `pTo` können eine Namensliste für eine oder mehrere Dateien und Verzeichnisse enthalten.

- Wenn das Attribut `FOF_MULTIDESTFILES` gesetzt ist, enthalten `pFrom` und `pTo` die gleiche Anzahl von Dateinamen. Jede Datei in der `pFrom`-Liste wird kopiert, umbenannt oder an die in der `pTo`-Liste angegebene Datei verschoben.
- `FOF_CONFIRMMOUSE` wird derzeit nicht verwendet.
- `FOF_SILENT` hindert die Funktion `SHFileOperation` daran, Dialogfelder zum Fortschritt einer Operation anzuzeigen.
- Über `FOF_RENAMEONCOLLISION` wird die Funktion dazu veranlasst, eine Kopie der Datei zu erstellen, wenn im Zielverzeichnis bereits eine gleichnamige Datei vorliegt. Das Namensformat der Kopie lautet »Kopie von ...« – dieser Name ist Ihnen mit Sicherheit durch das Arbeiten mit dem Windows Explorer bekannt.
- Durch `FOF_NOCONFIRMATION` wird die Funktion veranlasst, auch solche Operationen automatisch auszuführen, für die normalerweise eine Benutzerbestätigung erforderlich wäre. Hierzu zählt beispielsweise das Überschreiben von Dateien, das Erstellen von Verzeichnissen usw.
- Über `FOF_ALLOWUNDO` werden gelöschte Dateien und Verzeichnisse in den Papierkorb verschoben.
- Mit Hilfe von `FOF_FILESONLY` werden bei der Ausführung von Operationen mit Platzhaltern (z. B. *.*) nur Dateien kopiert. Verzeichnisse werden ignoriert.
- Durch `FOF_SIMPLEPROGRESS` wird die Anzeige der Dateinamen während eines Kopiervorgangs deaktiviert. Es werden nur Nachrichten angezeigt, die im Feld `lpszProgressTitle` angegeben werden.
- Über `FOF_NOCONFIRMMKDIR` wird die Funktion dazu veranlasst, automatisch neue Verzeichnisse zu erstellen, wenn dies zur Ausführung der Operation erforderlich ist.
- Mit `FOF_WANTMAPPINGHANDLE` wird das Feld `hNameMappings` so eingestellt, dass auf ein Array von `SHNAMEMAPPING`-Strukturen mit Quell- und Zieldateinamen (nach der Operation) verwiesen wird.

Was diese Funktion so interessant macht, ist das Format dieser Zeichenfolgen. Wie können durch eine Zeichenfolge mehrere Dateien angegeben werden? Im Gegensatz zu Visual Basic, bei dem die Länge jeder Zeichenfolge intern gespeichert wird, verwenden die Win32-API-Funktionen die C-Konvention, bei der eine Zeichenfolge mit einem NULL-Wert endet. Dies ist der Grund dafür, warum Win32-API-Zeichenfolgen im Gegensatz zu Visual Basic-Zeichenfolgen keine eingebetteten NULL-Werte enthalten können. Diese Funktion verwendet eine gemeinsame Konvention zur Übergabe mehrerer Zeichenfolgen in einem Puffer, wobei jede Zeichenfolge durch einen NULL-Wert abgetrennt wird und die letzte Zeichenfolge auf zwei NULL-Werte endet.

Das Beispielprogramm **FileOp1** enthält zwei Routinen zur Konvertierung von Zeichenfolgenarrays in diese auf NULL endenden Zeichenfolgenpuffer und umgekehrt. Mit Hilfe von `GetNullsString` wird ein Array aus Zeichenfolgen in einen Puffer konvertiert, wie nachstehend veranschaulicht wird.

```
Private Function GetNullsString(files() As String) As String
   Dim x&
   Dim res$
   For x = 1 To UBound(files)
      res$ = res$ & files(x) & Chr$(0)
   Next x
   ' Warum werden hier zwei Nullen eingefügt?
   ' Können Sie sich den Grund denken?
   res$ = res$ & Chr$(0) & Chr$(0)
   GetNullsString = res$
End Function
```

Bei der Beantwortung der oben genannte Frage sollten Sie berücksichtigen, was passiert, wenn sich im Array keine Zeichenfolgen befinden. Die umgekehrte Konvertierung ist komplexer und in der nachfolgenden Funktion `GetFileArray` beschrieben:

```
' Konvertieren einer auf zwei Nullen endenden Zeichenfolge
Private Function GetFileArray(NullsString As String, Optional _
ByVal SeparatorChar As Byte = 0) As Variant
   Dim nullpos&
   Dim curpos&
   Dim results() As String
   Dim itemcount&
   curpos = 1  ' Standort beim Start
   ReDim results(0)
```

```
   Do
      nullpos = InStr(curpos, NullsString, Chr$(SeparatorChar))
      If nullpos = 0 Then
         ' Dies sollte nie nie passieren
         ' Nur zurückgeben, was wir bisher haben
         GetFileArray = results()
         Exit Function
      End If
      If nullpos = curpos Then
         ' Die zweite Null - fertig!
         GetFileArray = results()
         Exit Function
      End If
      itemcount = itemcount + 1
      ReDim Preserve results(itemcount)
      results(itemcount) = Mid$(NullsString, curpos, nullpos - curpos)
      curpos = nullpos + 1
   Loop While True
End Function
```

Warum wird mit dieser Funktion ein benutzerdefiniertes Trennzeichen bereitgestellt? Da viele Programme Ihnen in einem Textfeld die Eingabe einer Dateiliste ermöglichen, die durch Leerzeichen voneinander getrennt sind. Die gleiche Funktion, die der Analyse von mit Nullen voneinander abgetrennten Zeichenfolgen dient, kann auch Zeichenfolgen analysieren, die mit Leerzeichen getrennt werden.

Das Beispielprogramm FileOp1

Das Programm enthält ein einziges Listenfeld, bei dem die `OLEDropMode`-Eigenschaft auf 1 gesetzt ist und zwar manuell. Die OLE-Ereignisse für das Listenfeld sind so definiert, dass Dateien, die vom Windows Explorer in das Listenfeld gezogen werden, mit folgendem Code ermittelt werden:

```
' Dieses Ereignis wird ausgelöst, wenn die Dateien abgelegt werden
Private Sub lstFiles_
OLEDragDrop(Data As DataObject, Effect As Long, Button As _
 Integer, Shift As Integer, x As Single, Y As Single)
   Dim count&, idx&

If (Effect And vbDropEffectCopy) <> 0 Then
      Effect = vbDropEffectCopy
```

```
      count = Data.files.count

      ReDim FileList(count)
      lstFiles.Clear
      ' Laden der Dateiliste
      For idx = 1 To count
         FileList(idx) = Data.files.Item(idx)
         ' Zum Listenfeld hinzufügen
         lstFiles.AddItem FileList(idx)
      Next idx
      lstFiles.Refresh
      ' Kopiervorgang ausführen
      DoACopy
End If
End Sub

' Dieses Ereignis wird ausgelöst, wenn sich die Maus über dem Listenfeld
' befindet.
Private Sub lstFiles_OLEDragOver(Data As DataObject, _
Effect As Long, Button As Integer, Shift As Integer, x As Single, _
Y As Single, State As Integer)
   If Data.GetFormat(vbCFFiles) Then
      ' Dies ist eine Dateiliste
      If (Effect And vbDropEffectCopy) Then
         ' Dateikopiervorgang bestätigen
         Effect = vbDropEffectCopy
         Exit Sub
      End If
   End If
   ' Dieser Datentyp kann nicht verarbeitet werden
   Effect = vbDropEffectNone
End Sub
```

Der Kopiervorgang wird gestartet, wenn Dateien im Listenfeld abgelegt werden. Auf diese Weise wird das `OLEDragDrop`-Ereignis dazu veranlasst, die `DoACopy`-Funktion aufzurufen, mit der die Programme auf eine Diskette in Laufwerk A: kopiert werden.

Eine Diskette wird deshalb verwendet, weil bei dem vorliegenden Beispiel die Kopiergeschwindigkeit herabgesetzt werden soll, damit Sie alle Fortschrittsdialogfelder sehen können, die bei derartigen Situationen üblicherweise angezeigt werden. Die `DoACopy`-Funktion wird wie folgt definiert:

```
Private Sub DoACopy()
   Dim sh As SHFILEOPSTRUCT
   Dim src$, dest$
   Dim res&

   src$ = GetNullsString(FileList)
   dest$ = "A:\" & Chr$(0) & Chr$(0)

   sh.hwnd = hwnd
   sh.wFunc = FO_COPY
   sh.pFrom = src$
   sh.pTo = dest$
   sh.fFlags = FOF_RENAMEONCOLLISION Or FOF_SIMPLEPROGRESS
   sh.hNameMappings = 0
    sh.lpszProgressTitle = "Files are being copied to floppy"

   res = SHFileOperation(sh)
   If res <> 0 Then
      MsgBox "Error occurred: " & GetErrorString(Err.LastDllError)
   End If

End Sub
```

Die Kopieroperation wird angegeben, indem Sie das Strukturfeld `wFunc` auf `FO_COPY` setzen. Durch `FOF_RENAMEONCOLLISION` wird eine Dateiumbenennung erzwungen, wenn auf der Diskette bereits eine gleichnamige Datei vorliegt. Mit dem Attribut `FOF_SIMPLEPROGRESS` wird festgelegt, dass im Fortschrittsdialogfeld nur die Nachricht angezeigt wird, dass die Dateien auf die Diskette kopiert werden, ohne dass dabei die einzelnen Dateinamen ausgegeben werden.

Ergebnisse

Die im Listenfeld abgelegten Dateien erscheinen im Listenfeld, werden aber nicht kopiert. Darüber hinaus stürzt die Anwendung unter Windows NT mit einem Speicherausnahmefehler ab, sobald Sie sie schließen.

Ihre Aufgabe besteht darin, dass Programm zu berichtigen, sodass es unter Windows NT fehlerfrei ausgeführt werden kann.

Puzzle 31

Dateioperationen, Teil 2

`ShFileOperation` wirkt auf den ersten Blick sehr unschuldig, da nur ein Parameter verwendet wird. Wie Sie jedoch bereits im letzten Puzzle sehen konnten, weist dieser Parameter eine geradezu haarsträubend komplexe Struktur auf:

```
Type SHFILEOPSTRUCT
        hwnd As Long
        wFunc As Long
        pFrom As String
        pTo As String
        fFlags As Integer
        fAnyOperationsAborted As Long
        hNameMappings As Long
        lpszProgressTitle As String '  only used if FOF_SIMPLEPROGRESS
End Type
```

Und wenn Sie dachten, dass Sie nach der Lösung des letzten Puzzle beruhigt schlafen gehen können, habe ich eine gute Nachricht für Sie – der Alptraum fängt gerade erst an.

Die `ShFileOperation`-Funktion unterstützt eine Funktion, die Sie wahrscheinlich schon oft im Windows Explorer beobachtet haben. Was geschieht, wenn Sie eine Datei in den Zwischenspeicher kopieren und diese in ein Verzeichnisfenster einfügen, in dem bereits eine gleichnamige Datei vorliegt? Die Datei wird automatisch umbenannt, wobei der Name in etwa »Kopie von ...« lautet. Mit Hilfe der Funktion `ShFileOperation` wird eine automatische Dateiumbenennung vorgenommen, wenn Sie im `fFlags`-Parameter die Konstante `FOF_RENAMEONCOLLISION` setzen.

Wenn aber durch eine Funktion eine automatische Umbenennung von Dateien erfolgt, sollte es einen Weg geben, mit der die Anwendung ermitteln kann, welche Dateien umbenannt wurden. Dies wird durch die Verwendung von Namenszuordnungen erreicht. Das `hNameMappings`-Feld von `SHFILEOPSTRUCT` wird mit Informationen zu den umbenannten Dateien geladen, wenn Sie im Parameter `fFlags` die Konstante `FOF_WANTMAPPINGHANDLE` angeben.

In der Win32 SDK wird die Verwendung der Konstante `FOF_WANTMAPPINGHANDLE` folgendermaßen definiert:

> *Sie füllt Mitglieder in die `hNameMappings`. Die Zugriffsnummer muss durch Verwenden der `SHFreeNameMapping`-Funktion freigegeben werden.*

Die Strukturbeschreibung enthält folgende zusätzliche Informationen zu diesem Parameter:

- *hNameMappings*
  Zugriffsnummer eines Objekts mit Dateinamenzuordnungen, das ein Array von SHNAMEMAPPING-Strukturen enthält. Jede Struktur enthält den alten und den neuen Pfadnamen für jede Datei, die verschoben, kopiert oder umbenannt wurde. Dieses Mitglied wird nur verwendet, wenn fFlags FOF_WANTMAPPINGHANDLE enthält.

Eine SHNAMEMAPPING-Struktur wird in C folgendermaßen definiert:

```
typedef struct _SHNAMEMAPPING { // shnm
    LPSTR pszOldPath; // Adresse des alten Pfadnamens
    LPSTR pszNewPath; // Zeiger auf den neuen Pfadnamen
    int   cchOldPath; // Anzahl der Zeichen im alten Pfadnamen
    int   cchNewPath; // Anzahl der Zeichen im neuen Pfadnamen
} SHNAMEMAPPING, FAR *LPSHNAMEMAPPING;
```

Folgende Beschreibung findet sich in der Datei **Win32api.txt** von Microsoft:

```
Private Type SHNAMEMAPPING
        pszOldPath As String
        pszNewPath As String
        cchOldPath As Long
        cchNewPath As Long
End Type
```

Die SHFreeNameMapping-Funktion ist leicht verständlich. Mit dieser Funktion werden die Werte aus dem Feld hNameMappings von SHFILEOPSTRUCT als Parameter verwendet und jegliche Daten freigegeben, die der Zugriffsnummer zugewiesen sind.

```
Private Declare Sub SHFreeNameMappings Lib "shell32.dll" _
(ByVal hNameMappings As Long)
```

Sie würden wahrscheinlich damit beginnen, die DoACopy-Funktion aus dem vorangegangenen Puzzle wie folgt zu bearbeiten:

```
' Kopiervorgang ausführen
Private Sub DoACopy()
   Dim sh As SHFILEOPSTRUCT
   Dim src$, dest$
   Dim res&
```

```
   src$ = GetNullsString(FileList)
   dest$ = "A:\" & Chr$(0) & Chr$(0)

   sh.hwnd = hwnd
   sh.wFunc = FO_COPY
   sh.pFrom = src$
   sh.pTo = dest$
   sh.fFlags = FOF_RENAMEONCOLLISION Or FOF_WANTMAPPINGHANDLE
   sh.hNameMappings = 0

   res = DoTheShellCall(sh, "Files are being copied to floppy")

   If sh.hNameMappings <> 0 Then
' Können Sie hier die Dateinamen lesen?
      Call SHFreeNameMappings(sh.hNameMappings)
   End If

End Sub
```

Durch das Entfernen des Attributs `FOF_SIMPLEPROGRESS` aus dem Programm können Sie während des Kopiervorgangs die einzelnen Dateinamen anzeigen. Die Konstante `FOF_WANTMAPPINGHANDLE` wird hinzugefügt, um den `hNameMappings`-Parameter zu aktivieren. Wenn `sh.hNameMappings` ungleich 0 ist, müssen Sie das Namenszuordnungsobjekt freigeben, um ein Speicherleck zu verhindern.

Ergebnisse

Hier können bisher keine Ergebnisse verzeichnet werden. Es handelt sich anscheinend um ein Puzzle mit offenem Ende. Denken Sie, dass Sie damit fertig werden?

- Kommentar der Technischen Revision
  Dan, das ist absolut unfair. Du hast nicht einmal erläutert, was eine Zugriffsnummer für Zuordnungen ist oder wie man diese verwendet. Ich konnte noch nicht einmal herausfinden, warum im Feld `hNameMappings` nie ein Wert angezeigt wird, der ungleich Null ist. Ich denke, die Leser brauchen schon ein paar mehr Informationen.
- Antwort des Autors
  Du hast recht, dieses Puzzle ist wirklich gemein. Ich habe mehr als zwei Stunden benötigt, um dieses Programm zum Laufen zu bringen, und ich fürchte, viele Leser werden das nie schaffen. Aber es ist nicht unfair. Ob Du es mir glaubst oder nicht, ich habe jede relevante Information angeführt, die ich in der Win32 SDK, MSDN und in der Knowledge Base von Microsoft finden konnte. Okay, die Dokumentation zur Verwendung dieser Funktion ist ziemlich übel. Aber meine Leser werden noch häufig in Situationen wie diese geraten, und die Lösung (wenn Sie sich ansehen) hilft Ihnen dabei, mit dieser Art von Situationen selbst fertig zu werden.

Puzzle 32

Animierte Rechtecke

Einen der interessantesten Effekte einer Benutzeroberfläche können Sie beobachten, wenn Sie unter Windows 95 oder Windows NT ein Fenster vergrößern oder verkleinern. Bei der Minimierung scheint das Fenster in der Taskleiste zu verschwinden. Dieser Animationseffekt wird durch eine Funktion erzeugt, die `DrawAnimatedRects` heißt und in der Win32-Dokumentation folgendermaßen definiert wird:

```
BOOL WINAPI DrawAnimatedRects(
    HWND hwnd,              // Zugriffsnummer für Clipping Window
    int idAni,              // Animationstyp
    CONST RECT *lprcFrom,   // Zeiger auf Rechteckkoordinaten (minimiert)
    CONST RECT *lprcTo      // Zeiger auf Rechteckkoordinaten
                            // (wiederhergestellt)
  );
```

Der `hwnd`-Parameter stellt das Fenster dar, in dem die Animation erfolgen soll. Wenn die Fensterzugriffsnummer 0 lautet, wird die Animation statt in einem Fenster auf dem Desktop durchgeführt.

Bei dem `idAni`-Parameter handelt es sich um eine der folgenden Konstanten:

- Private Const IDANI_OPEN = 1
- Private Const IDANI_CLOSE = 2
- Private Const IDANI_CAPTION = 3

Während `IDANI_OPEN` angibt, dass die Rechteckanimation vorgenommen werden sollte, wenn ein minimiertes Rechteck an seiner ursprünglichen Position wiederhergestellt wird, ist es bei `IDANI_CLOSE` genau umgekehrt: Die Rechteckanimation sollte vorgenommen werden, wenn ein wiederhergestelltes Rechteck minimiert wird. `IDANI_CAPTION` gibt an, dass nur der Titel animiert werden soll.

Bei den Parametern `IprcForm` und `IprcTo` handelt es sich um Zeiger auf `RECT`-Strukturen, die vier 32-Bit-Felder enthalten, mit denen die linke, die obere, Breite und Höhe des Rechtecks angegeben werden.

Im Beispielprogramm **Rects.vbp** (auf der Begleit-CD-ROM zu diesem Buch) wird die Verwendung dieser Funktion zur Animation einer Beschriftung – diese verfügt zunächst über eine Höhe und Breite von 0 in der Fenstermitte – demonstriert, um das gesamte Fenster auszufüllen. Die Eigenschaft `ScaleMode` des Formulars wird auf `vbPixels` gesetzt, um sicherzustellen, dass bei den Rechteckkoordinaten die Twips und Pixel nicht durcheinander gebracht werden.

```
' DrawAnimatedRects Example
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten
Option Explicit

Private Type RECT
   left As Long
   top As Long
   width As Long
   height As Long
End Type

Private Const IDANI_OPEN = 1
Private Const IDANI_CLOSE = 2
Private Const IDANI_CAPTION = 3

Private Declare Function DrawAnimatedRects Lib "user32" (ByVal hwnd _
As Long, ByVal idAni As Long, lprcFrom As RECT, lprcTo As RECT) As Long

Private Sub Command1_Click()
   Dim r1 As RECT
   Dim r2 As RECT
   ScaleMode = vbPixels
   r1.left = ScaleWidth / 2
   r1.top = ScaleHeight / 2
   r2.width = ScaleWidth
   r2.height = ScaleHeight
   Call DrawAnimatedRects(hwnd, IDANI_CAPTION, r1, r2)
End Sub
```

Ergebnisse

Die Ergebnisse, die sich durch Klicken auf die Befehlsschaltfläche ergeben, sind, gelinde gesagt, merkwürdig. Das Rechteck wurde in der Tat animiert. Aber die Start- und Endpunkte befinden sich außerhalb des Fensterausschnitts. Es scheint so, als handele es sich bei den Rechteckkoordinaten `r1` und `r2` um interne Bildschirmkoordinaten und nicht um Koordinaten im Verhältnis zum Fenster. Versuchen Sie, der `DrawAnimatedRects`-Funktion die `Command.hwnd`-Eigenschaft als Parameter zu übergeben. Die Erfahrung hat gezeigt, dass die Funktion den `hwnd`-Parameter zur Auswahl der zu animierenden Beschriftung verwendet, statt die Koordinaten der Parameter `r1` und `r2` zu verwenden.

Ihre Aufgabe ist folgende:

1. Finden Sie heraus, wie Sie die Funktion dazu veranlassen können, die Animation durchzuführen, wenn die `hwnd`-Eigenschaft – wie in der Dokumentation beschrieben – auf 0 gesetzt ist.
2. Führen Sie ein Beispiel an, in dem `IDANI_OPEN` und `IDANI_CLOSE` als Animationsparameter verwendet werden.
3. Erläutern Sie das Verhalten der Funktion im Beispiel anhand der Dokumentation.

Teil II
Die Lösungen

In diesem Teil des Buches finden Sie die Lösungen zu den Puzzlen aus Teil I. Bevor Sie weiterlesen, sollten Sie bedenken, dass der erzielte Lerneffekt direkt proportional zu der Zeit ist, die Sie darauf verwenden, die Puzzle selbst zu lösen.

Wenn Sie also bisher noch keines der Puzzle gelöst haben, schlage ich vor, die folgenden Schritte auszuführen, bevor Sie sich die Lösungen anschauen.

- Lesen Sie die Hinweise in Anhang A.
- Arbeiten Sie die Tutorien in Teil III durch, da Sie mit deren Hilfe die Fähigkeit erlangen können, die Puzzle selbständig zu lösen.

Dies kann ein wenig zeitaufwendiger sein, ist aber umso effektiver.

Lösung 1

Wo steckt denn jetzt der API-Aufruf?

Die Deklaration Erschien vom Aufbau her vielleicht sinnvoll, sie enthielt jedoch zwei Fehler. Obwohl die Funktion als `GetUserDefaultLCID` bezeichnet werden kann, stehen Ländereinstellungen doch in Zusammenhang mit dem Betriebssystem und sind daher Teil der `kernel32`-DLL und nicht, wie in der ursprünglichen Deklaration dargestellt, der `User32`-DLL.

Darüber hinaus muss bei den Win32-Funktionen die Groß- und Kleinschreibung berücksichtigt werden (im Gegensatz zu den 16-Bit-Windows-Funktionen). Bei der Mehrzahl der API-Funktionsnamen, die mehrere Wörter enthalten, wird der erste Buchstabe eines jeweiligen Wortes groß geschrieben, wie dies bei der `GetWindowText`-Funktion der Fall ist. Bei `LCID` handelt es sich jedoch um ein Akronym (`LoCale IDentifier`), daher müssen alle Buchstaben groß geschrieben werden.

Diese Art von Fehlern werden leicht übersehen. Das Gute hierbei ist, dass diese Fehler in Visual Basic nicht nur ermittelt werden können, sondern Sie sogar eine eindeutige Meldung diesbezüglich erhalten. Wenn Sie den Laufzeitfehler 453 empfangen (Angegebene DLL-Funktion nicht gefunden), wird das Problem mit großer Sicherheit durch die Deklaration verursacht, häufig durch einen Fehler im Funktions- oder Bibliotheksnamen.

Durch den nachstehenden Code wird nicht nur das Problem behoben, sondern gleichzeitig die Verwendung der `LCID` zur Ermittlung des englischen Namens der durch diese Ländereinstellung verwendeten Sprache verdeutlicht.

```
' Wo steckt denn jetzt der API-Aufruf?
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten

Option Explicit

Private Declare Function GetUserDefaultLCID Lib "kernel32" () As Long

Private Const LOCALE_SENGLANGUAGE = &H1001&
'  Englischer Name der Sprache
Private Declare Function GetLocaleInfo Lib "kernel32" Alias _
"GetLocaleInfoA" (ByVal Locale As Long, ByVal LCType As Long, _
ByVal lpLCData As String, ByVal cchData As Long) As Long

Private Sub Command1_Click()
   Dim lcid&
```

```
    Dim info$
    lcid = GetUserDefaultLCID()
    info = String$(255, 0)
    Call GetLocaleInfo(lcid, LOCALE_SENGLANGUAGE, info, 255)
    info$ = Left$(info, InStr(info, Chr$(0)) - 1)
    MsgBox lcid & ":" & info$, vbOKOnly, "User Default LCID"
End Sub
```

Lösung 2

Der letzte Fehler

Durch die Funktion `FindWindow` wird entweder eine Fensterzugriffsnummer oder eine Null zurückgegeben. Eine Null wird zurückgegeben, wenn kein Fenster gefunden werden kann oder ein Fehler aufgetreten ist. Woran können Sie nun erkennen, dass ein Fehler aufgetreten ist? Hierzu dient der Wert `LastError`. Die meisten API-Funktionen verfügen über eine interne Konstante, mit der die Bedeutung eines Fehlers angegeben wird. Dies ist ein Wert, der nur für Funktionen gültig ist, die über diese `LastError`-Funktionalität verfügen. Sie können anhand der Dokumentation zu einer Funktion erkennen, ob diese den Wert `LastError` verwendet oder nicht. Die Erläuterungen in der Dokumentation lauten hierbei in etwa wie folgt:

Ausführlichere Fehlerinformationen erhalten Sie durch den Aufruf von `GetLastError`.

Tatsächlich können Sie die API-Funktion `GetLastError` nicht von Visual Basic aus aufrufen, da Visual Basic den internen `LastError`-Wert nach jedem API-Aufruf löscht. Glücklicherweise ist es jedoch möglich, in Visual Basic den `LastError`-Wert für den vorangegangenen API-Aufruf abzurufen, indem Sie auf die `LastDllError`-Eigenschaft des `Err`-Objekts zugreifen.

Wie ermitteln Sie die Bedeutung der `LastError`-Werte?

Dies ist mittels der Datei **api32.txt** oder der Headerdatei **winerror.h** möglich (auf der Begleit-CD-ROM zu diesem Buch). In diesen Dateien können Sie nach den Werten für eine Konstante suchen. Diese Suche ist gar nicht so planlos, wie Sie vielleicht denken. Nachdem Sie die Datei etwa zur Hälfte durchgeblättert haben, finden Sie eine lange Liste mit Konstanten, die das Präfix `ERROR_` aufweisen. Die Liste beginnt folgendermaßen:

```
' The configuration registry database operation completed  successfully.
Public Const ERROR_SUCCESS = 0&

'   Incorrect function.
Public Const ERROR_INVALID_FUNCTION = 1 '  dderror

'   The system cannot find the file specified.
Public Const ERROR_FILE_NOT_FOUND = 2&

'   The system cannot find the path specified.
Public Const ERROR_PATH_NOT_FOUND = 3&
```

Führen Sie einen Bildlauf bis zu Nummer 123 durch:

```
'   The filename, directory name, or volume label syntax is incorrect.
Public Const ERROR_INVALID_NAME = 123&
```

Dies deutet darauf hin, dass einer oder beide Paramter falsch sind, was wiederum zwei interessante Fragen aufwirft:

1. Warum gibt diese Funktion unter Windows NT einen `LastError`-Wert von 123 zurück, was auf einen Parameterfehler hindeutet, und unter Windows 95/98 den Wert 0?
2. Deutet die Tatsache, dass unter Windows NT über `FindWindow` ein `LastError`-Wert zurückgegeben wird, tatsächlich auf ein Parameterproblem hin, oder konnte lediglich kein Fenster ermittelt werden?

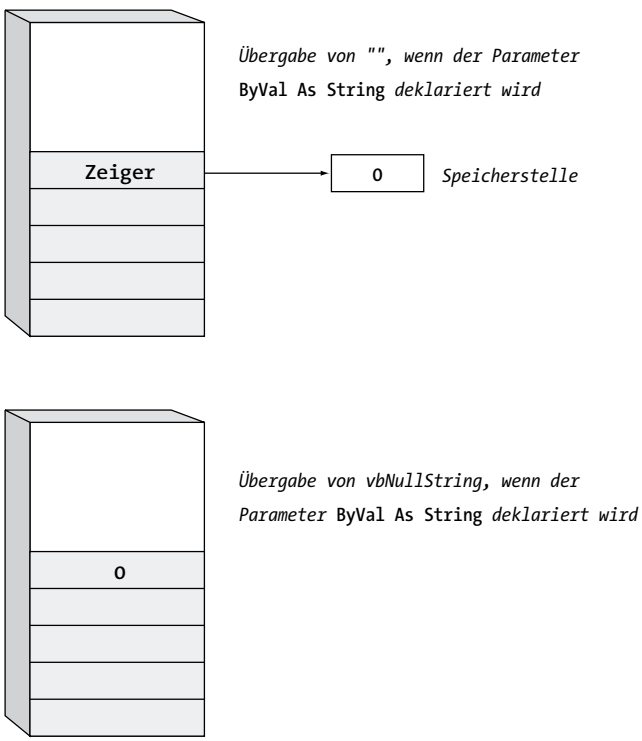


Abbildung S2-1 Übergabe von `vbNullString` und leeren Zeichenfolgen als Parameter

Es erfordert näheres Hinsehen, um herauszufinden, dass es sich tatsächlich um ein Parameterproblem handelt. Der Parameter für den Fenstertitel (Caption) ist richtig, aber Sie sollten sicherstellen, dass der Inhalt des Textfeldes auch einer Fensterbezeichnung entspricht.

Damit bleibt der Klassenname übrig. Aus der Dokumentation für die Funktion geht hervor, dass der Parameter für den Klassennamen ignoriert wird, wenn Sie NULL übergeben.

Aber was genau bedeutet NULL?

NULL bedeutet im Hinblick auf C++ immer 0 – der Wert 0 befindet sich im Stack für diesen Parameter.[1] Im Beispielprogramm wird als Klassenname "" übergeben. Handelt es sich hierbei um NULL?

Nein – "" stellt eine leere Zeichenfolge dar. In C bezeichnet "" einen Zeiger, der auf den abschließenden NULL-Wert einer Zeichenfolge verweist. Daher wird in diesem Fall ein Zeiger als Parameter übergeben, und der Zeiger erscheint im Stack. Dies ist nicht das gleiche wie NULL, dargestellt in Abbildung S2-1.

Glücklicherweise kann in Visual Basic auf einfache Weise ein NULL-Wert übergeben werden, wenn ein Parameter als Zeichenfolge deklariert wurde. Sie verwenden in diesem Fall die Konstante `vbNullString`.

Abbildung S2-1 zeigt den Inhalt des Parameters `lpClassName` im Stack, wenn eine leere Zeichenfolge (»«) und `vbNullString` übergeben werden. Letzteres entspricht dem, was in einer DLL-Funktion als NULL-Wert erwartet wird.

Sobald Sie diese Änderung vornehmen, funktioniert die Funktion einwandfrei. Wenn ein gültiger Fenstertitel angegeben wurde, wird das Fenster geschlossen (probieren Sie dies mit der Anwendung Editor aus). Wenn kein gültiger Fenstertitel angegeben wird, erscheint ein Meldungsfeld, der `LastError`-Wert lautet jedoch 0 – ein Hinweis darauf, dass die Funktion richtig ausgeführt wurde, auch wenn kein Fenster gefunden werden konnte.

Wie sieht die Sache mit einer leeren Zeichenfolge aus? Wenn Sie das Programm unter Windows 95/98 ausführen, wird eine leere Zeichenfolge als gültiger Fensterklassenname erkannt. Kann durch das Programm kein Fenster dieser Klasse gefunden werden, wird der Wert 0 zurückgegeben und der `LastError`-Wert wird auf 0 gesetzt, um anzugeben, dass kein Fenster ermittelt werden konnte.

1. Sie finden weitere Informationen zu Stacks und Stackframes in Tutorium 4 in Teil III dieses Buches.

Wenn Sie das Programm unter Windows NT ausführen, wird die leere Zeichenfolge als ungültiger Klassenname interpretiert, und der `LastError`-Wert wird auf 123 gesetzt, um anzuzeigen, dass es sich um einen ungültigen Parameter handelt.[2]

Eine weitere Möglichkeit, die »LastError«-Beschreibung zu erhalten

Eine andere Methode, die Beschreibung einer Fehlermeldung zu ermitteln, ist mit der API-Funktion `FormatMessage` gegebenstellt. Diese Funktion unterstützt eine Vielzahl von Optionen, die im Rahmen des vorliegenden Buches nicht besprochen werden können. Wir beschränken uns daher auf die Funktionalität, mit der Systemmeldungen abgerufen werden können. Die `FormatMessage`-Funktion wird wie folgt definiert:

```
Private Declare Function FormatMessage Lib "kernel32" Alias _
"FormatMessageA" (ByVal dwFlags As Long, lpSource As Any, ByVal _
dwMessageId As Long, ByVal dwLanguageId As Long, ByVal lpBuffer _
As String, ByVal nSize As Long, ByVal Arguments As Long) As Long

Private Const FORMAT_MESSAGE_FROM_SYSTEM = &H1000
```

Der `dwFlags`-Parameter ist auf die Konstante `FORMAT_MESSAGE_FROM_SYSTEM` eingestellt. Der `lpSource`-Parameter für diesen Modus muss NULL lauten. Da die Deklaration `As Any` lautet, müssen Sie darauf achten, dass Sie einen `Long`-Parameter `By Val` übergeben.[3] Der Parameter `dwMessageId` stellt den `LastError`-Wert dar. Setzen Sie den `dwLanguageId`-Parameter auf 0, um die Standardsystemsprache zu verwenden. Bei dem `lpBuffer`-Parameter handelt es sich um eine initialisierte Zeichenfolge, deren maximale Länge dem `nSize`-Parameter entspricht. Der `Arguments`-Parameter ist ebenfalls NULL.

```
' Systemfehleränderung abrufen
Private Function GetErrorString(ByVal LastErrorValue As Long) As String
   Dim bytes&
   Dim s As String
   s = String$(129, 0)
   bytes = FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM, ByVal _
   0&, LastErrorValue, 0, s$, 128, 0)
   If bytes > 0 Then
```

2. Wird der leere Zeichenfolgenparameter aufgrund von grundlegenden Unterschieden zwischen den Betriebssystemen Windows 95/98 und Windows NT unterschiedlich behandelt? Wahrscheinlich nicht. Oder hat sich der NT-Programmierer, der diese Funktion implementiert hat, nicht mit dem Windows 95-Programmierer abgesprochen? Wahrscheinlich. Und nein, dieser Unterschied wird nicht dokumentiert – ein weiteres Beispiel dafür, dass Sie Ihre Programme stets sowohl unter Windows 95/98 als auch unter Windows NT testen sollten.
3. Weiterführende Informationen zur Verwendung des `By Val`-Schlüsselwortes mit `As Any` deklarierten Parametern finden Sie in Tutorium 5.

```
      GetErrorString = Left$(s, bytes)
   End If
End Function
```

Die `FormatMessage`-Funktion gibt die Anzahl der Zeichen in der Zeichenfolge wieder, daher können Sie unter Verwendung der `Left$`-Funktion auf einfache Weise die gültigen Zeichen bis zum abschließenden NULL-Wert abrufen.

Lösung 3

Möchte Poly einen Keks?

Wo sollen wir anfangen?

Wenden wir uns zunächst der API-Deklaration zu. Der `hdc`-Parameter ist eine Zugriffsnummer, also ein 32-Bit-Wert, mit dem ein Objekt identifiziert wird. In diesem Fall handelt es sich um eine Zugriffsnummer für einen Gerätekontext, nämlich ein Objekt, das als Schnittstelle zwischen einem physischen Gerät (beispielsweise einem Fenster auf dem Bildschirm oder eine gedruckte Seite) und dem Subsystem für grafische Darstellung fungiert. Da Sie den Wert der Zugriffsnummer als Parameter übergeben möchten, müssen Sie für den Parameter den `By Val`-Operator angeben. Das gleiche gilt für den `nCount`-Parameter. Die richtige Deklaration lautet daher folgendermaßen:

```
Private Declare Function Polygon Lib "gdi32" (ByVal hdc As Long, _
lpPoint As POINTAPI, ByVal nCount As Long) As Long
```

Diese Änderung führt (Trommelwirbel, bitte) ... zu einem weiteren leeren Bildschirm.

Der nächste Schritt besteht darin, die `Polygon`-Funktion nachzuvollziehen und sich die Ergebnisse anzusehen. Diese Aufgabe ist einfacher zu lösen, wenn Sie den Polygon-Aufruf so ändern, dass Sie das folgende Funktionsformat verwenden:

```
Result = Polygon(hWnd, points(0), UBound(points) + 1)
```

Das Ergebnis lautet 0, d.h., die Funktion ist fehlgeschlagen. Wenn Sie sich mit Hilfe des Wertes der `Err.LastDllError`-Eigenschaft den letzten Fehlerwert anzeigen lassen, erhalten Sie als Ergebnis den Wert 6. Den entsprechenden Fehlerwert können Sie mit Hilfe der in Lösung 2 verwendeten `FormatMessage`-Funktion ermitteln, oder Sie schauen sich die entsprechende `ERROR_`-Konstante in Datei **api32.txt** oder **winerror.h** an (auf der Begleit-CD-ROM):

```
'   The handle is invalid
Public Const ERROR_INVALID_HANDLE = 6&
```

Sehen Sie sich den ersten Parameter, also die Zugriffsnummer, noch einmal genau an. In der Funktionsdokumentation wird dieser Parameter als Zugriffsnummer für einen Gerätekontext definiert. Im Code wird jedoch eine Zugriffsnummer auf ein Fenster verwendet. Fenster- und Gerätekontext sind zwei grundsätzlich verschiedene Dinge.

```
Call Polygon(hWnd, points(0), UBound(points) + 1)
```

Sie können mit Hilfe von API-Funktion den Gerätekontext für ein Fenster ermitteln, es ist jedoch viel einfacher, hierzu die `hdc`-Eigenschaft eines Formulars oder eines Bildsteuerelements einzusetzen.

```
Call Polygon(hdc, points(0), UBound(points) + 1)
```

Nehmen Sie keinerlei Änderungen an der Konfiguration eines Gerätekontextes vor, den Sie auf diese Weise erhalten – ansonsten könnte es zu Störungen mit dem Visual Basic-eigenen System kommen. Verwenden Sie entweder die API-Funktionen `SaveDC` und `RestoreDC`, um die aktuelle Konfiguration des Gerätekontextes zu speichern und wiederherzustellen, oder erstellen Sie mit Hilfe der Funktion `CreateCompatibleDC` einen separaten Gerätekontext. In Abbildung S3-1 sind die Ergebnisse nach dieser Änderung zu sehen.

Offensichtlich ist immer noch etwas nicht in Ordnung. Sie könnten sich den Algorithmus der Funktion `LoadPointArray` näher ansehen, aber dies würde pure Zeitverschwendung bedeuten. Die Vermutung liegt zwar nahe, aber bei dem vorliegenden Buch handelt es sich schließlich um ein API-Puzzlebuch. Das Einfügen eines üblichen Visual Basic-Bugs wäre unfair, ja irreführend, und ich würde es niemals wagen, an soetwas auch nur zu denken – jedenfalls nicht bei den ersten Puzzlen.

Vergleichen Sie Abbildung S3-1 mit Abbildung P3-1. Es scheint fast so, als handele es sich bei Abbildung S3-1 um ein stark vergrößertes Segment des oberen linken Bildausschnitts von Abbildung P3-1. Könnte es sich etwa um ein Skalierungsproblem handeln?

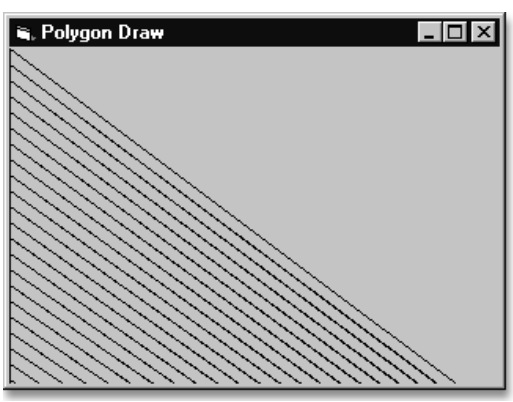


Abbildung S3-1 Zweite Version des Poly-Programms

Ganz genau. Die Punkte werden basierend auf den Höhen- und Breiteneigenschaften des Formulars generiert. Diese Eigenschaften werden in Twips angegeben – 1/1440 eines Zolls. API-Funktionen verwenden das aktuelle logische Koordi-

natensystem, das üblicherweise Pixel verwendet. Wie erfolgt eine Umrechnung von Twips in Pixel? Das `Screen`-Objekt verfügt über eine Eigenschaft mit dem Namen `TwipsPerPixelX` und `TwipsPerPixelY`, die dazu verwendet werden kann, die Anzahl der Twips pro Pixel anzugeben. Dividieren Sie die Höhe und Breite wie im folgenden Code in den entsprechenden `TwipsPerPixel`-Wert um, und das Ergebnis ist sehr viel überzeugender, wie in Abbildung S3-2 zu sehen ist.

```
LoadPointArray Width / Screen.TwipsPerPixelX, Height / _
Screen.TwipsPerPixelY, 5, points()
```

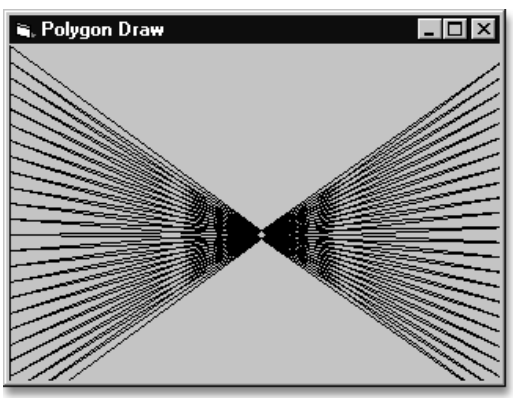


Abbildung S3-2 Eine weitere Version des Poly-Programms

Dennoch scheint das Bild immer noch nicht ganz richtig angezeigt zu werden. Im unteren Bildbereich sind die Vektoren etwas von der Grundlinie entfernt. Es sieht fast so aus, als basierten die Berechnungen nicht auf den tatsächlichen Dimensionen des Fensters. Wie sich herausstellt, liegt dies sehr wahrscheinlich daran, dass die Berechnungen nicht auf den tatsächlichen Dimensionen des Fensters basieren. Die Eigenschaften im Hinblick auf Höhe und Breite eines Formulars spiegeln die Dimensionen des gesamten Fensters wider. Das Bild wird jedoch nur im Clientbereich des Formulars angezeigt (nicht im Rahmen- und Titelbereich). Der Clientbereich eines Fensters kann mit Hilfe der API-Funktion `GetClientRect` oder mit Hilfe der Eigenschaften `ScaleWidth` und `ScaleHeight` des Formulars ermittelt werden. Die letztgenannte Methode wurde hier verwendet:

```
LoadPointArray ScaleWidth / Screen.TwipsPerPixelX, ScaleHeight / _
Screen.TwipsPerPixelY, 5, points()
```

Mit dieser Änderung funktioniert das Beispielprogramm einwandfrei.

Eine letzte Frage: Vielleicht ist Ihnen die vertikale Linie auf der linken Seite der Zeichnung aufgefallen. Können Sie sich erklären, was es mit dieser Linie auf sich hat?

Lösung 4

Nomen est Omen

Wie kann ein einzelner Codeabschnitt solche Probleme hervorrufen?

```
Private Declare Function GetComputerName Lib "kernel32" (ByVal _
ComputerName As String, ByVal BufferSize As Long) As Long

Private Const MAX_COMPUTERNAME_LENGTH = 15

Private Sub Form_Load()
   Dim s$
   Call GetComputerName(s$, MAX_COMPUTERNAME_LENGTH + 1)
   lblName.Caption = s$
End Sub
```

Der Ansatz bei der Lösung eines derartigen Problems ist immer gleich, unabhängig von der Komplexität eines Programms. Sie probieren etwas aus, interpretieren die auftretenden Fehler, und bearbeiten den Code zur Behebung des Fehlers. Anschließend versuchen Sie, den nächsten Fehler zu beheben.

Der DLL-Einsprungpunkt

Wenn Sie dieses Programm ausführen, erhalten Sie Fehler 453: Angegebene DLL-Funktion nicht gefunden `GetComputerName` in `kernel32`.

Wenn Sie sich bereits Tutorium 1, »Auffinden von Funktionen« (in Teil III dieses Buches) angesehen haben, wissen Sie, dass Sie mit dieser Fehlermeldung darauf hingewiesen werden, dass die angeforderte API-Funktion nicht in der angegebenen DLL vorliegt. Als Erstes würden Sie vielleicht denken, dass die Funktion `GetComputerName` in einer anderen DLL sich befindet. Doch bevor Sie mit der Suche beginnen, sollten Sie beachten, dass diese Funktion einen Zeichenfolgenparameter aufweist. Die meisten Funktionen mit Zeichenfolgenparameter verfügen über zwei Einsprungpunkte, den ANSI-Einsprungpunkt für Einbyte-Zeichen und den Unicode-Einsprungpunkt für Doppelbyte-Zeichen.

Versuchen Sie es mit folgender Deklaration:

```
Private Declare Function GetComputerName Lib "kernel32" Alias _
"GetComputerNameA" (ByVal ComputerName As String, ByVal BufferSize _
As Long) As Long
```

Auf diese Weise wird das Problem behoben. Laufzeitfehler 453 wird nicht mehr angezeigt, sobald die Funktion gefunden wurde.

Wenn Sie aber versuchen, das Programm unter Windows NT auszuführen, stürzt es sofort mit einem Speicherausnahmefehler ab.[4]

Initialisieren Sie die Zeichenfolge!

Wie Sie dem Beispielprogramm entnehmen können, wird die Funktion mit einer Puffergröße von `MAX_COMPUTERNAME_LENGTH` Zeichen aufgerufen. Woher stammt diese Zahl? Sie wurde der Dokumentation der Funktion entnommen:

Windows 95 und Windows 98: `GetComputerName` schlägt fehl, wenn die Eingabegröße kleiner als `MAX_COMPUTERNAME_LENGTH` + 1 ist.

Der Zeichenfolgenpuffer wird also mit bis zu 15 Zeichen (Wert von `MAX_COMPUTERNAME_LENGTH`-Puffer) plus dem abschließenden NULL-Wert geladen. Wenn Sie der API-Funktion die Variable `v$` übergeben, übergibt Visual Basic einen Zeiger auf einen auf NULL endenden Zeichenfolgenpuffer an die Funktion. Wie lang ist der Puffer während des Initialisierungsaufrufs? Ein Byte – der Puffer umfasst nur den abschließenden NULL-Wert. Die anfängliche Zeichenfolge ist leer.

Der API-Aufruf weiß jedoch nicht, dass der Puffer nur 1 Byte groß ist – Sie hatten ja ausdrücklich angegeben, dass der Puffer 16 Byte lang ist. Also lädt die API-Funktion fröhlich den Computernamen in den Puffer. Hierbei wird Speicher überschrieben, der anderweitig verwendet wird, und daher stürzt der Anwendungsspeicher entweder ab oder wird in irgendeiner Form beschädigt.

Es ist von größter Bedeutung, Zeichenfolgen vor deren Verwendung mit der benötigten Länge zu initialisieren.

Die folgende Zeile wird vor dem Aufruf der `GetComputerName`-Funktion eingefügt:

```
s$ = String$(MAX_COMPUTERNAME_LENGTH + 1, 0)
```

Problem gelöst? Nicht wirklich – das Programm stürzt immer noch mit einem Speicherausnahmefehler ab!

Was für eine Pufferlänge?

Sie wissen, dass die richtige Funktion verwendet (und gefunden) wird. Sie wissen auch, dass die Zeichenfolge richtig initialisiert wurde. Damit bleibt nur die Variable `BufferSize`. Was könnte einfacher sein, als die Größe eines Puffers als Parameter zu übergeben?

4. Windows NT eignet sich relativ gut zur Ermittlung unbefugter Zugriffsversuche auf ungültige Speicherstellen. Als Ergebnis wird in diesen Fällen eine Speicherausnahme ausgelöst. Unter Windows 95/98 wird diese Art von Problem nicht unbedingt erkannt, deshalb kommt es zu einer leichten Speicherbeschädigung, auf die Sie nicht hingewiesen werden. Später stürzt das System dann auf spektakuläre Weise ohne jeglichen Bezug zur ursprünglichen Fehlerquelle ab.

Der nSize-Parameter wird folgendermaßen deklariert.

```
LPDWORD nSize     // Adresse für die Größe des Namenspuffers
```

In Tutorium 6, »C++ Variablen treffen auf Visual Basic«, in Teil III dieses Buches, wird das Übertragen von C++-Variablen in Visual Basic erläutert. Bei DWORD handelt es sich um eine 32-Bit-Variable. Das Präfix LP deutet darauf hin, dass es sich bei dem nSize-Parameter tatsächlich um einen Zeiger auf die Puffergröße handelt, nicht jedoch um die Größe selbst. Im Moment wird der BufferSize-Parameter (unser Name für nSize) von der Funktion als By Val übergeben. Das bedeutet, dass wir die Größe des Puffers, nicht aber einen Zeiger auf die Größe übergeben. Die Lösung besteht darin, eine Long-Variable zur Speicherung der Größe zu erstellen und einen Zeiger auf diese Größe zu übergeben. Dies erreichen Sie ganz einfach, indem Sie das By Val-Kennwort entfernen und den Parameter als Verweis übergeben.

So gelangen Sie zu folgendem Code:

```
' Computer Name
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten

Option Explicit

Private Declare Function GetComputerName Lib "kernel32" Alias _
"GetComputerNameA" (ByVal ComputerName As String, BufferSize As Long) _
As Long

Private Const MAX_COMPUTERNAME_LENGTH = 15

Private Sub Form_Load()
   Dim s$
   Dim BufferLength As Long
   s$ = String$(MAX_COMPUTERNAME_LENGTH + 1, 0)
   BufferLength = MAX_COMPUTERNAME_LENGTH + 1
   Call GetComputerName(s$, BufferLength)
   lblName.Caption = s$
End Sub
```

Das fehlende Mosaiksteinchen

Was wird tatsächlich in den Variablenpuffer s$ geladen, wenn die Funktion ausgeführt wird? Eine Zeichenfolge, die auf NULL endet. Als VB-Programmierer möchten Sie weder einen abschließenden NULL-Wert noch irgendwelche anderen Zeichen erhalten, die anschließend folgen. Können Sie einen Code bereitstellen, mit dem der NULL-Wert und beliebige weitere Daten abgeschnitten werden?

In Puzzle 1, »Wo steckt denn jetzt der API-Aufruf?«, wurde bereits ein Versuch unternommen, den NULL-Wert einer Zeichenfolge abzuschneiden. Hierbei wurde die `Instr()`-Funktion zum Auffinden des abschließenden NULL-Wertes verwendet, die `Left$()`-Funktion wurde zum Auffinden des Teiles der Zeichenfolge eingesetzt, der sich vor diesem Zeichen befand. Vielleicht ist Ihnen schon der Gedanke gekommen, dass es einfacher wäre, die `Left$()`-Funktion zur Lokalisierung der Zeichenfolge zu verwenden, wenn Ihnen über die API-Funktion mitgeteilt würde, wie viele Zeichen tatsächlich in den Puffer geladen wurden.

Da auch dieses Puzzle wieder in einer Frage zu enden scheint, wenden wir uns schnell einer neuen Frage zu: Warum hat sich Microsoft bei dieser Funktion entschieden, den `BufferSize`-Parameter als Verweis zu übergeben? Auf diese Weise kann durch die Funktion der Wert der `BufferLength`-Variablen durch den Wert geändert werden, den Sie während des Aufrufs angeben.[5]

Wie sieht es mit dem Ergebnis aus?

Sie haben vielleicht bemerkt, dass die Funktion `GetComputerName` in diesem Beispiel mit Hilfe des Subroutinenformats aufgerufen wurde, bei dem der Rückgabewert ignoriert wird:

```
Call GetComputerName(s$, BufferLength)
```

API-Funktionen geben stets Rückgabewerte zurück, aus denen ersichtlich wird, ob die Funktion erfolgreich war oder fehlgeschlagen ist. Sie könnten darüber nachdenken, die Ergebnisse von API-Funktionen zu prüfen und Code zu schreiben, mit denen auftretende Fehler behandelt werden. Der Vorteil bei diesem Ansatz liegt darin, dass Sie so eine sehr stabile Anwendung erhalten. Der Nachteil dieser Vorgehensweise liegt in dem zusätzlichen Zeitaufwand, den Sie zum Schreiben von Code für alle Fehler benötigen, die möglicherweise auftreten könnten.

5. Am besten sehen Sie sich den Wert der `BufferLength`-Variable nach dem Funktionsaufruf an.

Ich wäre der letzte, der Sie davon abhielte, Ihre Anwendung mit einer guten Fehlerbehandlung zu versehen. Dennoch gibt es bei der Arbeit mit der Win32-API Fälle, in denen Sie vom Standpunkt der Entwicklung her fehlerfreien Code geschrieben haben und bei dem eine zusätzliche Fehlerbehandlung reine Zeitverschwendung wäre. Dies ist einer dieser Fälle.

Wie Sie in diesem Beispiel sehen konnten, kann die `GetComputerName`-Funktion fehlschlagen, wenn Sie einen ungültigen Parameter übergeben. Nachdem Sie jedoch Ihren Code dahingehend berichtigt haben, dass Sie der Funktion stets eine vorinitialisierte Zeichenfolge mit dem richtigen `BufferLength`-Parameter übergeben, dann sollte die Funktion einwandfrei ausgeführt werden. Aus diesem Grund habe ich in diesem Beispiel den kürzeren Ansatz unter Verwendung eines Subroutinenaufrufs vorgezogen.

Lösung 5

Finden Sie den Namen des ausführenden Programms!

Ist es Ihnen schwer gefallen, den API-Fehler zu finden? Ich hoffe ja, denn die Deklaration ist perfekt. Tatsächlich ist auch der Funktionsaufruf völlig in Ordnung. Das Problem liegt nicht im API-Aufruf, sondern in unserem Umgang mit dem Ergebnis.

Sehen wir uns den Code Schritt für Schritt an.

Zunächst wird die Zeichenfolge auf 261 Byte initialisiert:

```
ExecName = String$(MAX_PATH + 1, 0)
```

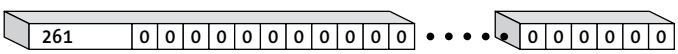


Abbildung S5-1 Anfänglicher Inhalt der `ExecName`-Zeichenfolge

Die `ExecName`-Zeichenfolge enthält nun 261 Zeichen, jedes mit einem 0-Wert. Diese werden intern im `BSTR`-Format gespeichert, d. h., Visual Basic speichert die Länge der Zeichenfolge, gefolgt von den Daten. Dieser Vorgang wird in Abbildung S5-1 veranschaulicht. Beachten Sie, dass die Zahl 0 auf ein Zeichen mit dem ASCII-Wert 0 hinweist, nicht, dass es das Zeichen »0« enthält.

Als Nächstes laden wird mit Hilfe der API-Funktion `GetModule FileName` den Dateinamen:

```
Call GetModuleFileName(0, ExecName, MAX_PATH)
```

Auf diese Weise wird die Zeichenfolge mit dem vollständigen Pfadnamen in die Anwendung geladen. Nehmen wird für dieses Beispiel an, dass das Programm in der Visual Basic-Entwicklungsumgebung ausgeführt wird, und dass sich das Programm **vb6.exe** im Stammverzeichnis von Laufwerk C: befindet. Die Zeichenfolge enthält nun die in Abbildung S5-2 gezeigten Daten.

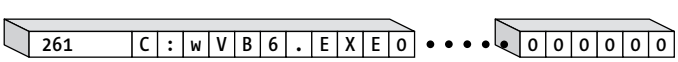


Abbildung S5-2 Inhalt der Zeichenfolge `ExecName`, nachdem diese über die Funktion `GetModuleFileName` geladen wurde

Im nächsten Schritt entfernen wir den Pfad, da wir nur den Namen der ausführbaren Datei benötigen, um zu ermitteln, ob es sich um das Programm Visual Basic handelt. Dies wird erreicht, indem – am Ende der Zeichenfolge beginnend – die Zeichenfolge rückwärts nach dem Backslash (\ ) durchsucht wird. In diesem Bei-

spiel befindet sich der Backslash an Position 3. Die Zeichenfolge wird anschließend mit Hilfe der Funktion `Mid` neu zugeordnet, um die Daten, die sich an den Backslash anschließen, abzurufen.

```
' Jetzt den Pfad abschneiden
LastBackslashPos = InStrRev(ExecName, "\ ")
If LastBackslashPos = 0 Then
   LastBackslashPos = InStrRev(ExecName, ":")
End If
ExecName = Mid$(ExecName, LastBackslashPos + 1)
```

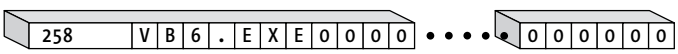


Abbildung S5-3 Inhalt der `ExecName`-Zeichenfolge nach dem Entfernen des Pfades

Die Ergebniszeichenfolge wird in Abbildung S5-3 dargestellt.

Wie lang ist die Zeichenfolge nach dieser Operation? Dies hängt von Ihrer Sichtweise ab. Würde die Zeichenfolge an eine API-Funktion übergeben werden, würde es sich um eine Zeichenfolge mit 7 Zeichen handeln, die auf einen NULL-Wert endet. Visual Basic sieht dies jedoch anders. In Visual Basic wird eine 258 Zeichen umfassende Zeichenfolge erkannt, die sich aus dem Dateinamen **vb6.exe** und weiteren 251 NULL-Zeichen zusammensetzt. Was geschieht, wenn Sie den folgenden Code ausführen?

```
If LCase$(ExecName) = "vb6.exe" Or _
      LCase$(ExecName) = "vb5.exe" Or _
      LCase$(ExecName) = "vb32.exe" Then
      IsThisVB = True
   End If
```

Im Vergleich mit **vb6.exe** unternimmt Visual Basic den Versuch, eine 258 Zeichen umfassende Zeichenfolge mit einer Zeichenfolge zu vergleichen, die 7 Zeichen enthält. Verständlicherweise wird hierbei keine Übereinstimmung ermittelt. Sie müssen demnach sämtliche der zusätzlichen NULL-Werte abschneiden. Eine der Möglichkeiten, dies zu erreichen, wird nachfolgend gezeigt:

```
NullPosition = InStr(ExecName, Chr$(0))
ExecName = Left$(ExecName, NullPosition - 1)
```

Wie viele API-Funktionen dieses Typs gibt die `GetModuleFileName`-Funktion die Anzahl der Zeichen zurück, die in die Zeichenfolge geladen wurden. In diesem Beispiel würde der Rückgabewert 10 lauten – die Länge des vollständigen Pfades und der Name der ausführbaren Datei. Wir könnten diesen Wert dazu verwenden, den NULL-Wert abzuschneiden, bevor wir den Pfad entfernen. Beide Ansätze führen gleichermaßen zum gewünschten Ergebnis.

Ich kann Ihnen nur empfehlen, bei der Lösung Ihrer eigenen API-Probleme dem hier angeführten Beispiel zu folgen. Sie können viele API-bezogene Probleme lösen, indem Sie sich die Inhalte einer Variablen und deren Struktur im Speicher genauer ansehen.

Lösung 6

Wo bleibt das Icon?

Der erste Schritt zur Lösung dieses Puzzles besteht zunächst aus einer Analyse des Codes, und zwar Zeile für Zeile. Die Funktion `LoadIcon` gibt den Wert 0 zurück, was darauf hinweist, dass das Icon nicht geladen werden konnte. Wenn Sie unter Windows NT arbeiten, wird über die Funktion `GetErrorString` im Modul `ErrString.bas` eine Beschreibung abgerufen, die auf dem LastError-Ergebnis basiert, das wiederum durch die Eigenschaft `Err.LastDllError` abgerufen wird.

Warum funktioniert `LoadIcon` nicht?

Lassen Sie uns zur Eingrenzung des Problems einen Teil des C-Headercodes durchgehen und genau untersuchen, wie die Iconkonstanten definiert sind: Beginnen wir mit den Konstantendefinitionen:

```
#ifdef RC_INVOKED
#define IDI_APPLICATION           32512
#define IDI_HAND                  32513
#define IDI_QUESTION              32514
#define IDI_EXCLAMATION           32515
#define IDI_ASTERISK              32516
#if(WINVER >= 0x0400)
#define IDI_WINLOGO               32517
#endif /* WINVER >= 0x0400 */
#else
#define IDI_APPLICATION           MAKEINTRESOURCE(32512)
#define IDI_HAND                  MAKEINTRESOURCE(32513)
#define IDI_QUESTION              MAKEINTRESOURCE(32514)
#define IDI_EXCLAMATION           MAKEINTRESOURCE(32515)
#define IDI_ASTERISK              MAKEINTRESOURCE(32516)
#if(WINVER >= 0x0400)
#define IDI_WINLOGO               MAKEINTRESOURCE(32517)
#endif /* WINVER >= 0x0400 */
#endif /* RC_INVOKED */
```

Die Konstante `#ifdef RC_INVOKED` erhält den Wert `true`, wenn die Headerdatei durch einen Ressourcencompiler ausgelöst wird – einem speziellen Compiler, der verschiedene Ressourcentypen, beispielsweise Icons und Bitmapdateien, in einem bestimmten Format miteinander kombiniert, das mit einer ausführbaren Datei verknüpft werden kann. Wir sind vor allem daran interessiert, wie die Konstanten vom C-Compiler selbst aufgerufen werden, deshalb lassen wir alles innerhalb von `RC_INVOKED` außer Acht. Dies führt uns zu folgenden Definitionen:

```
#define IDI_APPLICATION           MAKEINTRESOURCE(32512)
#define IDI_HAND                  MAKEINTRESOURCE(32513)
#define IDI_QUESTION              MAKEINTRESOURCE(32514)
#define IDI_EXCLAMATION           MAKEINTRESOURCE(32515)
#define IDI_ASTERISK              MAKEINTRESOURCE(32516)
#if(WINVER >= 0x0400)
#define IDI_WINLOGO               MAKEINTRESOURCE(32517)
#endif /* WINVER >= 0x0400 */
```

Die IDI_WINLOGO-Konstante beschreibt eine Konstante, die nur unter Windows 95 und höher sowie unter Windows NT und höher verwendet werden kann, demnach kann sie eingeschlossen werden.

Worauf bezieht sich MAKEINTRESOURCE? Bei MAKEINTRESOURCE handelt es sich um ein Makro für die Vorkompilierung. Ein Makro gleicht einer Funktion, jedoch mit dem Unterschied, dass ein Makro während der Kompilierung von einem Compiler verarbeitet wird. Das Ergebnis eines Makros ist kompilierter Text. Jetzt müssen wir uns das Makro MAKEINTRESOURCE nur noch ein wenig genauer ansehen:

```
#define MAKEINTRESOURCEA(i) (LPSTR)((DWORD)((WORD)(i)))
#define MAKEINTRESOURCEW(i) (LPWSTR)((DWORD)((WORD)(i)))
#ifdef UNICODE
#define MAKEINTRESOURCE           MAKEINTRESOURCEW
#else
#define MAKEINTRESOURCE           MAKEINTRESOURCEA
#endif // !UNICODE
```

Wir erstellen ein ANSI-Programm, daher ist die UNICODE-Bedingung nicht relevant. Die Makrodefinition für MAKEINTRESOURCE reduziert sich demnach auf Folgendes:

```
#define MAKEINTRESOURCEA(i) (LPSTR)((DWORD)((WORD)(i)))
#define MAKEINTRESOURCE  MAKEINTRESOURCEA
```

Was bedeutet das?

Berücksichtigen Sie, was geschieht, wenn Sie IDI_WINLOGO in Ihr Programm einschließen.

In diesem Fall wird IDI_WINLOGO durch MAKEINTRESOURCE(32517) ersetzt.

MAKEINTRESOURCE(32517) wiederum wird ersetzt durch MAKEINTRESOURCEA (32517).

Ein erneutes Ersetzen dieses Ausdrucks führt schließlich zu `(LPSTR)((DWORD)((WORD)32517))`, das vom Compiler kompiliert wird.

Dieser Ausdruck kann ebenfalls zu Verwirrung führen. Betrachten Sie zunächst `((WORD)32517)`. Welche Bedeutung hat dieser Ausdruck?

Wenn ein C-Header über einen durch Klammern umschlossenen Variablentyp verfügt, bedeutet dies, dass die folgende Variable den angegebenen Typ annehmen muss. Die Variablen werden hierbei nicht von einem Typ in einen anderen konvertiert, sondern der Compiler behandelt die Dateien einfach so, als gehörten Sie dem neuen Typ an. Daher bedeutet `((WORD)32517)`, dass der Wert 32517 wie eine Ganzzahl ohne Vorzeichen behandelt wird.

Das ganze Makro kann folgendermaßen interpretiert werden:

1. Nimm den ganzzahligen Wert »i« und behandle ihn als 16-Bit-Zahlenwert ohne Vorzeichen (unsigned integer).
2. Nimm diesen Zahlenwert und behandle ihn als 32-Bit-Zahlenwert (Festlegen der hohen 16 Bits auf 0).
3. Nimm den 32-Bit-Zahlenwert ohne Vorzeichen und behandle ihn als einen Zeiger auf eine Zeichenfolge (obwohl es sich nicht um einen Zeiger handelt).

Sie fragen sich vielleicht, wie Sie einen Zahlenwert, bei dem es sich nicht um einen Zeiger handelt, an eine API-Funktion übergeben können, die einen Zeichenfolgenparameter erwartet? Die Antwort liegt in der etwas undurchsichtigen Beschreibung des `lpIconName`-Parameters, die Sie bereits kennen:

Zeigt auf eine in NULL endende Zeichenfolge, die den Namen der zu ladenden Iconressource enthält. Alternativ kann dieser Parameter den Ressourcenbezeichner im Teil niederer Ordnung enthalten und 0 im Teil hoher Ordnung. Verwenden Sie zum Erstellen dieses Wertes das Makro `MAKEINTRESOURCE`.

Wie Sie sehen, ist der `lpIconName`-Parameter so konzipiert, dass zwei Parametertypen verwendet werden können. Wenn die `LoadIcon`-Funktion ermittelt, dass die hohen 16 Bits den Wert 0 aufweisen, setzt die Funktion voraus, dass Sie einen numerischen Bezeichner für eine Iconressource übergeben. Wenn die hohen 16 Bits Daten enthalten, wird angenommen, dass Sie einen Zeiger auf eine auf NULL endende Zeichenfolge übergeben. In Abbildung S6-1 wird dargestellt, welcher Wert an die Funktion übergeben wird, wenn der Konstantenwert `IDI_ASTERISK (32516)` lautet. Wie Sie sehen können, wird der Wert nicht geändert, sondern direkt an die `LoadIcon`-Funktion übergeben, obwohl der Compiler angewiesen wurde, den Wert der Konstanten als Zeiger auf eine Zeichenfolge zu interpretieren.

Wie sieht es mit Visual Basic aus?

Sie wissen jetzt, welcher Übergabewert von der API-Funktion erwartet wird. Welcher Wert wird jedoch tatsächlich an die Funktion übergeben?

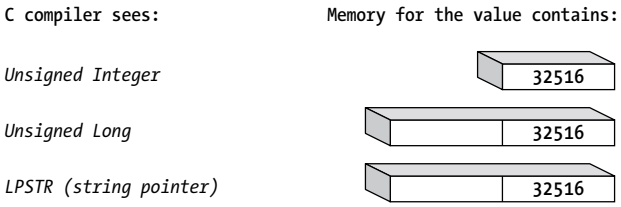


Abbildung S6-1 Übergabe der `IDI_ASTERISK`-Ressourcen-ID an die `LoadIcon`-Funktion

Der Parameter `lpIconParam` wird `ByVal As String` deklariert. Dies bedeutet, dass die Funktion immer als auf NULL endende Zeichenfolge übergeben wird. Was geschieht, wenn Sie einem als `ByVal As String` deklarierten Parameter eine numerische Konstante zuweisen? Visual Basic konvertiert die Konstante vor der Übergabe an die Funktion in eine Zeichenfolge. Dieses Verfahren unterscheidet sich vom `cast`-Operator in C, bei dem die Daten nicht verändert, sondern gemäß dem neuen Typ interpretiert werden. Visual Basic führt eine tatsächliche Konvertierung durch. Im vorliegenden Fall wird die `IDI_ASTERISK`-Konstante in die Zeichenfolge »32516« konvertiert und mit einem abschließenden NULL-Wert an die Funktion übergeben. Dieser Vorgang wird in Abbildung S6-2 veranschaulicht.

Da dieser Wert von der Funktion nicht erwartet wird, kommt es zu einem Fehler.

Warum wird von Visual Basic kein Fehler ausgegeben, wenn Sie versuchen, einen Zahlenwert an einen Zeichenfolgenparameter zu übergeben? Die Mehrzahl der Computerexperten stellt sich ebenfalls diese Frage – es handelt sich hierbei um einen Bereich, in dem viele Programmierer dem Ansatz von Microsoft grundsätzlich nicht zustimmen können. Microsoft hat diese automatische Typenkonvertierung an mehreren Stellen integriert, um dem Programmierer die Arbeit mit Visual Basic zu erleichtern. Unglücklicherweise kann genau aus diesem Grund über den Compiler eine Vielzahl von Fehlerarten nicht ermittelt werden. Visual Basic konvertiert fröhlich Parameter, ohne auf diese Tatsache hinzuweisen, und beschert dem Programmierer auf diese Weise einige schwer zu ermittelnde Bugs. Wir nennen dies »tückische Typenkonvertierung«, und viele Programmer (mich selbst eingeschlossen) halten dies für einen der größten Fehler von Visual Basic, einer Sprache, die sich ansonsten superb zur Programmierung eignet.

Wie sollten Sie in einer solchen Situation vorgehen? Sie könnten den `lpIcon`-Paramter einfach als `ByVal As Long` deklarieren, aber in diesem Fall kann die Funktion keine Zeichenfolgenparameter verwenden. Daher deklarieren Sie den Para-

meter als `As Any` oder `ByVal As Any`, damit die Funktion entweder mit einem Zeichenfolgen- oder mit einem numerischen Parameter aufgerufen werden kann.

Übergabe der IDI_ASTERISK-Ressourcen-ID, wenn der Parameter **ByVal As String** deklariert wurde

Sie können auch zwei separate Deklarationen erstellen: eine für `Long`-Parameter, die andere für Zeichenfolgenparameter. Von diesem Ansatz wurde bei der Lösung für dieses Puzzle ausgegangen. Mit Hilfe des `Alias`-Schlüsselwortes von Visual Basic können Sie einen Funktionsnamen für eine Visual Basic-Deklaration definieren, der sich von dem Funktionsnamen innerhalb der DLL unterscheidet. In diesem Fall verwendet die `LoadIconBynum`-Deklaration einen `Long`-Parameter, die `LoadIconBystring`-Deklaration einen Zeichenfolgenparameter. Beide Funktionen benutzen den `LoadIconA`-Einsprungpunkt in der DLL.

```
' IconView Puzzle
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten

Option Explicit

Private Declare Function LoadIconBynum Lib "user32.dll" Alias "LoadIconA" ( _
    ByVal hInstance As Long, _
    ByVal lpIconName As Long) As Long

Private Declare Function LoadIconByString Lib "user32.dll" Alias _
"LoadIconA" ( _
    ByVal hInstance As Long, _
    ByVal lpIconName As String) As Long

Private Declare Function DrawIcon Lib "user32" (ByVal hdc As Long, _
ByVal x As Long, ByVal y As Long, ByVal hIcon As Long) As Long

Private Const IDI_APPLICATION = 32512&
Private Const IDI_HAND = 32513&
Private Const IDI_QUESTION = 32514&
Private Const IDI_EXCLAMATION = 32515&
Private Const IDI_ASTERISK = 32516&
```

```
Private Const IDI_WINLOGO = 32517&

Private m_IconID As Long

Private Sub Form_Load()
   m_IconID = IDI_APPLICATION
End Sub

Private Sub optIcon_Click(Index As Integer)
   Select Case Index

Case 0
         m_IconID = IDI_APPLICATION
      Case 1
         m_IconID = IDI_ASTERISK
      Case 2
         m_IconID = IDI_EXCLAMATION
      Case 3
         m_IconID = IDI_HAND
      Case 4
         m_IconID = IDI_QUESTION
      Case 5
         m_IconID = IDI_WINLOGO
   End Select
   Picture1.Refresh
End Sub

Private Sub Picture1_Paint()
   Dim iconhandle As Long
   iconhandle = LoadIconBynum(0, m_IconID)
   If iconhandle = 0 Then
      MsgBox GetErrorString(Err.LastDllError)
   Else
      Call DrawIcon(Picture1.hdc, 0, 0, iconhandle)
   End If
End Sub
```

Lösung 7

Überladene Grafiken

Wie bei den meisten API-Bezogenen Problemen liegt die Ursache für den Fehler in der Deklaration. Sehen wir uns die SDK-Beschreibung der Poly-Funktion noch einmal genau an:

```
BOOL Polyline(

    HDC hdc,                    // Zugriffsnummer für Gerätekontext
    CONST POINT *lppt,          // Adresse des Arrays mit den Endpunkten
    int cPoints                 // Anzahl der Punkte im Array
   );
```

```
Private Declare Function Polyline Lib "gdi32" (ByVal hdc As Long, _
lppt() As POINTAPI, ByVal nCount As Long) As Long
```

Es handelt sich um eine Funktion, die einen BOOL-Wert zurückgibt. Ein Win32-BOOL-Wert ist ein Long-Wert, d.h., dies ist zumindest vorerst richtig. Die DLL lautet gdi32, was auch stimmt, ist doch diese Datei gdi32 genau jene DLL, die sämtliche der grundlegenden Grafikbibliotheksfunktionen enthält.

Der count-Wert ist eine Ganzzahl, und Win32-Ganzzahlen umfassen 32 Bit, d.h., der nCount-Parameter ist ebenfalls richtig.

Wie sieht es mit lpPoints aus?

Abbildung S7-1 zeigt den lppt-Parameter, wie er von der Polyline-Funktion gesehen wird. Die Funktion erwartet als Parameter einen Zeiger auf die erste POINTAPI-Struktur in einem Array von Strukturen im Speicher.

Durch die Visual Basic-Deklaration wird dieser Parameter als lppt() As POINTAPI definiert. Welcher Wert wird an eine API-Funktion übergeben, wenn Sie ein Array auf diese Weise deklarieren? Die Antwort auf diese Frage können sie Abbildung S7-2 entnehmen.

Abbildung S7-2 zeigt, dass es mit der Deklaration von Arrays mehr auf sich hat, als Sie zunächst denken mögen. Visual Basic speichert ein Array intern unter Verwendung des OLE-Speichersystem für Arrays. Dieses Speichersystem verwendet die Struktur SAFEARRAY, die zur Verwaltung von Arrays eingesetzt wird. Jedes Array verfügt über eine SAFEARRAY-Struktur, in der sämtliche Informationen gespeichert werden, die zur Beschreibung des Arrays vonseiten des Systems benötigt werden. Die OLE-API enthält verschiedene Funktionen, mit denen SAFEARRAY-Strukturen erstellt und bearbeitet werden können. Wenn Sie eine Variable in Visual Basic als

Array deklarieren, enthält die Variable tatsächlich einen Zeiger auf eine SAFEARRAY-Struktur, mit der das Array beschrieben wird. Die SAFEARRAY-Struktur wiederum enthält einen Zeiger auf die Arraydaten.

Wenn Sie einen API-Parameter als Array deklarieren, übergibt Visual Basic einen Zeiger an die Arrayvariable – an einen Zeiger auf eine SAFEARRAY-Struktur.

Dies ist ja alles sehr interessant, werden Sie sich sagen, aber was hat das mit der Übergabe eines Arrays an die Polyline-Funktion zu tun? Nichts! Die einzigen API-Funktionen, die SAFEARRAY-Strukturen verwenden können, sind die der OLE-API. Der Punkt hierbei ist, dass Sie keine Parameter als Array deklarieren sollten, wenn über API-Funktionen Zeiger auf Arrays verwendet werden.

Aber wie können Sie dann einen Zeiger auf ein Array übergeben? Sehen Sie sich erneut Abbildung S7-1 an. Aus der Perspektive der Polygonfunktion kann das Array als ein Zeiger auf das erste Element in einem Array betrachtet werden. Solange alle Strukturen sich zusammenhängend im Speicher befinden, müssen Sie lediglich sicherstellen, dass die Adresse der ersten Struktur im Array als Parameter übergeben wird. Dies führt zu folgender Deklaration:

```
Private Declare Function Polyline Lib "gdi32" (ByVal hdc As Long, _
lppt As POINTAPI, ByVal nCount As Long) As Long
```

Jetzt übergeben Sie eine einzelne POINTAPI-Struktur – oder etwa nicht? Da der Parameter als Verweis übergeben wird, reichen Sie die Adresse dieser Struktur. Solange auf diese im Speicher andere Strukturen des Arrays folgen, übergeben Sie einen Zeiger auf das gesamte Array weiter, genau wie in Abbildung S7-1 dargestellt.

Diese Änderung macht es jedoch erforderlich, dass Sie auch die Art des Funktionsaufrufs ändern. Wenn Sie lediglich die Deklaration ändern, erhalten Sie einen ByRef-Argumentfehler wegen Nichtübereinstimmung. Ändern Sie die Funktion daher von

```
Call Polyline(hdc, PointArray(), points)
```

in

```
Call Polyline(hdc, PointArray(0), points)
```

Sie müssen das erste Element im Array übergeben, nicht etwa das gesamte Array.

Dieser Schritt reicht aus, um das Programm zu berichtigen.

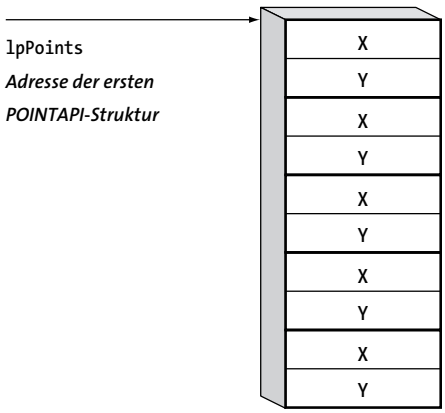


Abbildung S7-1 Der `lppt`-Parameter aus der Perspektive der `Polyline`-Funktion

Lösung 8

Bockspringen

Das erste Problem haben Sie bestimmt schnell lokalisiert. Es gibt zwei Gründe dafür, weshalb eine Funktion in einer DLL möglicherweise nicht aufgefunden werden kann: Sie haben entweder in der falschen DLL gesucht oder aber einen falschen Funktionsnamen verwendet. Die Versionsnummer des Betriebssystems ist bekanntlich sehr eng mit dem Betriebssystem verknüpft, daher sollte logischerweise `kernel32` die DLL sein, in der die Funktion enthalten ist. Und in dieser DLL ist die Funktion auch tatsächlich enthalten. Daher muss der Funktionsname falsch sein.

Es gibt nur zwei Gründe, weshalb eine Systemfunktion in der DLL über einen anderen Namen verfügen kann, als in der Dokumentation angegeben wird. Gelegentlich wird in der Dokumentation ein C-Makro beschrieben – ein Name, der in eine Reihe von C-Funktionsaufrufen übersetzt wird. Dies kommt glücklicherweise nur selten vor, und zudem wird dies wird üblicherweise in der Dokumentation beschrieben.

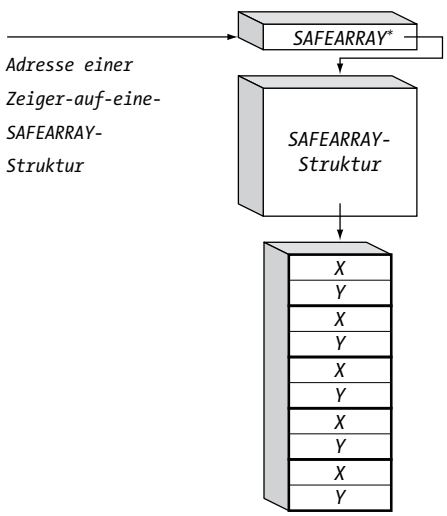


Abbildung S8-1 Arrayparameter werden als Zeiger auf Zeiger auf ein `SAFEARRAY` übergeben

Meistens liegt das Problem darin begründet, dass die Funktion einen Zeichenfolgenparameter verwendet und daher in der DLL mit zwei Einsprungpunkten auftaucht, nämlich mit einem ANSI-Einsprungpunkt mit dem Suffix A und als Unicode-Einsprungpunkt mit dem Suffix W.

Die `GetVersionEx`-Funktion verwendet jedoch keinen Zeichenfolgenparameter, wo liegt also das Problem?

Warten Sie mal! Sehen Sie sich noch einmal die OSVERSIONINFO-Struktur an:

```
Private Type OSVERSIONINFO
   dwOSVersionInfoSize As Long
   dwMajorVersion As Long
   dwMinorVersion As Long
   dwBuildNumber As Long
   dwPlatformId As Long
   szCSDVersion As String * 128 ' Vergessen Sie nicht, dass
                                ' VB-Arrays nullbasiert sind!
End Type
```

Die Struktur verfügt also über eine Zeichenfolge. Dies bedeutet, dass die Funktion tatsächlich zwei Einsprungpunkte benötigt, einen für eine ANSI-Zeichenfolge innerhalb einer Struktur, die andere für Unicode-Zeichenfolgen. Die richtige Deklaration lautet daher folgendermaßen:

```
Private Declare Function GetVersionEx Lib "kernel32" Alias _
"GetVersionExA" (os As OSVERSIONINFO) As Long
```

Die Funktion schlägt immer noch fehl, es wird eine 0 (ungültig) zurückgegeben.

Die C-Deklaration für die Struktur sieht vor, dass das dwOSVersionInfoSize-Feld die Größe der Struktur enthält. Bedeutet dies, dass die API-Funktion die Struktur zusammen mit ihrer tatsächlichen Größe lädt, oder müssen Sie die Größe festlegen, bevor Sie die Funktion aufrufen können? Wenn Sie die Hinweise in Anhang A und die Dokumentation zu dieser Funktion aufmerksam gelesen haben, dann wird Ihnen ein kurzer Hinweis aufgefallen sein, in dem erläutert wird, dass das dwOSVersionInfoSize-Feld mit der Strukturgröße initialisiert werden muss, bevor die Funktion aufgerufen wird.[6]

Durch die folgende einfache Codeänderung wird das Problem gelöst:

```
' Sie müssen das size-Feld initialisieren
   os.dwOSVersionInfoSize = Len(os)
```

6. Mein technischer Redakteur wies mich darauf hin, dass es eventuell unfair sein könnte, zu erwarten, dass jedem Leser mein API-Buch oder die Win32-SDK-Onlinedokumentation zur Verfügung steht. Ich entgegnete, dass dieses Buch niemandem etwas nützt, der keinen Zugang zur Win32-API-Dokumentation hat, von daher setze ich dies voraus.
   Er sagte mir außerdem, dass er es als äußerst unfair empfinden würde, dass der einzige Hinweis auf die Lösung bei diesem Puzzle im Anhang und nicht im Puzzle selbst genannt würde. Da hat er recht.
   Ich bin jedoch der Meinung, dass selbst wenn Sie aus diesem Puzzle nur lernen, dass Sie sich stets den jeweiligen Abschnitt in der API-Dokumentation genau durchlesen sollten, dann hat dieses Puzzle seinen Zweck bereits erfüllt.

Wieso ist unter Windows eine Initialisierung der Strukturgröße erforderlich? Sollte die Strukturgröße nicht bereits bekannt sein?

Die Antwort ist raffiniert und gleichzeitig elegant. Windows verwendet die Größe zur Versionsermittlung der Struktur, die von der aufrufenden Anwendung erwartet wird. Mal angenommen, die Windows-Entwickler entscheiden, dass Sie unter Windows NT 8.0 eine neue 128 Zeichen umfassende Zeichenfolge einführen möchten, mit der Sie die E-Mail-Adresse von Bill Gates abrufen könnten, um von ihm persönliche technische Unterstützung zu erhalten. Ältere Anwendungen würden weiterhin die aktuelle `OSVERSIONINFO`-Struktur verwenden, und wenn das System den Versuch unternähme, Daten in diesen neuen Puffer zu schreiben, würde die Anwendung aller Wahrscheinlichkeit nach abstürzen. Auf diese Weise kann das System die Größe des Puffers ermitteln. Die aktuelle `OSVERSIONINFO`-Struktur umfasst 148 Byte, die neue würde 276 Byte enthalten.

Windows setzt diese Methode bei Strukturen häufig ein. Wenn Sie also jemals mit einer Funktion konfrontiert werden, die einen Strukturparameter verwendet und fehlschlägt, stellen Sie sicher, dass alle erforderlichen Felder initialisiert wurden.

Lösung 9

Übersetzen von DEVMODE

Wenn Sie sich die Daten innerhalb einer Struktur ansehen und die Werte keinen Sinn zu ergeben scheinen, deutet dies darauf hin, dass ein Problem in der Deklarationsstruktur vorliegt und eine der Variablen in der Deklaration eine falsche Größe aufweist. Bei einfachen numerischen Deklarationen können Sie fast nichts falsch machen. Bei Zeichenfolgen und Arrays liegt der Fall dagegen anders. Das `dmDeviceName`-Feld ist in C beispielsweise folgendermaßen definiert:

```
#define CCHDEVICENAME 32
BYTE   dmDeviceName[CCHDEVICENAME];
```

Auf den ersten Blick scheint die folgende VB-Deklaration richtig zu sein:

```
Private Const CCHDEVICENAME = 32
dmDeviceName(CCHDEVICENAME) As Byte
```

Dieser Eindruck täuscht jedoch. Warum? Sehen Sie sich die Länge des Arrays an.

Die Arraygrenzen in einem C++-Array definieren die Anzahl der Elemente im Array. In diesem Fall enthält das Array also 32 Byte. Die Arraygrenzen in Visual Basic dagegen definieren die obere Grenze des Arrays. Wenn das erste Element im Array 0 lautet und die obere Grenze 32 (wie im vorliegenden Fall), umfasst das Array insgesamt 33 Elemente!

Eine Lösung besteht darin, die VB-Deklaration folgendermaßen abzuändern:

```
dmDeviceName(CCHDEVICENAME-1) As Byte
```

Es gibt jedoch eine einfachere Lösung. Da dieser Eintrag so konzipiert ist, dass er sowieso eine Zeichenfolge enthält, können Sie einfach eine Zeichenfolge mit festgelegter Länge verwenden. Im Gegensatz zu dynamischen Zeichenfolgen, die als BSTR-Zeiger (32-Bit-Zeiger) gespeichert werden, wird eine Zeichenfolge mit fester Länge direkt in der Struktur gespeichert. Wenn Sie also die Zeichenfolge wie folgt deklarieren,

```
Dim dmDeviceName As String * CCHDEVICENAME
```

werden 32 Byte in die Struktur aufgenommen.

Ändern Sie `dmFormName` gleichermaßen, und die Funktion wird einwandfrei funktionieren.

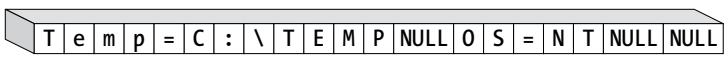
Lösung 10

Ein umweltbezogenes Problem

Was genau wird durch die `GetEnvironmentString`-Funktion zurückgegeben? Gemäß der Dokumentation handelt es sich bei dem Rückgabewert um einen Zeiger auf einen Speicherblock, der durch NULL-Werte voneinander getrennte Umgebungsvariablen enthält. Die letzte Variable endet auf zwei NULL-Werte. Nehmen wir an, Sie würden über die folgenden zwei Umgebungsvariablen verfügen:

```
Temp=C:\ TEMP
OS=NT
```

Diese würden folgendermaßen im Speicher erscheinen:



Aus Tutorium 6, »C++ Variablen treffen auf Visual Basic«, wissen Sie, dass es sich bei einer Visual Basic-Zeichenfolge tatsächlich um eine OLE-Zeichenfolge oder um `BSTR` handelt. Ein `BSTR`-Wert ist eine Zeichenfolge, die durch das OLE-Subsystem verwaltet wird (Zuweisung und Aufheben der Zuweisung). Ein `BSTR`-Zeiger verweist auf den Beginn der Zeichenfolgendaten. Die 32 Bit vor dieser Position enthalten die Länge der Zeichenfolge. Denken Sie außerdem daran, dass die von Visual Basic verwendeten `BSTR`-Zeichenfolgen üblicherweise Unicode-Daten enthalten.

Woher wissen wir, dass die `GetEnvironmentStrings`-Funktion keinen `BSTR`-Rückgabewert generiert?

- Der oben dargestellte Puffer enthält eindeutig keine Längendaten für die Zeichenfolge.
- Die `GetEnvironmentStrings`-Funktion ist Teil von `kernel32` (der Kern-DLL für das Betriebssystem), nicht des OLE-Subsystems. `BSTR`-Zeichenfolgen werden nur vom OLE-Subsystem verwendet.
- Die Dokumentation besagt, dass Sie die `FreeEnvironmentStrings`-Funktion verwenden müssen, um den Puffer freizugeben. Für eine `BSTR`-Zeichenfolge wäre keine besondere Funktion erforderlich, da diese mit der OLE-Funktion `SysFreeString` freigegeben werden könnte.

Wenn Sie die `GetEnvironmentStrings`-Funktion so deklarieren, dass eine Zeichenfolge zurückgegeben wird, erwartet Visual Basic einen `BSTR`-Wert und lädt diesen schon mal fröhlich in eine seiner internen Zeichenfolgenvariablen. Unglücklicherweise handelt es sich aber nicht um einen `BSTR`-Wert. Sobald Visual Basic also ver-

sucht, auf die Daten zuzugreifen und diese freizugeben, stehen die Chancen gut, dass es zu einem Speicherausnahmefehler kommt.

Was tun?

Der erste Schritt hin zur Problemlösung ist der, den Ergebniswert als `Long`-Wert zu deklarieren. Auf diese Weise erhalten Sie einen Zeiger auf den Speicherpuffer, den Sie anschließend mit Hilfe der Funktion `FreeEnvironmentStrings` freigeben können.

Aber was geschieht mit dem Puffer?

Im verwendeten Beispiel verfügen Sie bereits über den Code, mit dem Sie eine Zeichenfolge auf NULL-Werte hin prüfen und die Zeichenfolgen in das Listenfeld einfügen können. Das einzige Problem ist also, die Zeichenfolgendaten aus dem Umgebungspuffer in eine Zeichenfolge zu kopieren. Die `RtlMoveMemory`-Funktion kann dazu verwendet werden, den Speicher zu kopieren. Ihr Aufruf setzt jedoch voraus, dass Sie die Gesamtlänge des Puffers bis zu den zwei NULL-Werten kennen.

Die im überarbeiteten Projekt **EnvStr.vbp** (auf der Begleit-CD-ROM) enthaltene `FindDoubleNull`-Funktion zeigt eine Vorgehensweise:

```
' Ermitteln der Anzahl der Bytes einschließlich der Doppelnull
Private Function FindDoubleNull(ByVal Ptr As Long) As Long
   Dim bytebuffer(1) As Byte
   Dim offset As Long
   If Ptr = 0 Then Exit Function ' Error
   Do
      Call RtlMoveMemory(bytebuffer(0), ByVal (offset + Ptr), 2)
      If bytebuffer(0) = 0 And bytebuffer(1) = 0 Then
         FindDoubleNull = offset + 2 ' Add the two null bytes to the count
         Exit Function
      End If
      offset = offset + 1
   Loop While True
End Function
```

Die Funktion kopiert die Daten in Gruppen zu je zwei Bytes in das `bytebuffer`-Array und erhöht gleichzeitig bei jedem Durchlauf die Quelladresse. Werden zwei aufeinander folgende Nullbytes ermittelt, wird die Funktion beendet, und es wird die Gesamtzahl der Bytes von der Startadresse bis einschließlich der zwei NULL-Werte zurückgegeben. Die Funktion `RtlMoveMemory` kann nun dazu eingesetzt werden, die Daten aus dem Puffer in eine Zeichenfolge zu kopieren, der zuvor die

gewünschte Länge zugeordnet wurde. Beachten Sie, wie die durch das in der RtlMoveMemory-Funktion angegebene Ziel sowohl die Zeichenfolgen- als auch die Pufferadressvariablen ByVal übergeben werden. Die Umgebungs-Zeichenfolge wird ByVal übergeben, um die Konvertierung von der Visual Basic-Zeichenfolge in eine auf NULL endende C-Zeichenfolge (dem Format im Puffer) durchzuführen. Die Variable EnvironmentBuffer wird ByVal übergeben, da die Variable die Zeigeradresse enthält. Wenn Sie dies nicht tun, wird statt eines Zeigers auf die in der Variablen enthaltenen Adresse ein Zeiger auf die EnvironmentBuffer-Variable selbst weitergereicht. Mit der Funktion cmdGetStrings_Click werden die Umgebungsvariablen wie folgt abgerufen:

```
Private Sub cmdGetStrings_Click()
   Dim Environment As String
   Dim EnvironmentBuffer As Long
   Dim BufferLength As Long
   Dim CurrentPosition As Long
   Dim NewPosition
    list1.Clear
   EnvironmentBuffer = GetEnvironmentStrings()
   ' Pufferlänge abrufen
   BufferLength = FindDoubleNull(EnvironmentBuffer)
   ' Speicherplatz für die Daten zuweisen
   Environment = String$(BufferLength, 0)
   ' Zeichenfolge aus dem Puffer kopieren
   Call RtlMoveMemory(ByVal Environment, ByVal _
   EnvironmentBuffer, BufferLength)
   Do
      NewPosition = InStr(CurrentPosition + 1, Environment, Chr$(0))
      If NewPosition > CurrentPosition + 1 Then
         List1.AddItem Mid$(Environment, CurrentPosition + 1, _
         NewPosition - CurrentPosition - 1)
         CurrentPosition = NewPosition
      Else
         Exit Do
      End If
   Loop While True
   Call FreeEnvironmentStrings(EnvironmentBuffer)
End Sub
```

Sie fragen sich vielleicht, ob es nicht eine Möglichkeit gibt, den doppelten NULL-Wert auf schnellere Art zu ermitteln. Das Aufrufen einer separaten RtlMoveMemory-Funktion für jedes einzelne Byte immerhin eine recht mühsame Arbeit. Die Ant-

wort lautet glücklicherweise: Ja. Sie können die Länge einer Zeichenfolge mit Hilfe der `lstrlen`-API-Funktion ermitteln. Da es sich bei dem Umgebungspuffer lediglich um eine Liste der auf NULL endenden Zeichenfolgen handelt, können Sie eine Schleife durchlaufen und diese Funktion pro Zeichenfolge einmal aufrufen:

```
' Ermitteln der Anzahl der Bytes bis einschließlich der Doppelnull
Private Function FindDoubleNull2(ByVal ptr As Long) As Long
   Dim offset As Long
   Dim ThisLen As Long
   If ptr = 0 Then Exit Function ' Error
   Do
      ' Länge der vorliegenden Zeichenfolge ermitteln
      ThisLen = lstrlen(offset + ptr)
      ' Nächstes Offset nach dem Nullwert auf Byte setzen
      offset = offset + ThisLen + 1
      If ThisLen = 0 Then
         ' Dies gilt für zwei aufeinanderfolgende NULL-Werte
         FindDoubleNull2 = offset
         Exit Function
      End If
   Loop While True
End Function
```

Es gibt noch eine Methode, die sogar noch schneller zum Erfolg führt. Die DLL **apigid32.dll** (auf der Begleit-CD-ROM) beinhaltet die Funktion `agGetStringFrom2NullBuffer`, die folgendermaßen deklariert wird:

```
Declare Function agGetStringFrom2NullBuffer Lib "apigid32.dll" _
(ByVal Ptr As Long) As String
```

Diese Funktion verwendet als Parameter einen Zeiger auf einen Puffer mit den auf NULL endenden Zeichenfolgen, wobei die letzte Zeichenfolge zwei NULL-Werte aufweist – genau das Format, das von der `GetEnvironmentStrings`-Funktion verwendet wird. Hierbei wird eine Visual Basic-Zeichenfolge zurückgegeben. Moment mal – gibt es da nicht ein Problem mit DLL-Funktionen, die eine Zeichenfolge zurückgeben? Machen Sie sich keine Sorgen, diese Funktion wurde speziell entwickelt, um einen durch das OLE-Subsytem zugewiesenen BSTR-Wert zurückzugeben – genau das Format, das von Visual Basic erwartet wird. Bei der `agGetStringFrom2NullBuffer`-Funktion sind die zwei NULL-Werte nicht in der zurückgegebenen Zeichenfolge enthalten. Wenn Sie also diesen Ansatz wählen, müssen Sie die Schleife so bearbeiten, dass die letzte Zeichenfolge im Puffer abgerufen wird, da diese nicht durch die Funktion `lstrlen` ermittelt wird.

Lösung 11

Registrierungsspiele, Teil 1

Wenn Sie dieses Problem in weniger als fünf Minuten gelöst haben: Herzlichen Glückwunsch! Sie haben die wichtigste Lektion bei der API-Programmierung gelernt. Prüfen Sie Ihren Code immer noch ein zweites Mal auf Fehler hin, die durch Unachtsamkeit entstanden sein können.

Wenn Sie das Problem nicht sofort gefunden haben, werden Sie sich gleich an den Kopf fassen.

Erstes Problem

Den besten Hinweis auf das Problem gibt uns die Fehlermeldung, die besagt, dass der Registrierungsschlüssel nicht geöffnet werden konnte und das System die Datei nicht gefunden hat. Könnte es sein, dass der Fehler in den Parametern `hKey` oder `lpSubKey` liegt?

Der `hKey`-Parameter ist die Konstante aus der Datei **api32.txt**. Obwohl der Fehler hier liegen könnte, ist es vielleicht effektiver, zunächst den `lpSubKey`-Parameter zu prüfen, bevor Sie die Konstante `HKEY_LOCAL_MACHINE` mit den ursprünglichen Microsoft-Headerdateien vergleichen.

Der Teilschlüssel ist folgendermaßen definiert:

```
controlkey = "SYSTEM/CurrentControlSet/Control"
```

Sehen Sie das Problem?

```
controlkey = "SYSTEM\ CurrentControlSet\ Control"
```

Und jetzt?

Zweites Problem

Nachdem Sie in der Zeichenfolge die Schrägstriche durch Backslashes ersetzt haben, erhalten Sie unter Windows NT die Fehlermeldung, dass der Zugriff auf den Registrierungsschlüssel verweigert wurde.

Das ist merkwürdig. Die `KEY_READ`-Konstante ist eine logische `Or`-Kombination aller Konstanten, die zum Öffnen eines Schlüssels, zum Lesen und Aufzählen der zugehörigen Teilschlüssel sowie zum Lesen der Datenwerte benötigt werden könnten.

Sehen Sie sich zur Lösung des Problems noch einmal die Deklaration der `RegOpenKeyEx`-Funktion und den Code an, durch den die Funktion aufgerufen wird:

```
Private Declare Function RegOpenKeyEx Lib "advapi32.dll" Alias _
"RegOpenKeyExA" (ByVal hKey As Long, ByVal lpSubKey As String, _
ByVal ulOptions As Long, ByVal samDesired As Long, phkResult As _
Long) As Long

res = RegOpenKeyEx(HKEY_LOCAL_MACHINE, controlkey, KEY_READ, 0, rootkey)
```

Sehen Sie sich jetzt nacheinander die Funktionsparameter an. Da haben wir den Schlüssel, den Teilschlüssel ... Moment mal ... werden die Zugriffsrechte mit dem `ulOptions`- oder mit dem `samDesired`-Parameter angegeben?

Das ist es! Die Parameter `KEY_READ` und 0 wurden vertauscht. Ändern Sie den Code folgendermaßen ab:

```
res = RegOpenKeyEx(HKEY_LOCAL_MACHINE, controlkey, 0, KEY_READ, rootkey)
```

Es funktioniert!

Nebenbei gesagt ...

Sie denken vielleicht, dass beide Puzzle etwas künstlich wirken und ich diese Puzzle nur verwendet habe, um Sie durcheinander zu bringen, nur um Ihnen zu zeigen, dass selbst dann API-Programmierfehler auftreten können, wenn die Deklaration richtig ist.

Ich wünschte, ich könnte mich solcher Raffiniertheit rühmen (oder ist es Verschlagenheit?).

Die Wahrheit ist, dass mein erstes registrierungsbezogenes Puzzle Nummer 12 sein sollte, da die Aufzählung von Schlüsseln gewisse Schwierigkeiten mit sich bringt. Wie Sie sehen, folge ich meinen eigenen Ratschlägen und implementiere API-basierten Code in kleinen Abschnitten, die ich zwischendurch teste. Die zwei gezeigten Probleme befanden sich in meinem ursprünglichen Code. Und obwohl ich (in aller Bescheidenheit) zugebe, dass ich weniger als fünf Minuten gebraucht habe, um beide Probleme zu lösen, denke ich, dass beide Fehler realitätsnah sind und jederzeit auftreten können und deshalb ein eigenes Puzzle verdienen.

Lösung 12

Registrierungsspiele, Teil 2

Wenden wir uns zunächst dem Problem zu, die vorhandene `GetKeyInfo`-Funktion zu berichtigen, so dass diese unter Windows NT funktioniert.[7] Die Fehlermeldung besagt, dass der Fehler auf einen falschen Parameter zurückzuführen ist. In diesen Fällen gilt die Untersuchung jedes einzelnen Parameters als bester Ansatz. Der einzig fragwürdige Parameter ist `lpReserved`, der folgendermaßen beschrieben wird:

```
LPDWORD lpReserved,                    // reserviert - muss gleich NULL sein
```

Was bedeutet, dass der Parameter gleich NULL sein muss? Es bedeutet, dass der Parameter den Wert 0 aufweisen muss. Welcher Wert wird in der `GetKeyInfo`-Funktion als Parameter übergeben? Wir übergeben 0, oder? Das entspricht NULL, oder?

Oder nicht?

Sehen Sie sich die VB-Deklaration noch einmal an:

```
Private Declare Function RegQueryInfoKey Lib "advapi32.dll" Alias _
"RegQueryInfoKeyA" (ByVal hKey As Long, ByVal lpClass As String, _
lpcbClass As Long, lpReserved As Long, lpcSubKeys As Long, _
lpcbMaxSubKeyLen As Long, lpcbMaxClassLen As Long, lpcValues As Long, _
lpcbMaxValueNameLen As Long, lpcbMaxValueLen As Long, _
lpcbSecurityDescriptor As Long, lpftLastWriteTime As FILETIME) As Long
```

`lpReserved` ist als Verweis definiert. Was geschieht in Visual Basic, wenn Sie den Wert 0 an einen Parameter übergeben, der `By Reference As Long` deklariert wurde?

Es wird eine temporäre `Long`-Variable zugewiesen, die mit dem Wert 0 geladen wird. Anschließend wird die Adresse dieser temporären Variablen an die Funktion übergeben. Und die Adresse einer temporären Variablen ist niemals 0!

Wenn Sie einen NULL-Wert an einen Funktionsparameter vom Typ `Long` übergeben möchten, muss die Parameterdeklaration auf `ByVal As Long` abgeändert werden.

---

7. Merkwürdigerweise funktioniert die Funktion unter Windows 95 und 98 einwandfrei – ein weiterer Hinweis darauf, dass Sie Ihre Anwendungen immer unter beiden Betriebssystemen testen sollten. Wenn Sie Ihre Entwicklung unter Windows NT durchführen, können Sie diese Art von Fehlern bereits zu einem frühen Entwicklungszeitpunkt ermitteln.

Ein zweiter Lösungsweg

Dieser Ansatz stellt eine Möglichkeit dar, die GetKeyInfo-Funktion, wie im Puzzle gefordert, effizienter zu gestalten. Finden Sie die Parameter, die für die gestellte Aufgabe nicht geeignet sind, und ändern Sie deren Deklarationen in ByVal As Long, damit NULL-Parameter verwendet werden können. Sie können hierbei die Fähigkeit des Alias-Befehls zum Erstellen einer neuen Deklaration nutzen, mit der die RegQueryInfoKey-API-Funktion aufgerufen wird. Mit Hilfe des folgenden Codes wird eine neue RegQueryInfoKeyV2-Funktion deklariert, die so konzipiert ist, dass Sie an alle Parameter mit Ausnahme von lpSubKeys und lpcbMaxSubKeyLen NULL-Parameter übergeben können.

```
Private Declare Function RegQueryInfoKeyV2 Lib "advapi32.dll" _
Alias "RegQueryInfoKeyA" (ByVal hKey As Long, ByVal lpClass As String, _
ByVal lpcbClass As Long, ByVal lpReserved As Long, lpcSubKeys As Long, _
lpcbMaxSubKeyLen As Long, ByVal lpcbMaxClassLen As Long, ByVal _
lpcValues As Long, ByVal lpcbMaxValueNameLen As Long, ByVal _
lpcbMaxValueLen As Long, ByVal lpcbSecurityDescriptor As Long, _
ByVal lpftLastWriteTime As Long) As Long
```

Die Auswirkung auf die GetKeyInfo-Funktion ist, wie Sie der nachstehenden Version GetKeyInfo2 sehen können, beeindruckend:

```
Private Function GetKeyInfo2(ByVal hKey As Long, NumberOfKeys As _
Long, MaxKeyNameLength As Long) As Long
   Dim res As Long
   res = RegQueryInfoKeyV2(hKey, vbNullString, 0, 0, _
   NumberOfKeys, MaxKeyNameLength, 0, 0, 0, 0, 0, 0)
   If res <> 0 Then
      MsgBox "RegQueryInfoKey error: " & GetErrorString(res)
   End If
   GetKeyInfo2 = res
End Function
```

Ein dritter Lösungsweg

Der zuvor beschriebene Lösungsweg weist einen kleinen Nachteil auf. Jedes Mal, wenn Sie auf andere Teilinformationen der Funktion RegQueryInfoKey zugreifen müssen, müssen Sie einen neuen Alias erstellen. Gibt es eine Möglichkeit, die Effizienz dieses Ansatzes zu erhalten und dabei nur eine einzige RegQueryInfoKey-Deklaration zu verwenden?

Ja! Sie erreichen dies durch eine As Any-Parameterdeklaration, wobei der aufrufende Code zur Übergabe eines NULL-Wertes den Ausdruck ByVal 0& erhält. Der

Datentyp `As Any` birgt jedoch Gefahren und sollte, wann immer möglich, vermieden werden. In diesem Fall ist seine Verwendung nicht nötig, da ein anderer Ansatz verwendet werden kann, der leichter zu verstehen und einfacher umzusetzen ist. Sie verwenden die folgende Deklaration, bei der alle Parameter, bei denen es sich nicht um Zeichenfolgen handelt, `ByVal As Long` deklariert werden:

```
Private Declare Function RegQueryInfoKeyV3 Lib "advapi32.dll" _
Alias "RegQueryInfoKeyA" (ByVal hKey As Long, ByVal lpClass As String, _
ByVal lpcbClass As Long, ByVal lpReserved As Long, ByVal lpcSubKeys As _
Long, ByVal lpcbMaxSubKeyLen As Long, ByVal lpcbMaxClassLen As Long, _
ByVal lpcValues As Long, ByVal lpcbMaxValueNameLen As Long, _
ByVal lpcbMaxValueLen As Long, ByVal lpcbSecurityDescriptor As Long, _
ByVal lpftLastWriteTime As Long) As Long
```

Der Trick besteht darin, erstens explizit einen Zeiger auf die Variablen zu übergeben, die mit Informationen geladen werden sollen und zweitens den Wert 0 für die verbleibenden Parameter zu übergeben. Der `VarPtr2`-Operator[8] wird dazu verwendet, die Adressen für die Variablen abzurufen und ja, dieser Operator funktioniert, wie hier gezeigt, selbst mit Stackparametern. Wie Sie sehen können, kann die Funktion nun in einer einzelnen Codezeile implementiert werden:

```
Private Function GetKeyInfo3(ByVal hKey As Long, NumberOfKeys _
As Long, MaxKeyNameLength As Long) As Long

   GetKeyInfo3 = RegQueryInfoKeyV3(hKey, vbNullString, 0, 0, _
VarPtr(NumberOfKeys), VarPtr(MaxKeyNameLength), 0, 0, 0, 0, 0, 0)
End Function
```

Anmerkungen zur Entwicklung

Bei den meisten Beispielen in diesem Buch handelt es sich um eher banalen Code, doch dies ist kein Grund, das Softwaredesign völlig zu ignorieren. Wenn Sie das Beispielprogramm **Reg2** (auf der Begleit-CD-ROM) näher betrachten, wird Ihnen zusätzlicher Code auffallen, an dem Sie ablesen können, worauf das Design abzielt. Die CD-ROM enthält darüber hinaus das Klassenmodul `clsKeyValues`, das folgenden Code enthält:

8. Der `VarPtr`-Operator ist ein nicht dokumentierter Operator. Normalerweise würde ich Ihnen niemals empfehlen, eine nicht dokumentierte Funktion zu verwenden, da diese jedoch von Microsoft selbst in vielen Codebeispielen verwendet wird, ist deren Verwendung meiner Meinung nach relativ sicher. Die `agGetAddressForObject`-Funktion in der Datei **apigid32.dll** auf der Begleit-CD-ROM bietete eine Alternative zum Abrufen der Adresse einer Visual Basic-Variable.

```
' Klassenmodul clsKeyValues
' Schlüssel und Werte

Option Explicit
Public KeyName As String   ' Name dieses Schlüssels
Public ValueNames As Collection  ' Werte dieses Schlüssels
```

Den Lesern mit Erfahrung in der Komponentenerstellung mit Hilfe von Visual Basic wird das wenig anspruchsvolle Design dieser Klasse auffallen. Statt über Eigenschaftenanweisungen wird öffentlich auf die Felder zugegriffen. Mit der Klasse wird anstelle einer benutzerdefinierten Auflistung ein reines `Collection`-Objekt offengelegt. Würde es sich um ein öffentliches Objekt handeln, das durch einen ActiveX-Server offengelegt wird, würde ein solches Design zu erheblichen Problemen führen.

Es handelt sich jedoch nicht um ein öffentliches Objekt, das durch einen ActiveX-Server offengelegt wird. Es handelt sich um ein privates Objekt zur Verwendung innerhalb dieser Anwendung. Daher besteht keine Gefahr, dass eine außenstehende Person die Objekteigenschaften in ungeeigneter Form verwendet. Ich erwähne dies für den Fall, dass Sie Teile dieser Beispielanwendung erweitern und auf einem ActiveX-Objektserver verwenden möchten. (In meinem Buch »Developing COM/ActiveX Components with Visual Basic 6.0« wird u.a. auf diese Problematik eingegangen.)

Die `EnumerateKeys`-Funktion ruft über die Funktion `GetKeyInfo` (in sämtlichen Varianten) Informationen über die Teilschlüssel ab. Anschließend wird die eigentliche Aufzählung vorgenommen und eine Auflistung der `clsKeyValues`-Objekte erstellt, die an die aufrufende Funktion zurückgegeben werden können:

```
' Aufzählen aller Teilschlüssel für diesen Schlüssel,
' Zurückgeben einer Auflistung
' mit den clsKeyValues -Objekten
Private Function EnumerateKeys(ByVal hKey As Long) _
As Collection
   Dim NumberOfSubKeys As Long
   Dim MaxSubKeyLength As LongDim res As Long

   ' Abrufen der Anzahl der Teilschlüssel sowie der benötigten Pufferlänge
      res = GetKeyInfo(hKey, NumberOfSubKeys, MaxSubKeyLength)
   MsgBox "NumberOfSubKeys: " & NumberOfSubKeys & _
   "  MaxSubKeyLength: " & MaxSubKeyLength

   NumberOfSubKeys = 0
```

```
    res = GetKeyInfo2(hKey, NumberOfSubKeys, MaxSubKeyLength)
    MsgBox "NumberOfSubKeys: " & NumberOfSubKeys & _
    "  MaxSubKeyLength: " & MaxSubKeyLength

    NumberOfSubKeys = 0
    res = GetKeyInfo3(hKey, NumberOfSubKeys, MaxSubKeyLength)
    MsgBox "NumberOfSubKeys: " & NumberOfSubKeys & _
    "  MaxSubKeyLength: " & MaxSubKeyLength
End Function
```

Die vollständige Implementierung dieser Funktion sehen Sie in Puzzle 13.

Lösung 13

Registrierungsspiele, Teil 3

Sie verfügen über viele Zeilen mit neuem Code, wissen jedoch nicht, warum dieser nicht ausgeführt werden kann. In einem solchen Fall ist es unerlässlich, den Code schrittweise und sorgfältig zu überprüfen. Hierbei muss das Wort »sorgfältig« besonders betont werden. Dies bedeutet, dass Sie nicht nur die Funktionsergebnisse prüfen, sondern auch mögliche Nebeneffekte berücksichtigen sollten. Im vorliegenden Fall verdient die folgende Zeile besondere Beachtung:

```
res = GetValueInfo(hKey, NumberOfSubValues, MaxSubValueLength)
```

Die Funktion gibt stets einen richtigen Wert zurück (0), es werden jedoch niemals die Variablen `NumberOfSubValues` oder `MaxSubValueLength` gesetzt! Wir wissen aus Erfahrung, dass viele der Teilschlüssel in der Registrierung über Werte verfügen, also muss hier ein Problem vorliegen. Sehen wir uns die Funktion `GetValueInfo` genauer an:

```
' Abrufen der Werteanzahl sowie der maximalen Wertelänge
Private Function GetValueInfo(ByVal hKey As Long, _
NumberOfValues, MaxValueNameLength) As Long
   GetValueInfo = RegQueryInfoKeyV3(hKey, vbNullString, 0, 0, 0, 0, _
   0, VarPtr(NumberOfValues), VarPtr(MaxValueNameLength), 0, 0, 0)
End Function
```

Wie kann die Funktion stets ein richtiges Ergebnis zurückgeben aber dennoch nicht funktionieren? Die Antwort ist folgende: Sie kann nicht funktionieren. Es muss also ein Fehler im Code vorliegen. Aber wo liegt das Problem? Die Funktion sieht wie die `GetKeyInfo3`-Funktion aus, von der wir wissen, dass Sie funktioniert. Sehen wir uns die Funktion `GetKeyInfo3` erneut an:

```
Private Function GetKeyInfo3(ByVal hKey As Long, NumberOfKeys _
As Long, MaxKeyNameLength As Long) As Long
   GetKeyInfo3 = RegQueryInfoKeyV3(hKey, vbNullString, 0, 0, _
   VarPtr(NumberOfKeys), VarPtr(MaxKeyNameLength), 0, 0, 0, 0, 0, 0)
End Function
```

Schauen Sie genau hin. Erkennen Sie das Problem?

Die `GetKeyInfo3`-Funktion deklariert den `NumberOfKeys`-Parameter als Wert vom Typ `Long`. Wir haben es versäumt, in der `GetValueInfo`-Funktion einen Parametertyp anzugeben. Demnach werden die Parameter als Varianten deklariert.

Wenn wir nun den in Funktion GetKeyInfo2 gezeigten Ansatz verwenden, bei der ein Alias erzeugt wurde, damit die RegQueryInfoKeyGetKeyInfo2-Funktion mit Parametern aufgerufen wurde, die als Verweis übergeben wurden, würde diese Funktion funktionieren. Visual Basic würde den Parameter vom Typ Variant nehmen, eine temporäre Variable vom Typ Long erstellen, die Adresse der Long-Variablen an die Funktion übergeben und anschließend die Long-Variable zurück in den Variant-Parameter kopieren.

Wir übergeben die Adresse hier jedoch explizit, d.h., die Adresse für den Variant-Parameter wird über den VarPtr-Operator zurückgegeben. Ein Variant wird intern durch eine Struktur dargestellt, daher wird durch VarPtr die Adresse am Strukturbeginn zurückgegeben. Und der Long-Wert, den wir bekommen möchten, ist nicht am Beginn dieser Variant-Struktur gespeichert. Tatsächlich ist es erstaunlich, dass durch diesen Bug kein Systemabsturz erzeugt wird, da ungültige Werte in den Variant-Parameter geladen werden. Es ist ziemlich wahrscheinlich, dass bestimmte Kombinationen von Teilschlüsseln und Wertenamenlängen auch zu einem Speicherausnahmefehler führen könnten.

No Boom Today – Boom Tomorrow (There's Always a Boom Tomorrow)[9]

Sie können sich glücklich schätzen, wenn der Bug im vorherigen Puzzle keinen Speicherausnahmefehler hervorgerufen hat. Sobald Sie diesen Fehler jedoch beheben, wird Ihr System abstürzen (zumindest, wenn Sie unter NT arbeiten). Wenn Sie den Code durchgehen, werden Sie feststellen, dass der Absturz während des RegEnumValue-Aufrufs aufgetreten ist. Dies mag merkwürdig erscheinen, wenn man berücksichtigt, dass die Funktion fast identisch mit dem RegEnumKey-Aufruf ist.

Warten Sie, sagte ich »fast« identisch? In der Tat.

Ein genauerer Blick auf die Deklarationen zeigt, dass ein wichtiger Unterschied zwischen beiden Aufrufen besteht. Bei dem lpcbValueName-Parameter in der RegEnumValue-Funktion handelt es sich nicht um DWORD, sondern um LPDWORD. Dies bedeutet, dass dieser Parameterwert nicht als Wert, sondern als Verweis übergeben werden muss. Die lpcbValueName-Variable sollte auf die maximale Länge des lpValueName-Puffers gesetzt werden. Bei der Rückgabe enthält die Variable die aktuelle Länge der Zeichenfolge.

9. Diese Überschrift wurde schamloserweise einfach aus der TV-Serie Babylon 5 übernommen, bei der es sich nebenbei gesagt um eine der besten Serien handelt, die je produziert wurden.

Was geschieht, wenn wir diesen Parameter `ByVal` übergeben? Angenommen, die von der `GetValueInfo`-Funktion berechnete Länge beträgt 50 Byte. Wenn der Parameter `ByVal` deklariert wird, wird die Zahl 50 im Stack platziert. Da die `RegEnumValue`-Funktion jedoch einen Zeiger auf die Länge erwartet, wird der Wert 50 als Speicheradresse einer `Long`-Variable interpretiert, die die aktuelle Länge enthält. Wenn das Programm nun versucht, auf die Speicheradresse 50 zuzugreifen, stürzt das System mit einem Speicherausnahmefehler ab. Boom![10]

Die geprüfte `EnumerateValues`-Funktion verwendet die `CurrentValueLength`-Variable, um diese Änderung zu handhaben. Die Variable wird mit der Pufferlänge geladen und an die API-Funktion übergeben. Bei der Rückgabe enthält sie die aktuelle Länge des Variablennamens. Da wir die Länge kennen, besteht keine Notwendigkeit, mit Hilfe der `nulloffset`-Variable und der `Instr`-Funktion die tatsächliche Zeichenfolgenlänge zu bestimmen. Die `CurrentValueLength`-Variable muss vor jedem `RegEnumValue`-Aufruf auf die maximale Pufferlänge zurückgesetzt werden:

```
Private Declare Function RegEnumValue Lib "advapi32.dll" _
Alias "RegEnumValueA" (ByVal hKey As Long, ByVal dwIndex As Long, _
ByVal lpValueName As String, lpcbValueName As Long, ByVal lpReserved _
As Long, ByVal lpType As Long, ByVal lpData As Long, ByVal lpcbData As _
Long) As Long


' Aufzählen aller Werte für diesen Schlüssel, Zurückgeben einer Auflistung
' mit den Wertenamen
Private Function EnumerateValues(ByVal hKey As Long) As Collection
   Dim NumberOfSubValues As Long
   Dim MaxSubValueLength As Long
   Dim res As Long
   Dim ValueIndex As Long
   Dim Valuebuffer As String
   Dim ValueCollection As New Collection
   Dim currentValue As clsKeyValues
   Dim CurrentValueLength As Long

   res = GetValueInfo(hKey, NumberOfSubValues, MaxSubValueLength)
   If res <> 0 Then
      Exit Function
```

10. Die Windows 95/98-Version dieser Funktion ermittelt, dass es sich um eine ungültige Adresse handelt und gibt einen Parameterfehler zurück (Fehler 87), statt einen Speicherausnahmefehler zu verursachen.

```
    End If

    Valuebuffer = String$(MaxSubValueLength + 1, 0)

    For ValueIndex = 0 To NumberOfSubValues - 1
       CurrentValueLength = MaxSubValueLength + 1
       res = RegEnumValue(hKey, ValueIndex, Valuebuffer, CurrentValueLength, _
       0, 0, 0, 0)
       If res = 0 Then
          If CurrentValueLength = 0 Then
             ValueCollection.Add "Default"
          Else
             ValueCollection.Add Left$(Valuebuffer, CurrentValueLength)
          End If
       End If
    Next ValueIndex
    Set EnumerateValues = ValueCollection
End Function
```

Schlussfolgerung

Bei dem hier vorgeführten Parametertypfehler handelt es sich um einen der subtilsten Bugs, die beim Arbeiten mit API-Funktionen auftreten können, insbesondere, da es sich nicht wirklich um einen API-Deklarationsbug handelt. Das versehentliche Verwenden eines Parameters vom Typ `Variant` aufgrund einer fehlenden Parameterdeklaration führt üblicherweise nicht zu einem Programmabsturz. Dennoch ist dieses Vorgehen ineffizient und kann zu schwerwiegenden Problemen führen, wenn Sie eine öffentliche ActiveX-Komponente erstellen, da der Parametertyp `Variant` eine sehr viel intensivere Fehlerprüfung erfordert als Basisvariablentypen, die als Komponenteneigenschaften oder Parameter verwendet werden.

Lösung 14

Registrierungsspiele, Teil 4

Nun liege ich hier am Pool in Disney World, da fliegt plötzlich ein Flugzeug vorbei, hinter dem ein Transparent mit der Aufschrift »Dan, ruf' deinen Verleger an!« herflattert. Was für eine Art, einen Urlaub zu unterbrechen.

Okay, okay. Genug gescherzt. Sie haben mich erwischt, die ganze Sache war beabsichtigt (oder?). Wenn Sie bisher noch nicht versucht haben, die `DisplayValue`-Funktion selbst zu schreiben, möchte ich Ihnen dies dringend ans Herz legen.

So, und jetzt wird's ernst.

Wir wollen uns zunächst die VB-Deklaration ansehen. Der Trick hierbei besteht darin, das richtige Verfahren für den `lpData`-Parameter zu finden. In den vorangegangenen drei Puzzlen haben Sie alle weiteren Situationen kennengelernt. Der `lpData`-Parameter wird in der Dokumentation folgendermaßen definiert:

```
LPBYTE lpData,                    // Adresse des Datenpuffers
```

Das Problem liegt darin, dass wir über drei verschiedene Datentypen verfügen, die von dieser Funktion geladen werden können – Zeichenfolgen, Daten vom Typ `Long` sowie binäre Daten. Wir müssen außerdem in der Lage sein, einen NULL-Wert zu übergeben, da der erste Schritt darin besteht, die Funktion mit `lpData` gleich NULL aufzurufen, damit wir Datentyp und Puffergröße erhalten.

Lassen Sie uns sämtliche Möglichkeiten zur Behandlung der einzelnen Datentypen in Betracht ziehen, wie in Tabelle S14-1 dargestellt.

Sie können zwischen zwei allgemeinen Ansätzen wählen. Sie können die sicherste und einfachste Deklaration für jeden Datentyp auswählen und mehrere Aliasse für die Funktion `RegQueryValueEx` erstellen. In diesem Fall wählen Sie für den Nullwert `ByVal As Long`, für Zeichenfolgendaten `ByVal As String`, `As Long` für Daten vom Typ `Long` und `As Byte` für binäre Daten. Sie können die `RegQueryValueEx`-Funktion zunächst mit einem NULL-`lpData`-Wert aufrufen, um den Datentyp für den Wert zu ermitteln. Anschließend rufen Sie den geeigneten Alias für diesen Datentyp auf.

Der zweite Ansatz, der in dieser Lösung verwendet wird, sieht die Verwendung des Datentyps `As Any` vor und richtet sich danach, ob durch die aufrufende Routine der richtige Variablentyp im `lpData`-Parameter übergeben wird. Die `RegQueryValueEx`-Funktion verfügt demnach über die folgende Deklaration:

```
Private Declare Function RegQueryValueEx Lib "advapi32.dll" Alias _
"RegQueryValueExA" (ByVal hKey As Long, ByVal lpValueName As String, _
ByVal lpReserved As Long, lpType As Long, lpData As Any, _
lpcbData As Long) As Long
```

Datentyp	Deklaration	Aufruf
NULL	`ByVal As Long`	Null oder Variable, die Null enthält
	`ByVal As Any`	`0&` oder `Long`-Variable, die Null enthält
	`As Any`	`ByVal 0&` oder `ByVal` mit einer `Long`-Variable, die Null enthält
String	`ByVal As String`	Zeichenfolgenpuffer, der zuvor auf die benötigte Pufferlänge initialisiert wurde
	`ByVal As Any`	Zeichenfolgenpuffer, der zuvor mit der benötigten Pufferlänge initialisiert wurde
	`As Any`	`ByVal`-Operator, gefolgt von Zeichenfolgenpuffer, der zuvor mit der benötigten Pufferlänge initialisiert wurde
	`As Byte`	Erstes Byte in einem Bytearray, das mit einer ANSI-Zeichenfolge geladen wird. Die Zeichenfolge muss in Unicode konvertiert werden, bevor eine Zuweisung zur VB-Zeichenfolge erfolgt.
	`ByVal As Long`	Adresse des ersten Bytes in einem Bytearray, das zum Erhalt des verwendeten `VarPtr`-Operators mit einer ANSI-Zeichenfolge geladen wird. Die Zeichenfolge muss in Unicode konvertiert werden, bevor eine Zuweisung an die VB-Zeichenfolge erfolgt.
Long	`As Long`	Mit den Daten zu ladende `Long`-Variable
	`ByVal As Long`	Adresse der `Long`-Variablen, die unter Verwendung des `VarPtr`-Operators abgerufen wird
	`As Any`	Mit den Daten zu ladende `Long`-Variable
Binary	`As Byte`	Erstes Byte in einem Bytearray, das mit binären Daten geladen wird
	`As Any`	Erstes Byte in einem Bytearray, das mit binären Daten geladen wird.
	`ByVal As Long`	Adresse des ersten Bytes in einem Bytearray, das zum Erhalt des verwendeten `VarPtr`-Operators mit binären Daten geladen wird.

Tabelle S14-1 Verwendung verschiedener Datentypen

STOP!

Wenn Sie einen eigenen Lösungscode erarbeitet haben, sehen Sie nach, ob Sie dabei einen der beiden genannten Ansätze verwendet haben. Anschließend sollten Sie eine Pause machen und versuchen, die Lösung mit Hilfe des anderen Ansatzes zu implementieren. Dies wird Ihnen dabei helfen, die möglichen Optionen besser zu verstehen.

Durch den hier verwendeten Ansatz wird die `DisplayValue`-Funktion recht einfach.

Handelt es sich bei dem Datentyp um eine Zeichenfolge, wird durch die Funktion ein Zeichenfolgenpuffer mit der geeigneten Länge initialisiert und mit Hilfe des `ByVal`-Operators an die API-Funktion übergeben. Auf diese Weise wird sichergestellt, dass die erforderliche Unicode-in-ANSI-Konvertierung stattfindet, wenn die Funktion aufgerufen wird. In diesem Beispiel wird die `REG_MULTI_SZ`-Funktion deshalb nicht verwendet, da eine Analyse dieses Zeichenfolgentyps eine einfache Visual Basic-Übung darstellt (die Sie ausprobieren sollten). Diese Art der Zeichenfolge enthält mehrere Zeichenfolgen, die durch NULL-Zeichen separiert werden, wobei am Ende der letzten Zeichenfolge zwei NULL-Werte enthalten sind.

Handelt es sich bei dem Datentyp um einen `Long`-Wert, übergibt die Funktion eine `Long`-Variable als Parameter, was denselben zu einen Verweis macht.

Handelt es sich um den binären Datentyp, wird ein Bytearray durch die Funktion auf die richtige Länge dimensioniert, anschließend wird das erste Byte des Arrays als Verweis übergeben. So wird auf effektive Weise ein Zeiger auf den Beginn des Bytearrays weitergereicht. Die `DisplayValue`-Funktion sieht folgendermaßen aus:

```
Private Sub DisplayValue(ByVal hKey As Long, ByVal ValueName As String)
   Dim BufferSize As Long
   Dim DataType As Long
   Dim buffer As String
   Dim longdata As Long
   Dim binarydata() As Byte
   Dim res As Long
   Dim LabelOutput As String
   Dim x As Long
   Dim hexval As String

   ' Abrufen von Datentyp und Puffergröße
   res = RegQueryValueEx(hKey, ValueName, 0, DataType, ByVal 0&, BufferSize)
```

```
   ' Dieser Schritt sollte immer erfolgreich sein, da zuvor der Wertename
   ' geladen wurde
   If res = 0 Then
      Select Case DataType
         Case REG_SZ, REG_EXPAND_SZ, REG_MULTI_SZ
            LabelOutput = "String: "
            buffer = String$(BufferSize, 0)
            res = RegQueryValueEx(hKey, ValueName, 0, DataType, _
            ByVal buffer, BufferSize)
            LabelOutput = LabelOutput & Left$(buffer, Len(buffer) - 1)
            ' Die Analyse der REG_MULTI_SZ-Zeichenfolge wird zur Übung
            ' für den Leser ausgelassen
         Case REG_DWORD
            LabelOutput = "DWORD: "
            res = RegQueryValueEx(hKey, ValueName, 0, DataType, _
            longdata, BufferSize)
            LabelOutput = LabelOutput & Hex$(longdata)
         Case REG_BINARY
            LabelOutput = "Binary: "
            ReDim binarydata(BufferSize - 1)
            res = RegQueryValueEx(hKey, ValueName, 0, DataType, _
            binarydata(0), BufferSize)
            For x = 0 To BufferSize - 1
               hexval = Hex$(binarydata(x))
               If Len(hexval) = 1 Then hexval = "0" & hexval
               hexval = " " & hexval
               LabelOutput = LabelOutput & hexval
            Next x
         Case Else
            LabelOutput = "Unsupported data type"
      End Select
   lblValue.Caption = LabelOutput
   End If
End Sub
```

Schlussfolgerung

Sicher, ich hätte auch eine fehlerhafte Version der `DisplayValue`-Routine schreiben können, die Sie hätten reparieren müssen, aber diese Funktion bietet eine perfekte Möglichkeit, sich mit Parametern vertraut zu machen, die verschiedene Datentypen verwenden. Wenn Sie in der Lage waren, unter Verwendung eines der (oder beider) Ansätze eine eigene Lösung für dieses Puzzle zu finden, dann gratuliere ich Ihnen – Sie sind jetzt ein erfahrener API-Programmierer und befinden sich auf dem besten Wege, den Guru-Status zu erlangen (falls Sie nicht schon soweit sind).

Lösung 15

Welche Zeitzone ist es denn nun?

Wenn Sie mit einer Struktur arbeiten und am Anfang der Struktur Daten sehen, die anscheinend richtig sind, die Daten gegen Ende der Struktur hin jedoch weniger sinnvoll erscheinen, ist es möglich, dass ein Fehler in der Strukturdeklaration vorliegt. API-Funktionen hängen davon ab, dass sämtliche Felder in der Struktur von Beginn an in exakt der Reihenfolge erscheinen, in der sie im Speicheroffset vorliegen. Wenn eines der Felder in der Visual Basic-Deklaration falsch deklariert worden ist, ist die Position aller folgenden Felder ebenfalls falsch. Im vorliegenden Fall lautet die C-Deklaration für die `TIME_ZONE_INFORMATION`-Struktur folgendermaßen:

```
typedef struct _TIME_ZONE_INFORMATION {  // tzi
    LONG       Bias;
    WCHAR      StandardName[ 32 ];
    SYSTEMTIME StandardDate;
    LONG       StandardBias;
    WCHAR      DaylightName[ 32 ];
    SYSTEMTIME DaylightDate;
    LONG       DaylightBias;
}  TIME_ZONE_INFORMATION;
```

Das `StandardName`-Feld wurde wie folgt deklariert:

```
StandardName(32) As Integer
```

Bei `WCHAR` handelt es sich um einen »wide« 16-Bit-Unicode-Zeichenwert, also erscheint auf den ersten Blick die Verwendung des Datentyps `Integer` als angemessen. Auf der anderen Seite jedoch wissen Sie, dass viele Win32-API-Funktionen über separate ANSI- und Unicode-Einsprungpunkte verfügen. Wenn diese Funktionen nun mit Strukturparametern versehen wird und diese wiederum Zeichenfolgen enthalten, richtet sich der verwendete Zeichenfolgentyp nach dem Einsprungpunkt.

Die Frage ist nun, welcher Typ wird hier verwendet? Verfügt die Funktion `GetTimeZoneInformation` über separate ANSI- und Unicode-Einsprungpunkte? Falls ja, so müssen die Felder `StandardName` und `DaylightName` der `TIME_ZONE_INFORMATION`-Struktur wahrscheinlich in den Datentyp `Byte` geändert werden. Verfügt die Funktion jedoch nur über einen Einsprungpunkt, handelt es sich bei den Feldern `StandardName` und `DaylightName` wahrscheinlich tatsächlich um 16-Bit-Unicode-Zeichen. Und was stimmt nun?

Sie wissen die Antwort schon. Als Erstes werden die Felder `StandardName` und `DaylightName` als vom Typ `WCHAR` definiert, was sie zu einem 16-Bit-Unicode-Zeichen macht. Wenn sich der Typ nach dem Einsprungpunkt richten würde, müssten sie als vom Typ `TCHAR` deklariert werden, mit dem ein Zeichen angegeben wird, dessen Größe vom Einsprungpunkt abhängt. Zweitens wurde die `GetTimeZoneInformation`-Funktion im Puzzle folgendermaßen deklariert:

```
Private Declare Function GetTimeZoneInformation Lib _
"kernel32" (lpTimeZoneInformation As TIME_ZONE_INFORMATION) As Long
```

Wie Sie sehen können, weist die Deklaration keine Aliasdefinition auf. Würde die Funktion über zwei Einsprungpunkte verfügen, würden Sie den Aliasbegriff »GetTimeZoneInformationA« in der Deklaration erwarten. Das Beispielprogramm erzeugte keinen Fehler vom Typ »DLL-Einstiegspunkt nicht gefunden«, daher sind verschiedene Einsprungpunkte in dieser `GetTimeZoneInformation`-Funktion tatsächlich nicht vorhanden.

Demnach ist alles, was wir bisher bewiesen haben, dass die Visual Basic-Deklaration richtig ist. Dennoch funktioniert das Beispiel nicht, also muss ein Fehler vorliegen.

Die C++-Deklaration zeigt, dass es sich bei den Feldern `StandardName` und `DaylightName` um 32-Bit-Zeichen vom Typ `Integer` handelt. Bei der Visual Basic-Deklaration liegt ebenfalls ein 32-`Integer`-Array vor.

Oder etwa nicht?

Visual Basic-Arraydeklarationen sind nullbasiert. Das Feld

```
StandardName(32) As Integer
```

ist ein also ein 33-`Integer`-Array!. Ändern Sie die Arraygrenzen auf 31, und das Programm funktioniert.

Lesen der Namen

Die Felder `StandardName` und `DaylightName` werden mit den Unicode-Zeichenfolgen geladen. Wie können diese in Visual Basic-Zeichenfolgen konvertiert werden?

Es gibt zwei mögliche Lösungswege. Der einfachere basiert auf einer interessanten Visual Basic-Funktion, die es Ihnen ermöglicht, Zeichenfolgen einem Array zuzuweisen und umgekehrt.

Wenn Sie eine Zeichenfolge definieren

```
Dim timezonename As String
```

können Sie dieses folgendermaßen direkt vom Array aus zuordnen:

```
timezonename = tz.StandardName
```

Und was ist mit der Unicode-Konvertierung? Diese erfolgt automatisch – bzw. ist irrelevant. Wenn Sie in Visual Basic eine Zuordnung zwischen Zeichenfolgen und Arrays vornehmen, wird keine Datenkonvertierung durchgeführt. Eine Zeichenfolge enthält intern Unicode-Daten, die bei einer Übermittlung aus dem Array an die Zeichenfolge mit Hilfe einer Zuordnung automatisch richtig formatiert werden.

Da ist nur ein Haken – Sie können ein Array vom Typ `Integer` nicht in eine Zeichenfolge kopieren.

Die Lösung ist einfach. Statt die Felder `StandardName` und `DaylightName` als `Integer` zu deklarieren, deklarieren wir Sie als `Byte` und stellen sicher, dass die Arrays die gleiche Größe haben. Im vorliegenden Fall sind die Arrays 64 Byte lang, d.h., die Deklaration lautet:

```
StandardName(63) As Byte
```

Nachdem die das Array einer Zeichenfolge zugewiesen haben, können Sie nach dem abschließenden NULL-Zeichen suchen (dieses befindet sich am Ende einer jeden C-Zeichenfolge) und die Zeichenfolge an diesem Punkt abschneiden.

Die richtigen Ergebnisse werden in Abbildung S15-1 gezeigt.

Hier nun das vollständige Programm:

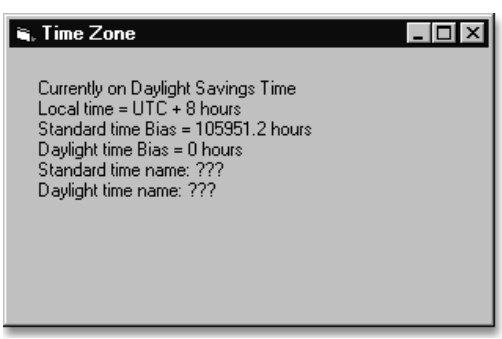


Abbildung S15-1 Die Endversion des Programms **TimeZone**

```
' Get Time Zone Information
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten

Option Explicit
```

```
Private Type SYSTEMTIME
        wYear As Integer
        wMonth As Integer
        wDayOfWeek As Integer
        wDay As Integer
        wHour As Integer
        wMinute As Integer
        wSecond As Integer
        wMilliseconds As Integer
End Type


Private Type TIME_ZONE_INFORMATION
        Bias As Long
        StandardName(63) As Byte
        StandardDate As SYSTEMTIME
        StandardBias As Long
        DaylightName(63) As Byte
        DaylightDate As SYSTEMTIME
        DaylightBias As Long
End Type

Private Declare Function GetTimeZoneInformation Lib _
"kernel32" (lpTimeZoneInformation As TIME_ZONE_INFORMATION) As Long

Private Const TIME_ZONE_ID_INVALID = &HFFFFFFFF
Private Const TIME_ZONE_ID_UNKNOWN = 0
Private Const TIME_ZONE_ID_STANDARD = 1
Private Const TIME_ZONE_ID_DAYLIGHT = 2

Private Sub Form_Load()
   Dim tz As TIME_ZONE_INFORMATION
   Dim info As String
   Dim res As Long
   Dim timezonename As String
   Dim nulloffset As Long
   res = GetTimeZoneInformation(tz)
   Select Case res
      Case TIME_ZONE_ID_INVALID
         lblTimeZone.Caption = "GetTimeZoneInformation function failed." _
         & vbCrLf
```

```
            lblTimeZone.Caption = lblTimeZone.Caption & "Error: " _
            & GetErrorString(Err.LastDllError)
         Case TIME_ZONE_ID_UNKNOWN
            lblTimeZone.Caption = "Time zone information unavailable."
         Case TIME_ZONE_ID_STANDARD
            info = "Currently on Standard Time" & vbCrLf
         Case TIME_ZONE_ID_DAYLIGHT
            info = "Currently on Daylight Savings Time" & vbCrLf
      End Select

      info = info & "Local time = UTC + " & tz.Bias / 60 & " hours" & vbCrLf
      info = info & "Standard time Bias = " & tz.StandardBias / 60 & _
      " hours" & vbCrLf
      info = info & "Daylight time Bias = " & tz.DaylightBias / 60 & _
      " hours" & vbCrLf
      timezonename = tz.StandardName
      nulloffset = InStr(timezonename, Chr$(0))
      info = info & "Standard time name: " & Left$(timezonename, _
      nulloffset - 1) & vbCrLf
      timezonename = tz.DaylightName
      nulloffset = InStr(timezonename, Chr$(0))
      info = info & "Daylight time name: " & Left$(timezonename, _
      nulloffset - 1) & vbCrLf
   lblTimeZone.Caption = info
   End Sub
```

Lösung 16

Seriennummern

Sehen wir uns die SDK-Beschreibung der `GetVolumeInformation`-Funktion noch einmal genau an:

```
BOOL GetVolumeInformation(

  LPCTSTR lpRootPathName,           // Adresse des Stammverzeichnisses
                                    // des Dateisystems
  LPTSTR lpVolumeNameBuffer,        // Adresse des Volumenamens
  DWORD nVolumeNameSize,            // Länge von lpVolumeNameBuffer
  LPDWORD lpVolumeSerialNumber,     // Adresse der Volumeseriennummer
  LPDWORD lpMaximumComponentLength, // Adresse der maximalen
                                    // Dateinamenlänge für das System
  LPDWORD lpFileSystemFlags,        // Adresse der Dateisystemflags
  LPTSTR lpFileSystemNameBuffer,    // Adresse des Dateisystemnamens
  DWORD nFileSystemNameSize         // Länge von
                                    // lpFileSystemNameBuffer
);
```

Was können Sie allein der Deklaration entnehmen, ohne in der SDK-Dokumentation nachzusehen?

Der einzige Parameter, der vorher gesetzt wird, ist der Parameter `lpRootPathName`, der das Stammverzeichnis enthält. Der Parametertyp lautet `LPCTSTR`. Eine Analyse ergibt Folgendes:

- `LP` Zeiger vom Typ `Long` (far)
- `C` Konstant, d.h., der Parameter wird durch die API-Funktion nicht geändert
- `T` ANSI, wenn der ANSI-Einsprungpunkt aufgerufen wird, Unicode, wenn der Unicode-Einsprungpunkt aufgerufen wird. Das Erscheinen eines T-Parameters impliziert, dass die Funktion sehr wahrscheinlich über zwei Einsprungpunkte verfügt und dass Sie den Einsprungpunkt mit dem A-Suffix verwenden, in diesem Fall `GetVolumeInformationA`.
- `STR` Zeichenfolge

Da Sie den ANSI-Einsprungpunkt verwenden, erwartet die Funktionen einen Zeiger auf eine auf NULL endende ANSI-Zeichenfolge, deren Inhalt von der API-Funktion nicht geändert wird.

Die Deklaration muss offensichtlich `ByVal As String` lauten.

Der `lpVolumeNameBuffer`-Parameter ist identisch, abgesehen davon, dass in der Typendefinition das C fehlt. Das bedeutet, dass es sich bei dem Parameter um keine Konstante handelt, die Zeichenfolge kann also durch die API-Funktion verändert werden. Eine solche Änderung setzt aber zwingend voraus, dass die Funktion um die Länge eines solchen Zeichenfolgepuffers weiß ... Diese Information wird mit Hilfe des Parameters `nVolumeNameSize` an die Funktion übergeben, der `ByVal As Long` deklariert ist.

Im ursprünglichen Puzzle wurde der `nVolumeNameSize`-Parameter auf 0 gesetzt, daher gibt die Funktion einen Fehler zurück, durch den angezeigt wird, dass weitere Informationen verfügbar sind.[11] Dieser Parameter sollte auf die Länge des Puffers gesetzt werden, in diesem Fall auf 256, da die Zeichenfolgen auf 256 Zeichen initialisiert wurden. Warum? Da die Win32-API häufig inkonsistent ist, wenn es darum geht, ob die Länge den abschließenden NULL-Wert enthält oder nicht. Sie sollten sich daher angewöhnen, Ihrem Puffer immer ein (oder zwei) Byte mehr zuzuweisen als nötig, für alle Fälle.

Die gleiche Technik wird zur Definition der Parameter `lpFileSystemNameBuffer` und `nFileSystemNameSize` angewandt.

Die Parameter `lpVolumeSerialNumber`, `lpMaximumComponentLength` und `lpFileSystemFlag` werden als Typ LPDWORD deklariert. Eine Analyse ergibt Folgendes:

- `LP` Zeiger vom Typ `Long` (far)
- `DWORD` Doppelwort ohne Vorzeichen- oder vorzeichenloser 32-Bit-Wert

Zur Übergabe eines numerischen Zeigers auf eine API-Funktion müssen Sie diesen nicht als Verweis, sondern als `ByVal` deklarieren. Entfernen Sie also die `ByVal`-Operatoren sämtlicher Parameter.

Und die Seriennummer ...

Sehen Sie sich die Seriennummer an, wie sie in einem DOS-Fenster als Rückgabe auf den Befehl Dir angezeigt wird:

```
D:\ >dir
 Volume in drive D is D1P1

The volume serial number is 12EC-0868.
```

11. Unter Windows 95 wird durch den Fehler angegeben, dass der Dateiname zu lang ist. Der Grund für diesen Unterschied liegt in einem Puzzle, das die Fähigkeiten des Autors übersteigt.

Der Hinweis in diesem Fall ist der Buchstabe, der zwischen A und F in der Seriennummer angezeigt wird, sowie die Tatsache, dass die Seriennummer aus zwei Zahlen mit je vier Ziffern besteht. Ein aus vier Ziffern bestehender Zahlenwert enthält im Hexadezimalformat 16 Bits. Zwei Zahlenwerte mit je vier Ziffern entsprechen also einer 32-Bit-Zahl.

Der Unterschied zwischen der im Puzzle angezeigten Seriennummer und der Seriennummer, die in einem DOS-Fenster angezeigt wird besteht also darin, dass es sich bei der einen um eine Dezimalzahl, bei der anderen um eine hexadezimale Zahl handelt.

Wie teilen Sie einen 32-Bit-Wert in zwei hexadezimale 16-Bit-Zahlenwerte auf?

Die hohen 32 Bits können mit Hilfe einer Division durch &H10000& abgerufen werden. Auf diese Weise werden alle Bits im Zahlenwert um 16 Bits nach rechts verschoben. Sie müssen außerdem die hohen 16 Bits unter Verwendung des `And`-Operators demaskieren, da durch die Division die hohen Bits nicht auf 0 gesetzt werden, wenn das höchste Bit ursprünglich 1 lautete. Dies kann durch den folgenden Code geschehen:

```
(SerialNumber \  &H10000) And &HFFFF&
```

Die niedrigen 16 Bits können mit Hilfe des `And`-Operators abgerufen und mit dem Wert `&HFFFF&` maskiert werden.

Die Endversion lautet folgendermaßen:

```
' Volume Information
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten

Option Explicit

Private Declare Function GetVolumeInformation Lib "kernel32" Alias _
"GetVolumeInformationA" (ByVal lpRootPathName As String, ByVal _
lpVolumeNameBuffer As String, ByVal nVolumeNameSize As Long, _
lpVolumeSerialNumber As Long, lpMaximumComponentLength As Long, _
lpFileSystemFlags As Long, ByVal lpFileSystemNameBuffer As String, _
ByVal nFileSystemNameSize As Long) As Long

Private Sub LoadVolumeInfo()
   Dim res&
   Dim VolumeName As String
   Dim FileSystem As String
   Dim SerialNumber As Long
```

```
   Dim ComponentLength As Long
   Dim FileSystemFlags As Long
   VolumeName = String$(256, Chr$(0))
   FileSystem = String$(256, Chr$(0))
   res = GetVolumeInformation(Left$(Drive1.Drive, 2) & "\ ",
         VolumeName, 255, SerialNumber, ComponentLength,
         FileSystemFlags, FileSystem, 255)
   If res = 0 Then
      MsgBox "GetVolumeInformation Error: " & _
      GetErrorString(Err.LastDllError)
      Exit Sub
   End If
   List1.Clear
   List1.AddItem "Drive " & Drive1.Drive
   List1.AddItem "Name: " & VolumeName
   List1.AddItem "Component Length: " & ComponentLength
   List1.AddItem "Serial #: " & Hex$((SerialNumber \
  &H10000) And &HFFFF&) _
   & "-" & Hex$(SerialNumber And &HFFFF&)
   List1.AddItem "FileSystem: " & FileSystem

End Sub

Private Sub Drive1_Change()
   LoadVolumeInfo
End Sub

Private Sub Form_Load()
   LoadVolumeInfo
End Sub
```

Sie haben vielleicht bemerkt, dass das Programm die Zeichenfolge nach dem NULL-Abschlusszeichen nicht abschneidet, nachdem über die `GetVolumeInformation`-Funktion Daten in die Variablen `FileSystem` und `VolumeName` geladen werden. Dies ist bei diesem Programm nicht nötig, da die Informationen direkt in einem Listenfeld platziert werden. Das Listenfeld basiert auf dem Windows-Steuerelement `LISTBOX`. Dieses Steuerelement erwartet eine auf NULL endende Zeichenfolge, daher »sieht« das Steuerelement nur den Teil der Zeichenfolge bis zum NULL-Zeichenwert, auch dann, wenn Sie dem Steuerelement mit Hilfe der `AddItem`-Methode eine längere Zeichenfolge übergeben.

Lösung 17

Der DEVMODE im Detail

Was Sie auf jeden Fall schnell lernen werden, wenn Sie mit API-Funktionen arbeiten, ist eine ausgesprochen sorgfältige Lektüre der Dokumentation. Vorbehaltlos einen Tipp oder eine Technik aus einer Fachzeitschrift oder dem Internet zu befolgen, ist nicht sehr verlässlich, da Sie nicht sicher sein können, ob der Autor sich Zeit genommen hat, die Dokumentation zu lesen und die Technik eingehender zu testen als nur in einer simplen Anwendung.

In diesem Fall finden Sie das Geheimnis an zwei Stellen.[1]

In der Beschreibung der Funktion `DocumentProperties` in meinem Win32-API-Buch finden Sie die folgende Beschreibung des Parameters `pDevModeOutput`:

> *Zeiger des Typs `Long` auf eine `DEVMODE`-Datenstruktur für das Gerät ... Beachten Sie, dass dieser Zeiger auf einen Puffer verweisen muss, der groß genug ist, um die privaten Druckertreiberdaten und darüber hinaus die standardmäßige `DEVMODE`-Struktur aufzunehmen. Ermitteln Sie die Adresse einer Struktur bzw. eines Bytearrays mittels der Funktion `agGetAddressForObject` (oder des Operators `VarPtr`), oder übergeben Sie die Adresse eines Speicherblocks unter Verwendung von `GlobalAlloc`.*
>
> *Wenn `fMode` den Wert Null hat, gibt diese Funktion die Größe der `DEVMODE`-Struktur für dieses Gerät zurück. Beachten Sie, dass diese Struktur größer sein kann als in der Typdefinitionsdatei API32.TXT angegeben.*

Wenn Sie einen Blick in den Win32 SDK werfen, finden Sie im Kommentarabschnitt Folgendes:

> *Wenn der Parameter fMode gleich Null ist, entspricht der Rückgabewert der Puffergröße, die erforderlich ist, um die Daten zur Druckertreiberinitialisierung aufzunehmen. Beachten Sie, dass dieser Puffer größer sein kann als die `DEVMODE`-Struktur, wenn der Druckertreiber private Daten an die Struktur anhängt.*
>
> *Beachten Sie, dass die tatsächlich von einem Druckertreiber verwendete `DEVMODE`-Struktur den geräteunabhängigen Teil enthält (wie oben definiert), gefolgt von einem treiberspezifischen Teil, der je nach Treiber und Treiberversion in Größe und Inhalt variiert. Aufgrund dieser Abhängigkeit vom Treiber ist es sehr wichtig, dass*

1. Vielleicht finden Sie es unfair, dass Sie eine weitere Dokumentation zur Lösung dieses Puzzles heranziehen müssen. Doch dieses Buch verfolgt unter anderem das Ziel, Ihnen die notwendigen Vorgehensweisen zu zeigen, mit denen Sie all dieser Probleme Herr werden können, und ohne jegliche Referenz ist eine ernsthafte API-Programmierung nicht möglich. Darüber hinaus wurde in Anhang A ausdrücklich betont, dass Sie bei diesem Puzzle zusätzliche Informationsquellen zu Rate ziehen müssen.

Anwendungen die tatsächliche Größe der DEVMODE-Struktur beim Treiber abfragen, bevor sie einen Puffer für die Struktur belegen.

Gehen Sie folgendermaßen vor, um für eine Anwendung lokale Änderungen der Druckereinstellungen vorzunehmen:

1. *Ermitteln Sie die Anzahl der für die vollständige DEVMODE-Struktur erforderlichen Bytes, indem Sie DocumentProperties aufrufen und für den Parameter fMode den Wert Null angeben.*
2. *Ordnen Sie der vollständigen DEVMODE-Struktur Speicher zu.*
3. *Ermitteln Sie die aktuellen Druckereinstellungen durch Aufruf von DocumentProperties. Übergeben Sie einen Zeiger als Parameter pDevModeOutput an die DEVMODE-Struktur, der Sie in Schritt 2 Speicher zugeordnet haben, und geben Sie den Wert DM_OUT_BUFFER an.*
4. *Bearbeiten Sie die entsprechenden Mitglieder der zurückgegebenen DEVMODE-Struktur, und geben Sie durch Setzen der entsprechenden Bits im Mitglied dmFields von DEVMODE an, welche Mitglieder geändert wurden.*
5. *Rufen Sie DocumentProperties auf, geben Sie die modifizierte DEVMODE-Struktur sowohl als Parameter pDevModeInput als auch pDevModeOutput zurück, und geben Sie die Werte DM_IN_BUFFER und DM_OUT_BUFFER an (die mittels des Operators OR kombiniert werden).*

Die C-Funktionsdeklaration beschreibt zwar die Parameter pDevModeInput und pDevModeOutput als Typ PDEVMODE (Zeiger auf eine DEVMODE-Struktur), doch bei näherer Betrachtung der Funktionsbeschreibung wird Ihnen klar, das dies noch nicht alles ist. Es handelt sich bei diesen Parametern tatsächlich um Zeiger auf Speicherpuffer, die am Anfang des Puffers eine DEVMODE-Struktur haben, gefolgt von zusätzlichen treiberspezifischen Daten. Aus der Perspektive von C ist die Beschreibung dieser Parameter als PDEVMODE-Datentyp völlig korrekt – der Parameter ist in der Tat ein Zeiger auf eine DEVMODE-Struktur. Auf die Struktur folgen einfach zusätzliche Daten.

Aus der Perspektive von Visual Basic ist die Angelegenheit ein wenig komplizierter. Wenn Sie den Parametertyp als DEVMODE-Struktur deklarieren, schließen Sie die zusätzlichen Daten nicht ein. Einige Druckertreiber verwenden in der DEVMODE-Struktur keine privaten Daten. Diese Drucker funktionieren hervorragend mit dem im Puzzle enthaltenen Code. Aber Treiber, die private Daten verwenden, werden versuchen, auf außerhalb der Struktur liegenden Speicher zuzugreifen. In manchen Fällen bleibt dies ohne Auswirkung oder führt zu einigen merkwürdigen Druckereinstellungen. In anderen Fällen kann ein Speicherausnahmefehler auftreten, Ihre Anwendung bricht plötzlich ab oder sonstige bizarre Phänomene treten in Ihrer Anwendung auf.

Der nachstehende Beispielcode zeigt Ihnen, wie Sie beim Aufruf von DocumentProperties richtig mit der DEVMODE-Struktur umgehen:

```
Dim hPrinter&
Dim res&
Dim dmIn As DEVMODE
Dim dmOut As DEVMODE
Dim DevmodeLength As Long
```

Wir ordnen der DEVMODE-Struktur und den privaten Daten des Druckers zwei dynamische Bytepuffer zu:

```
Dim dmInBuf() As Byte
Dim dmOutBuf() As Byte

' Bei Erfolg ungleich Null
res = OpenPrinter(Printer.DeviceName, hPrinter, 0)
If res = 0 Then
   lblStatus.Caption = "Printer could not be opened"
   Exit Sub
End If
```

Die Funktion DocumentProperties gibt die Länge des erforderlichen Puffers zurück, wenn Sie den Parameter fMode auf 0 setzen. Diese Länge wird zur Dimensionierung der zwei Bytearrays verwendet:

```
' Länge der DEVMODE-Struktur für diesen Drucker ermitteln
   DevmodeLength = DocumentProperties(hwnd, hPrinter, _
   Printer.DeviceName, VarPtr(dmOut), VarPtr(dmIn), 0)
   lblComment.Caption = "The DEVMODE structure for this printer is " _
   & DevmodeLength & " bytes. " & _
   "The standard DEVMODE structure is " & Len(dmOut) & " bytes."
   ReDim dmInBuf(DevmodeLength)
   ReDim dmOutBuf(DevmodeLength)
```

Übergeben Sie am Anfang der Bytearrays einen Zeiger als DEVMODE-Parameter.

```
' dmOutBuf-Puffer mit den Daten laden
res = DocumentProperties(hwnd, hPrinter, Printer.DeviceName, _
VarPtr(dmOutBuf(0)), VarPtr(dmInBuf(0)), DM_OUT_BUFFER Or DM_IN_PROMPT)
```

Die Daten befinden sich im Bytearray. Es ist leider ziemlich schwierig, auf Felder einer Struktur zuzugreifen, die in einem Bytearray begraben sind. Deshalb kopieren wir das Bytearray folgendermaßen unter Verwendung der Funktion RtlMoveMemory in eine DEVMODE-Struktur:

```
   ' Kopieren des DEVMODE-Anteils des Puffers in
   ' den dmOut-Puffer, damit wir ihn verwenden können
   Call RtlMoveMemory(dmOut, dmOutBuf(0), Len(dmOut))
   If dmOut.dmOrientation = DMORIENT_LANDSCAPE Then
      lblStatus.Caption = "Printer is in landscape mode"
   Else
      lblStatus.Caption = "Printer is in portrait mode"
   End If
   ' Sie können die Inhalte des dmout-Puffers bearbeiten und dann
   ' die Daten in den dmInBuf-Puffer kopieren und
   ' anschließend DocumentProperties mit DM_IN_
BUFFER aufrufen, um die neuen Werte festzulegen
   Call ClosePrinter(hPrinter)
```

Dieses Beispiel zeigt, wie Sie Gerätedaten abrufen können. Der Prozess wird umgekehrt, wenn Sie die Geräteparameter setzen möchten: Kopieren Sie die Daten aus dem Ausgabepuffer in den Eingabepuffer (sämtliche Daten, nicht nur den `DEVMODE`-Anteil). Nehmen Sie ggf. Modifikationen des `DEVMODE`-Anteils des Puffers vor, und setzen Sie die entsprechenden Bits im Feld `dmFieldsbits`, sodass das Gerät weiß, welche Parameter Sie ändern. Rufen Sie zum Schluss die Funktion `DocumentProperties` auf, wobei das `DM_IN_BUFFER`-Flag im Parameter `fMode` gesetzt wird.

Es mag Situationen geben, in denen Sie eine Visual Basic-Deklaration für eine Funktion haben, die Ihnen jemand übergeben hat, bzw. die Sie selbst konzipiert haben, und Sie entschlossen sind, sie in Ihrem Programm zu verwenden. Dann raufen Sie sich die Haare, weil Sie die Funktion nicht zum Laufen bringen oder weil sie ständig Speicherausnahmefehler verursacht. In vielen Fällen ist die Lösung durch sorgfältige Lektüre der Dokumentation zu finden, weil die Antwort (wie es bei der `DEVMODE`-Struktur der Fall ist) im Detail liegt.

Lösung 18

DT muss nach Hause telefonieren

DT und ich erholten uns schnell. Es war kein Gift. Es war nur ein wenig Brain-Freeze, um die Dramatik der Story zu steigern. Und es stellte sich heraus, dass eine schnelle Abkühlung jetzt genau das Richtige für mein Gehirn war.

»Das Geheimnis liegt im Packing.« Ich zeigte es DT.

Das Packing beschreibt, wie Variablen in einer Struktur angeordnet werden. Die meisten API-Strukturen unterstützen das Packing von einem Byte, d.h., jede Variable in der Struktur folgt im Speicher unmittelbar auf die vorhergehende. Das 4-Byte-Packing zwingt Variablen, entweder an der kleineren Variablengröße oder an der nächsten 32-Bit-Grenze zu beginnen. Das bedeutet, dass eine Einzelbytevariable an beliebiger Position beginnen kann, eine 16-Bit-Variable an einem geraden Byte beginnen muss, eine 32-Bit-Variable an einer 32-Bit-Grenze und Strukturen als Ganzes an 32-Bit-Grenzen beginnen. Visual Basic verwendet intern das 4-Byte-Packing, konvertiert jedoch Strukturen während API-Funktionsaufrufen in das 1-Byte-Packing. Das in diesem Fall erforderliche 4-Byte-Packing hat keinen Einfluss auf die Zeichenfolge `szEntryName`, weil sie nicht nur bereits nach dem 32-Bit-Parameter `dwSize` an einer 32-Bit-Grenze auftritt, sondern auch aus Einzelbyte-Elementen besteht, die an beliebiger Adresse beginnen können. Nun ist es jedoch so, dass eine Struktur, die mit ihrer Größe nicht bis an die folgende 32-Bit-Grenze heranreicht, genau bis dahin aufgefüllt wird, womit sichergestellt wird, dass die nächste Struktur genau dort beginnt. Aus diesem Grund wurde eine Fehlermeldung bezüglich einer ungültigen Strukturgröße zurückgegeben – die API-Funktion erwartet, dass die Struktur bis zur nächsten 32-Bit-Grenze aufgefüllt wird. Sie muss 264 Bytes lang sein, nicht 261. Die einfachste Lösung dieses Problems ist die Änderung der Strukturdeklaration wie folgt:

```
Public Type RASENTRYNAME
    dwSize As Long
    szEntryName As String * RAS_MaxEntryNameBuffer
    Padding(2) As Byte
End Type
```

»Hervorragend!« rief DT aus. »Die Fehlermeldung 632 erhalte ich nicht mehr. Nur die Fehlermeldung 603, die darauf hinweist, dass ein größerer Puffer erforderlich ist.«

Ich warf wieder einen Blick auf den Code. Nach Auftreten des Fehlers 603 berechnet das Programm die Anzahl der benötigten Einträge und ändert die Größe des Arrays `RasEntryBuffer` entsprechend:

```
Res = RasEnumEntries(0, vbNullString, RasEntryBuffer(0), BufferSize, _
      RasEntries)
If Res = 603 Then
   ' Puffer ist zu klein - Verändern der Größe zur ausgewählten Größe
   RasEntryCount = BufferSize / RasEntrySize
   ReDim RasEntryBuffer(RasEntryCount - 1)
   Res = RasEnumEntries(0, vbNullString, RasEntryBuffer(0), _
   BufferSize, RasEntries)
End If
```

Ich überflog den Code. »Oh-oh,« sagte ich, »Ich erhalte die Fehlermeldung 632 beim zweiten Aufruf von `RasEnumEntries`. Das ergibt keinen Sinn.«

»Na, und ob!« lachte DT. »Der Parameter `dwSize` im ersten Eintrag des Arrays wird mit der richtigen Größe initialisiert, die anderen jedoch nicht.«

»Aber die Dokumentation sagt nicht, dass die `dwSize`-Felder in jedem Eintrag der Struktur initialisiert werden müssen, oder?«

»Nein, aber sie behauptet auch nicht das Gegenteil. In diesem Punkt ist die Dokumentation ein wenig schwammig.«

»Und wie sollen wir dann wissen, das es notwendig ist?«

»Raten. Du probierst beides aus,« DT grinste. »Daran müsstest du dich eigentlich längst gewöhnt haben.«

Ich fügte der Funktion schnell den nötigen Code zur Initialisierung des `dwSize`-Feldes für jeden Eintrag in der Struktur hinzu:

```
Res = RasEnumEntries(0, vbNullString, RasEntryBuffer(0), BufferSize, _
      RasEntries)
If Res = 603 Then
   ' Puffer ist zu klein - Verändern der Größe zur ausgewählten Größe
   RasEntryCount = BufferSize / RasEntrySize
   ReDim RasEntryBuffer(RasEntryCount - 1)
   For CurrentEntry = 0 To RasEntryCount - 1
      RasEntryBuffer(CurrentEntry).dwSize = RasEntrySize
   Next CurrentEntry
   Res = RasEnumEntries(0, vbNullString, RasEntryBuffer(0), _
         BufferSize, RasEntries)
End If
```

Es funktionierte! Das Ergebnis war 0, Indiz dafür, dass die Funktion einwandfrei lief. Wir wollten gerade die Gläser heben und auf unseren Erfolg anstoßen, als DT verblüfft auf den Bildschirm starrte.

»Es funktioniert nur beim ersten Telefonbucheintrag! Der zweite Eintrag ist leer!«

Technische Anmerkung

Unter Umständen erzielen Sie auf Ihrem System nicht die oben beschriebenen Ergebnisse. Lesen Sie die folgenden Ausführungen, um die Ursache zu ermitteln.

Arrays der Strukturen

Wie kann die Funktion ein richtiges Ergebnis zurückgeben, obwohl ein Struktureintrag leer bleibt? Die Antwort: Er ist nicht leer. Die einzige Erklärung dafür ist die, dass der Struktureintrag, den Sie sehen, nicht mit dem Struktureintrag identisch ist, den die API-Funktion sieht.

Wie kann das sein?

Denken Sie einen Augenblick daran, wie Zeichenfolgen in Visual Basic gespeichert werden. Sie werden stets als Unicode-Zeichenfolgen gespeichert. Gilt dies auch für Zeichenfolgen innerhalb von Strukturen?

Ja.

Wie kann also eine Struktur an eine API-Funktion übergeben werden, die eine ANSI-Zeichenfolge erwartet?

Visual Basic muss die Struktur von Unicode in ANSI konvertieren. Hierzu wird, wie in Abbildung S18-1 gezeigt, eine temporäre Kopie der Struktur mit den nach ANSI konvertierten Zeichenfolgen erstellt.

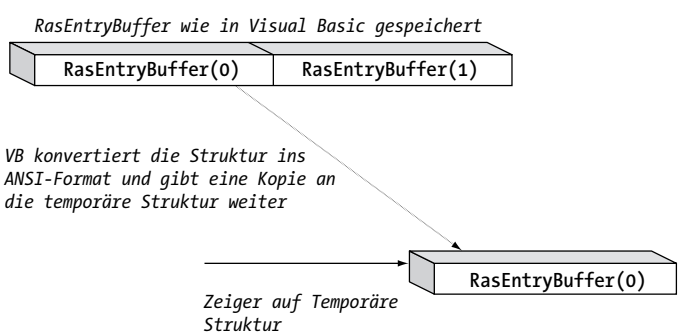


Abbildung S18-1 Übergeben von Strukturen in Arrays an eine API-Funktion

Doch was geschieht, wenn Sie einen Zeiger an ein Array solcher Strukturen übergeben möchten? Normalerweise können Sie einen Zeiger an ein Array übergeben, indem Sie einen Zeiger an das erste Element des Arrays übergeben. Doch in diesem Fall reicht Visual Basic einen Zeiger an die temporäre Struktur weiter. Visual Basic konvertiert nicht alle Einträge im Array – dazu gibt es keinen Grund, weil Sie, was VB betrifft, nur das erste Element an die API-Funktion übergeben.

Da stellt sich die Frage, wie mit der vorherigen Lösung ein richtiges Ergebnis erzielt werden konnte. Die Einstellung des Parameters `dwSize` für jeden Eintrag im Array hätte keinen Unterschied machen dürfen, wenn tatsächlich nur ein Eintrag übergeben wurde.

Ehrlich gesagt, ich weiß es nicht. Vielleicht haben wir einfach nur Schwein gehabt. Vielleicht enthielt der unmittelbar an die temporäre Struktur angrenzende Speicher einfach übriggebliebene Daten von einem vorherigen Aufruf, die der Datensatzgröße entsprachen. Mit anderen Worten, in den meisten Fällen stehen die Chancen für einen Misserfolg bestens – besonders dann, wenn das Telefonbuch mehr als zwei Einträge enthält.

Und was nun?

Eine Lösung wäre es, das Array der Strukturen explizit in einen Speicherpuffer zu kopieren.[2]

```
Private Declare Function RasEntryToMemBuffer Lib "kernel32" Alias _
"RtlMoveMemory" (dest As Byte, source As RASENTRYNAME, _
ByVal count As Long) As Long
Private Declare Function RasEntryFromMemBuffer Lib "kernel32" _
Alias "RtlMoveMemory" (dest As RASENTRYNAME, source As Byte, _
ByVal count As Long) As Long

Dim TransferBuffer() As Byte

For CurrentEntry = 0 To RasEntryCount - 1
        RasEntryBuffer(CurrentEntry).dwSize = RasEntrySize
        Call RasEntryToMemBuffer(TransferBuffer( _
        CurrentEntry * RasEntrySize), RasEntryBuffer(CurrentEntry), _
        RasEntrySize)
Next CurrentEntry
```

2. Eine andere Lösung wäre, das Feld `szEntryName` der Struktur `RASENTRYNAME` von einer Zeichenfolge mit fester Länge in ein Bytearray zu verwandeln. Nun muss Visual Basic zwar keine temporäre Struktur erstellen, doch dann ist es erforderlich, dass Sie explizite ANSI-zu-Unicode-Zeichenfolgenkonvertierungen für das Feld `szEntryName` durchführen.

Warum müssen Sie zum Kopieren eine separate Funktion erstellen und können `RtlMoveMemory` nicht direkt verwenden? In Wirklichkeit müssen Sie das gar nicht. Mit diesem Ansatz wird der Prozess nur anschaulicher und sicherer. Sie hätten `RtlMoveMemory` ebensogut direkt mit den Parametern `As Any` deklariert verwenden können. Aber Sie hätten nicht die Parameter `ByVal As Long` deklarieren und den Operator `VarPtr` verwenden können. Warum? Weil der Operator `VarPtr` die Adresse der Struktur im Speicher zurückgeben würde, und das wäre die Visual Basic-Struktur mit ihrer internen Unicode-Zeichenfolge.

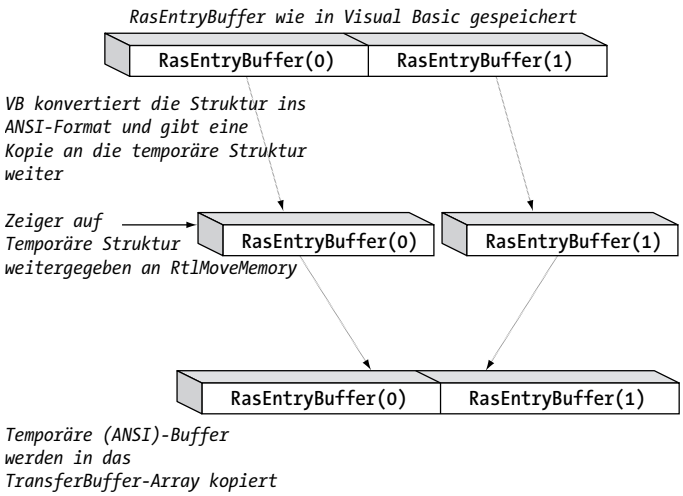


Abbildung S18-2 Laden von Strukturen in ein Bytearray

Bei Aufruf der Funktion `RasEntryToMemBuffer` wird der angegebene `RasEntryBuffer`-Eintrag in ANSI konvertiert und ein Zeiger auf diese temporäre Struktur an die Funktion `RtlMoveMemory` übergeben, die anschließend die Daten an die angegebene Position im `TransferBuffer`-Bytearray kopiert. Nach vollständigem Laden des `TransferBuffer`-Bytearrays kann die Funktion `RasEnumEntries`, wie in Abbildung S18-2 dargestellt, mit den richtigen Daten aufgerufen werden.

Nachdem `RasEnumEntries` das `TransferBuffer`-Array mit den Telefonbucheinträgen geladen hat, kann der Prozess umgekehrt werden. Mit Hilfe der Funktion `RasEntryFromMemBuffer` werden die Daten in das `RasEntryBuffer`-Array zurückkopiert. Das endgültige Programm sieht folgendermaßen aus:

```
' RasTest #1
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten

Option Explicit
```

```
Private Declare Function RasEntryToMemBuffer Lib "kernel32" Alias _
"RtlMoveMemory" (dest As Byte, source As RASENTRYNAME, _
ByVal count As Long) As Long
Private Declare Function RasEntryFromMemBuffer Lib "kernel32" _
Alias "RtlMoveMemory" (dest As RASENTRYNAME, source As Byte, _
ByVal count As Long) As Long

Private Sub Form_Load()
   Dim BufferSize As Long
   Dim RasEntries As Long
   Dim RasEntryBuffer() As RASENTRYNAME
   Dim RasEntrySize As Long
   Dim Res As Long
   Dim CurrentEntry As Integer
   Dim RasEntryCount As Long
   Dim TransferBuffer() As Byte

   ReDim RasEntryBuffer(0)
   RasEntrySize = Len(RasEntryBuffer(0))
   RasEntryBuffer(0).dwSize = RasEntrySize

   Res = RasEnumEntries(0, vbNullString, VarPtr(RasEntryBuffer(0)), _
         BufferSize, RasEntries)

' Jetzt wird der Vorgang so oft wiederholt, wie nach Anzahl der Einträge
' erforderlich
   If (Res = 0 Or Res = 603) And BufferSize > 0 Then
      RasEntryCount = BufferSize / RasEntrySize
      ReDim RasEntryBuffer(RasEntryCount - 1)
      ReDim TransferBuffer(BufferSize)
      For CurrentEntry = 0 To RasEntryCount - 1
         RasEntryBuffer(CurrentEntry).dwSize = RasEntrySize
         Call RasEntryToMemBuffer(TransferBuffer(CurrentEntry * _
         RasEntrySize), RasEntryBuffer(CurrentEntry), RasEntrySize)
      Next CurrentEntry
      Res = RasEnumEntries(0, vbNullString, VarPtr(TransferBuffer(0)), _
            BufferSize, RasEntries)
   End If

   If Res <> 0 Then
      MsgBox "RAS Error #" & Res, vbOKOnly, "Error"
```

```
    End If

    For CurrentEntry = 0 To RasEntries - 1
        Call RasEntryFromMemBuffer(RasEntryBuffer(CurrentEntry), _
        TransferBuffer(CurrentEntry * RasEntrySize), RasEntrySize)
        lstEntries.AddItem RasEntryBuffer(CurrentEntry).szEntryName
    Next CurrentEntry
End Sub
```

Schlussfolgerung

Nachdem ich einen letzten Drink mit DT genommen hatte, kehrte ich in mein Büro zurück. Ich hatte ein hartes Stück Arbeit hinter mir und wie so oft keinen zahlenden Klienten, dem ich dafür eine Rechnung schreiben konnte. Ich warf einen Blick in meine E-Mail und schmunzelte. Es war ein Auftrag. DT hatte sich mit ihrem Klienten in Hollywood in Verbindung gesetzt und ihm erklärt, dass sie mit ein wenig Aufwand ein Programm erstellen könnte, mit dem sein Computer nicht nur in der Lage sein würde, ihren Computer sowie die Computer aller seiner Freunde anzurufen, sondern das es sogar die Basis einer neuen Sitcom bilden könnte, in der Computerfamilien sich gegenseitig anrufen und dabei gelegentlich von dem spinnerten alten Atari von nebenan unterbrochen werden. Als er erfuhr, dass sie ein Programm entwickelte, stellte er keine Rückfragen. Er erteilte sofort einen Auftrag, und DT heuerte mich als Subunternehmer an, um die Einwahlsoftware zum Laufen zu bringen.

Ich denke, der Tag hatte sich alles in allem gelohnt.

Und ich hatte nicht einmal an einem Meeting teilnehmen müssen.

Lösung 19

Die RASDIALPARAMS-Struktur

Werfen wir zu Beginn erneut einen Blick auf die Struktur:

```
typedef struct _RASDIALPARAMS {
   DWORD  dwSize;
   TCHAR  szEntryName[RAS_MaxEntryName + 1];
   TCHAR  szPhoneNumber[RAS_MaxPhoneNumber + 1];
   TCHAR  szCallbackNumber[RAS_MaxCallbackNumber + 1];
   TCHAR  szUserName[UNLEN + 1];
   TCHAR  szPassword[PWLEN + 1];
   TCHAR  szDomain[DNLEN + 1] ;
#if (WINVER >= 0x401)
   DWORD  dwSubEntry;
   DWORD  dwCallbackId;
#endif
}  RASDIALPARAMS;
```

Zwei Dinge sind klar: Zuerst müssen wir die Werte einiger Konstanten herausfinden. Anschließend müssen wir überlegen, wie wir mit den Feldern `dwSubEntry` und `dwCallbackId` umgehen. Was bedeutet der folgende Ausdruck?

```
#if (WINVER >= 0x401)
   DWORD  dwSubEntry;
   DWORD  dwCallbackId;
#endif
```

Das Schlüsselwort `#if` leitet einen bedingt zu kompilierenden Block ein, ähnlich wie bei der von Visual Basic unterstützten bedingten Kompilierung. Es gibt an, dass alles Folgende bis zum Schlüsselwort `#endif` nur dann berücksichtigt wird, wenn die Kompilierungszeitkonstante `WINVER` größer als bzw. gleich &H401 ist. In Version 4.01 von Windows wurden dieser Struktur zwei neue Felder hinzugefügt. Woher weiß die Win32-API, ob diese beiden Felder vorhanden sind? Sie erfährt es durch das Feld `dwSize`! Diese beiden Felder erhöhen die Gesamtgröße der Struktur um acht Bytes.

Wie sieht es mit der `WINVER`-Konstante aus? Von welchem Wert sollten Sie bei der Übersetzung von Strukturen in Visual Basic ausgehen? Die Berücksichtigung dieses Wertes ist wichtig, denn wenn Sie eine Struktur einer höheren Version verwenden, ist sie in Verbindung mit älteren Betriebssystemversionen nicht einsetzbar. Ein C++-Programmierer wählt vor der Kompilierung der Anwendung für diese Konstante einen Wert, der der ältesten Betriebssystemversion gerecht wird, die

von der Anwendung unterstützt werden soll. Für Visual Basic sollten Sie den Wert vor Erstellen der Strukturdeklarationen wählen.

Die Variable WINVER ist eine 2-Byte-Zahl, in der das hohe Byte die Hauptversion von Windows angibt. Es ist 3 für Windows NT 3.*x*, 4 für Windows 95 und Windows NT 4 und 5 für Windows NT 5.*x*. Das niedrige Byte gibt die Unterversionsnummer an.

Was ist denn nun genau Version 4.01 von Windows? Ehrlich gesagt, ich weiß es nicht. Dieser bedingte Wert tritt nur an wenigen Positionen der C-Headerdateien auf, meistens in den RAS-Deklarationen. Ich vermute, dass er für ein Service Pack gültig ist, oder vielleicht ist er ein Indiz für ein noch erscheinendes Service Pack. Wie dem auch sei, unter den gegebenen Umständen sollten wir sicherheitshalber davon ausgehen, dass WINVER gleich 0x400 ist. Damit werden die Felder dwSubEntry und dwCallbackID ausgelassen, und es wird damit sichergestellt, dass die Struktur unter Windows NT 4 und Windows 95/98 gültig ist.

Dies ist eine wichtige Entscheidung, denn, wie Sie an den Konstantendeklarationen sehen, ändert sich die Größe ebenfalls in Abhängigkeit von der Windows-Version:

```
#if (WINVER >= 0x400)
#define RAS_MaxEntryName      256
#define RAS_MaxDeviceName     128
#define RAS_MaxCallbackNumber RAS_MaxPhoneNumber
#else
#define RAS_MaxEntryName      20
#define RAS_MaxDeviceName     32
#define RAS_MaxCallbackNumber 48
#endif
```

In unserem Fall hängt die Größe davon ab, ob die Variable WINVER größer als oder gleich 0x400 ist. Wird WINVER also auf 4.0 gesetzt, bedeutet dies, dass das Beispielprogramm unter NT 3.*x* nicht ordnungsgemäß ausführbar ist.

Basierend auf den Headerdateien können Sie die folgenden Konstantendeklarationen in Visual Basic erstellen. Die Konstanten mit dem Puffersuffix sind um 1 größer als die Basiskonstante, da die Zeichenfolgendeklarationen in der Struktur RASDIALPARAMS um 1 Byte größer sind als der Konstantenwert (um für das abschließende NULL-Zeichen Raum zu lassen).

```
Public Const RAS_MaxEntryName = 256
Public Const RAS_MaxEntryNameBuffer = 257
Public Const RAS_MaxPhoneNumber = 128
```

```
Public Const RAS_MaxPhoneNumberBuffer = 129
Public Const UNLEN = 256
Public Const UNLENBuffer = 257
Public Const PWLEN = 256
Public Const PWLENBuffer = 257
Public Const DNLEN = 15
Public Const DNLENBuffer = 16
```

RASDIALPARAMS sieht am Ende so aus:

```
Public Type RASDIALPARAMS
   dwSize As Long ' 4
   szEntryName As String * RAS_MaxEntryNameBuffer ' 257
   szPhoneNumber As String * RAS_MaxPhoneNumberBuffer '129
   szCallbackNumber As String * RAS_MaxPhoneNumberBuffer ' 129
   szUserName As String * UNLENBuffer ' 257
   szPassword As String * PWLENBuffer ' 257
   szDomain As String * DNLENBuffer '16
   Padding(2) As Byte
   'dwSubEntry As Long
   'dwCallbackId As Long
   '#if (WINVER >= 0x401)
   'DWORD  dwSubEntry;
   'DWORD  dwCallbackId;
   '#End If
End Type
```

Was at es nun mit dem zusätzlichen Füllarray am Ende der Struktur auf sich? Bei dieser Struktur ist dasselbe Problem zu beachten wie bei der Struktur RASENTRY-NAME in Puzzle 18. Das Packing ist auf 4 Bytes eingestellt. Bei den einzelnen Feldern innerhalb der Struktur treten keine Anordnungsprobleme auf, da sie alle aus Arrays einzelner Bytes bestehen, die sich an jeder Adresse befinden können. Die Struktur als Ganzes muss jedoch an einer 32-Bit-Grenze angeordnet werden. Wenn Sie die Längen aller Einträge addieren, erhalten Sie 4 + 257 + 129 + 129 + 257 + 257 + 16 = 1049. Die 32-Bit-Grenze ist 1052, Sie benötigen also noch 3 Bytes zum Auffüllen.

Mit dieser Struktur ist die Funktion fast verwendbar. An Stelle der Fehlermeldung 632 (ungültige Strukturgröße) beim Start erhalten Sie die Fehlermeldung 623 (ungültiger Telefonbucheintrag), wenn Sie versuchen, einen Telefonbucheintrag auszuwählen. Wie kann das sein?

Beachten Sie das Feld `szEntryName` in der Struktur. Es handelt sich um eine Zeichenfolge mit fester Länge, die mit folgender Zeile festgelegt wird:

```
rs.szEntryName = lstEntries.Text
```

Nehmen wir an, der Telefonbucheintrag sei »MyISP.« Das Listenfeld wurde beim Laden auf die Zeichenfolge »`MyISP`« gefolgt von einem NULL-Zeichen gesetzt. Doch das NULL-Zeichen ist nicht im Listenfeld gespeichert! Warum? Weil die Listensteuerung wie die meisten Steuerungen in Windows in C geschrieben ist und C-Zeichenfolgen verwendet. So wurde das abschließende NULL-Zeichen beim Speichern der Zeichenfolge im Listenfeld weggelassen.

Was geschieht, wenn Sie eine Zeichenfolge von fünf Zeichen einer 257 Zeichen langen Zeichenfolge mit fester Länge zuweisen? Visual Basic füllt den Rest der Zeichenfolge mit Leerzeichen! Wenn die API-Funktion versucht, den Telefonbucheintrag mit einer Zeichenfolge zu vergleichen, die den Eintrag gefolgt von einer Reihe Leerzeichen enthält, schlägt der Vergleich fehl, und der Eintrag wird nicht gefunden. Damit die Funktion einsetzbar ist, müssen Sie, wie nachstehend gezeigt, explizit ein abschließendes NULL-Zeichen hinzufügen:

```
rs.szEntryName = lstEntries.Text & Chr$(0)
```

Das ist alles.

Vielleicht fällt Ihnen auf, dass die Funktion `RasGetEntryDialParams` keine weiteren Informationen zurückgibt. Das liegt daran, dass die Funktion Informationen vom letzten erfolgreichen Einwahlvorgang zurückgibt, die von den im Telefonbuch gespeicherten Informationen abweichen. Sind die Informationen identisch, ist die zurückgegebene Zeichenfolge leer.

Lösung 20

Verbindungsherstellung

Einer meiner Grundsätze für alle diese Puzzle ist, dass sie fair sein müssen. Hart, aber fair. Und ich wette, dass einige unter Ihnen denken, dass dieses Puzzle eben gerade nicht fair ist. Ich habe Ihnen einen Block C-Code präsentiert (nachstehend wiederholt, um Ihnen nerviges Blättern zu ersparen), ihn als Aufzählung bezeichnet und absolut keine Hilfestellung zur Konvertierung in Visual Basic geboten:

```
#define RASCS_PAUSED 0x1000
#define RASCS_DONE   0x2000

#define RASCONNSTATE enum tagRASCONNSTATE
RASCONNSTATE
{
    RASCS_OpenPort = 0,
    RASCS_PortOpened,
    RASCS_ConnectDevice,
    RASCS_DeviceConnected,
    RASCS_AllDevicesConnected,
    RASCS_Authenticate,
    RASCS_AuthNotify,
    RASCS_AuthRetry,
    RASCS_AuthCallback,
    RASCS_AuthChangePassword,
    RASCS_AuthProject,
    RASCS_AuthLinkSpeed,
    RASCS_AuthAck,
    RASCS_ReAuthenticate,
    RASCS_Authenticated,
    RASCS_PrepareForCallback,
    RASCS_WaitForModemReset,
    RASCS_WaitForCallback,
    RASCS_Projected,

#if (WINVER >= 0x400)
    RASCS_StartAuthentication,
    RASCS_CallbackComplete,
    RASCS_LogonNetwork,
#endif
    RASCS_SubEntryConnected,
```

```
    RASCS_SubEntryDisconnected,

    RASCS_Interactive = RASCS_PAUSED,
    RASCS_RetryAuthentication,
    RASCS_CallbackSetByCaller,
    RASCS_PasswordExpired,

    RASCS_Connected = RASCS_DONE,
    RASCS_Disconnected
} ;
```

Es stimmt, dass ich keine Hilfestellung zur Interpretation dieses Codes geboten habe, weder im Puzzle noch in den Tutorien. Aber unfair? Ich glaube nicht. Dieses Buch soll Sie unter anderem dazu ermutigen, ja sogar zwingen, bei der Konfrontation mit kniffligen Problemen kreativ zu denken. Da es unmöglich ist, in diesem Buch (oder in irgendeinem anderen) jedes API-Problem zu behandeln, dem sie in Visual Basic gegenüberstehen, ist es um so wichtiger, dass Sie lernen, Ihre bereits vorhandenen Kenntnisse bei der Lösung neuer Probleme heranzuziehen.

Vielleicht wissen Sie nichts über Aufzählungen in C, doch sollten Sie schon ein wenig über Aufzählungen in Visual Basic wissen, die mit Hilfe der Anweisung `Enum` erstellt werden. In einer Aufzählung in Visual Basic empfängt jede Variable innerhalb der Aufzählung entweder einen explizit zugewiesenen Wert oder einen Wert, der größer ist als der Wert der vorherigen Variablen. Die C-Struktur, die Sie hier sehen, verwendet außerdem das Schlüsselwort `enum`. Einige der Variablen werden explizit zugewiesen, andere verfügen über keine Zuweisung. Das kommt Ihnen bekannt vor? Werfen Sie einen Blick auf den folgenden Visual Basic-Code, und vergleichen Sie ihn mit dem C-Code:

```
Private Const RASCS_PAUSED = &H1000
Private Const RASCS_DONE = &H2000

Private Enum RasConnState
    RASCS_OpenPort = 0
    RASCS_PortOpened
    RASCS_ConnectDevice
    RASCS_DeviceConnected
    RASCS_AllDevicesConnected
    RASCS_Authenticate
    RASCS_AuthNotify
    RASCS_AuthRetry
    RASCS_AuthCallback
```

```
    RASCS_AuthChangePassword
    RASCS_AuthProject
    RASCS_AuthLinkSpeed
    RASCS_AuthAck
    RASCS_ReAuthenticate
    RASCS_Authenticated
    RASCS_PrepareForCallback
    RASCS_WaitForModemReset
    RASCS_WaitForCallback
    RASCS_Projected
    RASCS_StartAuthentication
    RASCS_CallbackComplete
    RASCS_LogonNetwork
    RASCS_SubEntryConnected
    RASCS_SubEntryDisconnected
    RASCS_Interactive = RASCS_PAUSED
    RASCS_RetryAuthentication
    RASCS_CallbackSetByCaller
    RASCS_PasswordExpired
    RASCS_Connected = RASCS_DONE
    RASCS_Disconnected
End Enum
```

Sehen Sie? Der Code ist tatsächlich identisch, von geringfügigen, durch die Sprachsyntax bedingten Unterschieden einmal abgesehen.

Hier wird ein wichtiger Aspekt deutlich: Sprachen lassen sich in der Regel in bestimmte Klassifikationen einteilen. Visual Basic und C zählen beide zur Klasse der Blockstruktursprachen. Sie weisen zahlreiche Gemeinsamkeiten auf. Eines der wichtigsten Dinge, die Sie wie jeder andere Visual Basic-Programmierer lernen müssen, wenn Sie sich mit der Microsoft-Dokumentation befassen, ist zu erkennen, dass vieles, was Sie bereits wissen, direkt anwendbar ist. Mit ein wenig Raten und Experimentieren können Sie die meisten API-Probleme lösen.

Damit kommen wir zum nächsten Problem – der Rückruffunktion. Wenn Ihnen die Aufzählung gelungen ist und Sie die Anwendung ausführen konnten, war das Ergebnis wahrscheinlich ein sofortiger Speicherausnahmefehler. Warum?

Schauen wir noch einmal auf die Beschreibung von RasDialFunc:

```
VOID WINAPI RasDialFunc(
    UINT unMsg,                    // Aufgetretener Ereignistyp
    RASCONNSTATE rasconnstate,     // Noch einzugebender Verbindungsstatus
    DWORD dwError                  // Eventuell aufgetretener Fehler
   );
```

Vielleicht sind Sie mit Rückruffunktionen an sich nicht vertraut, doch Sie sollten es schon einigermaßen mit dem sein, was eine API-Funktion in diesem Fall erwartet. Denken Sie einen Augenblick daran, dass es sich hierbei um eine exportierte Funktion handelte. Wie würden Sie eine solche deklarieren? Die API-Funktion würde erwarten, die tatsächliche Meldung, rasconnstate und dwError-Werte auf dem Stack zu finden, da es sich bei diesen Parametern nicht um Zeiger handelt. Dies bedeutet, Sie würden ByVal in der Deklaration all dieser Parameter verwenden.

```
Public Declare RasDialFunc Lib "some library" (ByVal unMsg As Long, _
ByVal rasconnstate As Long, ByVal dwError As Long) As Long
```

Gemäß der C-Funktionsdeklaration gibt die Funktion einen Void-Wert zurück, was keiner Werterückgabe entspricht. Sie könnten hier anstelle einer Funktionsdeklaration eine Sub-Deklaration verwenden, doch können Sie gleichermaßen konsistent bleiben und eine Funktion deklarieren, die einen Long-Wert zurückgibt. Ignorieren Sie einfach das Ergebnis.

Wie würden Sie eine Public-Funktion in einem Standardmodul definieren, sodass sie den Wert vom Stack übernimmt? Exakt in gleicher Weise. Die Funktion muss folgendermaßen definiert werden:

```
Public Function RasFunc(ByVal unMsg As Long, ByVal RasConnState As _
Long, ByVal dwError As Long) As Long
```

Heißt das nun, das alles, was Sie mit einer Rückruffunktion tun müssen, sich darauf reduziert, dass Sie eine Declare-Anweisung erstellen und in eine reguläre Funktion ändern? Solange Sie mit numerischen Variablen arbeiten, ja. Doch dieser Ansatz ist bei Zeichenfolgen und Strukturen nicht anwendbar. API-Funktionen, die Rückrufe verwenden, übergeben Zeichenfolgen oder Strukturen als Zeiger an eine Zeichenfolge oder Struktur. Sie sollten Ihren Rückruffunktionsparameter ByVal As Long deklarieren und die Variable, die Sie empfangen, als Zeiger interpretieren. Dann können Sie die Daten mit Hilfe der Funktionen lstrcpy oder RtlMoveMemory in eine lokale Zeichenfolge oder Struktur kopieren.

Lösung 21

Welches ist das zugeordnete Laufwerk? Teil 1

War es ein Windows-bug, ein Bug im Projekt `MapInfo1` oder ein Fehler in der Dokumentation?

Das hängt von Ihrer Sichtweise und von Ihrer Interpretation der Dokumentation ab.

Der Kern des Problems liegt im folgenden Code aus der Funktion `GetMappedInfo` im Projekt **MapInfo1.vbp** (auf der Begleit-CD-ROM) :

```
res = WNetGetConnection(Drive, Buffer, BufferLength)
   If res = ERROR_MORE_DATA Then
      Buffer = String$(BufferLength, 0)
      res = WNetGetConnection(Drive, Buffer, BufferLength)
   End If
```

Die API-Dokumentation sagt Folgendes über den Parameter `BufferLength`:

Wenn die Funktion fehlschlägt, weil der Puffer zu klein ist, gibt dieser Parameter die erforderliche Puffergröße zurück.

Eine Interpretation dieses Kommentars ist, dass die zurückgegebene Puffergröße für die erfolgreiche Ausführung der Funktion ausreicht. Mit anderen Worten, sobald Sie diese Funktion aufrufen, um die richtige Puffergröße zu erhalten, sollte die Funktion nicht mehr den Rückgabewert `ERROR_MORE_DATA` zurückgeben können. Ich denke, dass die meisten vernünftigen Leute dies so wie ich sehen.

Aber Sie könnten dies auch so interpretieren, dass die erforderliche Puffergröße für den nächsten Versuch zurückgegeben wird. Wenn die Puffergröße immer noch nicht groß genug ist, gibt die Funktion das Ergebnis `ERROR_MORE_DATA` erneut mit einer neuen Puffergröße zurück, die hoffentlich ausreicht, alle Daten aufzunehmen.

Nun gut, vielleicht ist diese zweite Interpretation ein wenig gezwungen und nicht besonders intuitiv. Sie entspricht jedoch dem Verhalten dieser Funktion. Die Funktion `GetMappedInfo` im Projekt **MapInfo1.vbp** kann folgendermaßen neu geschrieben werden, um mehrere Versuche durchführen zu können:

```
Do
   Buffer = String$(BufferLength, 0)
   res = WNetGetConnection(Drive, Buffer, BufferLength)
Loop While res = ERROR_MORE_DATA
```

BufferLength beginnt bei 0 mit einem leeren Zeichenfolgenparameter Buffer. Jedesmal, wenn die Funktion WNetGetConnection den Wert ERROR_MORE_DATA zurückgibt, wird die Pufferlänge auf die aktuelle geschätzte Pufferlänge erhöht und ein weiterer Versuch unternommen, die Funktion aufzurufen.

Die Microsoft-Win32-API-Dokumentation ist die Standardreferenz für API-Funktionsaufrufe. Doch auch bei sorgfältiger Lektüre werden Sie wahrscheinlich gelegentlich experimentieren müssen, um exakt bestimmen zu können, was der Funktionsaufruf tatsächlich bewirkt.

Der zweite Lösungsweg

Ein weiterer gängiger Ansatz zur Lösung dieses Problems ist potenziell effizienter. Setzen Sie einfach für die Zeichenfolge Buffer einen sehr großen Wert, bevor Sie die API-Funktion WNetGetConnection zum ersten Mal aufrufen. Sie möchten vielleicht für alle Fälle testen, ob der ERROR_MORE_DATA-Fehler auftritt, obgleich das Risiko, dass dieser Fehler auftritt, sehr gering ist, wenn die Länge der Zeichenfolge auf mindestens MAX_PATH Bytes gesetzt ist (Konstante MAX_PATH ist gleich 260).

Wenn Sie diesen Ansatz wählen, müssen Sie das Ende der Zeichenfolge mit Hilfe der Funktion Instr suchen, um das abschließende NULL-Zeichen zu finden und alles zu entfernen, was zwischen diesem Zeichen und dem Ende der Zeichenfolge liegt. Im Beispiel **MapInfo1.vbp** wird die Variable BufferLength mit der Länge des Puffers geladen. Da das letzte Zeichen das abschließende NULL-Zeichen ist, kann die Funktion Left$ verwendet werden, um die Zeichenfolge links von diesem Zeichen abzurufen.

Lösung 22

Welches ist das zugeordnete Laufwerk? Teil 2

Wenn Sie den Code betrachten, werden Sie feststellen, dass der Absturz in dieser Zeile geschieht:

```
Call RtlMoveMemory(nr, OutputBuffer(0), Len(nr))
```

Was könnte diesen Absturz verursacht haben? Der Parameter `nr` ist richtig – es handelt sich um eine als Verweis übergebene `NETRESOURCE`-Struktur. `OutputBuffer(0)` ist das erste Byte im Bytearray. Als Verweis übergeben, entspricht es einem Zeiger an das erste Byte im Array, was ebenfalls richtig ist. Die Anzahl der übergebenen Bytes entspricht der Länge der `NETRESOURCE`-Struktur. Sie wissen, dass das Array `OutputBuffer` mindestens so groß ist wie die `NETRESOURCE`-Struktur, weil es gemäß Definition die Struktur enthält, darum sollte die Funktion `WNetGetResourceInformation` eine erforderliche Größe zurückgeben, die mindestens dieser Größe entspricht. Beim Überprüfen der Variablen `OutputBufferSize` sehen Sie, dass sie definitiv größer ist als eine `NETRESOURCE`-Struktur.

Dies bedeutet, dass das Problem innerhalb der Struktur liegen muss.

Wenn Sie die `NETRESOURCE`-Struktur mit Hilfe von `RtlMoveMemory` kopieren, wird die Variable `nr` mit den exakten Daten in den Puffer geladen. Die als `Long` definierten Felder stellen kein Problem dar. Aber wie sieht es mit den Zeichenfolgenvariablen aus? Wenn sie als Zeichenfolgen definiert sind (wie in diesem Beispiel), erwartet Visual Basic, dass alle diese Felder einen `BSTR`-Zeiger auf eine Unicode-Zeichenfolge enthalten. Die `NETRESOURCE`-Struktur definiert sie jedoch einfach als Zeichenfolgenzeiger, und da wir die ANSI-Version der Funktion `WNetGetResourceInformation` verwenden, handelt es sich um Zeiger auf mit NULL abgeschlossene ANSI-Zeichenfolgen.

Nicht die ANSI-zu-Unicode-Konvertierung bereitet uns hier Ärger, es ist die Tatsache, dass diese Zeiger definitiv keine BSTR-Zeiger sind. Visual Basic erkennt diese Werte und wird so gut wie sicher sofort abstürzen. Die Länge der Zeichenfolge wird vor den Daten erwartet. Welche Werte werden in diesem Fall gelesen? Wer weiß das schon? Aber Sie können sicher sein, dass es nicht die aktuelle Länge der Zeichenfolge sein wird. (Weitere Informationen zu diesem Problem finden Sie in Tutorium 7, »Klassen, Strukturen und benutzerdefinierte Typen,« in Teil III dieses Buches.)

Die Lösung besteht darin, den Datentyp dieser Parameter wie folgt in `Long` zu ändern:

```
Private Type NETRESOURCE
        dwScope As Long
        dwType As Long
        dwDisplayType As Long
        dwUsage As Long
        lpLocalName As Long
        lpRemoteName As Long
        lpComment As Long
        lpProvider As Long
End Type
```

Dabei entstehen zwei neue Probleme.

Zunächst müssen Sie vor dem Aufruf der Funktion `WNetGetResourceInformation` den Parameter `lpRemoteName` als Zeiger auf eine auf NULL endende ANSI-Zeichenfolge festlegen. Dies bedeutet, dass Sie einen ANSI-Zeichenfolgenpuffer erstellen und seine Adresse explizit ermitteln müssen.

Eine der einfachsten Methoden zur Erstellung von Speicherpuffern besteht in der Verwendung von Bytearrays.

Hier sehen Sie ein Codesegment, dass ein dynamisch zugeordnetes Bytearray mit einer auf NULL endenden ANSI-Zeichenfolge lädt:

```
' Laden eines dynamisch zugeordneten Bytearrays mit einer ANSI-Zeichen-
folge.
Private Sub CreateAnsiArray(StringVar As String, ByteArray() As Byte)
   ReDim ByteArray(Len(StringVar))
   Call lstrcpy(VarPtr(ByteArray(0)), StringVar)
End Sub
```

Hier wird eine raffinierte Technik angewandt.

Erstens, was ist die exakte Länge des auf die Operation `ReDim` folgenden Bytearrays?

Ist das Ergebnis der Operation `Len(StringVar)` gleich 5, wird der Parameter `ByteArray` mit einem Index von 5 neu dimensioniert. Doch in Visual Basic wird in diesem Fall ein 6 Byte langes Bytearray erstellt! Das zusätzliche Byte soll das abschließende NULL-Zeichen aufnehmen.

Die Funktion `lstrcpy` wird folgendermaßen definiert:

```
Private Declare Function lstrcpy Lib "kernel32" Alias "lstrcpyA" _
(ByVal lpString1 As Long, ByVal lpString2 As String) As Long
```

Die Funktion `lstrcpy` hat die Aufgabe, Zeichenfolgen von einem Quellzeiger in einen Zielzeiger zu kopieren. Der Zielzeiger (`lpString1`) wird als Long angegeben, sodass wir den ermittelten Zeiger mit Hilfe des Operators `VarPtr` übergeben können. Der Parameter `lpString2` wird `ByVal` als Zeichenfolge deklariert. Was geschieht, wenn Sie eine VB-Zeichenfolge mit Hilfe des Operators `ByVal` an eine API-Funktion übergeben? Sie wird in eine auf NULL endende ANSI-Zeichenfolge konvertiert! – das ist genau das, was die Funktion `lstrcpy` als Parameter erwartet.

Mit anderen Worten, anstatt eine explizite Unicode-zu-ANSI-Konvertierung zu verwenden, mogelt diese Funktion mit der Verwendung der Unicode-zu-ANSI-Konvertierung, die Visual Basic automatisch bei der Übergabe von Zeichenfolgen an API-Funktionen durchführt.

Das Feld `nr.lpRemoteName` kann mit Hilfe des folgenden Codes unter Verwendung eines dynamisch zugeordneten Bytearrays `ResourceNameArray` gefüllt werden:

```
Call CreateAnsiArray(resourcename, ResourceNameArray)
   nr.lpRemoteName = VarPtr(ResourceNameArray(0))
```

Die Funktion `RtlMoveMemory` wird nun ohne Absturz ausgeführt. Sie sehen dies in Abbildung S22-1, die den von der Funktion `WNetGetResourceInformation` geladenen Inhalt des Puffers zeigt. Der Puffer beginnt mit einer `NETRESOURCE`-Struktur, gefolgt von den Zeichenfolgendaten. Die Zeiger in der Struktur enthalten – durch die Pfeile gekennzeichnet – die Adressen dieser Zeichenfolgen im Puffer. Wenn Sie den ersten Teil des Puffers in eine Variable des Typs `NETRESOURCE` kopieren, weisen die Zeiger immer noch richtig auf die Zeichenfolgendaten im ursprünglichen Puffer.

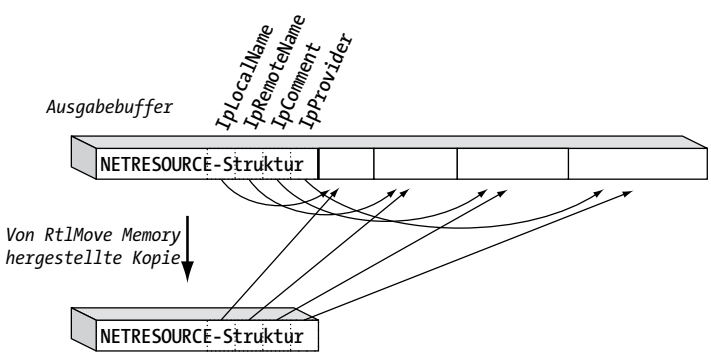


Abbildung S22-1 Die Zeichenfolgenadressen im Puffer bleiben nach dem Aufruf der Funktion `RtlMoveMemory` gültig

Jetzt müssen die Zeichenfolgendaten nur noch in eine Form konvertiert werden, auf die Visual Basic zugreifen kann.

Dazu dient die folgende Funktion:

```
' Abrufen einer VB-
Zeichenfolge unter Verwendung eines Zeigers auf eine auf NULL endende Zeich
enfolge
Private Function GetVBString(ByVal ptr As Long)
   Dim StringLength As Long
   Dim TempString As String
   If ptr = 0 Then Exit Function
   StringLength = lstrlen(ptr)
   TempString = String$(StringLength + 1, 0)
   Call lstrcpystring(TempString, ptr)
   GetVBString = Left$(TempString, StringLength)
End Function
```

Bei einem Zeiger können Sie mit der Funktion `lstrlen` die Länge der Zeichenfolge (ohne NULL-Zeichen) ermitteln. Dann können Sie die Zeichenfolge mit der erforderlichen Länge vorinitialisieren. Diese Funktion kehrt die in der Funktion `CreateAnsiArray` gezeigte Operation um, indem sie die Funktion `lstrcpy` verwendet und eine Visual Basic-Zeichenfolge als erste Variable sowie einen Zeiger als zweite Variable übergibt. Hierzu ist eine andere Deklaration für `lstrcpy` erforderlich, sodass wir einen neuen Alias für die Funktion erstellen, der folgendermaßen deklariert wird:

```
Private Declare Function lstrcpystring Lib "kernel32" Alias "lstrcpyA" _
(ByVal lpString1 As String, ByVal lpString2 As Long) As Long
```

Schließlich wird das abschließende NULL-Zeichen aus der Zeichenfolge entfernt.

Die Funktion `agGetStringFromPointer` aus der Bibliothek **apigid32.dll** (in Anhang C dieses Buches enthalten) führt dieselbe Operation durch und ist weitaus effizienter, da sie mit einem einzigen API-Aufruf arbeitet.

Daraus resultiert die folgende einsetzbare Version der Funktion `GetResourceInfo`:

```
' Netzwerkproviderinformationen abrufen
Private Sub GetResourceInfo(resourcename As String)
   Dim nr As NETRESOURCE
   Dim ResourceNameArray() As Byte
   Dim lpSystem As Long
   Dim OutputBufferSize As Long
   Dim OutputBuffer() As Byte
```

```
    Dim res As Long

    lstResource.Clear
    If resourcename = "" Then Exit Sub

    nr.dwType = RESOURCETYPE_DISK
    ' Erstellen eines Puffers mit dem lokalen ANSI-Namen
    Call CreateAnsiArray(resourcename, ResourceNameArray)
    nr.lpRemoteName = VarPtr(ResourceNameArray(0))

    ' Puffer vorbereiten
    OutputBufferSize = 1024
    Do
       ReDim OutputBuffer(OutputBufferSize)
       res = WNetGetResourceInformation(nr, OutputBuffer(0), _
             OutputBufferSize, lpSystem)
    Loop While res = ERROR_MORE_DATA
    If res <> 0 Then
       MsgBox "No Resource Information Available"
       Exit Sub
    End If

    ' Ergebnis-NETRESOURCE abrufen
    Call RtlMoveMemory(nr, OutputBuffer(0), Len(nr))

    lstResource.AddItem "Remote Name: " & GetVBString(nr.lpRemoteName)
    lstResource.AddItem "Provider: " & GetVBString(nr.lpProvider)
    lstResource.AddItem "System Name: " & GetVBString(lpSystem)

End Sub
```

Lösung 23

Fragen gibt es immer wieder

Zum letzten Mal eine kurze Erinnerung an die ursprüngliche Frage:

Ich versuche, die Struktur `USER_INFO_2` mit der VBA-Funktion `NetAddUser` einzusetzen. Teil dieser Struktur ist: PBYTE `usri2_logon_hours`; zeigt auf eine 21-Byte-Zeichenfolge (168 Bits), die angibt ...

Wie schließe ich dies in meine Typdeklaration ein? Darüber hinaus liegt in der C-Headerdatei eine solche Konstante vor:

```
#define USER_MAXSTORAGE_UNLIMITED ((unsigned long) -1L)
```

Ist dies kein Widerspruch?

Sollte ich einfach eine `Long`-Konstante mit dem Wert 1 deklarieren? Vielen Dank im Voraus!

Beginnen wir mit der zweiten Frage. Das Feld `usri2_max_storage` des Typs `Long` der Struktur `USER_INFO2` gibt die maximale Menge an Festplattenspeicher an, die der Benutzer auf dem System verwenden kann. Die Dokumentation schlägt vor, das Feld auf den Konstantenwert `USER_MAXSTORAGE_UNLIMITED` zu setzen, um dem Benutzer unbegrenzten Zugriff auf die Festplatte zu ermöglichen, und definiert die Konstante wie in der obigen Frage dargestellt.

Offensichtlich ergibt –1 keinen Sinn – Sie können einem Benutzer keinen Zugriff auf negativen Festplattenspeicher gewähren, es überrascht darum nicht, dass die Person, die die Frage eingesandt hat, dadurch irritiert war.

Doch erscheint die Verwendung des Wertes 1 für den gesunden Menschenverstand ebenso unsinnig, sofern Sie dem Benutzer nicht ausschließlich die Nutzung eines einzelnen Byte auf der Festplatte zugestehen möchten.

Um diese Frage beantworten zu können, muß man zunächst verstehen, wie Daten im Speicher abgelegt werden. Zunächst betrachten wir, wie Visual Basic dies mit bestimmten Daten in einer `Long`-Variable macht.

Der größtmögliche Wert, den eine Variable des Typs `Long` in Visual Basic speichern kann, ist 2147483647. Dies können Sie mit Hilfe des folgenden Codes testen:

```
Dim l As Long
l = &H7FFFFFFF
Debug.Print l
```

Ich weiß, was Sie denken: Warum ist `&H7FFFFFFF` die größte Zahl, die eine `Long`-Variable speichern kann? Was hat es mit `&HFFFFFFFF` auf sich?

Probieren Sie es aus, und sehen Sie, was geschieht.

Das Ergebnis ist –1.

Wie Sie sehen, ist eine `Long`-Variable in Visual Basic eine Vorzeichenvariable, d.h., das hohe Bit gibt an, dass die Zahl negativ ist. Wenn Visual Basic eine 32-Bit-Variable ohne Vorzeichen unterstützen würde, wäre das Ergebnis beim Laden der Variablen mit `&HFFFFFFFF` die Zahl 4 294 967 294.

So entspricht der Vorzeichenwert –1 dem Wert 4 294 967 294 ohne Vorzeichen. Beide werden identisch im Speicher unter `&HFFFFFFFF` abgelegt – der einzige Unterschied besteht in der Interpretation der Daten durch die Software.

Wenn Sie die folgende Konstantendefinition betrachten,

```
#define USER_MAXSTORAGE_UNLIMITED ((unsigned long) -1L)
```

sehen Sie, dass –1 der Ausdruck »(unsigned long)« vorangestellt ist. Was bedeutet dies? Es handelt sich um einen »Cast«. Ein Typ, der in Klammern einer Zahl oder Variablen in der C-Sprache vorangestellt ist, weist den Compiler an, die Daten diesem Typ gemäß zu interpretieren. Zwar wird die Zahl –1 angezeigt (das L zeigt an, dass es sich um eine –1 im 32-Bit-Format handelt), doch teilt der Cast dem Compiler mit, dass der Programmierer die Interpretation dieses Wertes als Wert ohne Vorzeichen wünscht, in diesem Falle als 4 294 967 294.

Für uns als Visual Basic-Programmierer ist es im Grunde unerheblich, wie der C-Compiler die Daten interpretiert. Wir sorgen nur dafür, dass der Variablen der richtige Wert zugewiesen wird. Der richtige Wert ist in diesem Fall `&HFFFFFFFF`, und der einfachste Weg, einer `Long`-Variablen in Visual Basic diesen Wert zuzuweisen, besteht darin, sie auf –1 zu setzen. Folglich lautet die richtige Deklaration:

```
Private Const USER_MAXSTORAGE_UNLIMITED = -1
```

Und jetzt das Array ...

Das Feld `usri2_logon_hours` der Struktur `USER_INFO2` kann optional auf einen Zeiger gesetzt werden, der auf ein 21-Byte (168-Bit)-Array zeigt. In der ursprünglichen Lösung verwendeten wir `Long`-Variablen für alle Felder der Struktur `USER_INFO2`. Numerische Daten wurden mit direkter Zuweisung an `Long`-Variablen behandelt. Zur Behandlung von Zeichenfolgen wurde die `Long`-Variable mit der unter Verwendung des `StrPtr`-Operators ermittelten Adresse einer Unicode-Zeichenfolge geladen. Kann das Feld `usri2_logon_hours` in ähnlicher Weise auf einen Zeiger auf ein 21-Byte-Array gesetzt werden? Ganz genau.

Erstellen Sie zunächst das Array wie folgt:

```
Dim HourList(20) As Byte
```

Beachten Sie, dass hiermit tatsächlich ein 21-Byte-Array erstellt wird, dessen Bytes von 0 bis 20 nummeriert sind. Weisen Sie dem Array als Nächstes einen Zeiger zu, der mit Hilfe des `VarPtr`-Operators wie nachstehend gezeigt, ermittelt wird:

```
ul2.usri2_logon_hours = VarPtr(HourList(0))
```

Alle Bytes im Array werden in einem zusammenhängenden Speicherblock abgelegt, sodass ein Zeiger auf das erste Byte im Array einem Zeiger auf das gesamte Array entspricht.

Arrayinhalte

Ich habe noch gar nicht erwähnt, was Sie in diesem Array speichern sollten. Dies gehört zwar nicht zur ursprünglichen Frage, verdient aber nichtsdestoweniger Beachtung, weil damit einige sehr interessante Techniken illustriert werden können, die vielen Visual Basic-Programmierern nicht geläufig sind.

Die Array-Inhalte werden in der Win32 SDK-Referenz folgendermaßen definiert:

Zeigt auf eine 21-Byte (168-Bit)-Zeichenfolge, die angibt, zu welchen Zeiten sich der Benutzer anmelden kann. Jedes Bit steht für eine bestimmte Stunde der Woche. Das erste Bit (Bit 0, Wort 0) ist Sonntag, 0.00 bis 0.59; das zweite Bit (Bit 1, Wort 0) ist Sonntag, 1.00 bis 1.59 usw. Ein Nullzeiger in diesem Element für `NetUserAdd`-Aufrufe bedeutet, dass keine zeitliche Beschränkung vorliegt. Ein Nullzeiger in diesem Element für `NetUserSetInfo`-Aufrufe bedeutet, dass keine Änderung erforderlich ist.

Dies ist eine äußerst interessante Methode zur Angabe von Zeiten. Es lohnt sich, wenn Sie darüber nachdenken. Bei 24 Stunden pro Tag und 7 Tagen pro Woche (wie üblicherweise in den meisten Gegenden außerhalb des Silicon Valley, wo man dem Tag routinemäßig noch einige Stunden hinzufügt, um genügend Zeit für Dinge wie das Schreiben dieses Buches übrig zu haben) benötigen Sie insgesamt 24 x 7 = 168 Bits, um für jede Stunde ein Bit zur Verfügung zu haben.

Wie gehen Sie vor, um einzelne Bits im Array zu setzen bzw. zu löschen?

Gehen wir einmal davon aus, dass die Tage von 1 bis 7 nummeriert sind (Sonntag bis Samstag) und die Stunden von 0 bis 23. (Dieses System verwendet die 24-Stunden-Zeit.)

Wir müssen Tag und Stunde in einen Index in einem Array sowie in eine Bitposition im indexierten Byte konvertieren.

In diesem Fall sind die Zahlen glücklicherweise ziemlich einfach zu berechnen. Jedes Byte enthält 8 Bits für 8 Stunden. Drei Bytes repräsentieren einen Tag. Wir können mittels der folgenden Gleichung das zu verwendende Byte berechnen:

```
ByteIndex = Int(Hour / 8) + ((Day - 1) * 3)
```

Der zweite Ausdruck, `((Day - 1 ) * 3)`, liefert uns die Nummer des ersten Bytes für einen bestimmten Tag (jeder Tag hat 3 Bytes). Jedes Byte nimmt 8 Stunden auf, also teilen Sie durch 8, um den Byteindex zu erhalten. Um Rundungen zu verhindern, müssen Sie die Funktion `Int` verwenden. Beispiel:

```
Montag 10.00 Uhr ist Tag 2, Stunde 10.
ByteIndex = Int(10/8) + ((2-1) * 3) = 1 + 3 = 4
```

Welches Bit entspricht dieser Uhrzeit? Wir haben den Index der Bytes mittels Division durch 8 ermittelt – im Rahmen dieser Operation muss nur noch die Bitposition ermittelt werden. Dies geschieht am einfachsten mit Hilfe des `mod`-Operators: 10 `mod` 8 ist 2, das richtige Bit ist also 2 (das dritte Bit von Bit 0 an gerechnet).

Aber die Bitnummer benötigen wir eigentlich nicht – tatsächlich benötigen wir ein Byte, in dem das angegebene Bit gesetzt ist, um das Bit mittels einer AND- bzw. OR-Operation zu löschen oder zu setzen. Da Visual Basic nicht über einen Shift-Operator verfügt, können wir den Exponenzialoperator wie folgt einsetzen:

```
ByteMask = 2 ^ Int(Hour Mod 8)
```

Jetzt wird mir eines klar: inwieweit Ihnen dies einleuchtet oder nicht, hängt davon ab, wie lange Ihr Mathematikunterricht zurückliegt. Nach meiner Erfahrung besteht die einfachste Lösung für Probleme dieser Art darin, einige Beispielwerte aufzuschreiben und zu prüfen, ob sie sinnvoll sind. Verwenden Sie Tabelle S23-1 als Vorlage für Ihre eigenen Beispiele.

Hour	Day	Int(Hour / 8)	ByteIndex	Hour Mod 8	2 ^ (Hour Mod 8)
0	1	0	0	0	1
0	2	0	3	0	1
7	2	0	3	7	128 = &H80
8	2	1	4	0	1

Tabelle S23-1 Mit Hilfe von Prüfwerten können Sie feststellen, ob eine Gleichung richtig ist

Um ein Bit zu setzen, kombinieren Sie einfach mit Hilfe des `OR`-Operators den Maskenwert mit dem vorhandenen Wert.

Um ein Bit zu löschen, benötigen Sie eine Maske, in der alle Bits mit Ausnahme des zu löschenden gesetzt sind. Sie erhalten diesen Wert, indem Sie den NOT-Operator für die Maske verwenden. Anschließend können Sie mit Hilfe des AND-Operators die Maske mit dem ursprünglichen Wert kombinieren, um das Bit zu löschen.

Hier sehen Sie die endgültige Funktion zum Setzen bzw. Löschen von Bitwerten:

```
' Day ist 1 - 7
' Hour ist 0 - 23
Private Sub SetBitValue(HourList() As Byte, Day As Integer, Hour As _
Integer, Allow As Boolean)

   Dim ByteIndex As Integer
   Dim ByteMask As Integer
   ' Berechnen der Byteposition
   ByteIndex = Int(Hour / 8) + ((Day - 1) * 3)
   ' Berechnen der Maske für die logische Operation
   ByteMask = 2 ^ Int(Hour Mod 8)
   Debug.Print ByteIndex
   Debug.Print ByteMask
   If Allow Then
      HourList(ByteIndex) = HourList(ByteIndex) Or ByteMask
   Else
      HourList(ByteIndex) = HourList(ByteIndex) And (Not ByteMask)
   End If
End Sub
```

Ein letztes Detail

Beim Testen des endgültigen Programms stellte ich schnell fest, dass ich mich tatsächlich nicht bei dem Konto anmelden konnte, das ich erstellt hatte – ich erhielt ständig die Meldung, das Konto sei abgelaufen. Ich fand heraus, dass ich zwei Änderungen vornehmen musste: Erstens musste ich den Benutzeradministrator bitten, das neue Konto einer Gruppe hinzuzufügen (Sie können dies mittels API-Funktionen bewerkstelligen, aber ich habe es in diesem Fall nicht getan). Außerdem musste ich das Feld usri2_acct_expires auf den Wert der Konstanten TIMEQ_FOREVER (ebenfalls -1) setzen, damit das Konto nie ablaufen würde.

Die endgültige Funktion zum Hinzufügen eines Benutzers sieht so aus:

```
Private Sub Command1_Click()
   Dim ul2 As USER_INFO_2
   Dim res As Long
   Dim parm_err As Long
   Dim username As String
   Dim userpass As String
   Dim HourList(20) As Byte
   username = "Puzzle" & Chr$(0)
   userpass = "pass" & Chr$(0)

   ul2.usri2_name = StrPtr(username)
   ul2.usri2_password = StrPtr(userpass)
   ul2.usri2_priv = USER_PRIV_USER
   ul2.usri2_flags = UF_NORMAL_ACCOUNT Or UF_SCRIPT
   ul2.usri2_max_storage = USER_MAXSTORAGE_UNLIMITED
   ul2.usri2_acct_expires = TIMEQ_FOREVER
   ul2.usri2_logon_hours = VarPtr(HourList(0))
   ' Anmeldung nur samstags von 23.00 Uhr bis Mitternacht ermöglichen!
   Call SetBitValue(HourList(), 7, 23, True)

   res = NetUserAdd(0, 2, VarPtr(ul2), parm_err)
   Debug.Print "Result: " & res
   If res <> 0 Then Debug.Print "Error at index: " & parm_err
End Sub
```

Lösung 24

Rufen Sie diese Zeichenfolge zurück!

Das Problem liegt im Rückrufparameter, der `ByVal As String` definiert ist. Ist ein Rückrufparameter `ByVal As String` deklariert, dann erwartet die Funktion einen Zeiger auf eine `BSTR`-Zeichenfolge. Aufgrund der `ByVal`-Übergabe setzt die Funktion voraus, dass es sich bei dem BSTR um eine Kopie der ursprünglichen Zeichenfolge handelt. Daher liegt es in der Verantwortung der Funktion, die Zeichenfolge vor der Rückgabe freizugeben. Unglücklicherweise ist der Zeiger kein BSTR – es ist eine reguläre, mit NULL abgeschlossene C-Zeichenfolge und obendrein noch eine ANSI-Zeichenfolge. Visual Basic wird die Zeichenfolge nicht nur falsch interpretieren, sondern auch spektakulär abstürzen bei dem Versuch, das freizugeben, was es für einen BSTR hält.

Die Lösung liegt in der Änderung des Parameters in `ByVal As Long`. Damit wird der Parameter mit einem Zeiger auf eine mit NULL abgeschlossene ANSI-C-Zeichenfolge geladen. Mit Hilfe der Funktion lstrlen kann die Länge der Zeichenfolge bestimmt werden, und mit der Funktion `lstrcpy` können die Daten kopiert werden, wie in dem nachstehenden richtigen Code gezeigt.

Nachfolgend ein Standardmodul:

```
' Puzzle Ländereinstellungen
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten
Option Explicit

Public Declare Function EnumSystemLocales Lib "kernel32" Alias _
"EnumSystemLocalesA" (ByVal lpLocaleEnumProc As Long, ByVal dwFlags _
As Long) As Long

Public Declare Function lstrcpy Lib "kernel32" Alias "lstrcpyA" _
(ByVal dest As String, ByVal source As Long) As Long
Public Declare Function lstrlen Lib "kernel32" Alias "lstrlenA" _
(ByVal source As Long) As Long

Public Const LCID_INSTALLED = &H1  ' IDs installierter Ländereinstellungen
Public Function EnumLocalesProc(ByVal Locale As Long) As Long
    Dim LocaleName As String
    LocaleName = String$(lstrlen(Locale) + 1, Chr$(0))
    Call lstrcpy(LocaleName, Locale)
```

```
    Debug.Print LocaleName
    EnumLocalesProc = True
End Function
```

Die Form enthält

```
' Puzzle Ländereinstellungen
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten

Option Explicit

Private Sub Command1_Click()
   Dim lRet As Long
   lRet = EnumSystemLocales(AddressOf EnumLocalesProc, LCID_INSTALLED)
End Sub
```

Die Variable `LocaleName` in der Funktion `EnumLocalesProc` verfügt immer noch über ein abschließendes NULL-Zeichen, dass Sie, falls erforderlich, löschen können. Dieses Programm zeigt die Ergebnisse in Form von acht Hexadezimalziffern an. Die US-Englischversion von Windows zeigt z. B. die Ländereinstellung 00000409 an. (Ausführliche Informationen zu Ländereinstellungen finden Sie in der Win32-Dokumentation sowie in Kapitel 6 meines Buches »Visual Basic Programmer's Guide to the Win32 API«.)

Lösung 25

Universelle Bezeichner, Teil 1

Es gibt wirklich nur zwei Ansätze beim Umgang mit 128-Bit-Variablen in Visual Basic: Sie müssen entweder ein Array oder einen benutzerdefinierten Typ erstellen (Struktur). Die Windows-Lösung ist die Verwendung einer Struktur. Die Kunst, eine C-Headerdatei in Visual Basic zu übersetzen, besteht vorwiegend darin, zu ermitteln, welche Anweisungen für die Aufgabe wirklich entscheidend sind. Werfen Sie erneut einen Blick auf die Liste, denn jetzt enthält sie Anmerkungen:

```
#ifndef GUID_DEFINED     // Folgendes überspringen, wenn
                         // GUID_DEFINED bereits definiert ist
#define GUID_DEFINED     // Definieren der Konstanten GUID_DEFINED
                         // kein Wert
typedef struct  _GUID    // Definieren einer Struktur
                         // (benutzerdefinierter Typ)
    {
    DWORD Data1;         // 32-Bit-Wert
    WORD Data2;          // 16-Bit-Wert
    WORD Data3;          // 16-Bit-Wert
    BYTE Data4[ 8 ];     // 8 x 8 Bits = 64 Bits
    }   GUID;            // Die GUID-Struktur ist 32+16+16+64 = 128 Bits

#endif                   // !GUID_DEFINED
                         // Ende des Teils, der übersprungen werden soll,
                         // wenn GUID_DEFINED bereits definiert ist
```

Die Verwendung von Konstanten zur bedingten Kompilierung, wie `GUID_DEFINED` im obigen Beispiel, ist in Windows-Headerdateien sehr gebräuchlich. Wenn diese Headerdatei zum ersten Mal in ein Programm einbezogen wird, ist `GUID_DEFINED` nicht definiert. Dies bedeutet, dass die Zeilen bis zur Anweisung `#endif` vom Compiler abgearbeitet werden, und zwar inklusive einer Zeile, in der die Konstante `GUID_DEFINED` definiert wird. Wird die Headerdatei erneut in ein Programm einbezogen, erkennt der Compiler, dass die Konstante `GUID_DEFINED` bereits definiert ist, und überspringt den Code bis zur Anweisung `#endif`. Dies verhindert, dass die GUID-Struktur ein zweites Mal definiert wird – ein Fehler in C++.

Hier liegt ein Unterschied zum System von Visual Basic, wo Sie einfach Module einem Projekt hinzufügen. In C++ können Dateien Headerdateien einbeziehen. Diese Dateien werden so behandelt, als wären sie ein Teil der Hauptprogrammdatei. Doch Headerdateien können auch zusätzliche Headerdateien einbeziehen.

Wenn mehrere Dateien dieselbe Headerdatei einbeziehen (wie es im Fall der hier verwendeten Datei **wtypes.h** sehr wahrscheinlich ist, die Deklarationen wichtiger, von Windows verwendeter Datentypen enthält), besteht das Risiko, dass Funktionen oder Strukturen mehrfach deklariert werden. Die hier gezeigte Verwendung von Konstanten zur bedingten Kompilierung trägt dazu bei, diesen Fehler zu vermeiden:

```
// Die folgenden vier Zeilen definieren einen neuen, LPGUID genannten Typ,
// wenn die Konstante __LPGUID_DEFINED__ noch nicht definiert ist.
// LPGUID ist ein Long-Zeiger auf eine GUID
#if !defined( __LPGUID_DEFINED__ )
#define __LPGUID_DEFINED__
typedef GUID __RPC_FAR *LPGUID;
#endif // !__LPGUID_DEFINED__
```

Für den Fall, dass LPGUID später als Funktionsparameter eingesetzt wird, ist es nützlich zu wissen, worum es sich dabei handelt. Aber da es sich um einen Zeiger handelt, wissen Sie, dass es ein 32-Bit-Zeigerwert sein wird. Der Ausdruck `__RPC_FAR` wird in Kürze beschrieben.

```
// Die folgenden Zeilen definieren eine Struktur mit der Bezeichnung
// OBJECTID. Es ist für dieses Puzzle völlig unwichtig. In der Tat
// erinnere ich mich nicht daran, diese Struktur jemals verwendet zu haben.
#ifndef __OBJECTID_DEFINED
#define __OBJECTID_DEFINED
#define __OBJECTID_DEFINED
typedef struct  _OBJECTID
    {
    GUID Lineage;
    unsigned long Uniquifier;
    }   OBJECTID;
#endif // !_OBJECTID_DEFINED

#if !defined( __IID_DEFINED__ )  // If the __IID_DEFINED__ _
// Wenn die Konstante bereits definiert ist,
// Folgendes überspringen
#define __IID_DEFINED__    // Definieren der Konstanten __IID_DEFINED__
typedef GUID IID;          // Eine IID ist eine GUID

typedef IID __RPC_FAR *LPIID;  // Eine LPIID ist ein Zeiger auf eine IID

typedef GUID CLSID;            // Eine CLSID ist auch eine GUID
#endif
```

Was bedeutet der Ausdruck __RPC_FAR in den Deklarationen von LPIID und LPGUID?

In der C-Headerdatei **rpc.h** finden Sie folgende Definition:

```
#define __RPC_FAR
```

Doch es folgt nichts auf das Wort __RPC_FAR! Bedeutet dies, dass der Ausdruck als »Nichts« definiert ist? Das ist richtig. Es handelt sich um eine Konstante, die auf einigen Plattformen verwendet werden kann, um den Compiler anzuweisen, bestimmte Typen von Code zu erzeugen. Doch dies ist unter Win32 ohne Bedeutung. Also verwirft der Compiler den Ausdruck einfach während der Kompilierung.

Die Visual Basic-Deklaration einer GUID entspricht der folgenden einfachen Struktur:

```
Public Type GUID
   Data1 As Long     ' 32 bits
   Data2 As Integer  ' 16 bits
   Data3 As Integer  ' 16 bits
   Data4(7) As Byte  ' 8 x 8 bits
End Type
```

Haben Sie Data4 richtig definiert? Denken Sie daran, das Array ist nullbasiert. Würden Sie es als Data4(8) As Byte definieren, hätten Sie 72 Bits anstatt 64.

Wie verhält es sich mit CLSID und IID?

Sie können identische Strukturen für diese Typen erstellen. Doch warum sollten Sie unnötige Umstände machen? Da sich alle auf 128-Bit-Werte beziehen, können Sie einfach einen einzelnen benutzerdefinierten Typ mit der Bezeichnung GUID oder CLSID erstellen und ihn jederzeit verwenden, wenn Sie mit einem dieser universellen Bezeichner arbeiten müssen. Die Puzzle 26–28 verwenden für diesen Zweck eine Struktur mit der Bezeichnung CLSID.

Lösung 26

Universelle Bezeichner, Teil 2

Wenn Sie das Kommentierungszeichen im `Err.Raise`-Code in der `GUIDObject`-Klasse löschen würden, sähe das Programm folgendermaßen aus:

```
res = CLSIDFromProgID(MyProgramId, cid)
If res = 0 Then
   GetCLSID = cid
Else
   ' Was geschieht, wenn Sie hier das Kommentierungszeichen entfernen?
   Err.Raise res
End If
```

Seit wann lösen Sie Fehler mit Ergebnissen von API-Funktionen aus? Normalerweise würden Sie die Funktion `GetErrorString` in **ErrString.bas** verwenden, um die Beschreibung eines Fehlers aufzurufen.

Werfen wir einen weiteren Blick auf die Dokumentation der Funktion `CLSIDFromProgID` und die VB-Deklaration:

```
HRESULT CLSIDFromProgID(
    LPCOLESTR lpszProgID,          //Zeiger auf die ProgID
    LPCLSID pclsid                 //Zeiger auf die CLSID
   );
Private Declare Function CLSIDFromProgID Lib "ole32.dll" _
(ByVal lpszProgId As String, lpclsid As CLSID) As Long
```

Was ist ein `HRESULT`? Gemäß der Deklaration handelt es sich um einen `Long`-Wert, den Sie im Hinblick auf die Tatsache erwarten würden, dass API-Funktionen fast immer `Long`-Werte zurückgeben.

Betrachten Sie die Definition von `HRESULT` in der Datei **wtypes.h:**

```
typedef LONG HRESULT;
```

`HRESULT` ist offensichtlich eine `Long`-Variable. Warum sollte ich versuchen, einen Fehler mit einem `HRESULT`-Wert auszulösen? Wie unterscheidet sich ein `HRESULT`-Wert von einem normalen API-Fehlercode?

`HRESULT` ist ein Fehlerwert des Typs `Long`, doch im Gegensatz zu regulären API-Ergebniscodes wird ein `HRESULT` durch COM definiert, das Component Object Model. COM definiert ein spezielles Fehlercodeformat, das vollständig unabhängig von den Win32 API-Fehlerergebnissen ist, die Sie mit Hilfe der `Err.LastDllError`-Methode erhalten. `HRESULT`-Fehlercodes sind dieselben Werte, die auch von Visual

Basic und sonstigen COM-basierten Anwendungen bei der Behandlung von Objektfehlern verwendet werden. Die Anweisung `Err.Raise` in der `GUIDobject`-Klasse löst den Fehler »`ActiveX Component Can't Create Object`« (ActiveX-Komponente kann das Objekt nicht erstellen) aus. Dies lässt darauf schließen, dass ein Problem mit der Programm-ID vorliegt.

Ein weiterer Ansatz zur Bestimmung der Bedeutung von `HRESULT` ist, den Wert im Hexadezimalformat zu betrachten. In diesem Fall beträgt er `&H800401F3`, das entspricht der Konstanten `CO_E_CLASSSTRING` in der Datei **api32.txt**. Laut Win32-Dokumentation für die Funktion `CLSIDFromProgID` ist ein möglicher Rückgabewert der Funktion wirklich `CO_E_CLASSSTRING`, beschrieben als »The registered `CLSID` for the `ProgID` is invalid« (Die registrierte `CLSID` für die `ProgID` ist ungültig).

Also ist die Registrierung hoffnungslos beschädigt, oder mit der Programm-ID stimmt etwas nicht.

So wollen wir den ein wenig im Wunschdenken schwelgen und davon ausgehen, dass die Registrierung nicht beschädigt ist und untersuchen, wo der Fehler der Programm-ID liegen könnte.

Der erste Parameter der Funktion `CLSIDFromProgID` wird als `LPCOLESTR` beschrieben. Was ist `LPOLESTR`? LP weist auf einen Zeiger hin. Das C ist Indiz für einen konstanten Wert. Sie wissen, dass es sich bei `LPSTR` um eine Zeichenfolge handelt, und `LPCSTR` ist eine konstante Zeichenfolge – die von der API-Funktion nicht modifiziert wird. Aber welche Art von Zeichenfolge ist `OLESTR`?

Die Deklaration in der Datei **wtypes.h** lautet

```
typedef WCHAR OLECHAR;
typedef /* [string] */ OLECHAR __RPC_FAR *LPOLESTR;
```

`LPOLESTR` ist ein Zeiger auf `OLESTR`. `OLESTR` setzt sich aus `OLECHAR`-Zeichen zusammen, die als `WCHAR`-Datentyp definiert sind. `WCHAR` ist ein `wide`-Zeichen.

Mit anderen Worten, es muss eine Unicode-Zeichenfolge sein. Damit könnten Sie auch vertraut werden, wenn es um Funktionen geht, die ein Teil des OLE-Subsystems sind. Im Gegensatz zu allgemeinen Win32 API-Funktionen, die über separate ANSI- und Unicode-Einsprungpunkte verfügen und im Allgemeinen ANSI-Zeichenfolgen mit ANSI-Einsprungpunkten verwenden, verfügen die OLE-Funktionen fast immer über einen einzelnen Einsprungpunkt und akzeptieren Unicode-Zeichenfolgenparameter.

Wie sollten Sie eine Unicode-Zeichenfolge an eine API-Funktion übergeben?

Der Parameter `lpszProgId` wird, wie hier gezeigt, `ByVal As Long` deklariert. Die aufrufende Routine muss die Adresse einer Unicode-Zeichenfolge als Parameter übergeben.

```
Private Declare Function CLSIDFromProgID Lib "ole32.dll" _
(ByVal lpszProgId As Long, lpclsid As CLSID) As Long
```

Sie könnten mit dem folgenden Code ein temporäres Bytearray definieren, die Zeichenfolge zuweisen und dann einen Zeiger an das erste Byte des Arrays übergeben:

```
Dim TempBuffer() As Byte
TempBuffer() = MyProgramId
res = CLSIDFromProgID(VarPtr(TempBuffer(0)), cid)
```

Aber der Code im tatsächlichen Beispiel verfolgt einen deutlich effizienteren Ansatz durch Verwendung des `StrPtr`-Operators. Dieser Operator gibt einen Zeiger auf die Zeichendaten zurück, wie sie wirklich unter Visual Basic gespeichert sind. Da Visual Basic Zeichenfolgen im Unicode-Format speichert, kann so mühelos eine Unicode-Zeichenfolge an eine Funktion übergeben werden.

```
Option Explicit

Public Function GetCLSID() As CLSID
   Dim cid As CLSID
   Dim res As Long
   Dim MyProgramId As String

   MyProgramId = "GUIDPuzzle.GUIDObject" & Chr$(0)
   res = CLSIDFromProgID(StrPtr(MyProgramId), cid)
   If res = 0 Then
      GetCLSID = cid
   End If
End Function
```

Warum haben wir im Aufruf von `CLSIDFromProgID` nicht `VarPtr(MyProgramId)` verwendet?

Weil der `VarPtr`-Operator die Adresse der Variablen `MyProgramId` zurückgibt. Diese Variable enthält einen Zeiger auf die BSTR-Zeichendaten. Der Unterschied zwischen diesen beiden Operatoren ist in Abbildung S26-1 zu erkennen.

Die Ergebnisse dieser Lösung sehen Sie in Abbildung S26-2.

Jetzt werfen wir in den Abbildungen und in Tabelle S26-1 einen Blick auf die Strukturdaten und die Ursprungsdaten. Beachten Sie, dass die tatsächlichen Werte auf Ihrem System in jedem Fall abweichen.

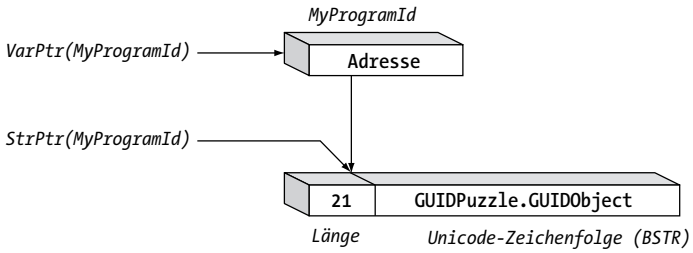


Abbildung S26-1 Der Unterschied zwischen `VarPtr` und `StrPtr`

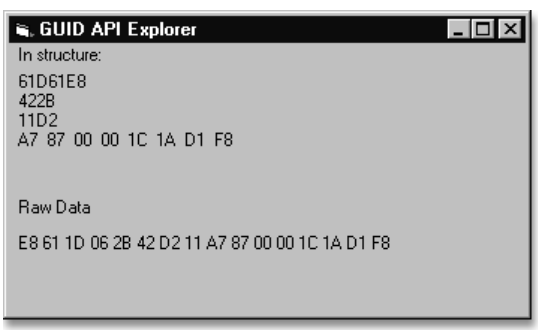


Abbildung S26-2 Jetzt funktioniert das `GUIDPuzzle`-Programm

Feldname	Wert	Ursprungsdaten
Data1	61D61E8	E8 61 1D 06
Data2	422B	2B 42
Data3	11D2	D2 11
Data4	A7 87 00 00 1C 1A D1 F8	A7 87 00 00 1C 1A D1 F8

Tabelle S26-1 Darstellungen der `GUID`-Daten

Hier sehen Sie die Auswirkungen der Byteanordnung darauf, wie die Daten angezeigt werden. (Mehr darüber finden Sie in Tutorium 2, »Der Arbeitsspeicher – da, wo alles beginnt,« in Teil III dieses Buches.)

Warum plage ich mich damit, Ihnen diese beiden unterschiedlichen Arten der Datenbetrachtung zu zeigen? Weil Sie unter Umständen gelegentlich mit beiden Darstellungen konfrontiert werden. Obwohl in Programmen in der Regel mittels der Programm-ID auf Objekte Bezug genommen wird, werden Sie jedesmal,

wenn Sie in einer Visual Basic-Projektdatei oder der Systemregistrierung einen Blick unter die Motorhaube werfen müssen, der `CLSID` in einem Format begegnen, das etwa so aussieht:

```
{ 061D61E8-422B-11D2-A787-00001C1AD1F8}
```

Sie erkennen, dass die Daten in diesem Format dem in der zweiten Spalte von Tabelle S26-1 angezeigten Wert sehr ähneln. Der einzige Unterschied ist, dass das Array `Data4` in eine Gruppe von 2 Bytes und eine Gruppe von 6 Bytes aufgeteilt ist.

Die Zeichenfolgendarstellung einer `CLSID` wird nahezu universell von Anwendungen und Datendateien verwendet, die `CLSID`-Informationen in einem für Personen lesbaren Format anzeigen. Wo treffen Sie aber dann auf die Ursprungsdaten? Sie sehen CLSIDs im Ursprungsdatenformat, wenn Sie den Speicher in einem Debugger wie etwa dem Visual C++-Debugger untersuchen.

Da das Zeichenfolgenformat häufig verwendet wird, fragen Sie sich bestimmt, ob es eine Funktion zur Konvertierung einer `CLSID`-Struktur in die Zeichenfolgendarstellung einer `CLSID` gibt. Dies ist der Fall und deshalb auch Thema von Puzzle 27.

Lösung 27

Universelle Bezeichner, Teil 3

Sie haben bereits in vorhergehenden Beispielen erfahren, wie REFCLSID-Parameter behandelt werden, sodass Sie sicher sein können, dass das Problem nicht bei den Parametern liegt.

Dass diese Lösung zum Absturz führt, illustriert eines der häufigsten Probleme bei der Arbeit mit neuen bzw. komplexen API-Funktionen – die Tendenz, Schlussfolgerungen zu ziehen und Deklarationen zu erstellen, ohne vorher gründlich nachzudenken. Der Parameter ppsz ist als LPOLESTR *-Typ definiert. Der Glaube, eine Visual Basic-Zeichenfolge als Verweis übergeben zu können, ist verlockend, führt aber meistens in die Irre.

Um die Gründe zu verstehen, betrachten wir einige mögliche Verfahren, nach denen man bei der Übergabe eines Zeichenfolgenparameters an eine API-Funktion vorgehen kann, und was im Einzelfall hinter den Kulissen geschieht. Abbildung S27-1 illustriert den gesamten Prozess, der bei der Übergabe einer Zeichenfolge an eine API-Funktion abläuft. (Dieses Thema wird auch in Tutorium 5, »Das ByVal-Schlüsselwort: Die Lösung für 90 % aller API-Probleme,« in Teil III dieses Buches behandelt.)

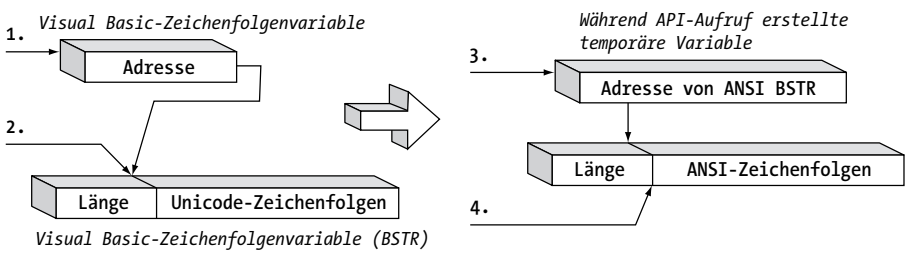


Abbildung S27-1 Übergabe von Zeichenfolgen an API-Funktionen

Visual Basic speichert Zeichenfolgen intern im BSTR-Format. Ein BSTR ist eine OLE-Zeichenfolge, also eine Zeichenfolge, die vom OLE-Subsystem zugewiesen wird. Eine Visual Basic-Zeichenfolgenvariable enthält einen BSTR, der auf den Anfang der BSTR-Zeichenfolgendaten zeigt. Die 4 Bytes vor dem Anfang der Zeichenfolgendaten enthalten die Länge der Zeichenfolge. Die Zeichenfolgendaten in einer Visual Basic-Zeichenfolge werden immer im Unicode-Format gespeichert. Wenn Visual Basic eine Zeichenfolge an eine API-Funktion übergibt, wird eine temporäre BSTR-Zeichenfolge erstellt, die eine ANSI-Zeichenfolge enthält. Bei Rückgabe durch die API-Funktion wird der Inhalt der ANSI-Zeichenfolge in die Unicode-Zeichenfolge von Visual Basic kopiert.

Die nummerierten Pfeile in Abbildung S27-1 stellen die verschiedenen Zeiger dar, die an API-Funktionen übergeben werden können:

- **Zeiger 1** ist die Adresse der Visual Basic-Zeichenfolgenvariablen, die den Zeiger auf die Visual Basic-Zeichenfolgendaten (`BSTR`-Zeichenfolge) enthält. Sie können diese Adresse in gleicher Weise ermitteln wie die Adresse jeder beliebigen Visual Basic-Variablen: durch Verwendung des `VarPtr`-Operators.
- **Zeiger 2** ist der Wert des Zeigers für eine Zeichenfolge, nämlich die Adresse des Anfangs der Visual Basic-Zeichenfolgendaten (BSTR-Zeichenfolge). Sie können diese Adresse in Visual Basic mit Hilfe des `StrPtr`-Operators ermitteln.
- **Zeiger 3** ist die Adresse der temporären Variablen, die den Zeiger auf die von Visual Basic erstellte temporäre ANSI-Zeichenfolge (BSTR-Zeichenfolge) enthält. Diese Adresse übergeben Sie, wenn Sie eine Zeichenfolgenvariable als Verweis an eine API-Funktion übergeben.
- **Zeiger 4** ist der Wert des Zeigers für die temporäre ANSI-Zeichenfolge (BSTR-Zeichenfolge). Diese Adresse übergeben Sie, wenn Sie eine Zeichenfolgenvariable als Wert an eine API-Funktion übergeben.

Die überwiegende Mehrzahl der Zeichenfolgenzeiger verwendenden API-Funktionen wird als `LPSTR`, ANSI oder Zeichenfolgen deklariert. Der einzige auf eine ANSI-Zeichenfolge weisende Zeiger in der Abbildung ist Zeiger 4, was erklärt, warum die meisten Zeichenfolgenvariablen als Wert übergeben werden.

Die Funktion `StringFromCLSID` benötigt eine Unicode-Zeichenfolge, sodass die Zeiger 3 und 4 eindeutig nicht in Frage kommen. Wäre der Parameter `ppsz` genau wie `LPOLESTR` deklariert, ließe sich Zeiger 2 verwenden, da es sich bei einem `LPOLESTR` um einen Zeiger auf die eigentlichen Unicode-Zeichenfolgendaten handelt. Aus diesem Grund können Sie den Operator `StrPtr` verwenden, um gültige Parameter für viele API-Funktionen zu ermitteln, die mit OLE-Zeichenfolgen arbeiten. Deklarieren Sie einfach den Parameter `ByVal As Long`, und übergeben Sie den durch Anwendung des Operators `StrPtr` auf die Zeichenfolge ermittelten Wert. Sie müssen natürlich sicherstellen, dass der Zeichenfolge vorher eine ausreichende Länge zur Aufnahme jeglicher Daten, die von der API-Funktion in die Zeichenfolge geladen werden können, zugewiesen wird.

Damit bleibt Zeiger 1 als einzig mögliche Lösung für die Funktion `StringFromCLSID` übrig. Auf den ersten Blick scheint dies zu funktionieren. Zeiger 1 ist ein Zeiger auf einen `BSTR`, der einen Zeiger auf die Zeichenfolge enthält. Und ein `LPOLESTR` *-Parameter ist ein Zeiger auf einen Zeiger. Das müsste doch funktionieren, oder?

Falsch. Warum? Weil ein `LPOLESTR` kein Zeiger auf einen `BSTR` ist. Er ist ein Zeiger auf eine reguläre, mit NULL abschließende Zeichenfolge. Die Zuweisung eines

BSTR erfolgt durch das OLE-Zeichenfolgenverwaltungssystem, und den Zeichenfolgendaten ist die Länge der Zeichenfolge vorangestellt. Ein LPOLESTR ist einfach ein generischer Zeiger auf eine Zeichenfolge. Abbildung S27-2 zeigt den Parameter ppsz aus der Perspektive der Funktion StringFromCLSID.

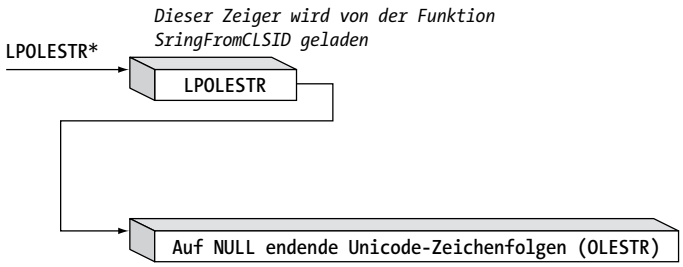


Abbildung S27-2 Funktion StringFromCLSID erhält den Zeiger LPOLESTR *

Die Funktion erhält den Zeiger LPOLESTR *, also die Adresse einer LPOLESTR-Variablen, die die Adresse der Zeichenfolgendaten enthält. Dies bedeutet, dass die Funktion die LPOLESTR-Variable ändern kann. Wenn Sie einen Blick auf die Beschreibung dieses Parameters werfen, sehen Sie Folgendes:

```
LPOLESTR * ppsz    // Indirekter Zeiger auf die zurückgegebene Zeichenfolge
```

Dies bedeutet, dass der LPOLESTR-Parameter, auf den bei Aufruf der Funktion ein Verweis erfolgt, von der Funktion mit einem Zeiger auf die Zeichenfolgendaten geladen wird.

Im ursprünglichen Puzzlecode lädt StringFromCLSID die Variable MyString mit einem Zeiger auf eine mit NULL abschließende Unicode-Zeichenfolge. Doch nichtsdestoweniger scheitert die Funktion in doppelter Hinsicht. Erstens wird die jedem BSTR vorangestellte Länge nicht von dieser Funktion gesetzt (weil es sich nicht um einen BSTR handelt). Zweitens wird der Zeiger nicht vom OLE-Subsystem zugewiesen. Unter Windows NT tritt, sobald Visual Basic versucht, den Zeiger als BSTR zu behandeln, ein Ausnahmefehler auf. Unter Windows 95 tritt kein Ausnahmefehler auf – zumindest nicht sofort – aber die Zeichenfolgenlänge ist falsch, und die vorherige Zeichenfolge, auf die die Variable verweist, ist »verloren«, da der Zeiger überschrieben wurde.

Da Visual Basic keine Möglichkeit der automatischen Behandlung des LPOLESTR *-Parametertyps bietet, liegt es an Ihnen, eine Deklaration zu erstellen, die der API-Funktion die gewünschten Informationen liefert. In diesem Fall benötigt die Funktion einen Zeiger auf eine 32-Bit-Variable, die sie mit einem Zeiger auf eine Zeichenfolge laden kann.

Am einfachsten erfolgt die Übergabe eines Zeigers an eine 32-Bit-Variable durch Übergabe einer `Long`-Variablen als Verweis, was zu folgender Deklaration führt:

```
Private Declare Function StringFromCLSID Lib "ole32.dll" _
(lpclsid As CLSID, lpOlestr As Long) As Long
```

Jetzt müssen Sie nur noch einen Weg finden, die Zeichenfolgendaten aus der `Long`-Variablen abzurufen.

Nehmen Sie sich vor dem Weiterlesen doch einige Minuten (oder Stunden) Zeit, selbst eine Lösung zu finden.

Die »GetCLSIDAsString«-Methode

Mit Hilfe des folgenden Codes kann der von der Funktion `StringFromCLSID` gelieferte Zeichenfolgenzeiger `LPOLESTR` in eine Visual Basic-Zeichenfolge konvertiert werden:

```
Private Declare Function lstrlenW Lib "kernel32" _
(ByVal lpString As Long) As Long
Private Declare Function RtlMoveMemory Lib "kernel32" _
(dest As Any, source As Any, ByVal count As Long) As Long

Public Function GetCLSIDAsString() As String
   Dim cid As CLSID
   Dim StringBuffer() As Byte
   Dim StringLength As Long

   cid = GetCLSID()
   Dim MemoryPointer As Long

   Call StringFromCLSID(cid, MemoryPointer)
   StringLength = lstrlenW(MemoryPointer) * 2
   ReDim StringBuffer(StringLength)
   Call RtlMoveMemory(StringBuffer(0), ByVal MemoryPointer, _
   StringLength)
   GetCLSIDAsString = StringBuffer()
End Function
```

Bei der Variablen `MemoryPointer` handelt es sich um eine `Long`-Variable, die von der Funktion `StringFromCLSID` mit der Adresse der `LPOLESTR`-Zeichenfolge geladen wird. Der erste Schritt bei der Konvertierung dieser Zeichenfolge in eine Visual Basic-Zeichenfolge ist die Ermittlung der Länge. Dies kann mit Hilfe der Funktion `lstrlenW` durchgeführt werden. Die `lstrlen`-API-Funktion gibt die Anzahl der in

einer Zeichenfolge enthaltenen Zeichen zurück. Sie verfügt über zwei Einsprungpunkte: `lstrlenA` für ANSI-Zeichenfolgen und `lstrlenW` für Unicode-Zeichenfolgen. Da die Variable `MemoryPointer` die Adresse einer Unicode-Zeichenfolge aufnehmen soll, wird die Funktion `lstrLenW` verwendet. Der Parameter `lpString` der Funktion `lstrlenW` wird `ByVal As Long` deklariert, weil Sie die Adresse der Zeichenfolgendaten übergeben, die in diesem Fall in der `Long`-Variablen `MemoryPointer` enthalten ist.

Die Anzahl der Bytes in der Zeichenfolge ist doppelt so groß wie die Länge der Zeichenfolge. Denken Sie daran, dass die Funktion `lstrlenW` die Anzahl der Zeichen in der Zeichenfolge zurückgibt, und die Unicode-Zeichenfolgen verwenden 2 Bytes für jedes Zeichen. Die Funktion dimensioniert ein Bytearray auf die zur Aufnahme der Unicode-Zeichenfolgendaten erforderliche Größe. Anschließend wird der Text mit der Funktion `RtlMoveMemory` aus dem `MemoryPointer`-Puffer in das neue `StringBuffer`-Array kopiert. Schließlich nutzt die Funktion die Fähigkeit von Visual Basic, ein Array direkt einer Zeichenfolge zuzuweisen, um den Inhalt des Bytearrays in eine Visual Basic-Zeichenfolge zu kopieren – in diesem Fall die Zeichenfolge, die die Ergebnisse der Funktion aufnimmt.

Ein weiteres Problem

Das hier vorgestellte Programm läuft perfekt.

Oder etwa nicht?

Fügen Sie der `GUIDPuzzle`-Form folgende Funktion hinzu:

```
Private Sub cmdMemTest_Click()
   Dim obj As New GUIDObject
   Do While True
      lblGUIDString.Caption = obj.GetCLSIDAsString()
      DoEvents
   Loop
End Sub
```

Wenn Sie auf die Schaltfläche `cmdMemTest` klicken, wird die Funktion `GetCLSIDAsString` erneut aufgerufen. Bei der Ausführung unter Windows NT rufen Sie den NT-Systemmonitor auf (im Menü **Start** unter **Programme · Verwaltung (Allgemein)**). Fügen Sie ein Diagramm zur Überwachung des verfügbaren Arbeitsspeichers hinzu. Skalieren Sie das Diagramm so, dass die Speicherauslastungskurve etwa in der Mitte des Fensters beginnt.

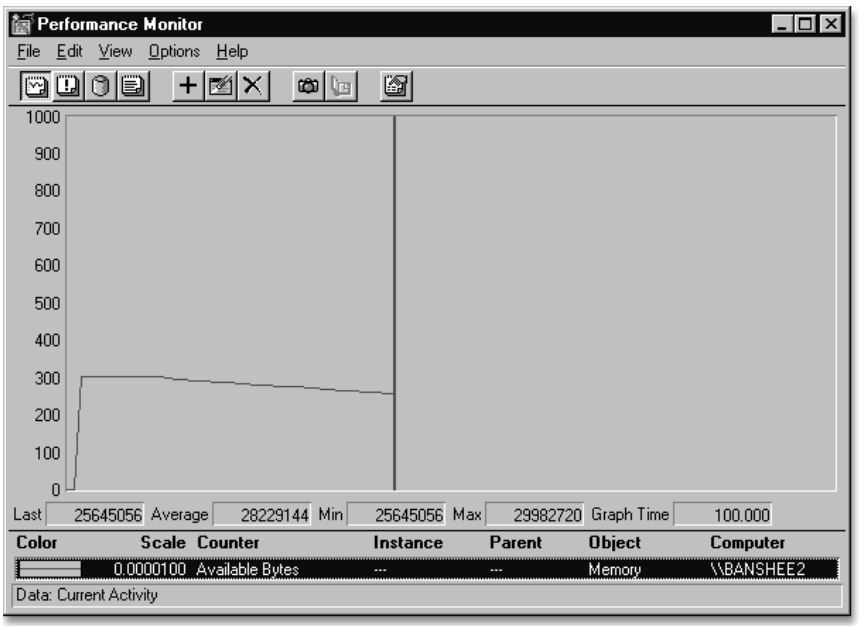


Abbildung S27-3 NT-Systemmonitor veranschaulicht ein Speicherleck

Unter Windows 95 können Sie durch Aufruf des Feldes »Info für Windows« im Explorerfenster regelmäßig die verfügbare Speichermenge überprüfen. Das Ergebnis ähnelt dem in Abbildung S27-3 dargestellten. Die verfügbare Speichermenge nimmt allmählich ab.

Dieses als »Speicherleck« bezeichnete Phänomen wird sehr leicht außer Acht gelassen. Ein Speicherleck entsteht, wenn ein Teil des Programms Speicher belegt, ohne ihn wieder freizugeben. Wo ist in diesem Fall das Speicherleck?

Betrachten Sie erneut Abbildung S27-2. Die Funktion `StringFromCLSID` lädt die Variable `MemoryPointer`, die Sie als Parameter übergeben haben, mit einem Zeiger auf die Zeichenfolge, die eine Darstellung des `CLSID`-Parameters enthält. Diese Zeichenfolge legt Daten im Speicher ab. Woher kommen diese Daten?

Die Funktion `StringFromCLSID` ordnet sie zu!

Werfen Sie einen weiteren Blick auf die Beschreibung der Parameter:

Parameter

- `rclsid`
- `[in] CLSID`, die konvertiert werden soll
- `ppsz`
- `[out]` Zeiger auf die Ergebniszeichenfolge

Was bedeuten die Präfixe `[in]` und `[out]`? Denken Sie daran, dass Sie es jetzt mit dem OLE-Subsystem zu tun haben – einem Teil von Win32-API, der in mancher Hinsicht anders funktioniert als die regulären Win32-API-Funktionen. Sie haben bereits im vorherigen Puzzle gesehen, dass sich die Fehlerbehandlung im OLE-Subsystem von der in normalen Win32-API-Funktionen unterscheidet. Es stellt sich heraus, dass die Parameter ebenfalls anderen Regeln unterliegen. Jeder Parameter ist als `[in]`-Parameter, `[out]`-Parameter oder `[in-out]`-Parameter definiert.

- `[in]`-Parameter werden vom Aufrufer zugeordnet und freigegeben.
- `[out]`-Parameter werden von der API-Funktion zugeordnet und müssen vom Aufrufer mit Hilfe des OLE-Speicherzuordners freigegeben werden.
- `[in-out]`-Parameter werden vom Aufrufer zugeordnet und können von der API-Funktion freigegeben und neu zugeordnet werden. Der Aufrufer ist letztendlich für die Freigabe der Parameter verantwortlich.

In jedem Fall müssen Sie die Dokumentation der jeweiligen Funktion zu Rate ziehen, um zu bestimmen, wie der fragliche Speicher zugeordnet bzw. freigegeben wird. OLE kennt tatsächlich mehrere verschiedene Speicherzuordnungssysteme, deren Einsatz vom verwendeten Subsystem abhängt.

In diesem Fall wird die Zeichenfolge, deren Zeiger in die Variable `MemoryPointer` geladen wird, der API-Funktion zugeordnet. Sie sind dafür verantwortlich, dass der Puffer nach Gebrauch wieder freigegeben wird. Dies ist glücklicherweise problemlos möglich.

Über die `CoTaskMemFree`-API-Funktion können Sie die Freigabe des durch den OLE-Speicherzuordner zugeordneten Speichers erreichen. Sie wird in Visual Basic folgendermaßen deklariert:

```
Private Declare Sub CoTaskMemFree Lib "ole32.dll" (ByVal ptr As Long)
```

Fügen Sie einfach die folgende Codezeile vor dem Ende der Methode `GetCLSIDAs-String` ein, um die Zeichenfolge freizugeben, auf die die Variable `MemoryPointer` weist. Das Speicherleck wird verschwinden.

```
Call CoTaskMemFree(MemoryPointer)
```

Lösung 28

Zeichnen von OLE-Objekten

Was ist ein Objekt?

Ein Objekt ist etwas, das irgendwo im Arbeitsspeicher Ihres Systems oder aber eines anderen existiert. Eine Objektvariable enthält einen Zeiger auf eine Position im lokalen Speicher, die eine Schnittstelle enthält. Es handelt sich tatsächlich um ein Array von Zeigern auf Funktionen, doch als VB-Programmierer müssen Sie sich über solche Implementierungsdetails keine Gedanken machen. Der Zugriff auf ein Objekt kann nur über diese Zeiger auf eine Schnittstelle erfolgen. Ein bestimmtes Objekt kann dabei viele verschiedene Schnittstellen unterstützen.

Wenn Sie verschiedene Typen von Objektvariablen deklarieren, geben Sie im Grunde nur den Schnittstellenzeigertyp an, den die Variable aufnimmt. Eine `As Object` deklarierte Variable enthält z.B. immer einen Zeiger auf eine `IDispatch`-Schnittstelle. Wenn Sie eine Variable als einen bestimmten Klassentyp deklarieren, enthält diese Variable immer einen Zeiger auf die individuelle Schnittstelle dieser Klasse. Beachten Sie dabei, dass jede Schnittstelle auf `IUnknown` basiert, d.h., eine `As IUnknown` deklarierte Objektvariable kann einen Verweis auf jedes COM-Objekt enthalten, und zwar unabhängig von der verwendeten Schnittstelle. Abbildung S28-1 zeigt, wie eine Objektvariable auf eine Schnittstelle zeigt.[3]

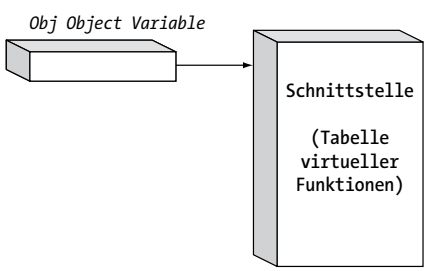


Abbildung S28-1 Objektvariablen zeigen auf einen Speicherblock, der eine Schnittstelle definiert

Jetzt betrachten wir wieder die C- und Visual Basic-Deklaration für die `OleDraw`-Funktion:

```
WINOLEAPI OleDraw(
    IUnknown * pUnk,      //Zeiger auf das zu zeichnende Objekt
    DWORD dwAspect,       //Darstellungsform für das Objekt
```

3. Ich muss hier erneut betonen, dass der folgende Absatz ohne Kenntnisse von COM wahrscheinlich unverständlich ist. Mein Buch Developing COM/ActiveX Components with Visual Basic 6.0 liefert Ihnen die erforderlichen Hintergrundinformationen.

```
    HDC hdcDraw,          //Gerätekontext, in dem gezeichnet werden soll
    LPCRECT lprcBounds    //Zeiger auf das Rechteck, in dem das
                          //Objekt gezeichnet wird
   );
Private Declare Function OleDraw Lib "ole32.dll" _
   (pUnk As Object, ByVal dwAspect As Long , _
      ByVal hdcDraw As Long, _
      lprcBounds As RECT) As Long
```

Die Variable `pUnk` ist als Zeiger auf eine `IUnknown`-Schnittstelle definiert. Kann ein `As Object` deklarierter Parameter hier eingesetzt werden? Ja, denn jede Objektschnittstelle basiert auf `IUnknown`. Doch was wird bei dieser Deklaration wirklich übergeben? Ist es die Objektvariable oder ein Zeiger auf die Objektvariable?

Stimmt – `pUnk` wird als Verweis deklariert, d.h., ein Zeiger auf die Variable `pUnk` wird an die API-Funktion übergeben, nicht der Zeiger auf die Schnittstelle selbst (die in der Variablen `pUnk` enthalten ist). Wenn es auch ein wenig merkwürdig aussieht, lautet die richtige Deklaration also folgendermaßen:

```
Private Declare Function OleDraw Lib "ole32.dll" _
   (ByVal pUnk As Object, _
      ByVal dwAspect As Long, _
      ByVal hdcDraw As Long, _
      lprcBounds As RECT) As Long
```

Es macht deshalb einen etwas merkwürdigen Eindruck auf uns, weil wir für gewöhnlich denken, dass mit `ByVal` eine »Kopie« der Variablen übergeben wird. Aber Sie wissen ja bereits, dass Zeichenfolgen eine Ausnahme bilden. Betrachten Sie Objekte als eine weitere Ausnahme. Wenn eine OLE-Funktion einen Parameter als `IUnknown *` deklariert, sollte das Objekt als Wert übergeben werden. Wenn der Parameter jedoch als `IUnknown **` deklariert wird, handelt es sich bei dem Parameter um einen Zeiger, der auf einen auf eine Schnittstelle gerichteten Zeiger gerichtet ist (mit anderen Worten, ein Zeiger auf eine Variable, die einen Zeiger auf eine Schnittstelle enthält), und in diesem Fall sollte das Objekt als Verweis übergeben werden.

Objektbeziehungen

Sobald Sie die Deklaration festlegen, erhalten Sie Fehlermeldungen, die darauf hinweisen, dass das Objekt die `IviewObject`-Schnittstelle nicht unterstützt. Die `OleDraw`-Funktion zeichnet das Objekt mit Hilfe der `IviewObject`-Schnittstelle, also ist die Funktion offensichtlich nicht für Objekte einsetzbar, die sie nicht unterstützen. Bedeutet dies, dass Sie die Funktion nicht zum Zeichnen von Steuerelementen verwenden können?

Oder bedeutet es, dass Sie der Funktion das falsche Objekt übergeben?

Dieses Problem ist in der Tat spitzfindig. Wenn Sie ein Steuerelement als Parameter an eine Funktion übergeben, dann übergeben Sie eigentlich einen Verweis auf das Erweiterungsobjekt des Steuerelements, also ein Objekt, das die Methoden und Eigenschaften des Steuerelements mit den von Visual Basic bereitgestellten Methoden und Eigenschaften kombiniert. Die Erweiterung unterstützt die `IviewObject`-Schnittstelle nicht. Statt dessen müssen Sie einen Verweis auf das Steuerelementobjekt selbst übergeben. Dieser Verweis kann, wie im folgenden Code gezeigt, mit Hilfe der Objekteigenschaft des Steuerelements ermittelt werden:

```
Private Sub cmdDrawText_Click()
   DrawTheObject Privatectl1.object
End Sub
Private Sub cmdDrawWord_Click()
   DrawTheObject MonthView1.object
End Sub

Private Sub Command1_Click()
   PrintTheObject MonthView1.object
End Sub
```

Jetzt werden Sie feststellen, dass die beiden Zeichenbefehle funktionieren. Aber das Steuerelement `MonthView` druckt nicht.

Visual Basic soll drucken

Betrachten wir erneut die Funktion `PrintTheObject`:

```
Private Function PrintTheObject(obj As Object)
   Dim r As RECT
   Dim res As Long
   Printer.ScaleMode = vbPixels
   r.Right = Printer.ScaleWidth
   r.Bottom = Printer.ScaleHeight
   res = OleDraw(obj, Aspects(cmbAspect.ListIndex), Printer.hDC, r)
   If res <> 0 Then
      Err.Raise res
   End If
   Printer.EndDoc
End Function
```

Wie erfährt Visual Basic, dass Sie jetzt etwas drucken möchten? Bei Einstellung von `ScaleMode` wird VB nur mitgeteilt, welcher Skalierungsmodus zu verwenden ist, wenn Sie beginnen, in das Druckerobjekt zu zeichnen. Der Befehl `OleDraw` legt Daten im Gerätekontext des Druckers ab, doch damit wird das Zeichensystem von Visual Basic übergangen, sodass VB nicht wissen kann, dass Sie tatsächlich eine Zeichenoperation durchgeführt haben. Wenn dann der Befehl `EndDoc` erreicht ist, erkennt Visual Basic nicht, dass etwas gezeichnet worden ist, und führt keine weiteren Aktionen durch.

Visual Basic muss mittels eines Tricks dazu veranlasst werden, den Gerätekontext zu initialisieren, bevor die `OleDraw`-Funktion aufgerufen wird. Dazu muß die Funktion wie folgt geändert werden:

```
Private Function PrintTheObject(obj As Object)
   Dim r As RECT
   Dim res As Long
   Printer.ScaleMode = vbPixels
   Printer.PSet (0, 0)
   r.Right = Printer.ScaleWidth
   r.Bottom = Printer.ScaleHeight
   res = OleDraw(obj, Aspects(cmbAspect.ListIndex), Printer.hDC, r)
   If res <> 0 Then
      Err.Raise res
   End If
   Printer.EndDoc
End Function
```

Das Drucken dieses einzelnen Pixels informiert Visual Basic darüber, dass eine Druckoperation stattgefunden hat, woraufhin der Gerätekontext richtig initialisiert und die Seite gedruckt wird, wenn die Methode `EndDoc` aufgerufen wird. Sie sehen jetzt den ganzseitigen Ausdruck eines wunderschönen Kalenderblattes.[4]

Schlussfolgerung

Sie haben vielleicht bemerkt, dass das Microsoft-Steuerelement `MonthView` sich zum Drucken in das von Ihnen vorgesehene Rechteck selbst skalieren kann, während das in Visual Basic geschriebene private Steuerelement diese Fähigkeit nicht aufweist. Hier liegt in der Tat eine der Beschränkungen von Visual Basic: Ihr Steuerelement wird nicht mit einem Mechanismus zur Bestimmung des für die `OleDraw`-

4. »Schön« bezieht sich in diesem Zusammenhang nicht auf die ästhetische Qualität des gezeichneten Bildes, die zugegebenermaßen zu wünschen übrig läßt. Es bezieht sich vielmehr auf die Tatsache, dass nach Ihren wahrscheinlich frustrierenden Bemühungen, die Lösung zu finden, praktisch jedes Bild für Sie ein Augenschmaus ist.

Funktion bereitgestellten Rechtecks ausgestattet. Sie können die ActiveX-Erweiterungstechnologie in Version 6 von Desaware's SpyWorks verwenden, um dieses Rechteck zu erkennen, ja sogar die `IViewObject`-Zeichenoperation zu überschreiben, und es Ihrem Steuerelement so ermöglichen, sich genau wie `MonthView` selbst zu skalieren.

Seien Sie also nicht überrascht, wenn diese Funktion nicht mit jedem ActiveX-Steuerelement einsetzbar ist. Obwohl jedes ActiveX-Steuerelement `IViewObject` richtig implementieren sollte, enthalten viele von ihnen interne Voraussetzungen, die die Funktion in Verbindung mit beliebigen Gerätekontexten verhindern.

Obwohl dieses Puzzle im OLE-Abschnitt liegt, gehört es eigentlich mehr in den folgenden Abschnitt »High Technology«, da alle Probleme, mit denen Sie bei der Lösung dieses Puzzles konfrontiert wurden, spitzfindig sind und eine umfassende Windows-Kenntnis voraussetzen. Wenige Visual Basic-Programmierer arbeiten häufig mit der OLE-API – in der Tat ein komplexes Gebiet. Doch es gibt viele versteckte Kleinode – wie dieses – für Visual Basic-Programmierer, die bereit sind, ein wenig in der nahezu unfassbaren Dokumentation zu graben.

Lösung 29

Was tun, wenn's richtig wehtut?

Der erste Schritt zur Lösung dieses Problems ist zu wissen, wie eine C-Funktion eine Struktur zurückgibt. Abbildung S29-1 zeigt den Stackframe für eine C-Funktion, die eine Struktur zurückgibt (vielleicht möchten Sie Tutorium 4, »Die Arbeitsweise einer DLL: In einem Stackframe«, in Teil III dieses Buches lesen, bevor Sie die Lektüre an dieser Stelle fortsetzen). Damit eine Funktion eine Struktur zurückgeben kann, muss irgendwo im Speicher eine temporäre Struktur zur Aufnahme der Daten vorhanden sein (da Strukturdaten nicht in Register passen). Die aufrufende Funktion ordnet dieser temporären Struktur Speicherplatz auf dem Stack zu und übergibt der Struktur einen Zeiger als versteckten Parameter.

Hier folgt noch eine andere Sichtweise. Betrachten Sie wieder die Deklaration der Funktion cpuspeed:

```
struct FREQ_INFO cpuspeed(int clocks);
```

Es scheint sich hier um eine Funktion zu handeln, die einen Ganzzahlparameter annimmt und eine Struktur zurückgibt. Doch ein Microsoft C-Compiler interpretiert diese Funktion beim Aufruf folgendermaßen:

```
struct FREQ_INFO *cpuspeed(struct FREQ_INFO *hiddenstructure, int clocks);
```

Das aufrufende Programm weiß, dass es zusätzlichen Speicherplatz auf dem Stack für den Rückgabewert zuweisen muss. Die Funktion `cpuspeed` weiß, dass sie in einem zusätzlichen Parameter die Ergebnisdaten ablegen soll, und dass sie den Wert dieses Parameters als Ergebnis zurückgeben sollte. Woher wissen Sie das? Weil es ein Teil der von Microsoft-Compilern verwendeten Implementierung der Aufrufkonvention ist, tief in der Sprachdokumentation versteckt. Glücklicherweise reicht das Verständnis dieser Aufrufkonvention aus, damit Sie herausfinden können, wie diese Funktion aus Visual Basic aufgerufen wird.

In Visual Basic können Sie keine API-Funktion deklarieren, die eine Struktur zurückgibt. Der Trick zur Verwendung von Funktionen, die Strukturen zurückgeben, in Visual Basic basiert auf der Tatsache, dass die temporäre Struktur nicht auf dem Stack liegen muss. Sie können eine Struktur zur Aufnahme des Rückgabewertes zuweisen. Anstatt dass die Sprache einen Zeiger auf die Struktur in einem versteckten Parameter übergibt, müssen Sie den Zeiger auf die Struktur explizit übergeben. Hierzu fügen Sie der Funktion einen Pseudoparameter hinzu, der einen Zeiger auf die Struktur enthält, die von der Funktion »zurückgegeben« wird. Der tatsächliche Stackframe für die Funktion `cpuspeed` ist in Abbildung S29-2 dargestellt.

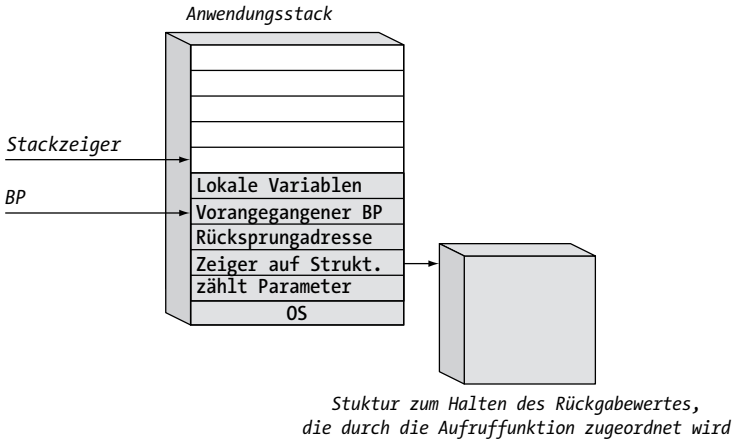


Abbildung S29-1 So gibt eine C-Funktion eine Struktur zurück

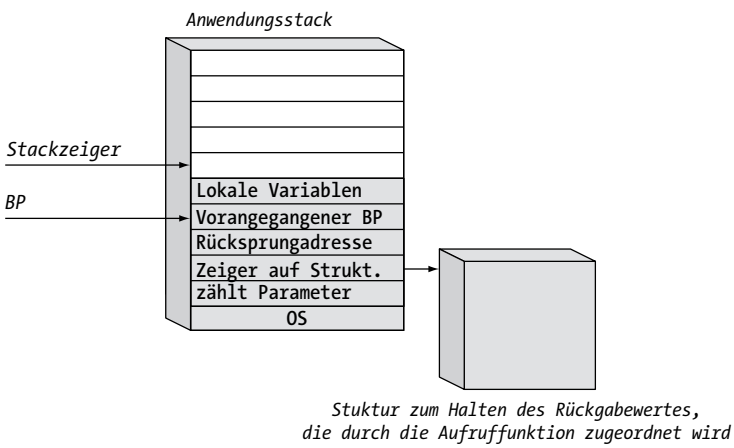


Abbildung S29-2 Der Stackframe für die Funktion `cpuspeed`

```
Declare Function cpuspeed Lib "cpuinf32.dll" (HiddenStruct As _
FREQ_INFO, ByVal counts As Long) As Long
```

Die Funktion `cpuspeed` erkennt den Zeiger auf die Struktur `FREQ_INFO` und legt den Ergebniswert in der Struktur ab, bevor sie einen Wert zurückgibt. Der tatsächliche Rückgabewert für die Funktion ist die Adresse der versteckten Struktur, doch da die Funktion die Ergebnisdaten in der als Parameter `HiddenStruct` übergebenen Struktur ablegt, müssen Sie diese Adresse nicht verwenden.

Warten Sie, das ist noch nicht alles ...

Beim Aufruf der Funktion werden Sie feststellen, dass sie einen »Bad DLL«-Aufrufkonventionsfehler auslöst. Dieser Fehler tritt auf, wenn die aufgerufene Funktion den Stackzeiger nicht wieder richtig positioniert.

Doch warum sollten Sie eine Aufrufkonventionsfehlermeldung erhalten? Stimmen die Parameter nicht?

Sie stimmen. Die Ursache des Problems liegt darin, dass die Funktion `cpuspeed` die C-Aufrufkonvention verwendet. Normalerweise verwendet API die Aufrufkonvention `stdcall` (`standard call`), bei der die aufrufende Funktion Parameter auf dem Stack ablegt und der aufgerufene Parameter die Parameter vom Stack stößt (sodass der Stackzeiger wieder auf dieselbe Position weist wie vor der Ablage von Parametern auf dem Stack). Unter der C-Aufrufkonvention ist die aufrufende Funktion dafür verantwortlich, die Parameter vom Stack zu stoßen. Die Funktion `cpuspeed` stößt die beiden Parameter nicht, wie Visual Basic erwartet, vom Stack, sodass der »Bad DLL«-Aufrufkonventionsfehler auftritt.

Doch hier sind zwei Feinheiten zu berücksichtigen:

- Visual Basic ist raffiniert genug, diese Situation zu erkennen und den Stackzeiger für Sie zu positionieren.
- Wenn der »Bad DLL«-Aufrufkonventionsfehler erkannt wird, sind die Ergebnisdaten bereits in der temporären Struktur abgelegt.

Mit anderen Worten, diesen Fehler können Sie getrost ignorieren!

Das Projekt `CPUInfo` zeigt die Verwendung einiger in **cpuinf32.dll** enthaltener Funktionen zur Ermittlung von Informationen über einen Intel-Prozessor. Die Funktion `ShowSpeed` ermittelt mit Hilfe der Funktion `cpuspeed` die Geschwindigkeit der CPU und fügt die Informationen wie nachstehend gezeigt in ein Listenfeld ein:

```
Private Sub ShowSpeed()
   Dim freq As FREQ_INFO
   Dim res&
   On Error Resume Next ' "Bad DLL"-Aufrufkonventionsfehler tritt hier auf!
   ' UND WIR KÜMMERN UNS NICHT DRUM!!!
   res = cpuspeed(freq, 0)  ' Hier immer 0 verwenden
   lstDisplay.AddItem "Raw CPU Speed: " & freq.raw_freq & " Mhz"
   lstDisplay.AddItem "Normalized CPU Speed: " & freq.norm_freq _
   & " Mhz"
End Sub
```

Nebenbei gesagt ...

Diese Lösung ist nicht für Strukturen geeignet, die höchstens 8 Bytes lang sind. In diesen Fällen ist der C-Compiler raffiniert genug, die Daten in Register zu übertragen.

Wie behandeln Sie diese kleineren Strukturen?

Der Trick besteht darin, in der Visual Basic-Deklaration die Rückgabe eines `Currency`-Wertes festzulegen. Der `Currency`-Wert hat eine Länge von 8 Bytes. Dann können Sie die Daten von einem temporären `Currency`-Wert in eine Struktur des richtigen Typs kopieren.

Dieser kurze Beispielcode zeigt die Vorgehensweise:

```
Private Declare Function Test Lib "mylib" ( ) As Currency

Private Sub Command2_Click()
   Dim m As mystruct
   Dim c As Currency
   c = Test( )
' Sie speichern das Ergebnis in einem temporären Currency-Wert
   ' Dann kopieren Sie die Daten von dem Currency-Wert in die Struktur.
   RtlMoveMemory m, c, 8
End Sub
```

Lösung 30

Dateioperationen, Teil 1

Beginnen sollte die Suche sinnvollerweise bei der Struktur `SHFILEOPSTRUCT` und der zugehörigen VB-Deklaration. Wo könnte die Ursache des Problems liegen?

Die C-Deklaration der Struktur sieht folgendermaßen aus:

```
typedef struct _SHFILEOPSTRUCT { // shfos
    HWND         hwnd;
    UINT         wFunc;
    LPCSTR       pFrom;
    LPCSTR       pTo;
    FILEOP_FLAGS fFlags;
    BOOL         fAnyOperationsAborted;
    LPVOID       hNameMappings;
    LPCSTR       lpszProgressTitle;
} SHFILEOPSTRUCT, FAR *LPSHFILEOPSTRUCT;
```

Und die VB-Deklaration für die Struktur lautet:

```
Type SHFILEOPSTRUCT
    hwnd As Long
    wFunc As Long
    pFrom As String
    pTo As String
    fFlags As Integer
    fAnyOperationsAborted As Long
    hNameMappings As Long
    lpszProgressTitle As String '  nur verwendet, wenn FOF_SIMPLEPROGRESS
End Type
```

Zunächst haben Sie vielleicht die Zeichenfolgen in Verdacht. Wenn Sie Tutorium 7, »Klassen, Strukturen und benutzerdefinierte Typen« (in Teil III dieses Buches) gelesen haben, wissen Sie, dass die Verwendung dynamischer Zeichenfolgen innerhalb von Strukturen mit Risiken verbunden sein kann – besonders dann, wenn die API-Funktion den Inhalt dieser Felder in irgendeiner Weise modifiziert. Der Grund liegt darin, dass eine dynamische Zeichenfolge in einer Struktur einen 32-Bit-BSTR-Wert enthält. Die Win32-API kann BSTRs nicht verwenden, sodass jeder Versuch, die Inhalte dieser Felder zu modifizieren, mit hoher Wahrscheinlichkeit einen Speicherausnahmefehler auslöst.

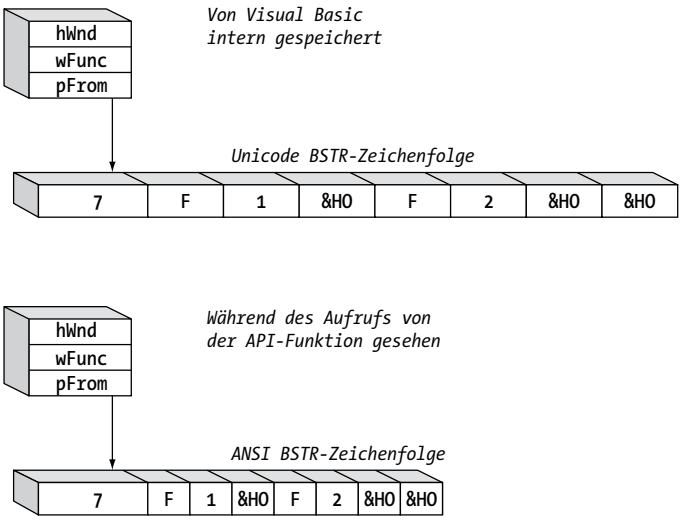


Abbildung S30-1 Übergabe der Struktur `SHFILEOPSTRUCT` an eine API-Funktion

Was geschieht, wenn sich eine dynamische Zeichenfolge in einer als Parameter übergebenen Struktur befindet? Visual Basic erstellt eine temporäre Struktur und ersetzt das `BSTR`-Feld durch eine `ANSI-BSTR`-Zeichenfolge. Abbildung S30-1 stellt diesen Sachverhalt für die ersten drei Felder in der Struktur `SHFILEOPSTRUCT` dar, wobei das Feld `pFrom` zwei Dateinamen enthält, `F1` und `F2`.

In der Abbildung können Sie erkennen, dass das Feld `pFrom` beim Aufruf der Funktion auf eine ANSI-Zeichenfolge zeigt, die zwei durch NULL-Zeichen getrennte Dateinamen und am Ende zwei abschließende NULL-Zeichen enthält. Genau dies erwartet die API-Funktion, sodass nur dann ein Problem auftreten könnte, wenn die Funktion den Inhalt dieser Struktur in irgendeiner Weise modifizieren würde. Ist dies möglich?

Die Zeichenfolge in der Struktur ist in der C-Deklaration als `LPCSTR` (`LP` = Long (far) Pointer, `C` = Constant, `STR` = String) definiert. Der Constant-Code ist Indiz dafür, dass die API-Funktion den Wert der Zeichenfolge bzw. des Feldes in der Struktur nicht ändert. Gemäß Deklaration und Dokumentation ist es sicher und richtig, wie in diesem Beispiel gezeigt dynamische Zeichenfolgen zu verwenden. Nichtsdestoweniger misslingt es.

Noch ein Blick auf die Strukturanordnung

Jetzt werden wir näher untersuchen, wie die Felder der Struktur `SHFILEOPSTRUCT` im Speicher angeordnet werden. Am Anfang der Headerdatei **shellapi.h** finden Sie folgende Zeile:

```
#include <pshpack1.h>
```

Hiermit wird die Einzelbyteanordnung aktiviert. Dies bedeutet, dass die Struktur wie in Abbildung S30-2 gezeigt im Speicher vorliegt. Jedes Datenfeld in der Struktur folgt im Speicher unmittelbar auf seinen Vorgänger.

Aber Visual Basic verwendet beim Schreiben der Daten in Dateien, die auf der Festplatte gespeichert sind, nur die Einzelbyteanordnung. Bei der Übergabe von Strukturen an API-Funktionen verwendet Visual Basic die Regeln der natürlichen Anordnung, d. h. jedes Feld befindet sich an einer durch die Größe des Feldes teilbaren Speicherposition. Aus diesem Grund befinden sich 32-Bit-Variablen an einer durch 4 teilbaren Byteadresse, und die Struktur erhält eine zusätzliche Füllung, um diese Anordnungsregel zu erzwingen. Visual Basic übergibt also an die Funktion `SHFileOperation` eine Struktur, die der Darstellung in Abbildung S30-3 entspricht.

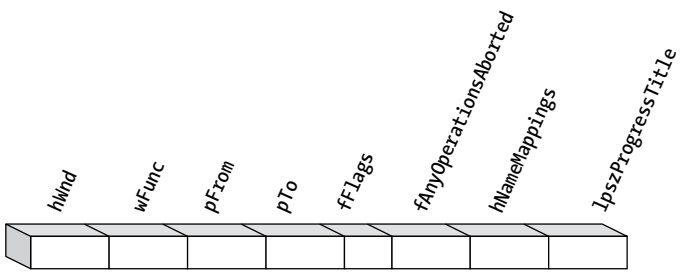


Abbildung S30-2 Erwartete Anordnung der Struktur `SHFILEOPSTRUCT` im Speicher

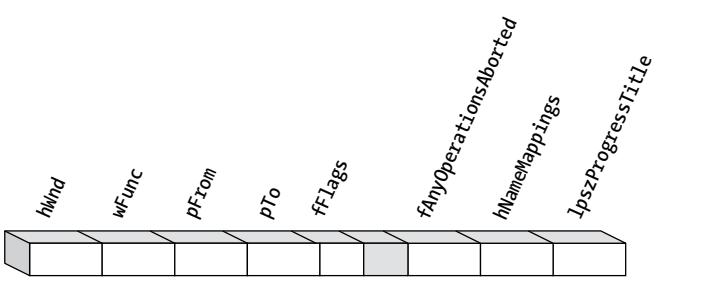


Abbildung S30-3 Tatsächliche Anordnung der Struktur `SHFILEOPSTRUCT` bei Übergabe an die Funktion `SHFileOperation`

Die Felder `fAnyOperationsAborted`, `hNameMappings` und `lpszProgressTitle` scheinen an falscher Stelle im Speicher zu liegen! Führen Sie das Programm zur Überprüfung erneut aus, doch setzen Sie das Feld `lpszProgressTitle` diesmal nicht auf eine Zeichenfolge (löschen Sie die Zeile, in der es auf eine Zeichenfolge gesetzt wird).

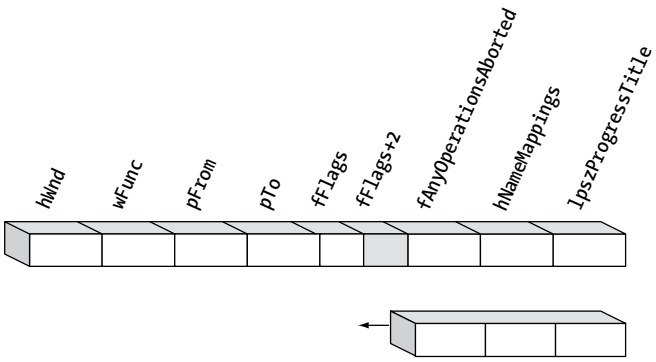


Abbildung S30-4 Verschieben von Daten in der Struktur `SHFILEOPSTRUCT`, um Einzelbyteanordnung zu erzielen

Jetzt funktioniert die Funktion! Nun ja, sie funktioniert in dem Sinne, dass die Datei kopiert wird und die Funktion beim Beenden des Programms nicht abstürzt. Die Zeichenfolge wird immer noch nicht angezeigt.

Die Lösung des Problems ist vom Konzept her einfach – Sie müssen auf irgendeine Art und Weise die Struktur komprimieren, damit das Füllen vor der Übergabe an die API-Funktion überflüssig wird, wie in Abbildung S30-4 gezeigt.

Dies erreichen Sie am einfachsten durch die Verwendung speziell für das UDT-Packing (User-Defined Type Packing – Anordnung nach benutzerdefiniertem Typ) konzipierter Software wie z. B. SpyWorks von Desaware. Mit diesem Ansatz definieren Sie die Strukturfelder für das UDT-Packingprogramm und übergeben sie an die Struktur. Die Software ordnet sie in einem Speicherpuffer in Einzelbyteanordnung an und gibt einen Speicherpuffer zurück, den sie an die API-Funktion übergeben können. Dann wird die Anordnung wieder rückgängig gemacht, und die Daten werden wieder in der Struktur abgelegt, wenn die Rückgabe durch die Funktion erfolgt.

Diese Software ist besonders dann von Nutzen, wenn mehrere Fehlanordnungen in einer Struktur vorliegen. Wenn wie hier nur eine Fehlanordnung vorliegt, können Sie diese Operation selbst durchführen.

Die `DoACopy`-Funktion wird wie folgt geändert:

```
' Kopiervorgang ausführen
Private Sub DoACopy()
   Dim sh As SHFILEOPSTRUCT
   Dim src$, dest$
   Dim res&
```

```
    src$ = GetNullsString(FileList)
    dest$ = "A:\ " & Chr$(0) & Chr$(0)

    sh.hwnd = hwnd
    sh.wFunc = FO_COPY
    sh.pFrom = src$
    sh.pTo = dest$
    sh.fFlags = FOF_RENAMEONCOLLISION Or FOF_SIMPLEPROGRESS
    sh.hNameMappings = 0

    res = DoTheShellCall(sh, "Files are being copied to floppy")

End Sub
```

Wir ändern `SHFILEOPSTRUCT` auch wie hier gezeigt, sodass das Feld `lpszProgressTitle` den Typ `Long` erhält. Warum? Weil die Funktion `DoTheShellCall` Daten innerhalb der Struktur verschieben wird, d.h. dass sich die Daten im Feld `lpszProgressTitle` ändern werden (zumindest aus der Perspektive von Visual Basic). Dieser Weg führt mit Sicherheit zu einem Speicherausnahmefehler. Änderung zu `Long` und explizites Setzen der Variablen auf einen Zeiger, der auf eine ANSI-Zeichenfolge weist, nimmt Ihnen die Sorge, dass der Inhalt dieses Feldes Visual Basic irritieren könnte.

```
Private Type SHFILEOPSTRUCT
        hwnd As Long
        wFunc As Long
        pFrom As String
        pTo As String
        fFlags As Integer
        fAnyOperationsAborted As Long
        hNameMappings As Long
        lpszProgressTitle As Long '  nur verwendet, wenn FOF_SIMPLEPROGRESS
End Type
```

Die Funktion `DoTheShellCall` führt den Aufruf von `SHFileOperation` aus und wird wie folgt definiert:

```
Private Declare Sub lstrcpy Lib "kernel32" (dest As Any, src As Any)

' Beachten Sie hier die Anordnungstricks
Private Function DoTheShellCall(sh As SHFILEOPSTRUCT, Optional _
 SimpleCaption As String) As Long
   Dim titlebuf() As Byte
```

```
    Dim res&
    ReDim titlebuf(Len(SimpleCaption) + 1)
    Call lstrcpy(titlebuf(0), ByVal SimpleCaption)
    sh.lpszProgressTitle = VarPtr(titlebuf(0))
    RtlMoveMemory ByVal (VarPtr(sh.fFlags) + 2), ByVal _
    VarPtr(sh.fAnyOperationsAborted), 12
    res = SHFileOperation(sh)
    RtlMoveMemory ByVal VarPtr(sh.fAnyOperationsAborted), ByVal _
    (VarPtr(sh.fFlags) + 2), 12
    DoTheShellCall = res
End Function
```

Die Funktion weist ein Bytearray zur Aufnahme der ANSI-Zeichenfolge zu, die den gewünschten Titel enthalten wird. Die Funktion `lstrcpy` kopiert die Zeichenfolge in den Bytepuffer. In diesem Fall sind beide Parameter der Funktion `lstrcpy` `As Any` definiert. Der erste Parameter wird als Byte übergeben, sodass die Funktion `lstrcpy` einen Zeiger auf das erste Byte im Array erkennt. Der zweite Parameter wird als `ByVal As String` übergeben (beachten Sie, dass `ByVal` explizit in den Funktionsaufruf einbezogen werden muss, da es nicht in der Deklaration enthalten ist). Dies bewirkt, dass die Zeichenfolge als mit dem NULL-Zeichen abgeschlossene ANSI-Zeichenfolge übergeben wird. Die Funktion `lstrcpy` kopiert diese Zeichenfolge in den Bytepuffer.

In dem Feld `lpszProgressTitle`, jetzt eine `Long`-Variable, wird die Adresse des ersten Bytes in dieser Zeichenfolge abgelegt. Dann komprimiert die Funktion `RtlMoveMemory` die Struktur. Das Ziel ist die Adresse der Füllung, die 2 Bytes hinter dem Feld `fFlags` liegt, sodass die Adresse als Adresse des Feldes `fFlags` plus 2 berechnet werden kann. Die Quelle ist das Feld `fAnyOperationsAborted`. Insgesamt werden 12 Bytes kopiert. Damit wird die in Abbildung S30-4 dargestellte Verschiebung erzielt.

Die Speicherverschiebung wird nach Aufruf der Funktion rückgängig gemacht, sodass Sie die Daten in der Struktur richtig lesen können, falls Sie etwaige Änderungen der Felder untersuchen müssen.

Hiermit ist das Problem des Dateikopierens gelöst, doch wie Sie bald sehen werden, haben wir erst mit der Betrachtung der Funktion `ShFileOperation` begonnen.

Lösung 31

Dateioperationen, Teil 2

Welchen Ansatz wählen Sie zur Lösung eines solchen Problems? Machen Sie's wie ich: Gehen Sie Schritt für Schritt vor. In der Beschreibung des Puzzles erwähnte ich, dass ich für die Lösung zwei Stunden benötigte. Das Resultat ist, dass ich mich noch gut an die einzelnen Lösungsschritte erinnere.

Überprüfen Sie die Strukturen

Der erste Schritt ist stets die gründliche Überprüfung Ihrer Deklarationen. In diesem Fall ist nur die Struktur `SHNAMEMAPPING` neu deklariert, die folgendermaßen in C und Visual Basic definiert wurde:

```
typedef struct _SHNAMEMAPPING { // shnm
    LPSTR pszOldPath; // Adresse des alten Pfadnamens
    LPSTR pszNewPath; // Zeiger auf den neuen Pfadnamen
    int   cchOldPath; // Anzahl der Zeichen im alten Pfadnamen
    int   cchNewPath; // Anzahl der Zeichen im neuen Pfadnamen
} SHNAMEMAPPING, FAR *LPSHNAMEMAPPING;
```

```
Private Type SHNAMEMAPPING
        pszOldPath As String
        pszNewPath As String
        cchOldPath As Long
        cchNewPath As Long
End Type
```

Wenn Sie dynamische Zeichenfolgen in Strukturen sehen, fragen Sie sich zuerst, ob die API-Funktion versuchen könnte, diese Felder zu modifizieren. Wenn ja, können Sie keine dynamischen Zeichenfolgen verwenden. Dies wird ausführlich behandelt in Tutorium 7, »Klassen, Strukturen und benutzerdefinierte Typen« (in Teil III dieses Buches). Da die Struktur hier von der API-Funktion selbst geladen wird und es sich bei den Parametern `pszOldPath` und `pszNewPath` jeweils um reguläre Zeichenfolgen mit abschließendem NULL-Zeichen (`LPSTR`) handelt, ist es klar, dass diese Parameter nicht als `String` definiert werden müssen. Sie müssen als `Long`-Variablen deklariert werden. Sie müssen die Zeichenfolgendaten mit Hilfe des VB-Codes aus den Zeigern extrahieren.

Erster Codetest

Mein nächster Schritt war das Setzen eines Unterbrechungspunktes in der Zeile `Call SHFreeNameMappings` der Funktion `DoACopy`, wie nachstehend gezeigt:

```
' Kopiervorgang ausführen
Private Sub DoACopy()
   Dim sh As SHFILEOPSTRUCT
   Dim src$, dest$
   Dim res&

   src$ = GetNullsString(FileList)
   dest$ = "A:\ " & Chr$(0) & Chr$(0)

   sh.hwnd = hwnd
   sh.wFunc = FO_COPY
   sh.pFrom = src$
   sh.pTo = dest$
   sh.fFlags = FOF_RENAMEONCOLLISION Or FOF_WANTMAPPINGHANDLE
   sh.hNameMappings = 0

   res = DoTheShellCall(sh, "Files are being copied to floppy")

   If sh.hNameMappings <> 0 Then
 ' Können Sie hier die Dateinamen lesen?
      Call SHFreeNameMappings(sh.hNameMappings)
   End If

End Sub
```

Warum testete ich den Code an diesem Punkt, als noch kein Code zum Arbeiten mit den Daten geschrieben war? In erster Linie war es Neugier. Außerdem wuchs mein Misstrauen gegenüber der Dokumentation. Sie war sehr vage – bezeichnete das Feld `sh.hNameMappings` als »Handle« (Zugriffsnummer). Was bedeutet dies? Handelt es sich um einen Zeiger auf ein Array von Strukturen? Wenn ja, woher weiß man, wie viel Strukturen das Array enthält? Vielleicht ist die letzte Struktur eine NULL-Struktur oder eine Struktur mit leeren Zeichenfolgen? Vielleicht ist die Zugriffsnummer ein Array von Zeigern auf Strukturen? Ich verbrachte eine gute halbe Stunde damit, MSDN und die Microsoft-Website nach Hinweisen zu durchkämmen, bevor ich aufgab. Also versuchte ich herauszufinden, welche Typen von Werten das Feld enthielt.

Die Ergebnisse? Das Feld `hNameMappings` hatte immer den Wert Null, unabhängig davon, was ich versuchte, oder welche Dateien ich kopierte. Null. NULL. Na denn.

Ich war fast überzeugt, dass es sich um einen Windows-Bug handelte, als ich es spaßeshalber unter Windows 95 versuchte (meine Entwicklungen führe ich in der Regel unter NT 4.0 durch). Bingo! Ein anderer Wert als Null wurde immer dann angezeigt, wenn ein Dateiname auftauchte.

Und die Microsoft-Dokumentation enthält absolut keinen Hinweis darauf, dass dieses Merkmal nicht unter NT 4.0 unterstützt wird. Nicht den geringsten.[5]

Lesen des Arrays

Jetzt hatte ich eine »Zugriffsnummer« und musste nur noch herausfinden, was ich damit anfangen sollte. Der Dokumentation zufolge enthielt die Zugriffsnummer ein Array der `SHNAMEMAPPING`-Strukturen. Darum lag der Schluss nahe, dass es sich bei der Zugriffsnummer um einen Zeiger auf ein Array dieser Strukturen handelte. Also schrieb ich folgenden Code:

```
' Kopiervorgang ausführen
Private Sub DoACopy()
   Dim sh As SHFILEOPSTRUCT
   Dim src$, dest$
   Dim res&
   Dim NameMapping As SHNAMEMAPPING
   Dim OriginalName As String
   Dim FinalName As String

   src$ = GetNullsString(FileList)
   dest$ = "A:\ " & Chr$(0) & Chr$(0)

   sh.hwnd = hwnd
   sh.wFunc = FO_COPY
   sh.pFrom = src$
   sh.pTo = dest$
   sh.fFlags = FOF_RENAMEONCOLLISION Or FOF_WANTMAPPINGHANDLE
   sh.hNameMappings = 0
```

5. Oh, Sie finden den Hinweis für dieses Kapitel in Anhang A unfair (den mit der Anmerkung, dass die Funktion als unter NT und Windows 95/98 ausführbar dokumentiert ist)? Nun, ich hielt ihn solange für unfair, bis ich entdeckte, das die Dokumentation Unrecht hatte. Das gehört zum springenden Punkt dieses Puzzles – wenn die Ergebnisse Ihrer Experimente der Dokumentation widersprechen, ist die Möglichkeit naheliegend, dass die Dokumentation Unrecht hat und Sie Recht haben.

```
    res = DoTheShellCall(sh, "Files are being copied to floppy")

    If sh.hNameMappings <> 0 Then
       RtlMoveMemory NameMapping, ByVal sh.hNameMappings, _
       Len(NameMapping)
       If NameMapping.cchOldPath <> 0 Then
           OriginalName = String$(NameMapping.cchOldPath + 1, Chr$(0))
           Call lstrcpy(ByVal OriginalName, ByVal _
           NameMapping.pszOldPath)
       End If
       If NameMapping.cchNewPath Then
           FinalName = String$(NameMapping.cchNewPath + 1, Chr$(0))
           Call lstrcpy(ByVal FinalName, ByVal NameMapping.pszNewPath)
       End If
       lstFiles.AddItem OriginalName
       lstFiles.AddItem "- renamed to -"
       lstFiles.AddItem FinalName
       Call SHFreeNameMappings(sh.hNameMappings)
    End If

End Sub
```

Ich hatte immer noch keine Idee, wie ich die Anzahl der im Array enthaltenen Strukturen ermitteln sollte, doch war ich zuversichtlich, dass zumindest der erste Eintrag im Array gültig sein würde, wenn nach Ausführung des Programms der Wert von `hNameMappings` gleich Null wäre.

Die Funktion `RtlMoveMemory` behandelt den Wert im Feld `hNameMappings` als Zeiger und kopiert ihn in eine `SHNAMEMAPPING`-Struktur mit Namen `NameMapping`. Das Feld `cchOldPath` in der Struktur `NameMapping` enthält die Länge des ursprünglichen Dateinamens, also initialisierte ich die Zeichenfolge `OriginalName` mit dieser Länge (plus 1 für das abschließende NULL-Zeichen). Die Funktion `lstrcpy` in diesem Beispielprogramm ist wie folgt definiert:

```
Private Declare Sub lstrcpy Lib "kernel32" (dest As Any, src As Any)
```

Beide Parameter sind `As Any` definiert, sodass der Funktionsaufruf für Zeichenfolgen- und `Long`-Parameter explizit `ByVal`-Ausdrücke enthalten muss. Damit wird die Zeichenfolge, auf die das Feld `pszOldPath` weist, in die Zeichenfolge `OriginalName` kopiert. Der Vorgang wird für die Zeichenfolge `FinalName` wiederholt. Schließlich werden die Zeichenfolgen dem Listenfeld hinzugefügt.

Die ersten Ergebnisse der Ausführung auf meinem System bestanden in der Anzeige unsinniger Zeichen im Listenfeld. Auf Ihrem System könnte es sogar zum Absturz führen. Beim Durchsehen des Programms und Prüfen der tatsächlichen Werte der vier Felder kam ich zu folgenden Ergebnissen:

- `pszOldPath` = 1
- `pszNewPath` = &H828509FC
- `cchOldPath` = 4
- `cchNewPath` = 16

Ein wenig gesunder Menschenverstand hilft hier weiter. Keine gültige Zeichenfolge hat die Adresse 1. Die Quelldatei in meinem Test hatte 15 Zeichen, nicht 4. In diesem Fall handelt es sich bei einer Zugriffsnummer offensichtlich nicht um einen Zeiger.

Erforschen der Zugriffsnummer

Die Versuchung aufzugeben war zwar groß, doch die Projektleitung bestand auf diesem Puzzle, und ich hatte bereits viel Zeit in das Problem investiert. So entschloss ich mich, den Inhalt dieser so genannten Zugriffsnummer näher zu betrachten. Ich fügte den folgenden Code zur Funktion `DoACopy` hinzu und brachte das Programm auf den Stand, den Sie im Lösungsverzeichnis des Projektes **FileOp2.vbp** finden (auf der Begleit-CD-ROM). Die Funktion `GetDataString` ist eine im Modul **errstring.bas** enthaltene Speicheranzeigefunktion und wird definiert in Tutorium 2, »Der Arbeitsspeicher – da, wo alles beginnt« (in Teil III dieses Buches).

```
If sh.hNameMappings <> 0 Then
      OriginalName = GetDataString(sh.hNameMappings, 16)
      Debug.Print OriginalName
      RtlMoveMemory NameMapping, ByVal sh.hNameMappings, _
      Len(NameMapping)
      OriginalName = GetDataString(NameMapping.pszNewPath, 16)
      Debug.Print OriginalName
```

Der Code basiert auf der Voraussetzung, dass es sich bei dem einzigen Feld im Speicher, auf dass die Struktur `hNameMappings` weist, und das zumindest annähernd wie ein Zeiger aussieht, um die Daten im Feld `pszNewPath` handelt.

Die erste `Debug`-Anweisung erbrachte folgende Daten:

```
01 00 00 00 FC 09 85 82 04 00 00 00 10 00 00 00
```

Die Daten, auf die das Feld `pszNewPath` wies, die sich in diesem Fall an Adresse &H828509FC befanden, sahen folgendermaßen aus:

```
B4 A4 94 82 CC 22 85 82 0F 00 00 00 1B 00 00 00
```

Dies erinnert stark an eine `NameMapping`-Struktur. Die ersten beiden 32-Bit-Werte sind `&H8294A4B4` und `&H828522CC`, wobei es sich um Zeiger handeln könnte. Die nächsten beiden 32-Bit-Werte sind 15 und 27. Der Name der Quelldatei bestand aus 15 Zeichen, ebenfalls ein Indiz dafür, dass es sich um eine `NameMapping`-Struktur handeln könnte.

Welche Fakten hatte ich gewonnen?

Bei der Zugriffsnummer im Feld `hNameMappings` der `SHFILEOPSTRUCT`-Struktur handelt es sich fast mit Sicherheit um einen Speicherzeiger. Er zeigt auf einen Datenblock, der mindestens zwei 32-Bit-Werte des Typs `Long` enthält. Der zweite dieser Werte ist ein Zeiger auf eine `SHNAMEMAPPING`-Struktur.

Was ist der erste Wert?

Ich versuchte, zwei Dateien auf einmal zu kopieren, einfach um zu sehen, was dann geschieht. Der erste Wert im Speicherpuffer, auf den die Zugriffsnummer zeigte, änderte sich sofort zu 2.

Das Geheimnis ist gelüftet

Die Zugriffsnummer kann als von mir selbst erfundener Zeiger auf eine Struktur betrachtet werden, der wie nachstehend beschrieben in Projekt **FileOp2B.vbp** (auf der Begleit-CD-ROM) definiert ist. Die Definition ist auch im Lösungsverzeichnis dieses Puzzles enthalten:

```
Private Type SomeSortOfHandle
   mappingcount As Long
   MappingAddress As Long
End Type

Die Funktion DoACopy wurde wie folgt umgeschrieben:

' Kopiervorgang ausführen
Private Sub DoACopy()
   Dim sh As SHFILEOPSTRUCT
   Dim src$, dest$
   Dim res&
   Dim NameMapping As SHNAMEMAPPING
   Dim MappingInfo As SomeSortOfHandle
   Dim mappingcounter As Long
```

```
    Dim OriginalName As String
    Dim FinalName As String
    src$ = GetNullsString(FileList)
    dest$ = "A:\ " & Chr$(0) & Chr$(0)

    sh.hwnd = hwnd
    sh.wFunc = FO_COPY
    sh.pFrom = src$
    sh.pTo = dest$
    sh.fFlags = FOF_RENAMEONCOLLISION Or FOF_WANTMAPPINGHANDLE
    sh.hNameMappings = 0


    res = DoTheShellCall(sh, "Files are being copied to floppy")

    If sh.hNameMappings <> 0 Then
' Kopieren der Zuordnungsstruktur in eine von uns erstellte Struktur
       RtlMoveMemory MappingInfo, ByVal sh.hNameMappings, _
       Len(MappingInfo)
       ' Betrachten der Daten jeder SHNAMEMAPPING-Struktur
       For mappingcounter = 1 To MappingInfo.mappingcount
          RtlMoveMemory NameMapping, ByVal _
          CLng(MappingInfo.MappingAddress + Len(NameMapping) _
          * (mappingcounter - 1)), Len(NameMapping)
          If NameMapping.cchOldPath <> 0 Then
             OriginalName = String$(NameMapping.cchOldPath + 1, Chr$(0))
             Call lstrcpy(ByVal OriginalName, ByVal _
             NameMapping.pszOldPath)
          End If
          If NameMapping.cchNewPath Then
             FinalName = String$(NameMapping.cchNewPath + 1, Chr$(0))
             Call lstrcpy(ByVal FinalName, ByVal NameMapping.pszNewPath)
          End If
          lstFiles.AddItem OriginalName
          lstFiles.AddItem "- renamed to -"
          lstFiles.AddItem FinalName
       Next mappingcounter
       Call SHFreeNameMappings(sh.hNameMappings)
    End If

End Sub
```

Eine Struktur des Typs `SomeSortOfHandle` mit Namen `MappingInfo` wird mit Hilfe der Funktion `RtlMoveMemory` von der Adresse geladen, auf die die Zugriffsnummer zeigt. Das Feld `mappingcount` dieser Struktur enthält die Anzahl der `SHNAMEMAPPING`-Strukturen im Array, auf die das Feld `MappingAddress` zeigt. Jede `SHNAMEMAPPING`-Struktur wird jeweils in einem Schleifendurchlauf in die Struktur `NameMapping` kopiert.

Die folgende Zeile verdient besondere Beachtung:

```
RtlMoveMemory NameMapping, ByVal CLng(MappingInfo.MappingAddress _
+ Len(NameMapping) * (mappingcounter - 1)), Len(NameMapping)
```

Der zweite Parameter zeigt eine Berechnung der Adresse jeder Struktur im Array. Bei dem Ausdruck `MappingInfo.MappingAddress` handelt es sich um die Adresse der ersten Struktur im Array. Der Ausdruck `Len(NameMapping) * (mappingcounter - 1)` berechnet basierend auf dem Wert der Schleifenvariablen mappingcounter einen Offset zur richtigen Struktur.

Das endgültige Ergebnis ähnelt der Anzeige in Abbildung S31-1.

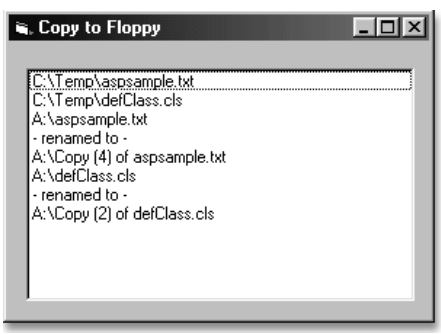


Abbildung S31-1 Typische Anzeige nach dem Kopieren der Dateien auf eine Diskette

Schlussfolgerung

Dies ist vielleicht das komplizierteste Puzzle im Buch. Doch es veranschaulicht am besten die Probleme, mit denen Sie konfrontiert sind, wenn Sie die neuesten Merkmale von Windows nutzen möchten. Die Win32-API ist umfangreich, und Dokumentation sowie Beispiele von Microsoft sind manchmal unzulänglich. Wenn Sie zuweilen Schwierigkeiten haben, präzise Antworten von Microsofts technischem Support zu erhalten, seien Sie nicht zu kritisch – vielleicht wissen die es ja selbst nicht. Vertrauen Sie in solchen Fällen auf Ihre eigene Experimentierfähigkeit, forschen Sie, und finden Sie am Ende selbst heraus, welche Schritte durchzuführen sind, damit die Funktion funktioniert. Dann ist es durchaus möglich, dass Sie und der 22-jährige, vor sechs Monaten von Microsoft eingestellte Hochschulabsolvent, der den Code geschrieben hat, die einzigen Menschen sind, die die Lösung kennen.

Lösung 32

Animierte Rechtecke

Sie wurden dreifach herausgefordert:

1. Finden Sie heraus, wie Sie die Funktion dazu veranlassen, die Animation durchzuführen, wenn die `hwnd`-Eigenschaft – wie in der Dokumentation beschrieben – auf 0 gesetzt ist.
2. Führen Sie ein Beispiel an, in dem `IDANI_OPEN` und `IDANI_CLOSE` als Animationsparameter verwendet werden.
3. Geben Sie eine Erklärung, die das Verhalten der Funktion im Beispiel mit der Dokumentation in Einklang bringt.

Die Antworten sind einfach.

1. Ich weiß es nicht.
2. Ich habe keine Möglichkeit gefunden, diese Parameter unter Windows 95, 98 oder NT 4 einzusetzen.
3. Meine besten Spekulationen (und ich betone, es sind nur Spekulationen) sind (a), dass es sich um einen Windows-Bug handelt oder (b), dass die Funktion gemäß einer bestimmten Spezifikation definiert, jedoch nie vollständig implementiert wurde – und niemand hat sich darum gekümmert, die Autoren der Dokumentation zu informieren.

Bis zum nächsten Mal

Als ich mein Buch »Visual Basic Programmer's Guide to the Win32 API« schrieb, habe ich die Mehrzahl der API-Funktionen eingehend behandelt, die für einen Visual Basic-Programmierer von Interesse sein könnten. In meinen kühnsten Träumen hätte ich nicht erwartet, dass Windows so rasant an Komplexität zunehmen würde. Der Leistungsumfang der Win32-API hat immens zugenommen, weit über den Punkt hinaus, an dem ein Autor noch ihre gesamten Inhalte für andere Visual Basic-Programmierer praxisgerecht aufbereiten kann.

Wenn Sie versucht haben, alle Puzzle in diesem Buch zu lösen, bin ich überzeugt, dass das Spektrum der API-Funktionen, die Sie effektiv anwenden können, im Vergleich zu vorher wesentlich breiter geworden ist, auch wenn Sie nicht jedes Puzzle selbst mit Erfolg gelöst haben.

Damit komme ich zum letzten Puzzle.

Ich weiß, dass viele von Ihnen versucht haben, einige dieser Puzzle zu lösen – besonders die späteren – und dabei von der Frustration eingeholt wurden. Diese Frustration ist qualvoll, aber die erworbenen Kenntnisse und Fertigkeiten dürften die Mühe wert sein. Und doch ist dies nur ein Anfang. Täglich kommen neue API-Funktionen hinzu. Dass einige davon sich für mich als unlösbar herausstellen, steigert, wie ich hoffe, Ihr Vertrauen in Ihre eigene Fähigkeit, auch das kniffligste Puzzle anzugehen, sowie Ihre Bereitschaft, zu misstrauen und das, was andere (und sei es Microsoft) ihnen sagen, mit gebotener Skepsis zu betrachten.

Zum Schluss möchte ich Sie einladen, unsere Website unter **http://www.desaware.com/** aufzusuchen, wo Sie nach der Veröffentlichung entstandene Überarbeitungen und Korrekturen sowie möglicherweise ein neues Puzzle oder zwei finden (ein nicht in diesem Buch enthaltenes BeispielPuzzle ist sogar bereits vorhanden).

Äh, übrigens – falls Sie eine Lösung für dieses Puzzle finden, schreiben Sie mir bitte ein paar Zeilen.

Mit besten Grüßen

Daniel Appleman

Januar 1999

Teil III
Die Tutorien

Sie fragen sich vielleicht, warum dieser Teil des Buches nicht Teil I ist. Immerhin enthalten die Tutorien die Informationen, die Sie zum Lösen der Puzzle benötigen.

Naja, dies ist ein Puzzlebuch.

Ich bin sicher, dass viele Leser sich direkt auf die Puzzle stürzen und versuchen, dieselben ohne Hilfe zu lösen. Und wissen Sie was? Viele von Ihnen werden dabei erfolgreich sein.

Dennoch besteht die Chance, dass Sie bei einem oder mehreren Puzzlen nicht weiterkommen. Ich möchte Ihnen raten, in diesem Fall eine Pause einzulegen und es sich vor einem lauschigen Kaminfeuer bequem zu machen (oder auf einem Badetuch am Swimmingpool oder Strand, je nach Klima) und diese Tutorien zu lesen. Ich denke, selbst die Experten unter Ihnen werden die Tutorien zu schätzen wissen.

Und für die absoluten API-Anfänger: Herzlich Willkommen. Sie sollten diesen Teil des Buches lesen, bevor Sie sich auch nur das erste Puzzle in diesem Buch ansehen. Nach der Lektüre dieser Tutorien sollten Ihnen die Puzzle sehr viel weniger einschüchternd erscheinen als das auf den ersten Blick der Fall sein mag.

Tutorium 1

Auffinden von Funktionen

Wenn Sie die Komplexität der Win32-API im Hinblick auf den internationalen Handel mit dem Weltfinanzsystem vergleichen müssten, würden Sie mit Sicherheit zu dem Schluss kommen, dass das Weltfinanzsystem weitaus komplizierter ist.

Zumindest auf den ersten Blick.

Müssten Sie den Welthandel mit der Art und Weise vergleichen, mit der in Windows Funktionen verwaltet werden, würden Sie vielleicht eine gewisse Übereinstimmung feststellen.[1]

Denken Sie statt dessen darüber nach, wie der Güteraustausch zwischen zwei Ländern abgewickelt wird.

Ein US-Exporteur verfügt beispielsweise über einen großen Bestand an Jumbo-Jets, die er verkaufen möchte. Ein chinesischer Importeur hat dringenden Bedarf für diese Jets. Einige Faxe werden über den Pazifik gesendet, und das Geschäft ist abgeschlossen.

Im weiteren Verlauf des Tages verzeichnet ein US-Importeur einen hohen Bedarf an Schokoladeriegeln. Er kontaktiert seinen chinesischen Freund, um anzufragen, ob dieser ihm ein Kontingent an Schokoladenriegeln zur Verfügung stellen kann. Sein Freund teilt ihm mit, dass er heute keine Schokolade exportiert. Der US-Importeur verschickt sofort E-Mails an andere Exporteure in der ganzen Welt, um festzustellen, wer das Produkt zur Verfügung stellen kann. Wenn er keinen Exporteur findet, werden Tausende von potenziellen Käufern keine Schokolade erhalten, und der Importeur muss aus diesem Grund vielleicht sein Geschäft aufgeben.

Ein Handel kann nur dann stattfinden, wenn das Produkt, das ein Exporteur anbietet, exakt dem Produkt entspricht, das der Importeur benötigt.

Bei einem Zugriff auf die Funktionen einer DLL (Dynamic Link Library) wird genau der gleiche Prozess abgewickelt. Eine DLL kann interne Funktionen anderen DLLs außerhalb des Programms zur Verfügung stellen, indem die Funktionen exportiert werden. Dies geschieht durch Auflistung der Funktionen in einer internen Exportliste, die von Windows untersucht werden kann. Ein Visual Basic-Programm gibt an, dass eine Datei importiert werden soll, indem auf ebendiese in einer `Declare`-Anweisung oder durch eine Typenbibliothek verwiesen wird. Die `Declare`-Anweisung enthält zwei Informationen, mit denen die zu importierende Funktion spezifiziert wird: den Namen der DLL-Datei und den Namen der Funktion. Der Funktionsname wird im `Alias`-Feld der `Declare`-Anweisung angegeben. Ist kein `Alias`-

1. Nein, ich beziehe mich nicht auf die Tatsache, dass Microsoft aus beidem Profit zieht.

Feld vorhanden, setzt VB voraus, dass der in der Anweisung angegebene Funktionsname mit dem Funktionsnamen in der DLL übereinstimmt.

Beispiel:

```
Declare Sub myfunc Lib "mydll.dll" Alias "otherfunc" ()
```

Diese Anweisung gibt an, dass Sie die Funktion »otherfunc« aus der Datei **mydll.dll** importieren möchten und auf diese Funktion im VB-Programm mit dem Namen »myfunc« verweisen.

Die folgende Anweisung dagegen sagt etwas anderes aus:

```
Declare Sub myfunc Lib "mydll.dll" ()
```

Diese Anweisung gibt an, dass Sie die Funktion »myfunc« aus der Datei **mydll.dll** importieren möchten, und dass Sie auf diese Funktion im VB-Programm mit dem gleichen Namen verweisen.

Im ersten Fall muss durch die DLL **mydll.dll** die Funktion »otherfunc« exportiert werden, da ansonsten die Operation fehlschlägt – VB wird die Funktion nicht finden. Im zweiten Fall muss durch die DLL **mydll.dll** die Funktion »myfunc« exportiert werden.

Die »Declare«-Anweisung im Detail

Die Visual Basic-Dokumentation bietet für die `Declare`-Anweisung die folgende Syntax an:

```
[Public | Private] Declare Sub name Lib "libname" [Alias "aliasname"]
[([arglist])]

[Public | Private] Declare Function name Lib "libname" [Alias "aliasname"]
[([arglist])] [As type]
```

Lassen Sie uns eine genaue Untersuchung der einzelnen Elemente vornehmen – ▲Tabelle T1-1.

`[Public | Private]`	Die `Declare`-Anweisung kann ein vorangestelltes Schlüsselwort `Public` oder `Private` aufweisen. Die Standardeinstellung hierbei lautet `Public`. Bei der Verwendung von `Declare`-Anweisungen in einer Anwendung können zwei Ansätze voneinander unterschieden werden. Beim ersten Ansatz werden die `Declare`-Anweisungen als öffentlich deklariert und in einem Standardmodul platziert, das von allen weiteren Modulen in der Anwendung genutzt werden kann. Bei Verwendung des zweiten Ansatzes werden die Anweisungen als privat deklariert. Die für jedes Modul benötigten Deklarationen werden in das Modul eingefügt. Dieser Ansatz wird häufig bei größeren Anwendung eingesetzt. Die `Declare`-Anweisungen in Formular- und Klassenmodulen müssen als privat deklariert werden.
`Declare`	Das Schlüsselwort, mit dem angegeben wird, dass diese Anweisung zur Definition des Zugriffs auf eine API- oder DLL-Funktion verwendet wird.
`Sub | Function`	Praktisch jede Win32-API-Funktion gibt einen `Long`-Wert zurück und sollte daher als Funktion deklariert werden. Sie können die Anweisung als `Sub` deklarieren, wenn Sie nicht auf den Rückgabewert verweisen müssen. Wenn Sie die Anweisung als `Function` deklarieren, stellen Sie sicher, dass Sie für den Rückgabewert den richtigen Typ angeben. Der Standardrückgabewert (`Variant`) führt wahrscheinlich zu einem Fehler oder einer Speicherausnahme.
`name`	Der Name, durch den die Funktion in Visual Basic bezeichnet wird. Weitere Informationen erhalten Sie unter den Anmerkungen zum `Alias`-Ausdruck. Dieser Name muss den gleichen Namenskonventionen entsprechen, die für die Visual Basic-Funktionen gelten. Beispiel: Der Name darf im gleichen Modul nicht bereits verwendet worden sein (unabhängig von Groß- oder Kleinschreibung).
`Lib`	Der Begriff, der diesem Schlüsselwort folgt, ist die Bibliothek, also der Name der DLL, die die Funktion enthält.
`"libname"`	Der in Klammern angegebene Name der DLL. Wenn Sie eine der drei Haupt-DLLs angeben (Kernel32, User32 oder GDI32), muss keine Erweiterung angegeben werden. Andernfalls muss unbedingt die Erweiterung angegeben werden (in den meisten Fällen **.dll**). Sie können auch einen vollständigen Pfadnamen angeben, aber in den meisten Fällen ist hier nur der Name der DLL enthalten, damit die Anwendung die DLL auch unabhängig vom Installationsort auffinden kann. Windows sucht unter Verwendung der Standardsuchregeln des verwendeten Betriebssystems nach der DLL. Die DLL befindet sich üblicherweise entweder im Systemverzeichnis (typischerweise **\windows\system** für Windows 95/98 und **\Winnt\system32 für NT**) oder in dem Verzeichnis, in dem sich auch die Anwendung befindet. Freigegebene DLLs sollten im Systemverzeichnis gespeichert werden.

Tabelle T1-1 Elemente der `Declare`-Anweisung

`Alias`	Fehlt dieser Begriff, setzt Visual Basic voraus, dass es sich bei dem unter »name« angegebene Ausdruck gleichzeitig um den Namen der Funktion in der DLL handelt. Beachten Sie hierbei, dass der Funktionsname unter Berücksichtigung der Groß- und Kleinschreibung angegeben werden muss (die Groß- oder Kleinschreibung muss exakt mit der exportierten Funktion übereinstimmen).
`"aliasname"`	Falls angegeben, handelt es sich um den Namen der exportierten Funktion in der DLL. Bei diesem Begriff muss die Groß- und Kleinschreibung berücksichtigt werden.
`[([arglist])]`	Die Funktionsparameter. Diese werden in den folgenden Tutorien noch ausgiebig erläutert.
`[As Type]`	Bei einer Funktion sollte stets ein Datentyp für den Rückgabewert angegeben werden, üblicherweise `As Long`. Einer der häufigsten Fehler im Umgang mit der `Declare`-Anweisung besteht darin, dass kein Rückgabetyp angegeben wird. Sie können den Funktionstyp auch angeben, indem Sie ein Typenzeichen oder den Namen der Funktion hinzufügen (beispielsweise % für `Integer` oder & für `Long`).

Tabelle T1-1 Elemente der `Declare`-Anweisung

Auffinden exportierter Funktionen

Das Windows-Betriebssystem umfasst eine Vielzahl von DLLs, und herauszufinden, welche DLL eine bestimmte API-Funktion enthält, kann eine ziemliche Herausforderung darstellen. Glücklicherweise gibt es verschiedene Tools und Methoden, die Sie bei dieser Aufgabe unterstützen.

Die weitaus einfachste Methode zum Auffinden einer DLL-Funktion ist die, eine richtige Deklaration als Referenz heranzuziehen. Ihnen stehen zwei primäre Deklarationsquellen zur Verfügung – die Datei **win32api.txt**, die zum Lieferumfang von Visual Basic gehört, und die Datei **api32.txt**, die auf der Begleit-CD-ROM zu diesem Buch enthalten ist. Ich muss jedoch erneut betonen, dass nur richtige Deklarationen als Referenz herangezogen werden können, da beide Dateien Fehler enthalten können. Mit der Datei **api32.txt** werden viele Fehler behoben, die in der Datei **win32api.txt** vorhanden sind (die Datei **api32.txt** basiert auf der Datei **win32api.txt**). Als dieses Buch in Druck ging, waren keine Fehler in der Datei **api32.txt** bekannt, ich möchte jedoch nicht beschwören, dass die Datei perfekt ist.

Die grundsätzliche Beschränkung dieser beiden Dateien zeigt sich darin, dass Sie nicht für jede API-Funktion Deklarationen enthalten. Dies ist auch gar nicht möglich, da Microsoft täglich neue Funktionen und DLLs hinzufügt.

Auf diese Weise wird Windows ständig vergrößert. Microsoft fügt in den seltensten Fällen vorhandenen DLLs neue Funktionen hinzu. Statt dessen werden neue DLLs erstellt, mit denen neue Funktionen implementiert werden. Dies führt uns

zum nächsten Ansatz beim Ermitteln der DLL mit einer bestimmten Funktion – dem gesunden Menschenverstand.

In der nachstehenden Tabelle T1-2 werden einige der häufig verwendeten DLLs mit ihrer Funktionalität aufgeführt. Wenn Sie wissen, welche Aufgabe durch eine Funktion ausgeführt wird, können Sie häufig die DLL erraten, in der diese Funktion enthalten ist. Wenn Sie beispielsweise über eine Funktion verfügen, mit der eine grafische Operation für ein Fenster ausgeführt wird, ist die Wahrscheinlichkeit sehr hoch, dass sich diese Funktion in der DLL **gdi32.dll** befindet, der Bibliothek für die grafische Geräteschnittstelle.

kernel32.dll	Diese DLL enthält Funktionen, mit denen der Kernel für das Betriebssystem implementiert wird. In dieser DLL befinden sich Funktionen für Prozessverwaltung, Systemfunktionalität, Speicherverwaltung und die meisten Eingabe-/Ausgabefunktionen für Geräte. Sie brauchen die Erweiterung **.dll** nicht anzugeben, wenn Sie in einer Visual Basic-Deklaration auf diese DLL verweisen.
user32.dll	Diese DLL enthält Funktionen, mit denen die fensterbasierte Benutzeroberfläche implementiert wird. In dieser DLL sind Funktionen zum Erstellen und Verwalten von Fenstern, Steuerelementen und Dialogfeldern enthalten. Sie brauchen die Erweiterung **.dll** nicht anzugeben, wenn Sie in einer Visual Basic-Deklaration auf diese DLL verweisen.
gdi32.dll	Diese DLL enthält Funktionen, mit denen die grafische Ausgabe sowohl an Fenster als auch an Geräte für die Druckausgabe implementiert wird. Hierzu zählt auch die Textausgabe. Sie brauchen die Erweiterung **.dll** nicht anzugeben, wenn Sie in einer Visual Basic-Deklaration auf diese DLL verweisen.
mpr.dll	Diese DLL enthält geräteunabhängige Netzwerkfunktionen, von denen die meisten mit dem Präfix `WNET` beginnen.
version.dll	Diese DLL enthält Funktionen, die zusammen mit Versionsressourcen arbeiten und die in diesen Ressourcen enthaltenen Informationen erhalten. Versionsressourcen können Ihnen bei der Ermittlung der aktuellsten Version einer ausführbaren Datei oder DLL helfen.
winmm.dll	Diese DLL enthält eine Vielzahl von Funktionen, die mit den von Windows unterstützten Multimediafunktionen in Zusammenhang stehen. Hierzu zählen beispielsweise Audio-, Video- und Joystickgeräte.
winspool.drv	Diese DLL enthält Funktionen zur Druckverwaltung und zur Verwaltung des Spoolers für den Systemdruck.
lz32.dll	Diese DLL enthält Funktionen zur Dekomprimierung von Dateien, die mit Hilfe des Windows-Dienstprogramms **compress.exe** komprimiert wurden.
shell32.dll	Diese DLL enthält Funktionen, die in Zusammenhang mit den Benutzerschnittstellenfunktionen hoher Ebene stehen, die von Windows zur Verfügung gestellt werden. Es handelt sich beispielsweise um Funktionen, mit denen Drag&Drop-Operationen für Dateien oder die Verwaltung von Verknüpfungen zwischen Dateien, Icons und Dokumenten gehandhabt werden.

Tabelle T1-2 Häufig verwendete DLLs

advapi32.dll	Diese DLL enthält Funktionen, mit denen eine Reihe von weiterführenden Aufgaben ausgeführt wird. Hier finden sich beispielsweise Funktionen für die Registrierungsverwaltung sowie Prozess- und Threadverwaltungsfunktionen. Darüber hinaus sind auch Funktionen für NT-Dienst und die NT-Sicherheit darin enthalten.
comdlg32.dll	Mit den Funktionen in dieser DLL werden die Standarddialogfelder von Windows implementiert, die in den meisten Windows-Anwendungen verwendet werden. Hierzu gehören beispielsweise die Dialogfelder zum Öffnen und Speichern von Dateien sowie das Dialogfeld für die Druckerauswahl.
Comctl32.dll	Mit dieser DLL werden verschiedene Steuerelemente implementiert, die als die Windows-Standardsteuerelemente bezeichnet werden. Hierzu zählen z. B. die Steuerelemente für die Symbolleiste und ein `ImageList`-Steuerelement.
Oleaut32.dll	Diese DLL enthält Funktionen, mit denen verschiedene Funktionen für die OLE-Automatisierung implementiert werden. Hierzu gehören SAFEARRAY, BSTR und OLE `Variant`-Funktionen.
Ole32.dll	Diese DLL enthält Funktionen, mit denen die Grundfunktionen der OLE-Technologie (auch als ActiveX bekannt) implementiert werden.
wsock32.dll	Mit dieser DLL wird die Winsock-API implementiert. Diese Funktionen werden für die Netzwerkkommunikation benötigt, üblicherweise im Rahmen des Internets oder bei Verwendung von Intranets.

Tabelle T1-2 Häufig verwendete DLLs

Tabelle T1-2 ist in gewisser Weise irreführend. Dies werden Sie feststellen, wenn Sie eine Zählung der DLLs (Dateien mit der Erweiterung **.dll**) im Systemverzeichnis Ihres Computers durchführen. Bei NT-Systemen müssen Sie sich das Verzeichnis **system32** ansehen. Sie werden wahrscheinlich mehr als 50 solcher Dateien finden. Viele dieser DLLs enthalten jedoch unwichtige Funktionen.

Fragen Sie die DLL

Ein anderer Ansatz zum Auffinden der richtigen DLL für eine Funktion stellt die Suche innerhalb der DLL selbst dar. Das Microsoft-Programm **DumpBin.exe** (im Lieferumfang von Visual Studio enthalten) sowie die Funktion `Schnellansicht` des Windows Explorers können dazu verwendet werden, eine Liste der Funktionen abzurufen, die von einer DLL exportiert werden können. Das Programm DumpInfo (auf der Begleit-CD-ROM) kann ebenfalls dazu verwendet werden, eine Liste der von einer DLL exportierten Funktionen abzurufen, und verfügt außerdem über die Fähigkeit, sämtliche DLLs in einem Verzeichnis nach einer speziellen Funktion zu durchsuchen. Dieses Programm, das den vollständigen Quellcode enthält, verwendet außerdem eine Reihe von erweiterten API-Programmierungsmethoden und wird in Tutorium 9, »In einer DLL-Datei«, ausführlich erläutert.

Was ist mit den Typenbibliotheken?

Tatsächlich ist eine weitere Methode vorhanden, mit der Sie auf die API-Funktionen von Visual Basic-Programmen zugreifen können – die Verwendung von Typenbibliotheken. Typenbibliotheken werden mit Hilfe der Sprache IDL (Interface Definition Language) geschrieben und unter Verwendung des Tools **midl.exe** kompiliert, das im Lieferumfang von Visual Basic enthalten ist. Typenbibliotheken weisen im Hinblick auf die `Declare`-Anweisung die folgenden Vorteile auf:

- Sie bieten eine etwas bessere Leistung.
- API-Aufrufe, die mit Hilfe von Typenbibliotheken durchgeführt werden, belegen weniger Speicher in der ausführbaren VB-Datei.

Sie können die API-Deklarationen mit einer Hilfedatei verknüpfen.

Typenbibliotheken weisen folgende Nachteile auf:

Sie müssen die IDL-Datei nach jeder Änderung erneut kompilieren.

Nach der Erstellung einer Funktionsdefinition in der Typenbibliothek sollten Sie diese nicht ändern, da Sie ansonsten Anwendungen beschädigen könnten, die ebenfalls diese Typenbibliothek verwenden.

Sie müssen IDL lernen.

In meinem Buch »Dan Appleman's Visual Basic Programmer's Guide to the Win32 API« ist ein Kapitel zur Erstellung von Typenbibliotheken enthalten.[2]

In diesem Buch werden ausschließlich `Declare`-Anweisungen zum Aufruf von API-Funktionen verwendet. Warum? Da das Lösen der Puzzle ein Herumexperimentieren mit den Deklarationen erfordert und es sehr viel einfacher ist, Experimente mit einer `Declare`-Anweisung durchzuführen als dabei eine Typenbibliothek zu verwenden. Meine persönliche Erfahrung hat gezeigt, dass es einfacher ist, Anwendungen mit Hilfe der `Declare`-Anweisung zu entwickeln und zu testen. Nach dem Entwickeln einer Anwendung wechsle ich nur selten zu einer Typenbibliothek.

2. Das Kapitel zu den Typenbibliotheken ist nur in der Onlineversion des Buches enthalten, die sich auf der CD-ROM zu dem Buch befindet – in der gedruckten Version erscheint dieses Kapitel nicht. Das Buch enthält außerdem eine API-Typenbibliothek, die sämtliche der im Buch beschriebenen Funktionen umfasst, einschließlich des Quellcodes. Die Deklarationen sind außerdem mit der CD-ROM-Edition des Buches verknüpft, wodurch Ihnen das Auffinden von Hilfeinformationen zu einem beliebigen Funktionsaufruf erheblich erleichtert wird.

Tutorium 2

Der Arbeitsspeicher – da, wo alles beginnt

Eine Warnung an Fortgeschrittene:

Dieses Tutorium deckt einige grundlegende Computerkonzepte ab. Daher kann es jetzt sehr langweilig werden. Dennoch schließe ich dieses Tutorium ein, weil ich eine erstaunliche Anzahl Programmierer kennengelernt habe, bei denen es am Verständnis dieser Konzepte hapert. Dies liegt jedoch nicht unbedingt an den Programmierern – Visual Basic macht es dem Benutzer sehr leicht, die Grundlagen zu überspringen, daher gibt es viele VB-Programmierer, die das, was jeder Informatikstudent im ersten Semester lernt, nie gelernt haben. Obwohl in diesem Buch viele weiterführende Themen behandelt werden, liegt es in meiner Verantwortung, dass Ihnen alle wichtigen Informationen zur Verfügung stehen. Wenn Sie die Konzepte bereits kennen, prima – aber vielleicht möchten Sie dennoch einen Blick darauf werfen. Man kann nie wissen, welche Überraschungen ich vielleicht noch aus dem Hut zaubere ...

Visual Basic. Eine Sprache, die die Windows-Programmierung revolutioniert hat. Eine Sprache, die so einfach ist, dass anscheinend jeder ohne viel Übung ein tolles VB-Programm schreiben kann. Es gibt Bücher, mit denen Sie Visual Basic angeblich in nur 21 Tagen erlernen können! Eine Sprache, bei der Sie nicht wissen müssen, was sich im Hintergrund abspielt, ist eine prima Sache, oder?

Angenommen, Sie möchten in einem Programm die Zahl 5 speichern. Wo liegt der Unterschied zwischen den folgenden Variablen?

- `Dim X As Integer`
- `Dim L As Long`
- `Dim F As Single`
- `Dim D As Double`
- `Dim C As Currency`
- `Dim B As Byte`
- `Dim V As Variant`
- `Dim S As String`
- `X = 5`
- `L = 5`
- `F = 5`
- `D = 5`
- `C = 5`

- `B = 5`
- `V = 5`
- `S = 5`

Der Unterschied besteht darin, dass Visual Basic diese Zahl jedes Mal anders speichert. Die Wahrheit ist, dass Sie sich als Visual Basic-Programmierer vielleicht gar nicht darum kümmern, welchen Variablentyp Sie verwenden. Einige der so genannten »Visual Basic-Experten« ermutigen tatsächlich andere dazu, stets Variablen vom Typ `Variant` zu verwenden, denn dann müssten sie sich keine Sorgen über den verwendeten Variablentyp machen!

Ich persönlich halte diesen Ansatz schlicht für dumm. Er ist ineffizient, auf lange Sicht schlechter zu unterstützen und erhöht das Risiko von Bugs. In vielen Fällen funktioniert es jedoch.

Wie dem auch sei, wenn Sie von Visual Basic aus API- oder DLL-Funktionen aufrufen möchten, führt das Ignorieren des verwendeten Variablentyps und deren Funktionsweise letztendlich dazu, dass Sie lediglich die Deklarationen anderer Programmierer blind kopieren und hoffen, dass diese funktionieren bzw. dass die Funktionen für Ihre Anwendung geeignet sind.

Wir werden die verschiedenen Variablentypen zu einem späteren Zeitpunkt näher beleuchten. Zunächst wenden wir uns dem wahren Zweck von Variablen zu.

Das Innenleben der Variablen

Die Aufgabe einer Variablen ist es, Daten zu speichern. Daten befinden sich im Speicher.

Aber worum handelt es sich eigentlich genau bei einem Speicher?

Wenn Sie zu Ihrem Computerhändler gehen, handelt es sich um eine kleine, niedliche PC-Karte mit einem winzigen Chip, die unglaubliche 90 DM kostet, aber Sie sind dankbar (weil Sie wissen, dass diese Karte letzte Woche noch das Doppelte gekostet hat) und gleichzeitig frustriert (weil Sie wissen, dass die gleiche Karte nächste Woche nur die Hälfte kosten wird).

Sie wissen, dass Ihr System über 16 MB oder 32 MB oder 64 MB Speicher oder so verfügt, aber Sie wissen vielleicht nicht genau, was MB eigentlich bedeutet (es bedeutet Megabyte – eine Million Byte Daten, nur, dass es gar nicht eine Million, sondern 1 048 576 Byte sind – aber ist ja auch egal, wenn Sie nicht wissen, was ein Byte überhaupt bedeutet).

Also vergessen Sie das alles. Die Speichergröße Ihres Systems wirkt sich auf dessen Leistung aus, hat jedoch keinerlei Bedeutung, wenn Sie ein VB-Programmierer sind.

Was Sie wissen müssen, ist die Bedeutung einer grundlegenden Einheit – dem Bit.

Bits

Das Bit bildet die Grundlage für sämtliches Computerwissen. Dieses Bit kann zwei Werte annehmen, 0 oder 1.

Mal angenommen, Ihnen stünde für die Datenspeicherung ein Bit zur Verfügung. Könnten Sie die Zahl fünf mit Hilfe eines einzigen Bits speichern? Nein – das Bit kann ja nur über einen von zwei Werten verfügen. Was ist jedoch, wenn Sie ein Bit dazu verwenden möchten, eine Aussage als wahr oder falsch zu bewerten? Kein Problem – wahr und falsch stellen zwei Werte dar. Sie können ein Bit auch dazu verwenden, etwas als »Ein« oder »Aus« zu kennzeichnen. In einem Bitmap mit den Farben schwarz und weiß (monochrom) könnte beispielsweise jedes Bit einen Pixel als schwarz oder weiß kennzeichnen.

Bits können sicherlich sehr nützlich sein, dies jedoch erst, wenn Sie in Gruppen angeordnet werden.

Nibbles

Sie werden das Wort »Nibble« immer nur dann hören, wenn jemand die Bedeutung von binären Zahlen erklärt. Was ist eine binäre Zahl? Eine binäre Zahl setzt sich aus Bits zusammen, die für sich genommen immer nur einen von zwei Werten (binär) annehmen können. Wir verwenden den Begriff »Nibble«, um die Funktionsweise von binären Zahlen zu erläutern, da diese Form der Erklärung weniger Zeit beansprucht als die Erläuterung mit Hilfe von Bytes. Warum dies so ist, werden Sie gleich sehen.

Sie wissen, dass in einem Bit nur einer von zwei Werten gespeichert werden kann. Was können Sie mit zwei Bits machen?

Zwei Bits können die folgenden möglichen Werte annehmen:

Zweites Bit	Erstes Bit
0	0
0	1
1	0
1	1

Es mag Ihnen merkwürdig erscheinen, warum das zweite Bit an die erste Stelle gerückt wurde, aber auch dies hat einen Grund. Belassen wir es im Moment dabei, dass wir bei dieser Erläuterung davon ausgehen, das erste Bit als das Bit zu sehen, das auf der rechten Seite zu sehen ist.

Logischerweise können mit zwei Bits vier Werte dargestellt werden. Sie könnten zwei Bits beispielsweise dazu verwenden, von 0 bis 3 oder von 1 bis 4 zu zählen. Sie könnten mit diesen Bits beispielsweise vier verschiedene Farben darstellen:

Zweites Bit	Erstes Bit	
0	0	Rot
0	1	Grün
1	0	Blau
1	1	Weiß

Wie Sie sehen können, ist es dem Computer völlig egal, wozu Sie diese Bits verwenden. Der Computer versteht die Bedeutung der Bits nicht. Darin besteht die Aufgabe eines Programmierers – den Bits eine Bedeutung zu geben. In den meisten Fällen wird Ihnen diese Aufgabe von den Visual Basic-Variablen abgenommen. Bei den API- und DLL-Aufrufen ist dies meist nicht der Fall. Dies ist einer der Gründe dafür, warum Sie verstehen sollten, was sich hinter den Kulissen abspielt.

Angenommen, Sie verfügen über vier Bits. Vier Bits bilden ein Nibble. Sagen wir, dass jedes dieser Bits über eine Bezeichnung verfügt, mit der die Position des Bits im Nibble angegeben wird, von Bit#0 bis Bit#3. Die möglichen Werte lauten folgendermaßen:

Bit #3	Bit #2	Bit #1	Bit #0	Wert
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8

Bit #3	Bit #2	Bit #1	Bit #0	Wert
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	0	14
1	1	1	1	15

Jetzt kommen wir zum Wesentlichen. Mit einem Nibble können Sie bis zu 16 verschiedene Werte darstellen, in diesem Fall von 0 bis 15. Sind weitere Werte möglich? Was geschieht, wenn wir beispielsweise sagen, dass durch Bit #3 gekennzeichnet werden kann, ob es sich um eine negative Zahl handelt?

Bit #3	Bit #2	Bit #1	Bit #0	Wert
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1

Mit einem Nibble können bis zu 16 verschiedene Werte dargestellt werden, die Bedeutung dieser Werte richtet sich jedoch danach, wie diese Daten von der Computersprache gespeichert werden. Wenn das höchste Bit (das Bit mit der höchsten Positionsnummer) dazu verwendet wird, negative Zahlen zu kennzeichnen, wird die Variable als Vorzeichenvariable (signed variable) bezeichnet.[1] Andernfalls spricht man von einer Variablen ohne Vorzeichen (unsigned variable). Visual Basic-Programmierer machen sich im Allgemeinen keine Gedanken über diese Zusammenhänge, da es sich bei den meisten Variablentypen um Vorzeichenvariablen handelt. API- und DLL-Funktionen können jedoch Variablen ohne Vorzeichen verwenden.

Angenommen, eine DLL-Funktion verfügte über einen Parameter, der ein Nibble ohne Vorzeichen verwendet, und Sie möchten an diesen Parameter die Zahl 14 übergeben. Wenn Sie eine Nibble-Vorzeichenvariable verwenden, können Sie den Wert –2 übergeben. Sehen Sie sich die Tabelle an –14 ohne Vorzeichen entspricht der –2 mit Vorzeichen!

Selbstverständlich verwenden weder DLLs noch die API Nibbles. Wir ziehen Sie hier nur als Beispiel heran, damit wir uns nicht mit größeren Tabellen befassen müssen. Aber dies hat einen sehr wichtigen Grund. Bevor wir uns mit größeren Bitzahlen befassen, sollten wir uns darauf einigen, dass diese in 4er-Gruppen angeordnet werden. Wenn wir mit 16 Bits arbeiten möchten, verwenden wir statt der Darstellung 0000000000000000 für 16 Bits vom Wert 0 die Darstellung 0000, wobei jede Null ein Nibble repräsentiert. Wenn wir die binäre Zahl 0011 0011 0011 0011 darstellen möchten, verwenden wir 3333, da jede Gruppe 0011 ohne Vorzeichenverwendung dem Wert 3 entspricht.

Aber was geschieht mit der binären 16-Bit-Zahl 1010 1011 1100 1101? Der erste Nibble ist 10, der zweite 11, der dritte 12, der vierte 13. Wir könnten 10111212 schreiben, aber das wäre sehr verwirrend – es ist sehr viel einfacher, wenn ein Nibble nur aus einer Ziffer besteht. Diese Methode funktioniert von 0 bis 9 einwandfrei, aber bei größeren Zahlen müssen Sie statt dessen Buchstaben verwenden:

1. Es ist leicht verständlich, warum die Wert 0 bis 7 den acht positiven Zahlen zugeordnet werden, aber warum zählen die negativen Zahlen von –8 bis –1? Dies ist kein Zufall, aber eine Erklärung kann im Rahmen dieses Buches leider nicht erfolgen. Das hier verwendete Format wird als Zweierkomplement bezeichnet. In diesem Format können Sie eine Zahl durch Komplementierung (Invertieren der einzelnen Bitwerte) und anschließendes Addieren einer Eins in eine negative Zahl umwandeln. Auf diese Weise wird verhindert, dass Sie einen positiven und einen negativen Nullwert erhalten, und es wird sichergestellt, dass arithmetische Operationen sowohl mit negativen als auch mit positiven Zahlen richtig durchgeführt werden können.

Bit #3	Bit #2	Bit #1	Bit #0	Hexadezimal
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	A
1	0	1	1	B
1	1	0	0	C
1	1	0	1	D
1	1	1	0	E
1	1	1	1	F

Jetzt wird die binäre 16-Bit-Zahl 1010 1011 1100 1101 zu ABCD. Wenn Sie mit einer 32-Bit-Zahl arbeiten, ist es sehr viel einfacher, diese als 049AB159 zu notieren als durch 00000100101010111000101011001 darzustellen. Tatsächlich ist dies derart vorteilhaft, dass kein Programmierer mit gesundem Verstand noch das binäre Zahlensystem verwendet. Wir verwenden Nibbles, bei denen jede Ziffer vier Bits darstellt. Aber wir nennen dieses System natürlich nicht so. Da jede Ziffer vier Bits umfasst und damit 16 mögliche Werte darstellen kann (0-F), bezeichnen wir dieses Format als hexadezimal (zur Basis 16). In Visual Basic wird das Präfix &H zur Kennzeichnung einer hexadezimalen Zahl verwendet; wir notieren demnach hex AB15 als &HAB15. Ein C-Programmierer verwendet zur Kennzeichnung einer Hexadezimalzahl das Präfix 0x. Sie würde dann als 0xAB15 notiert.

Bytes

Wenn Sie bisher folgen konnten, entspannen Sie sich – das Schlimmste ist überstanden. Sie haben die Bedeutung der Bits kennengelernt. Sie wissen, dass die Zahl der darstellbaren Werte mit der Zahl der verwendeten Bits steigt. Ein einzelnes Bit kann nur einen von zwei Werten annehmen. Mit vier Bits können insge-

samt 16 Werte dargestellt werden. Mit acht Bits können insgesamt 256 Werte dargestellt werden. Die Zahl der möglichen Werte ist tatsächlich 2^n, wobei **n** die Anzahl der Bits darstellt (d.h., jedes Mal, wenn Sie ein Bit addieren, verdoppelt sich die Anzahl der möglichen Werte).

Acht Bits werden als Byte bezeichnet. Visual Basic verfügt über einen Datentyp mit Namen `Byte`, bei dem es sich um den einzigen Datentyp ohne Vorzeichen handelt, der von Visual Basic grundsätzlich unterstützt wird. Das bedeutet, dass ein Bytewert von 0 bis 255 reichen kann.

Bytes sind im Hinblick auf den Computerspeicher die zugrundliegende Maßeinheit. Was bedeutet das?

Abbildung T2-1 zeigt, wie im ersten 8-Bit-Mikroprozessor der Speicher aufgebaut war. Der Speicher wurde in 65536 Byte (64K) Speicher unterteilt. Jedes Byte konnte durch den Mikroprozessor individuell angesteuert werden. Wenn der Wert eines bestimmten Bits in einem Byte überprüft werden sollte, musste das gesamte Byte geladen und der Bytewert untersucht werden, um zu überprüfen, ob das betreffende Bit gesetzt war. Der Vorgang des Prüfens und Setzens einzelner Bits wird in Tutorium 3 besprochen, »Ein Boolescher Wert und seine Bitfelder werden bald auseinandergehen«.

Abbildung T2-1 64K Speicher in einem 8-Bit-System

In einem 8-Bit-System wird über Bytes auf den Speicher zugegriffen. Jedes Byte im Speicher verfügt über eine eindeutige Adresse, mit der der jeweilige Standort angegeben wird. Daher lautet die Speicheradresse für das erste Byte 0, für das zweite Byte 1 usw., bis zum Ende des Speichers.

Abbildung T2-2 zeigt ein 32-Bit-System mit 4,2 GB Speicher. Warum 4,2 GB? Weil Sie mit 32 Bits 4,2 Billionen Werte darstellen können. Handelt es sich bei diesen Werten um Speicheradressen, können Sie auf bis zu 4,2 Billionen Byte zugreifen. Diese Art Speicher wird auch als 32-Bit-Adressraum bezeichnet. Beachten Sie, dass der Speicher etwas anders aufgebaut ist. Jede Speicherstelle umfasst 32 Bits, die Adresse erhöht sich jedoch pro Speicherstelle um vier. Was bedeutet das? Es bedeutet, dass der Prozessor Lese- und Schreibvorgänge in Blöcken von jeweils 32 Bits durchführt, je vier Byte gleichzeitig. Dennoch kann weiterhin auch auf einzelne Bytes zugegriffen werden. Warum ist das wichtig?

Wenn ein Prozessor vier Byte Daten gleichzeitig verschieben kann, können die Daten im Gegensatz zu einem Prozessor, der die Daten nur Byte pro Byte verschieben kann, mindestens viermal so schnell verschoben werden. Tatsächlich ist die Geschwindigkeit sogar noch höher, da nicht nur größere Datenblöcke verschoben werden, sondern gleichzeitig sämtliche Zusatzanweisungen gespeichert werden, die für den Zugriff auf die einzelnen Bytes nötig sind. Dies ist einer der Hauptgründe dafür, dass ein 32-Bit-Prozessor schneller ist als ein 8-Bit-Prozessor. Was geschieht, wenn Sie auf eine 32-Bit-Variable vom Typ `Long` zugreifen möchten, deren Adresse &H100 lautet? Kein Problem – der Prozessor geht direkt an diese Adresse und liest die Daten in einem Vorgang. Was passiert jedoch, wenn sich die `Long`-Variable an der Adresse &H101 befindet? Wenn Sie eine Intel-CPU besitzen, muss der Prozessor tatsächlich zwei separate Leseoperationen durchführen. Drei Byte der Variablen befinden sich unter &H100, ein Byte muss von der Speicherstelle &H104 abgerufen werden. Die CPU muss anschließend die Daten innerhalb der CPU in einer einzigen `Long`-Variablen anordnen, damit mit diesen Daten Operationen ausgeführt werden können. Dies ist, gelinde gesagt, zeitaufwendig. Bei einigen Prozessoren stellt sich diese Situation sogar noch schlimmer dar. Der Prozessor gibt statt dessen einfach einen Fehler aus, der besagt, dass eine ungültige Operation vorgenommen wurde. Dies führt zu sogenannten »Anordnungsfragen«, die in Tutorium 7, »Klassen, Strukturen und benutzerdefinierte Typen« näher erläutert werden.

Als Visual Basic-Programmierer brauchen Sie sich über diese Dinge keine Gedanken zu machen, da all dies gut vor Ihnen verborgen wird. Ein Verständnis dieser Sachverhalte ist jedoch bei der Verwendung von erweiterten API- und DLL-Aufrufen absolut unerlässlich.

Größe des Arbeitsspeichers

Was? Sie haben keine 4,2 GB Arbeitsspeicher auf Ihrem System? Ich auch nicht (obwohl ich schon soviel Festplattenspeicher besitze).

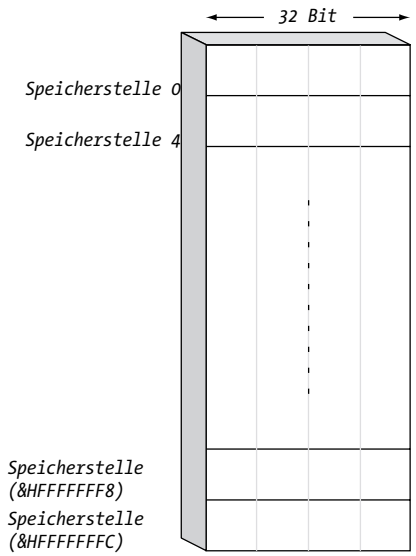


Abbildung T2-2 Speicheraufbau in einem 32-Bit-System mit 4,2 GB Arbeitsspeicher

Sie wissen vielleicht, dass Ihr System nur über 16 MB oder 64 MB Arbeitsspeicher verfügt. Und Sie wissen vielleicht auch, dass dieser Arbeitsspeicher gemeinsam von allen Anwendungen genutzt wird, die auf Ihrem System ausgeführt werden, einschließlich des Betriebssystems.

Ich möchte, dass Sie das vergessen.

Alles, was Sie von nun an wissen müssen, ist dass jede Ihrer Anwendungen über einen 32-Bit-Adressraum (4,2 GB) für sich ganz allein verfügt. Sicher, gewisse Teile dieses Bereiches sind für das Betriebssystem reserviert, und Sie werden sich aus diesem Adressraum kaum selbst Speicher zuweisen. Darüber hinaus ist mit dem größten Teil dieses Speicherbereiches kein realer Speicher verknüpft – wenn Sie versuchten, auf diese Adressen zuzugreifen, würden Sie einen Speicherausnahmefehler verursachen. Das Wichtige hierbei ist, zumindest im Hinblick auf eine Anwendung, dass dieser Anwendung der gesamte Systemspeicher zur Verfügung steht. Dies führt zu einigen interessanten Situationen, besonders dann, wenn die Programmierung auf einem 16-Bit-System vorgenommen wird. Es bedeutet, dass es sehr schwierig ist, den Speicher auf einem Win32-System gemeinsam zu nutzen. (Sie finden weitere Informationen zu diesem Thema in Kapitel 14 meines Buches »Dan Appleman's Visual Basic Programmer's Guide to the Win32 API«.)

Zurück zu den Datentypen

Zu Beginn dieses Tutoriums nahmen wir einen Code, in dem die Zahl 5 unter Verwendung verschiedener Variablentypen gespeichert wurde. Wie werden diese Werte also tatsächlich im Arbeitsspeicher gespeichert? Das Projekt **Memory.vbp** (auf der Begleit-CD-ROM zu diesem Buch) gibt Ihnen einen Einblick in die einzelnen Bytes einer Variablen. Die Variablen werden auf Formularebene definiert und im Formularereignis `Load` auf 5 gesetzt. Außerdem wird die API-Funktion `RtlMoveMemory` deklariert. Diese Funktion erhält zwei Speicheradressen und kopiert lediglich die Anzahl der Bytes, die in der Quelladresse angegeben sind, in die Zieladresse.

```
' MemoryTutorial - A Quick Memory Demo Program
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten

Option Explicit
Private Declare Sub RtlMoveMemory Lib "kernel32" (ByVal Dest As _
Long, ByVal Source As Long, ByVal Count As Long)
Dim I As Integer
Dim L As Long
Dim F As Single
Dim D As Double
Dim C As Currency
Dim b As Byte
Dim V As Variant
Dim S As String

Private Sub Form_Load()
   I = 5
   L = 5
   F = 5
   D = 5
   C = 5
   S = 5
   b = 5
   V = 5
End Sub
```

Die `GetDataString`-Funktion verwenden wir dazu, den Inhalt im Arbeitsspeicher zu untersuchen. Sie übergeben an diese Funktion eine Speicheradresse und eine Bytezahl. Begonnen wird an der Adresse, die durch den `Adress`-Parameter angegeben wird. Anschließend wird jedes Byte kopiert, um diese jeweils in eine `Byte`-

Variable mit Namen `thisbyte` umzuwandeln.[2] Die `thisbyte`-Variable wird unter Verwendung der `Hex$`-Funktion in eine hexadezimale Zeichenfolge konvertiert. Angenommen `thisbyte` enthält `&H8F`. In diesem Fall würde von `Hex$(thisbyte)` 8F zurückgegeben. Enthält `thisbyte` die Zahl 5, wird von `Hex$(thisbyte)` 5 zurückgegeben. Zum Ausgleich fügen wir eine zusätzliche Null hinzu, wenn der erste Nibble von `thisbyte` Null lautet (daher erscheinen `thisbyte` Werte von 0-F tatsächlich als 00-0F). In Tutorium 3 wird demonstriert, wie die `AND`-Operation eingesetzt werden kann, um festzustellen, ob das hohe Nibble Null lautet.

```
Private Function GetDataString(Address As Long, Bytes As Long) _
As String
   Dim thisbyte As Byte
   Dim offset As Integer
   Dim result$

   For offset = 0 To Bytes - 1
      ' Wir erhalten den Inhalt des Byte an Adresse+Offset
      Call RtlMoveMemory(VarPtr(thisbyte), Address + offset, 1)

      ' Falls zwischen 0-15, Hinzufügen einer Null.
      If (thisbyte And &HF0) = 0 Then result$ = result$ & "0"
      result$ = result$ & Hex$(thisbyte) & " "
   Next offset

   GetDataString = result$
End Function
```

Die Routine `Command1_Click` verwendet die `GetDataString`-Funktion zum Abrufen der tatsächlichen Speicherinhalte für jede Variable. Die `VarPtr`-Funktion ist eine wenig dokumentierte Visual Basic-Funktion, mit der die Adresse einer Variable im Speicher abgerufen wird. Sie verwenden diese Funktion häufig, wenn Sie mit API- und DLL-Aufrufen arbeiten. Die `LenB`-Funktion gibt die Größe der Variablen im Speicher für die meisten Variablentypen zurück. (Für Variablen vom Typ `Variant` wird die Größe der Daten zurückgegeben, die in der Variablen enthalten sind, nicht jedoch die Größe der Variablen selbst. Bei Zeichenfolgen wird die Länge der Zeichenfolge zurückgegeben.)

2. Puristen werden anmerken, dass diese Funktion wahrscheinlich die ineffizienteste Methode zum Abruf von Arbeitsspeicherdaten darstellt. In einer richtigen Anwendung würde die Funktion die Daten in ein Bytearray kopieren und die Bytes einzeln scannen. Der hier verwendete Ansatz mag zwar langsam sein, doch er bietet eine praktische Veranschaulichung der Aufgabe – und immerhin handelt es sich ja um ein Tutorium.

```
Private Sub Command1_Click()
   Dim newline As String
   Dim x&
   List1.Clear

   List1.AddItem "Integer: " & GetDataString(VarPtr(I), LenB(I))
   List1.AddItem "Long: " & GetDataString(VarPtr(L), LenB(L))
   List1.AddItem "Single (Float): " & GetDataString(VarPtr(F), LenB(F))
   List1.AddItem "Double: " & GetDataString(VarPtr(D), LenB(D))
   List1.AddItem "Currency: " & GetDataString(VarPtr(C), LenB(C))
   List1.AddItem "Byte: " & GetDataString(VarPtr(b), LenB(b))
   List1.AddItem "Variant: " & GetDataString(VarPtr(V), 16)
   List1.AddItem "String: " & GetDataString(VarPtr(S), LenB(S))

   For x = 0 To List1.ListCount
      newline = newline & List1.List(x) & vbCrLf
   Next x
   Clipboard.Clear
   Clipboard.SetText newline
End Sub
```

Die Ergebnisse, die im Listenfeld angezeigt werden, lauten folgendermaßen:

- `Integer`: 05 00
- `Long`: 05 00 00 00
- `Single (Float)`: 00 00 A0 40
- `Double`: 00 00 00 00 00 00 14 40
- `Currency`: 50 C3 00 00 00 00 00 00
- `Byte`: 05
- `Variant`: 02 00 00 00 00 00 00 00 05 00 00 00 00 00 00 00
- `String`: E4 05 18 00[3]

Das Wichtigste hierbei sind die Werte der `Integer`- und `Long`-Variablen. Wenn Sie die Zahl 5 einer 16-Bit-Ganzzahl zuordnen, lautet der Binärwert 0000000000000101 (oder &H0005). Erinnern Sie sich noch an das Zahlenschema, dass wir anfangs verwendeten, bei dem das ganz rechts aufgeführte Bit das Bit #0 war? Das ganz rechts angezeigte Bit wird als das unbedeutendste Bit (oder Nibble) bezeichnet, da es die kleinste Änderung am Wert der Gesamtzahl darstellt. Wie Sie an der Speicheranzeige für diese Variablen sehen können, erscheint das niedrigste Bit an der unters-

3. Dieser Wert lautet auf Ihrem Computer sehr wahrscheinlich anders.

ten Speicheradresse (die Bytes im Arbeitsspeicher werden beginnend mit der niedrigsten Speicheradresse angezeigt). Daher erscheint die Zahl &H0005 in einem Speicherabbild als 05 00. Es scheint, als wurden die Bytes vertauscht! Das ist nicht ganz richtig – tatsächlich richtet sich dies nur nach der Anordnung der Bytes.

Die Werte für `Float`, `Double` und `Currency` scheinen nicht viel Sinn zu ergeben. Das liegt daran, dass sie tatsächlich keinen Sinn ergeben. Das intern von diesen Datentypen verwendete Format ist etwas, worüber sich selbst erfahrene Programmierer nicht viele Gedanken machen.

Der `Variant`-Typ ist ebenfalls ein Datentyp, über den Sie nur wenig nachdenken werden. Die 02 zu Beginn kennzeichnet, dass es sich um eine `Integer`-Variable mit Vorzeichen handelt. Ein Variant kann viele Variablentypen enthalten, daher muss der Typ intern gespeichert werden. Die 05 wird später im Datenbereich des Variants angezeigt.

Denken Sie nicht über die Zeichenfolge nach – das können Sie später immer noch.

Schlussfolgerung

In diesem Tutorium wurden einige wichtige Konzepte vorgestellt, die die Grundlage der Computerwissenschaften bilden. Viele Visual Basic-Programmierer kennen diese Konzepte nicht, aber es ist unbedingt erforderlich, diese Konzepte zu verinnerlichen, um auf einer fortgeschrittenen Ebene arbeiten zu können.

Sie müssen verstehen, dass eine Variable nur eine Methode darstellt, mit der Daten im Arbeitsspeicher gespeichert werden.

Sie müssen die Beziehung zwischen binären und hexadezimalen Zahlen kennen und mit hexadezimalen Zahlen arbeiten können.

Sie müssen verstehen, dass der Arbeitsspeicher eine Sequenz von Bytes darstellt, in dem auf jedes dieser Bytes über die zugehörige Adresse zugegriffen werden kann, dass jedoch bei einem Win32-System Daten tatsächlich in Blöcken von 32 Bits übertragen werden.

Zu Beginn dieses Tutoriums haben wir festgestellt, dass die Zahl 5 in verschiedenen Variablentypen gespeichert werden kann und dass, solange Sie mit Visual Basic programmieren, die Konvertierung der Variablentypen automatisch stattfindet, sodass Sie immer mit den Zahlen arbeiten, die Sie erwarten. Wenn Sie jedoch eine API-Funktion deklarieren, wird über den Parametertyp in der `Declare`-Anweisung definiert, wie die Daten in die API-Funktion übertragen werden. Wenn das Datenformat im Speicher nicht mit dem durch die API-Funktion erwarteten Format übereinstimmt, schlägt die Funktion fehl oder das Ergebnis ist falsch.

Sie werden einige Zeit benötigen, um sich mit diesen Konzepten vertraut zu machen – so vertraut, dass Sie Ihnen in Fleisch und Blut übergehen. Machen Sie sich keine Sorgen, Ihnen werden in diesem Buch noch viele Beispiele begegnen, die auf diesen Konzepten beruhen. Zögern Sie nicht, sich dieses Tutorium noch einmal durchzusehen, wenn Sie an anderer Stelle Zweifel haben sollten. Je besser Sie diese Grundlagen verstehen, desto mehr Erfolg werden Sie bei der Verwendung von weiterführenden Techniken haben.

Tutorium 3

Ein Boolescher Wert und seine Bitfelder werden bald auseinandergehen

Boolesche Algebra

Dieser Begriff löst bei fast allen Nicht-Programmierern Panik aus. Und es ist erstaunlich, bei wie vielen Visual Basic-Programmierern genau die gleiche Angst ausgelöst wird. Keine Angst – Sie haben es vielleicht noch nicht bemerkt, aber durch die Lektüre von Tutorium 2, »Der Arbeitsspeicher – da, wo alles beginnt«, haben Sie schon 90 Prozent des Wissens erworben, dass Sie für die Arbeit mit der Win32-API auf höchster Ebene benötigen.

Und wenn Sie sich nicht mehr an die Schulalgebra erinnern, so ist das auch nicht schlimm. Das Wenige, das Sie wissen müssen, kann leicht mit diesem kurzen Kapitel abgedeckt werden, und gleichzeitig wird Ihnen die Tür zum Verständnis der verschiedensten Sachverhalte geöffnet – von Kombinationen von Flagkonstanten über die Funktionsweise von Cursorn bis zur Konvertierung komplexer C-Datenstrukturen.

Und wenn Sie wissen möchten, ob sich hinter dem Wort »Boole« eine mysteriöse Bedeutung verbirgt – die Boolesche Algebra wurde nach dem Mathematiker George Boole benannt.

Einfache Boolesche Operationen

Ihnen ist der Visual Basic-Typ `Boolean` wahrscheinlich bekannt. Dieser Variablentyp kann einen von zwei Werten annehmen: `True` oder `False`. Behalten Sie diesen Gedanken im Hinterkopf – wir werden später darauf zurückkommen.

Nachfolgend werden einige Boolesche Ausdrücke aufgeführt:

- Dim A As Boolean, B As Boolean
- A = True
- B = Not True

Wie lautet der Wert von B? Natürlich `False`. Dies ist ein logischer Gedankengang: Alles, was nicht `True` ist, muss `False` sein. Die Boolesche Algebra ist größtenteils so aufgebaut – nach dem Vernunftprinzip.

Wir verwenden eine spezielle Tabelle, genannt Wahrheitstabelle, um die Ergebnisse einer Booleschen Operation zu ermitteln. Nachstehend sehen Sie die Wahrheitstabelle für NOT-Operationen:

Variable A	Ergebnis (NOT A)
True	False
False	True

Tabelle T3-1 Die NOT-Operation

Die erste Spalte enthält sämtliche der möglichen Variablenwerte. Das Ergebnis bezeichnet die Daten, die Sie nach Durchführung der Operation erhalten. Daher ist `Not True=False`, `Not False=True`.

Der NOT-Operator ist am einfachsten zu verstehen. In Visual Basic sind drei Boolesche Operatoren vorhanden, die zwei Variablen verwenden, die Operatoren AND, OR und XOR.

Die Wahrheitstabellen für diese Operatoren lauten folgendermaßen:

Variable A	Variable B	Ergebnis (A And B)
False	False	False
False	True	False
True	False	False
True	True	True

Tabelle T3-2 Die AND-Operation

Variable A	Variable B	Ergebnis (A OR B)
False	False	False
False	True	True
True	False	True
True	True	True

Tabelle T3-3 Die OR-Operation

Variable A	Variable B	Ergebnis (A XOR B)
False	False	False
False	True	True
True	False	True
True	True	False

Tabelle T3-4 Die `XOR`-Operation

Sie können diese Angaben überprüfen, indem Sie das Beispielprojekt **Boolean.vbp** im Verzeichnis mit den Tutorienbeispielen ausführen. Diese Datei befindet sich auf der Begleit-CD-ROM zu diesem Buch.

Wenn Sie die Datei ausführen, brauchen Sie nur auf die Steuerelementtitel zu klicken, um für die Variablen A und B zwischen den Werten `True` und `False` zu wechseln (deaktivieren Sie nicht das Kontrollkästchen `Use Boolean Variables`).

Boolesche Werte und Zahlen

Die bisher beschriebenen Booleschen Operationen werden mit Variablen verwendet, die den Wert `True` oder `False` tragen können. Visual Basic-Variablen, die als `Boolean` deklariert werden, können nur einen dieser beiden Werte annehmen.

Sehen Sie sich das folgende Beispielprogramm an:

```
Dim A As Boolean
Dim B As Integer
A = 5
Debug.Print A
B = A
Debug.Print B
```

Wenn Sie der Booleschen Variablen A einen Wert zuordnen, der ungleich Null (5) ist, wird diese auf `True` gesetzt. Durch die erste `Debug.Print`-Anweisung wird `True` zurückgegeben. Wenn Sie der Variablen A einen numerischen Wert zuweisen, lautet der zugewiesene Wert -1.

Dies wirft eine interessante Frage auf. Wenn für jeden Wert ungleich Null `True` gilt, warum muss -1 verwendet werden?

Ob Sie es glauben oder nicht, ich stelle diese Frage nicht nur, um die Tutorien interessanter zu gestalten. Wenn Sie die Antwort auf diese Frage nicht verstehen, besteht die Gefahr, dass Sie in viele Ihrer VB-Programme mit API-Funktionen un-

beabsichtigt schwer auffindbare Bugs einbauen – die Art Bug, die man erst nach Stunden mühevoller Arbeit entdeckt.

Zur Beantwortung dieser Frage stellen wir zunächst eine andere Frage:

Sie wissen, wie Boolesche Operationen mit `Boolean`-Variablen funktionieren, die auf den Wahrheitstabellen beruht, die in einem der vorangegangenen Abschnitte gezeigt wurden. Wie funktionieren Sie jedoch bei Verwendung von numerischen Variablen?

Eine numerische Variable setzt sich aus Bits zusammen (wie beschrieben in Tutorium 2, »Der Arbeitsspeicher – dort, wo alles beginnt«). `Byte`-Variablen verfügen über 8 Bits, `Integer`-Variablen weisen 16 Bits, `Long`-Variablen 32 Bits auf. Jedes dieser Bits kann entweder den Wert 1 oder 0 tragen (was in gewisser Weise den Werten `True` oder `False` entspricht).

Beispiel: Wenn Sie den `NOT`-Operator für eine 16-Bit-Variable verwenden, die die Zahl 5 enthält, dann erhalten Sie folgendes Ergebnis:

```
5 = 0000000000000101 (binary)
Not 5 -> Not 0000000000000101 (binary) -> 1111111111111010 (binary) -
> &HFFFA (hex) -> -6 (decimal)
```

Wie Sie sehen, wird die NOT-Operation auf jedes Bit angewandt.

Was geschieht, wenn Sie einen Operator verwenden, der auf zwei Variablen angewandt wird? Die Operation wird für jedes Bitpaar der zwei Variablen durchgeführt. Beispiel:

```
5 Or 2 ->
0000000000000101 (binary) Or 0000000000000010 (binary) ->
0000000000000111 (binary) -> 7
5 Or 2 = 7
```

Sie können das Projekt **Boolean.vbp** dazu verwenden, ein wenig mit verschiedenen Zahlen und den Booleschen Operatoren herumzuexperimentieren (das Projekt finden Sie im Tutorienabschnitt auf der CD-ROM). Deaktivieren Sie das Kontrollkästchen `Use Boolean Variables` nur dann, wenn Sie mit anderen Zahlen experimentieren möchten. Ich empfehle Ihnen, zu Experimentierzwecken in die beiden Textfelder hexadezimale Werte einzugeben (Sie müssen vor den Werten lediglich ein `&H` ergänzen). Die Ergebnisse werden im Hexadezimalformat angezeigt, und Sie sollten sich im Umgang mit Booleschen Operationen an die Verwendung von Hexadezimalwerten gewöhnen.

Warum dies wichtig ist?

Weil Sie in viele Situationen geraten werden, in denen API-Funktionen und -Strukturen einzelne Bits und Bitgruppen innerhalb einer numerischen Variablen verwenden.

Kombinieren von Bits mit Hilfe des OR-Operators

Angenommen, Sie möchten die `SetWindowPos`-Funktion dazu verwenden, die Position eines Fensters auf dem Bildschirm zu ändern. Diese Funktion verfügt über eine Reihe von Fähigkeiten, die diejenigen übersteigen, die von der Visual Basic-Funktion `Move` bereitgestellt werden. Sie können ein Fenster beispielsweise so einstellen, dass es sich immer um das oberste Fenster handelt, d.h., dieses Fenster wird über allen anderen angezeigt, selbst dann, wenn es sich nicht um das aktive Fenster handelt. Die genaue Operation der Funktion wird durch einen so genannten »Flagparameter« definiert, der teilweise in der Win32-Dokumentation beschrieben wird:

`uFlags`

Gibt die Größen- und Positionsflags für ein Fenster an. Dieser Parameter kann eine Kombination der folgenden Werte darstellen:

Wert	Bedeutung
`SWP_HIDEWINDOW`	Verbirgt das Fenster.
`SWP_NOACTIVATE`	Keine Aktivierung des Fensters. Ist dieses Flag nicht gesetzt, wird das Fenster aktiviert und wird vor dem obersten Fenster oder der nicht obersten Gruppe angezeigt (je nach Einstellungen des `hWndInsertAfter`-Parameters).
`SWP_NOCOPYBITS`	Löscht den gesamten Inhalt des Clientbereichs. Ist dieses Flag nicht gesetzt, werden die gültigen Inhalte des Clientbereichs gespeichert und in den Clientbereich zurückkopiert, sobald die Fenstergröße angepasst oder die Fensterposition geändert wurde.
`SWP_NOMOVE`	Beibehaltung der aktuellen Position (`X-` und `Y-`Parameter werden ignoriert).
`SWP_NOSIZE`	Beibehaltung der aktuellen Größe (`cx-` und `cy-`Parameter werden ignoriert).
`SWP_NOZORDER`	Beibehaltung der aktuellen Z-Reihenfolge (der Parmeter `hWndInsertAfter` wird ignoriert).
`SWP_SHOWWINDOW`	Zeigt das Fenster an.

Machen Sie sich keine Sorgen, wenn Sie die Bedeutung dieser Parameter oder die Bedeutung des Wortes »Flag« in diesem Kontext nicht verstanden haben – hierzu kommen wir in Kürze. Ihnen wird dieser Parametertyp in vielen API-Funktionen begegnen. Was bedeutet die Aussage »eine Kombination der folgenden Werte«?

Diese Werte sind Namen von Konstanten, die in der Datei **api32.txt** definiert werden (diese Datei gehört zum Lieferumfang dieses Buches). Lassen Sie uns die aktuellen Werte dieser Konstanten betrachten:

```
Public Const SWP_NOSIZE = &H1
Public Const SWP_NOMOVE = &H2
Public Const SWP_NOZORDER = &H4
Public Const SWP_NOACTIVATE = &H10
Public Const SWP_SHOWWINDOW = &H40
Public Const SWP_HIDEWINDOW = &H80
Public Const SWP_NOCOPYBITS = &H100
```

Sehen Sie sich die Werte selbst einmal genau an. Es handelt sich in allen Fällen um Hexadezimalwerte. Stellen Sie sich die Werte als Binärwerte vor (Sie sollten in der Lage sein, die Umrechnung im Kopf durchzuführen, Sie können jedoch auch die Tabellen in Tutorium 2 für die Umrechnung heranziehen). Sie sollten folgende Ergebnisse erhalten.

```
Public Const SWP_NOSIZE =     &H1 =0000000000000001
Public Const SWP_NOMOVE =     &H2 =0000000000000010
Public Const SWP_NOZORDER =   &H4 =0000000000000100
Public Const SWP_NOACTIVATE = &H10 =0000000000010000
Public Const SWP_SHOWWINDOW = &H40 =0000000001000000
Public Const SWP_HIDEWINDOW = &H80 =0000000010000000
Public Const SWP_NOCOPYBITS = &H100 =0000000100000000
```

Sehen Sie es jetzt? Jede der Konstanten stellt eine Zahl dar, bei der nur ein einziges Bit gesetzt ist. Nicht jedes Bit in dieser Auflistung verfügt über eine Konstante, es handelt sich jedoch aus Platzgründen auch nur um einen Tabellenauszug. Ein 32-Bit umfassender `Long`-Wert kann auf diese Weise über bis zu 32 Optionen verfügen, eine für jedes Bit. Der `uFlags`-Parameter ist ein 32-Bit-Wert, auch wenn in der obigen Liste nur 16 Bits angezeigt werden.

Angenommen, Sie möchten die Parameter zum Verschieben und die Größenparameter ignorieren und das Fenster verbergen, ohne dessen Position in der Z-Reihenfolge zu ändern. Um dies zu erreichen, würden Sie die Konstanten `SWP_NOSIZE`, `SWP_NOMOVE`, `SWP_NOZORDER` und `SWP_HIDEWINDOW` miteinander kombinieren.

Was geschieht, wenn Sie die Konstanten mit Hilfe des `AND`-Operators miteinander verbinden?

```
&H1 And &H2 And &H4 And &H80 = 0
```

wegen jedes Bitpaares , 1 AND 0 = 0.

Wenn Sie jedoch den OR-Operator verwenden, erhalten Sie die folgenden Ergebnisse:

```
SWP_NOSIZE Or SWP_NOMOVE Or SWP_NOZORDER Or SWP_HIDEWINDOW ->
&H1 Or &H2 Or &H4 Or &H80  -> &H87 -> 000010000111 binär.
```

Sehen Sie sich die Binärwerte einer jeden Konstanten und den Binärwert des Ergebnisses an. Durch den OR-Operator werden in Wirklichkeit die individuellen Bits miteinander kombiniert.

Warum sollte man diesen Ansatz bei der Übergabe von Anweisungen an eine Funktion verwenden? Wenn Sie für jede Option einen separaten Parameter verwenden würden, hätten Sie bald Funktionen mit Dutzenden von Parametern. Darüber hinaus erfordert jeder Parameter beim Aufruf einer Funktion nicht nur zusätzlichen Speicherplatz, sondern auch zusätzliche CPU-Operationen zur Speicherung des Parameters im Stack. Das Verwenden einzelner Bits in Parametern stellt eine äußerst effiziente Methode dar. Durch das Angeben von Konstanten, durch die einzelne Bits repräsentiert werden, ist es möglich, mit Hilfe des OR-Operators jede beliebige Bitkombination genau einzustellen. Jeder Parameter bzw. jede Variable, deren Bedeutung durch individuelle Bits definiert wird, die über separate Definitionen verfügen, werden als Flagvariablen bezeichnet; die einzelnen Bits werden Flagbits genannt.

Wie sieht es mit der Bedeutung der einzelnen Konstanten und mit der Verwendung der SetWindowsPos-Funktion aus? Deklarationen und Verwendungsweise dieser Funktion sowie weitere API-Funktionen finden Sie in meinem Buch »Dan Appleman's Visual Basic Programmer's Guide to the Win32 API« oder in jeder anderen Win32-API-Referenz. Eine genauere Erläuterung dieser Beziehungen würde den Rahmen dieses Buches schlicht sprengen.

Auffinden von Bits mit Hilfe des AND-Operators

Angenommen, Sie möchten Informationen zu einem Fenster im System abrufen. Verfügt dieses Fenster über einen Rahmen? Ist es minimiert oder maximiert? Weist das Fenster eine vertikale oder horizontale Bildlaufleiste auf? Über die Win32-API können Sie separate Funktionen für jede dieser Fragen definieren, dies wäre jedoch eine sehr ineffiziente Vorgehensweise. Alle diese Fragen können mit Ja oder Nein beantwortet werden, was wiederum durch ein einzelnes Bit dargestellt werden kann. Tatsächlich werden diese Informationen fensterintern als 32-Bit-Long-Wert gespeichert, der als Fensterstil bezeichnet wird. Sie können diesen Wert mit Hilfe der Funktion GetWindowLong auf die folgende Weise abrufen:

```
Public Const GWL_STYLE = (-16)
Declare Function GetWindowLong Lib "user32" Alias _
"GetWindowLongA" (ByVal hwnd As Long, ByVal nIndex As Long) As Long
' Abrufen der Klasseninformationen
  style& = GetWindowLong&(useHwnd&, GWL_STYLE)
```

Hier werden einige der verfügbaren Stilmerkmale aufgeführt:

```
Public Const WS_MINIMIZE = &H20000000
Public Const WS_VISIBLE = &H10000000
Public Const WS_DISABLED = &H8000000
Public Const WS_MAXIMIZE = &H1000000
Public Const WS_BORDER = &H800000
Public Const WS_VSCROLL = &H200000
Public Const WS_HSCROLL = &H100000
```

Angenommen, Sie ermittelten, dass ein Fenster über das Stilmerkmal `&H1A00000` verfügt.

Können Sie darüber hinaus ermitteln, ob das Fenster eine horizontale Bildlaufleiste aufweist?

Um dies feststellen zu können, müssen Sie den Wert eines einzelnen Bits ermitteln können. Lassen Sie uns mit einem einfachen Beispiel beginnen. Nehmen Sie die Zahl 5 oder 101 (binär). Definieren Sie, wie nachfolgend gezeigt, drei Konstanten:

- `BIT0 = &H1`
- `BIT1 = &H2`
- `BIT2 = &H4`

Überlegen Sie nun, was geschieht, wenn Sie den `AND`-Operator mit jedem dieser Werte verwenden.

```
BIT0 And 5 -> 001 And 101 -> 001
BIT1 And 5 -> 010 And 101 -> 000
BIT2 And 5 -> 100 And 101 -> 100
```

Wenn ein Bit in der Konstante 0 lautet, ist das Ergebnisbit automatisch auch 0 – dies ist die Funktionsweise des `AND`-Operators. Lautet ein Bit in der Konstante 1, hängt das Ergebnisbit davon ab, wie der Wert des zweiten Parameters lautet. Im vorliegenden Fall die Zahl 5. Die Konstante dient tatsächlich als Maske, durch die nur die Bits verwendet werden können, die Sie im Ergebnis ausgewählt haben. Konstanten, die auf diese Weise verwendet werden, werden häufig als Maskierungskonstanten bezeichnet.

Verfügt das Fenster nun über eine horizontale Bildlaufleiste oder nicht? Um dies herauszufinden, verwenden wir den AND-Operator mit der Maskenkonstante WS_HSCROLL:

```
style &And WS_HSCROLL -> &H1A00000 And &H100000 -> 0
```

Sie können die binären Werte selbst berechnen, wenn Sie möchten.

Verfügt das Fenster über eine vertikale Bildlaufleiste?

```
style &And WS_VSCROLL -> &H1A00000 And &H200000 -> &H200000
```

Das WS_VSCROLL-Bit im style ist gesetzt, daher verfügt das Fenster über eine vertikale Bildlaufleiste. Sie können die Vorteile aus der Tatsache ziehen, dass der Ergebniswert nur dann ungleich Null ist, wenn die durch die Maske identifizierten Bits gesetzt sind. Durch die If/Then-Anweisung wird der Wert &H200000 als True interpretiert, daher wird durch die folgende Anweisung

```
If style& And WS_VSCROLL Then
```

der Code innerhalb des Bedingungsblocks ausgeführt, wenn eine vertikale Bildlaufleiste vorhanden ist.

Der folgende Auszug aus dem Programm **Winview.vbp** in Kapitel 5 meines Win32-API-Buches zeigt, wie Sie eine Zeichenfolge erstellen, mit der die Stilmerkmale eines Fensters beschrieben werden. Eine Einführung in Fenster- und Klassenstilmerkmale finden Sie in Kapitel 2 des genannten Buches.

```
If style& And WS_BORDER Then
    outstring$ = outstring$ + "WS_BORDER" + crlf$
End If
If style& And WS_DISABLED Then
    outstring$ = outstring$ + "WS_DISABLED" + crlf$
End If
If style& And WS_HSCROLL Then
    outstring$ = outstring$ + "WS_HSCROLL" + crlf$
End If
If style& And WS_MAXIMIZE Then
    outstring$ = outstring$ + "WS_MAXIMIZE" + crlf$
End If
If style& And WS_MINIMIZE Then
    outstring$ = outstring$ + "WS_MINIMIZE" + crlf$
End If
```

```
If style& And WS_VSCROLL Then
    outstring$ = outstring$ + "WS_VSCROLL" + crlf$
End If
```

Löschen eines Bits

Sie wissen, dass Sie ein Bit mit Hilfe der OR-Operation setzen können. Wenn Sie ein Bit auf 1 setzen möchten, verwenden Sie

```
Variable = Variable Or BIT1
```

Aber wie löschen Sie ein Bit? Der OR-Operator kann hierzu offensichtlich nicht verwendet werden.

Sie könnten Folgendes ausprobieren:

```
Variable = Variable And BIT1
```

Stellen Sie sich vor, die Variable enthielte bei Ausführung dieser Operation den Wert 6.

```
6 And BIT1 = 6 And 2 = 0110 (binary) And 0010 (binary) = 0010 = &H2.
```

Dieses Verfahren funktioniert nicht. Durch dieses Vorgehen wird jedes Bit mit Ausnahme von Bit 1 gelöscht. Es wird also das Gegenteil von dem erreicht, was eigentlich erzielt werden sollte. Mit Hilfe der AND-Operation kann ermittelt werden, ob ein Bit gesetzt ist, aber kann mit dieser Operation ein Bit auch gelöscht werden?

Ja. Wie Sie sich vielleicht erinnern, kann mit dem AND-Operator jedes Bit auf 0 gesetzt werden. Der Trick hierbei besteht darin, dass die Maske eine 0 für das zu löschende Bit bereitstellt, keine 1. Für die beizubehaltenden Bits muss eine 1 bereitgestellt werden, keine 0. Durch welchen Operator wird eine derartige Änderung der Bitwerte vorgenommen? Durch den NOT-Operator.

Versuchen Sie es mit dem folgenden Beispiel:

```
6 And (Not BIT1) = 6 And (Not 2) = 0110 And (Not 0010) = 0110 And 1101 = 01
00 = &H4.
```

Bit 1 wurde erfolgreich gelöscht!

Die Magie von XOR

Die letzte Boolesche Operation, mit der wir uns befassen werden, ist die XOR-Operation, auch »exklusive OR-Operation« genannt. Es handelt sich hierbei um einen etwas merkwürdigen Operator, und es ist häufig etwas schwierig, einem Anfänger dessen Bedeutung klarzumachen. Lassen Sie uns ein hypothetisches

Beispiel heranziehen, bei dem eine Variable über drei Flagbits verfügt. Die Bits weisen die Konstanten BIT0, BIT1 und BIT2 auf, die in einem der vorangegangenen Abschnitte in diesem Tutorium definiert wurden.

Jetzt lassen Sie uns etwas Neues ausprobieren. Was geschieht, wenn Sie den XOR-Operator mit der BIT1-Maske verwenden?

```
6 Xor BIT1 -> 0110 Xor 0010 -> 0100 = &H4
```

Durch den XOR-Operator wurde das Bit in nur einer Operation gelöscht. Was geschieht, wenn Sie den gleichen Operator erneut auf das Ergebnis anwenden?

```
4 Xor BIT1 -> 0100 Xor 0010 -> 0110 -> &H6
```

Das Bit wurde gesetzt!

Zusammenfassung

- Zum Setzen eines Bits verwenden Sie den OR-Operator mit einer Bitmaske.
- Zum Prüfen eines Bits verwenden Sie den AND-Operator mit einer Bitmaske.
- Zum Löschen eines Bits verwenden Sie den AND-Operator mit der invertierten Bitmaske (der Bitmaske, die sich ergibt, wenn Sie den NOT-Operator verwendet haben).
- Zum Wechseln zwischen den Bitzuständen verwenden Sie den XOR-Operator.

Das Wechseln zwischen den Bitzuständen wird häufig bei Grafikoperationen eingesetzt. Ein besonders Merkmal des XOR-Operators ist es, dass bei dessen Anwendung auf eine Variable und ein sich anschließendes erneutes Anwenden auf das zuvor erhaltene Ergebnis wieder der ursprüngliche Wert zurückgegeben wird. Wenn Sie den XOR-Operator bei Grafikoperationen einsetzen, können Sie bei dessen einmaliger Verwendung eine Änderung der Grafik erreichen (beispielsweise das Zeichnen einer Linie oder eine Kombination der ersten Grafik mit einer zweiten Grafik). Durch eine erneute Verwendung des Operators wird das ursprüngliche Bild wiederhergestellt. Der XOR-Operator wird daher häufig in Sprites, Animationen und bei Cursorn verwendet.

Dieses Verfahren wird im Beispielprogramm **XORLine.vbp** auf der Begleit-CD-ROM zu diesem Buch veranschaulicht. Wenn Sie den XOR-Operator selbst ausprobieren möchten, erstellen Sie ein Formular mit den folgenden Eigenschaften:

BorderStyle	3 – Fixed Dialog
DrawMode	7 – Xor Pen
ForeColor	&HooFFFFFF& (weißer Stift)

```
' XORLine Example
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten

Option Explicit

Dim PrevX As Single
Dim PrevY As Single
Dim centerX As Single, centerY As Single

Private Sub Form_Load()
   PrevX = -1
   PrevY = -1
   centerX = Width / 2
   centerY = Height / 2
End Sub
Private Sub Form_MouseMove(Button As Integer, Shift As Integer, _
X As Single, Y As Single)
   ' Bei gleichem Standort beibehalten
   If X = PrevX And Y = PrevY Then Exit Sub

   If PrevX >= 0 Then
      ' Löschen der vorherigen Linie
      Line (centerX, centerY)-(PrevX, PrevY)
   End If
   Line (centerX, centerY)-(X, Y)
   PrevX = X
   PrevY = Y
End Sub
```

Wenn Sie die Maus über das Formular bewegen, wird eine Linie vom Mittelpunkt des Formulars bis zur Maus gezogen. Durch die Routine wird verfolgt, wo sich die aktuelle Linie befindet, wenn also ein neues `MouseMove`-Ereignis eintritt, kann die zuvor gezeichnete Linie gelöscht werden, bevor die neue Linie eingezeichnet wird.

Wie wird die zuvor gezeichnete Linie gelöscht? Durch Wiederholung der zuvor durchgeführten Zeichenoperation. Diese Methode funktioniert, da der Zeichenmodus auf `XOR` eingestellt ist. Das bedeutet, dass die Stiftfarbe unter Verwendung der `XOR`-Operation auf die Formularfarbe eingestellt wird. Die Stiftfarbe ist weiß – alle Bits sind auf 1 gesetzt, das Ergebnis ist demnach, dass die Werte der Bits im Formular umgekehrt werden, wenn die Maus verschoben wird.

Ändern Sie die Hintergrundfarbe des Formulars und beobachten Sie, wie sich die aktuelle Linienfarbe ändert.

Boolescher Vergleich

Vor langer Zeit (zu Beginn dieses Tutoriums) erwähnte ich, dass es sehr wichtig ist, zu verstehen, warum Boolesche Variablen in Visual Basic immer den Wert –1 aufweisen.

Um diesen Umstand zu erklären, greifen wir erneut auf den Beispielcode zurück, mit dem ermittelt wurde, ob ein Fenster über einen Rahmen verfügt.

```
' Abrufen der Klasseninformationen
    style& = GetWindowLong&(useHwnd&, GWL_STYLE)

If style& And WS_BORDER Then
    ' Dieser Code wird ausgeführt, wenn das Fenster über einen Rahmen
    ' verfügt
End If
```

Dieser Code funktioniert, sofern das Fenster über einen Rahmen verfügt, da der Ausdruck »`style& And WS_BORDER`« den Ergebniswert `&H800000` aufweist. Dieser Wert ist ungleich Null, demnach funktioniert die bedingte Anweisung.

Was würde passieren, wenn Sie Ihren Code so umschreiben, dass die Anweisung ausgeführt wird, wenn das Fenster über keinen Rahmen verfügt?

```
If Not (style& And WS_BORDER) Then
    ' Dieser Code soll ausgeführt werden, wenn das Fenster keinen Rahmen
    ' aufweist
End If
```

Wenn das Fenster keinen Rahmen aufweist führt der Ausdruck »`Not (style& And WS_BORDER)`« zu dem Ergebnis »`Not 0`«, was wiederum zum Ergebnis –1 bzw. `True` führt. Daher wird die bedingte Anweisung ebenfalls ausgeführt.

Verfügt das Fenster über einen Rahmen, führt der Ausdruck zu dem Ergebnis `Not &H800000`, was `&H7FFFFF` entspricht – ebenfalls ein Wert, der ungleich Null, d.h. `True` ist. Daher wird die bedingte Anweisung ausgeführt.

Der Bedingungscode wird in diesem Fall immer ausgeführt. Es handelt sich offensichtlich um einen Bug. Und es handelt sich um einen Bug, der schwer auffindbar ist.

Darum verwenden in Visual Basic Boolesche Variablen immer den Wert –1 für `True` und den Wert 0 für `False`. Würden Sie dies nicht tun, würden bedingte Ausdrücke fehlschlagen.

Warum ist dies für API-Programmierer von Wichtigkeit? Da in Visual Basic konsistent der Wert –1 für `True` verwendet wird, während die Win32-API jeden Wert ungleich Null, besonders die Zahl 1, zur Kennzeichnung eines Ergebnisses verwendet, für das `True` gilt.

Daher sollten vor einem Vergleich die Ergebnisse einer API-Funktion oder Bitoperation stets mit Null verglichen werden. Das zuvor genannte Beispiel kann mit dem folgenden Code leicht gelöst werden:

```
If Not ((style& And WS_BORDER)<>0) Then
    ' Dieser Code soll ausgeführt werden, wenn das Fenster keinen Rahmen
    ' aufweist
End If
```

oder

```
If (style& And WS_BORDER)=0 Then
    ' Dieser Code soll ausgeführt werden, wenn das Fenster keinen Rahmen
    ' aufweist
End If
```

C und Boolesche Operationen

Nun, da Sie wissen, wie in Visual Basic Boolesche Operatoren verwendet werden, sollten wir uns kurz ansehen, worin die Unterschiede zwischen C und Visual Basic liegen. Boolesche Visual Basic-Variablen sind identisch mit `Integer`-Variablen. Und da die Booleschen Operatoren in Visual Basic für jedes Bit in einer Variable ausgeführt werden, müssen Variablen für `True` auf –1 und für `False` auf 0 gesetzt werden, damit ein ordnungsgemäßer Vergleich durchgeführt werden kann.

In der C-Sprache wird ein prinzipiell anderer Ansatz verfolgt. Es werden zwei unterschiedliche Sätze Boolescher Operatoren definiert. Der erste Satz dient dem Vergleich; deren Operation basiert auf dem Wert der gesamten Variablen. Hierbei entspricht der Wert 0 gleich `False` und jeder Wert, der ungleich Null ist, `True`. Diese Operatoren werden nachfolgend aufgeführt:

C-Operator	Bedeutung
&&	Logisches AND
||	Logisches OR
!	Logisches NOT

Beispiel: Sehen Sie sich die Ergebnisse des !-Operators für 0, 1 und -1 an.

- !0 = Jeder Wert ungleich 0
- !1 = 0
- !-1 = 0

Diese Operatoren werden nicht auf jedes Bit in der Variablen angewandt, sondern auf die Variable als Einheit. Die Win32-API wurde anfänglich zur Verwendung mit C konzipiert, daher wurde festgelegt, dass jeder Rückgabewert ungleich Null dem Zustand True entspricht. Es wurde hierbei vorausgesetzt, dass ein Programmierer in C einen logischen Booleschen Vergleich durchführen würde, der mit einem beliebigen Wert zu einem richtigen Ergebnis führt.

Was jedoch, wenn ein C-Programmierer mit den einzelnen Bits arbeiten muss – sollen die Werte mit Hilfe der OR-Operation oder durch Maskierung mit Hilfe der AND-Operation kombiniert werden? Die Sprache C bietet verschiedenen Operatoren für diese »bitweisen« Booleschen Operationen:

C-Operator	Bedeutung
&	Bitweise UND
|	Bitweise ODER
~	Bitweise NICHT (Negation)
^	Bitweise exklusive OR-Operation

Ihnen werden diese Operatoren gelegentlich bei der Lektüre von C-Headerdateien begegnen.

C-Bitfelder

In den Sprachen C und C++ wird gelegentlich eine Funktion verwendet, die als Bitfeld bezeichnet wird. Mit diesen Bitfeldern können Sie einem einzelnen Bit in einer Variablen einer Struktur eine bestimmte Bedeutung zuweisen. Sehen Sie sich die folgende `DCB`-Struktur an:

```
typedef struct _DCB {  // dcb
    DWORD DCBlength;              // Größe von (DCB)
    DWORD BaudRate;               // aktuelle Baudrate
    DWORD fBinary: 1;             // binärer Modus, keine EOF-Prüfung
    DWORD fParity: 1;             // Aktivieren der Paritätsprüfung
    DWORD fOutxCtsFlow:1;         // Flusssteuerung für CTS-Ausgabe
    DWORD fOutxDsrFlow:1;         // Flusssteuerung für DSR-Ausgabe
```

```
    DWORD fDtrControl:2;          // DTR-Typ für Flusssteuerung
    DWORD fDsrSensitivity:1;      // DSR sensitivity
    DWORD fTXContinueOnXoff:1;    // XOFF continues Tx
    DWORD fOutX: 1;               // XON/XOFF out flow control
    DWORD fInX: 1;                // XON/XOFF in flow control
    DWORD fErrorChar: 1;          // Aktivieren der Fehlerersetzung
    DWORD fNull: 1;               // Aktivieren von Null Stripping
    DWORD fRtsControl:2;          // RTS-Flusssteuerung
    DWORD fAbortOnError:1;        // Abbrechen der Lese-/
Schreibvorgänge bei Fehler
    DWORD fDummy2:17;             // reserviert
    WORD wReserved;               // derzeit nicht verwendet
    WORD XonLim;                  // Übertragen des XON Schwellwert
    WORD XoffLim;                 // Übertragen des  XOFF Schwellwert
    BYTE ByteSize;                // Anzahl der Bits/Byte, 4-8
    BYTE Parity;                  // 0-4=no,odd,even,mark,space
    BYTE StopBits;                // 0,1,2 = 1, 1.5, 2
    char XonChar;                 // Tx und Rx XON Zeichen
    char XoffChar;                // Tx und Rx XOFF Zeichen
    char ErrorChar;               // Zeichen für Fehlerersetzung
    char EofChar;                 // Ende des Eingabezeichens
    char EvtChar;                 // erhaltenes Ereigniszeichen
    WORD wReserved1;              // reserviert, nicht verwenden
}
```

Nähere Informationen zur Verwendung verschiedener Variablentypen finden Sie in Tutorium 6, »C++-Variablen treffen auf Visual Basic«. Für den Augenblick mag es genügen, wenn Sie wissen, dass ein »`struct`« in C++ in Visual Basic zu einem benutzerdefinierten Typ wird und dass `DWORD` sich auf eine 32-Bit umfassende `Long`-Variable bezieht. Sehen Sie sich nur die ersten drei Felder in dieser C-Struktur an:

Bei dem ersten Feld handelt es sich um einen 32-Bit-`Long`-Wert mit dem Namen `DCBlength`.

Das zweite Feld ist ein 32-Bit-`Long`-Wert mit dem Namen `BaudRate`.

Bei dem dritten Feld, `fBinary`, handelt es sich nicht um eine Variable. Es ist eine 32-Bit-`Long`-Variable, deren Bits folgendermaßen definiert sind:

- Bit 0 entspricht `fBinary`
- Bit 1 entspricht `fParity`
- Bit 2 entspricht `fOutxCtsFlow`

- Bit 3 entspricht `fOutxDsrFlow`
- Bit 4 und 5 entsprechen `fDtrControl`
- Bit 6 entspricht `fDsrSensitivity`
- Bit 7 entspricht `fTXContinueOnXoff`
- Bit 8 entspricht `fOutX`
- Bit 9 entspricht `flnX`
- Bit 10 entspricht `fErrorChar`
- Bit 11 entspricht `fNull`
- Bit 12 und 13 entsprechen `fRtsControl`
- Bit 14 entspricht `fAbortOnError`
- Bit 15-31 werden nicht verwendet

Der Doppelpunkt (:) hinter dem Variablennamen gibt an, dass die Variable eine spezielle Anzahl von Bits umfasst. Da Visual Basic das Konzept von Bitfeldern nicht unterstützt, werden sämtliche dieser Bits in einer einzigen 32-Bit-`Long`-Variable zusammengefasst. Ein VB-Programmierer benötigt Boolesche Operationen, um den Wert einzelner Bits zu setzen, zu löschen und zu ermitteln.

Die tatsächliche `DCB`-Struktur in Visual Basic wird, wie nachfolgend gezeigt, definiert. Sämtliche der oben aufgeführten Bitfelder werden in einem einzigen Feld namens »Bits1« miteinander kombiniert.

```
Type DCB
      DCBlength As Long
      BaudRate As Long
      Bits1 As Long
      wReserved As Integer
      XonLim As Integer
      XoffLim As Integer
      ByteSize As Byte
      Parity As Byte
      StopBits As Byte
      XonChar As Byte
      XoffChar As Byte
      ErrorChar As Byte
      EofChar As Byte
      EvtChar As Byte
wReserved1As Integer
End Type
```

Wenn Sie beispielsweise über eine DCB-Variable mit dem Namen DCB1 verfügen und Sie die Paritätsprüfung aktivieren wollten, würden Sie die folgende Operation ausführen:

```
DCB1.Bits1 = DCB1.Bits1 Or &HO2(set bit 1)
```

Zur Deaktivierung der Paritätsprüfung führen Sie die folgende Operation aus:

```
DCB1.Bits1 = DCB1.Bits1 And (Not &HO2&)
```

Wenn Sie einen Codeblock nur dann ausführen möchten, wenn Parität ermöglicht wird, verwenden Sie einen Code wie diesen:

```
If (DCB1.Bits1 And &H2)<>0 Then
' Parität wird  ermöglicht
End If
```

Sehen Sie sich das Feld mit der Bezeichnung fDtrControl an. Dieses Feld umfasst zwei Bits, da die Steuerung der DTR-Linie (einem Signal, dass bei der seriellen Kommunikation verwendet wird) keine einfache Boolesche Operation darstellt. Dieses Signal verfügt über drei mögliche Modi. Es werden zwei Bits zur Repräsentation von drei Werten eingesetzt – in diesem Fall die Werte 0 bis 2 –, die durch die folgenden Konstanten beschrieben werden:

- Public Const DTR_CONTROL_DISABLE = &H0
- Public Const DTR_CONTROL_ENABLE = &H1
- Public Const DTR_CONTROL_HANDSHAKE = &H2

Wir wissen, dass die fDtrControl-Bits in den Bits 4 und 5 der Variablen Bits1 enthalten sind. Wie extrahieren wir die Werte, um diese zu lesen?

Zunächst erstellen wir eine Maske, damit nur die fDtrControl-Bits betrachtet werden. Bei der Maske müssen die Bits 4 und 5 gesetzt sein. Im Binärformat entspricht dies 0000000000110000 bzw. &H30.

Die Bits können folgendermaßen extrahiert werden:

```
DtrBits = DCB1.Bits1 And &H30
```

Wie gelangen Sie nun dahin, dass die zwei Bits mit den drei Konstanten verglichen werden können? Die Bits müssen hierzu noch von Position 4 und 5 an die Bits 0 und 1 verschoben werden. Sie erreichen eine Verschiebung um einen Wert nach rechts, indem Sie die Zahl durch 2 dividieren.

Sehen Sie sich zu diesem Verfahren die nachstehende Tabelle an.

Zahl	Binärformat	Division durch 2	Binärformat
16	10000	8	1000
8	1000	4	100
4	100	2	10
2	10	1	1

Eine Division durch 2 führt zu einer Verschiebung der Bits nach rechts. Eine Multiplikation mit 2 führt zu einer Verschiebung der Bits nach links. Der einzige Fall, bei dem es knifflig werden kann, ist das hohe Bit (Bit 31 in einer `Long`-Variablen), da hierbei eine Überlaufsituation entstehen kann. Dies trifft hier jedoch nicht zu, da das hohe Bit nicht verwendet wird.

Bei den `fDtrControl`-Bits müssen wird den `DtrBits`-Wert um 4 Bits nach rechts verschieben:

```
DtrValue = DtrBits / &H10000
```

Sie können den Wert mit der Konstanten vergleichen, um zu prüfen, ob die DTR-Steuerung aktiviert ist:

```
If DtrValue = DTR_CONTROL_ENABLE Then
' DTR_CONTROL_ENABLE ist gesetzt
End If
```

Angenommen, Sie möchten die `fDtrControl`-Bits auf `DTR_CONTROL_HANDSHAKE` setzen. Hierzu müssen Sie den Prozess umkehren:

```
DtrValue = DTR_CONTROL_HANDSHAKE   ' Setzen des Wertes
DtrBits = DtrValue * &H10000       ' Verschieben der Bits nach links
DCB1.Bits1 = DCB1.Bits1 And (Not &H30)' Löschen der zwei Bits
DCB1.Bits1 = DCB1.Bits 1Or DtrBits ' Or im gewünschten Wert
```

Sie können sich diesen Prozess genauer vor Augen führen, wenn Sie dieses Verfahren für mehrere Werte auf dem Papier nachvollziehen.

Einfügen von »Integer«- in »Long«-Variablen

Es gibt einen besonderen Fall, der der Verwendung von Bitfeldern sehr ähnlich ist und in der Win32-API häufig auftritt. Sie haben gesehen, dass bei Verwendung von Bitfeldern Gruppen von mehreren Bits in einer einzigen 32-Bit-`Long`-Variablen zusammengeführt werden können. Sie wissen, dass eine VB-`Integer`-Variable 16 Bits umfasst. Es ist möglich, zwei 16-Bit-`Integer`-Variablen in eine 32-Bit-`Long`-Variable einzufügen. Ein klassisches Beispiel hierfür liegt vor, wenn Sie zwei 16-Bit-

Mauskoordinaten für interne Windows-Mausmeldungen in einen Long-Wert einfügen. Die X-Koordinate wird in der unteren 16-Bit-Integer-Variablen, der Y-Wert in der oberen Integer-Variablen gespeichert.

Zur Kombination der Integer-Werte X und Y in einer Long-Variable mit dem Namen L gehen Sie folgendermaßen vor:

```
Dim XL As Long, YL As Long
' Beide Werte werden in Long-Werte konvertiert, um einen Überlauf zu
' verhindern.
' Anschließende Maskierung mit &H7FFF, um Fehler aufgrund von
' Zeichenerweiterungen zu verhindern.
XL = X And &HO7FFF&
YL = Y And &HO7FFF&
YL = YL * &H10000&
```

Um die Long-Variable L in die Integer-Werte X und Y aufzuteilen, gehen Sie folgendermaßen vor:

```
Dim XL As Long, YL As Long
XL = L And &HO7FFF&                ' Extrahieren des X-Wertes
X = XL                             ' Aufgrund der gewählten Maske
                                   ' kann kein Überlauf entstehen
YL = L And &H7FFF0000&
YL = YL / &H10000&
Y = YL
```

Dieses Beispiel funktioniert nur mit positiven Integer-Variablen. Bei negativen Werten müssen zusätzliche Schritte ausgeführt werden, um mögliche Überlauffehler zu vermeiden. Dies ist der Grund, warum das Einfügen und Entfernen am besten mit einer DLL-Unterstützungsbibliothek vorgenommen wird, die in C++ geschrieben wurde, beispielsweise der DLL **apigid32.dll**, die zum Lieferumfang dieses Buches gehört. In Anhang C finden Sie Beschreibungen der Funktionen agDWORDto2Integers und agPOINTStoLong, die zur Durchführung dieser Operationen eingesetzt werden können.

Tutorium 4

Die Arbeitsweise einer DLL: In einem Stackframe

Laufzeitfehler 49
Falsche DLL-Aufrufkonvention.

Kommt Ihnen das bekannt vor?

Dies ist die Fehlermeldung, die Sie erhalten, wenn Ihre `Declare`-Anweisung für einen DLL- oder API-Aufruf falsch ist – wenn Sie Glück haben. Wenn Sie kein Glück haben, weist Ihre Deklaration einen der vielen Fehler auf, die von Visual Basic nicht ermittelt werden können. Aber das ist eine andere Geschichte.

Woher rührt also dieser Fehler? Warum werden in Visual Basic einige DLL-Deklarationsfehler erkannt und andere nicht? Um die Antwort auf diese Fragen verstehen zu können, müssen Sie sich den Mechanismus näher ansehen, mit dem Funktionen aufgerufen werden, und Sie müssen den so genannten Stackframe unter die Lupe nehmen.

Stackframe? Klingt wie einer dieser esoterischen Computerbegriffe, mit denen Nichteingeweihte eingeschüchtert werden sollen. Das kommt wahrscheinlich daher, dass es sich um einen dieser esoterischen Computerbegriffe handelt, mit denen Nichteingeweihte eingeschüchtert werden sollen.

Jetzt mal ernsthaft – wenn Sie verstehen möchten, was ein Stackframe ist, müssen Sie zunächst einmal wissen, was ein Stack ist.

Der Stack

Ein Stack bezeichnet eine Datenstruktur im Arbeitsspeicher, für die zwei Standardoperationen durchgeführt werden. Diese Operationen sind `push`, mit der Daten im Stack platziert werden, und `pop`, mit der Daten aus dem Stack entfernt werden. Bei einem Stack handelt es sich um eine so genannte »Last In First Out«-Struktur, bei der die zuletzt platzierten Daten immer als erste entfernt werden.

Abbildung T4-1 veranschaulicht die `Push`- und `Pop`-Operationen, die für einen einfachen Stack mit 32-Bit-`Long`-Werten durchgeführt werden.

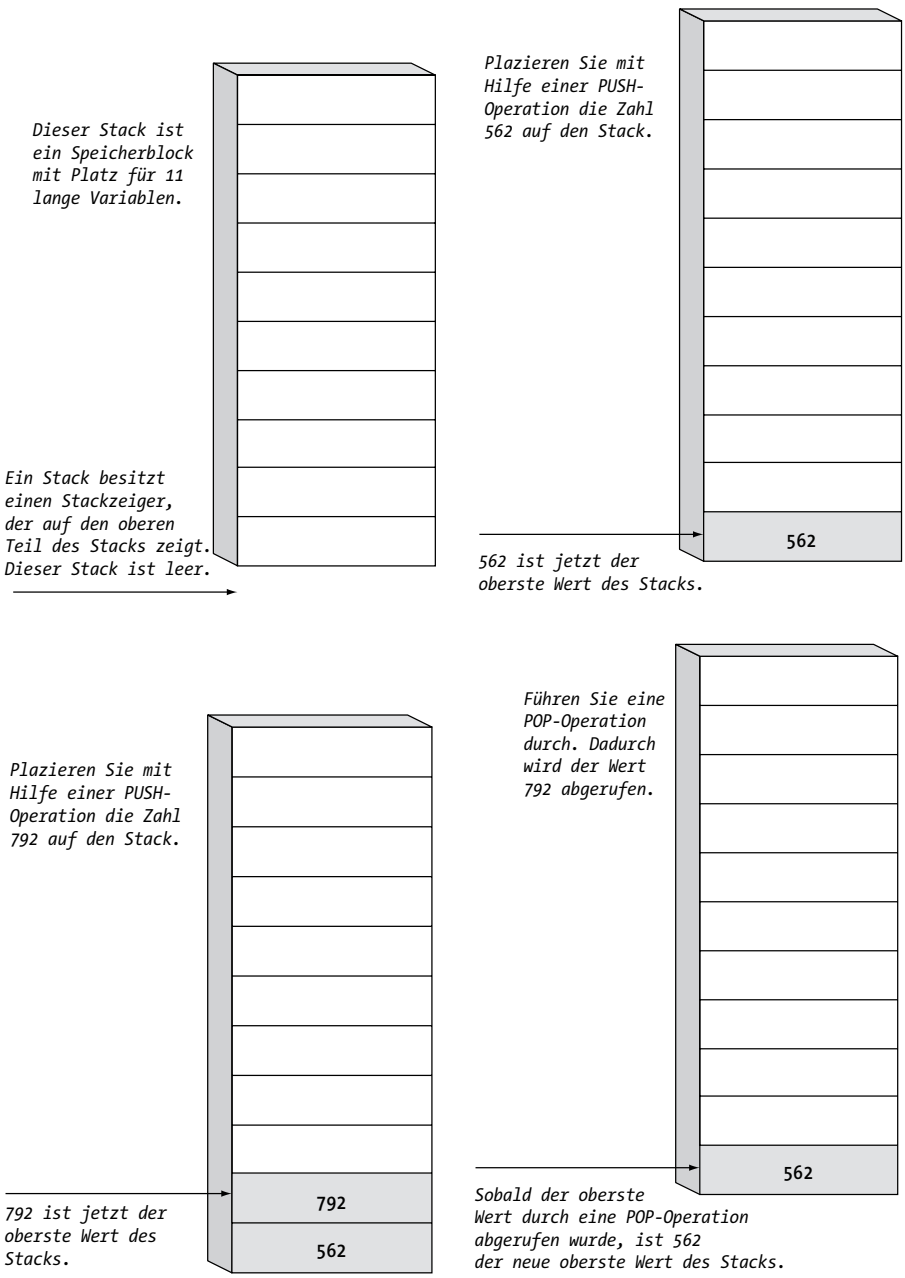


Abbildung T4-1a–d Die Arbeitsweise von Stacks

Die erste Abbildung zeigt das leere Stack. Jedes Stack verfügt über einen Stackzeiger, der auf den obersten Wert im Stack verweist. Bei diesem Zeiger kann es sich um eine Variable handeln, die Speicheradressen enthält, oder (falls der Stack in

einem Array implementiert ist) es handelt sich um einen Arrayindex des obersten Datenelements im Array. Es ist in Visual Basic nicht unüblich, Stack unter Verwendung von Arrays zu implementieren.

Werden Daten in einem Stack platziert, wird der Stackzeiger an die nächste Position verschoben, und die Daten werden an dem Standort abgelegt, auf den durch den Stackzeiger verwiesen wird. Wird eine `Pop`-Operation ausgeführt, wird das Datenelement, auf das durch den Stackzeiger verwiesen wird (als Ergebnis der `Pop`-Funktion) zurückgegeben, und der Stackzeiger wird wieder an die vorherige Position verschoben.

Zur Implementierung eines Stacks ist lediglich die Zuordnung eines Speicherblocks und die Ausführung von `Push`- und `Pop`-Operationen für diesen Block erforderlich. Stacks werden nicht oft in Anwendungen ohne speziellen Zweck verwendet, daher sind sie den meisten Visual Basic-Programmierern nicht geläufig. Dennoch stellen sie einen wichtigen Bestandteil jedes Funktions- oder Subroutinenaufrufs dar.

Sobald Sie eine Anwendung laden, wird einem Stack, der zu diesem Prozess gehört, im Speicher ein gewisser Bereich zugewiesen. Dieser Stack wird als Anwendungs- oder Prozessstack bezeichnet. Die CPU eines Computers bietet eine effiziente Hardwareunterstützung für diese Anwendungsstacks. Jede CPU verfügt über ein Hardwareregister, das dazu dient, als Stackzeiger zu fungieren, d.h. auf den Standort des obersten Datenelements in diesem Stack zu verweisen. Darüber hinaus werden `push`- und `pop`-Anweisungen in Maschinensprache bereitgestellt, zusammen mit Anweisungen, mit deren Hilfe in nur einem Arbeitsschritt mehrere Elemente in einem Stack platziert oder aus diesem entfernt werden können.

Des weiteren sind im Umgang mit Anwendungsstacks zwei weitere Faktoren zu berücksichtigen. Als erstes besteht jeder Stack aus 32 Bits. Wenn Sie ein kleineres Element im Stack platzieren möchten, bleibt der nicht benötigte Speicherplatz unbelegt. Wenn Sie ein größeres Element im Stack platzieren möchten, muss dieser Eintrag mehrere 32-Bit-Einträge umfassen. Zweitens nimmt die Größe eines Stacks auf x86-Systemen (486, Pentium usw.) ab. Dies bedeutet, dass bei der Platzierung eines Elements im Stack der Wert des Stackzeigers abnimmt. Wenn Sie dagegen ein Element entfernen, erhöht sich der Wert. Mit anderen Worten: der Stack verhält sich genau so, wie Sie es nicht erwarten würden.

Warum ist es so wichtig, dass der Systemstack in Bezug auf die Hardware effizient implementiert wird? Sie müssen bedenken, dass ein Stack nicht nur bei jedem Funktionsaufruf verwendet wird, sondern auch dann, wenn Parameter an Funktionen übergeben werden, um Speicherplatz für lokale Variablen zuzuweisen. Stacks werden gelegentlich sogar von Funktionen verwendet, um Ergebnisse an die aufrufende Routine zurückzugeben.

Aufrufen einer einfachen Funktion

Stellen Sie sich einen Funktionsaufruf vor, bei dem die Funktion weder über Parameter noch über lokale Variablen verfügt. Wir definieren drei dieser Funktionen – A , B und C:

```
Function A() As Long
End Function

Function B() As Long
Call A()
End Function

Sub Main()
Call B()
End Sub
```

Was geschieht, wenn Sie dieses Programm ausführen? Zunächst wird das Programm vom Betriebssystem in den Speicher geladen. Anschließend werden dem Code, eventuellen globalen Daten und dem Anwendungsstack Speicherplatz zugewiesen. Dann überträgt das System dem Startpunkt des Programm die Steuerung, worauf über das Programm einige interne Initialisierungsvorgänge durchgeführt werden, und anschließend wird `Sub Main()` aufgerufen.

Das oberste Element im Stack enthält zu diesem Zeitpunkt eine Rücksprungadresse. Was ist eine Rücksprungadresse? Bei jedem Funktionsaufruf springt die CPU zu dieser Funktion, um den zugehörigen Code auszuführen. Wenn Sie die Funktion beenden, benötigt der Prozessor die Information, von welchem Punkt aus der Absprung erfolgte, damit eine Rückkehr an diesen Standort und eine Ausführung genau der Anweisung stattfinden kann, die sich dem Funktionsaufruf unmittelbar anschließt. Die Adresse dieser Anweisung ist die Rücksprungadresse, die im Stack gespeichert wird, sobald eine Funktion aufgerufen wird. Dieses Verfahren wird in Abbildung T4-2a gezeigt. Beim Aufruf von `Sub Main()` verweist die Rücksprungadresse auf einen Standort in der Anwendung, mit dem Initialisierung und Cleanup durchgeführt werden und die Steuerung an das Betriebssystem zurückgegeben wird, wenn das Programm beendet wird.

Abbildung T4-2b zeigt, was passiert, wenn `Function B()` während `Sub Main()` aufgerufen wird. Die Adresse in `Sub Main()` nachdem der Funktionsaufruf im Stack gespeichert wurde, und die Ausführung werden zu Beginn von `Function B()` fortgeführt. Der gleiche Prozess findet statt, wenn `Function A()` von `Function B()` aufgerufen wird, wie dargestellt in Abbildung T4-2c.

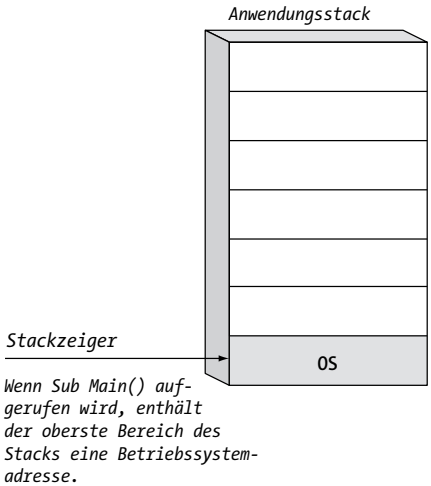


Abbildung T4-2a Der Prozessstack bei der Initialisierung

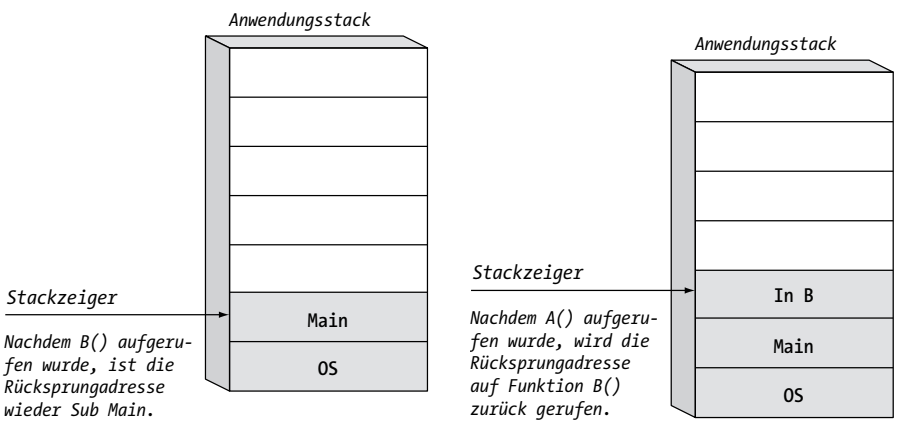
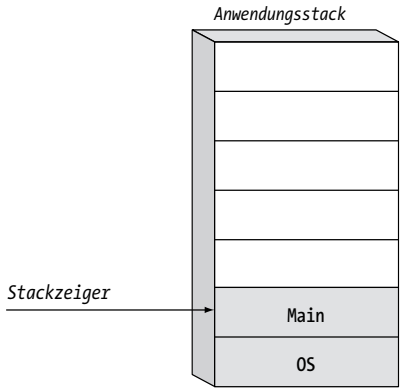


Abbildung P4-2b–c Bei Aufrufen der Funktion werden die Rücksprungadressen im Stack platziert

Sobald die Ausführung von `Function A()` beendet ist, wird der Rückgabewert aus dem obersten Stackelement entfernt, und die Ausführung wird in `Function B()` fortgesetzt, wie in Abbildung T4-2d dargestellt.

Wie Sie sehen können, übernehmen Stacks bei der Programmausführung eine wichtige Funktion. Dies ist grundlegend für die Art und Weise, mit der Subroutinen- und Funktionsausrufe ausgeführt werden. Aber das ist nur der Anfang.



Nachdem wir aus Funktion A() zurück gekehrt sind, befinden wir uns wieder im Kontext von B(), und die Rücksprungadresse lautet wieder Sub Main.

Abbildung T4-2d Bei Beenden einer Funktion wird die Rücksprungadresse aus dem Stack entfernt

Übergeben eines Parameters an eine Funktion

Lassen Sie uns nun untersuchen, was geschieht, wenn eine Funktion über einen Parameter verfügt. In diesem Fall verwenden wir einen `Long`-Parameter, da dieser 32 Bits umfasst und wir deshalb für den Augenblick ignorieren können, was passiert, wenn ein Parameter nicht genau der Größe eines Stacks entspricht.

`Function B()` ist einigen Visual Basic-Programmierern vielleicht nicht sehrgeläufig. Beim ersten Aufruf in diesem Beispiel weist der Parameter L den Wert 1 auf, sodass die Funktion sich selbst erneut aufruft, dieses Mal mit dem Wert 0. Nach dem zweiten Durchlauf wird der Parameterwert 0 ermittelt, was zu einer sofortigen Beendigung der Funktion führt. Wenn eine Funktion sich auf diese Weise selbst aufruft, spricht man von einer rekursiven Funktion. Dieser Begriff wird gelegentlich auch für mathematische Operationen und für bestimmte Datenstrukturtypen verwendet. Der Code für `Function B()` lautet folgendermaßen:

```
Function B(ByVal L As Long) As Long
   Debug.Print L
   If L = 0 Then Exit Function
   B = B(L - 1)
End Function

Sub Main()
   Call B(1)
End Sub
```

Abbildung T4-3a wird der Stack direkt nach dem ersten Aufruf von `Function B()` dargestellt. Wie Sie sehen können, wird der Parameter der Funktion kurz vor der Rücksprungadresse im Stack platziert. Wenn die Funktion auf Variable L verweist, gilt der Verweis dem Wert im Stack. Beim zweiten Aufruf von `Function B()` wird Parameter L erneut im Stack platziert, wie dargestellt in Abbildung T4-3b. Dieses Mal weist der Parameter den Wert 0 auf. Welcher der zwei Stackpositionen enthält den tatsächlichen Parameter L? Die Antwort lautet: beide. Die Parameterinstanz mit dem Wert 1 gehört zum ersten Aufruf von `Function B()`. Die Parameterinstanz mit dem Wert 1 gehört zum zweiten Aufruf von `Function B()`. Bei jedem Aufruf der Funktion verfügt diese über eine eigene lokale Kopie von Parameter L.

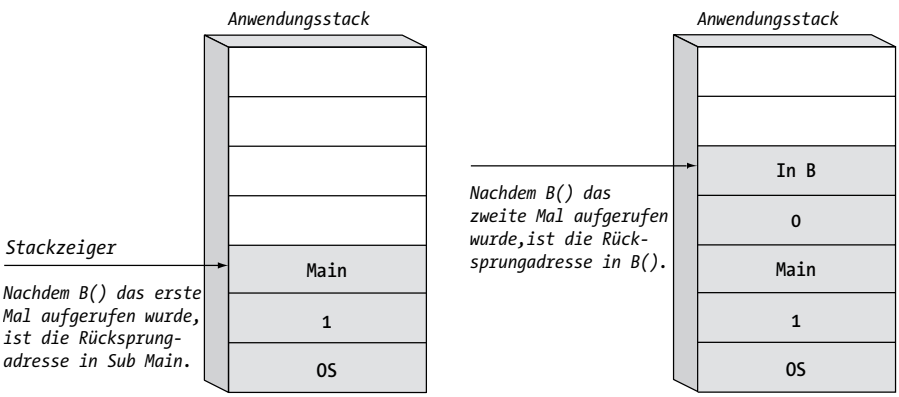


Abbildung T4-3a–b Übergabe eines einzelnen Parameters an eine Funktion

Verfügt eine Funktion über mehrere Parameter, werden diese in ihrer Reihenfolge im Stack platziert (hierzu später mehr). Obwohl der Prozessor nur Werte hinzufügen und entfernen kann, die sich oben im Stack befinden, können Werte an beliebigen anderen Stellen im Stack gelesen und geschrieben werden. Dies ist nicht erstaunlich, da es sich ja tatsächlich bei einem Stack nur um einen Speicherpuffer handelt.

Rufen wir uns einiges des zuvor Gesagten ins Gedächtnis zurück. Sie haben gesehen, dass ein Stack bei Funktionsaufrufen zur Speicherung von Rücksprungadressen unbedingt benötigt wird. Jetzt haben wir herausgefunden, dass ein Stack auch dazu verwendet wird, Parameter an Funktionen zu übergeben. Hierbei verfügt jede Funktion über ihre eigene Parameterkopie. Es stellt sich folgende Frage: Wenn ein Stack einen eindeutigen Satz Parameterwerte für jeden Funktionsaufruf speichern kann, kann der Stack auch andere Variablen speichern, die für jeden Funktionsaufruf eindeutig sind?

Die Antwort lautet Ja, und genau auf diese Weise werden lokale Variablen behandelt.

Lokale Variablen

Betrachten wir erneut `Function B()`. Wir verwenden den gleichen Code wie im vorangegangenen Beispiel, mit der Ausnahme, dass dieses Mal eine lokale Variable zur Speicherung eines Zwischenwertes verwendet wird. Dieser Zwischenwert ist auf den gleichen Wert gesetzt wie Parameter L.

```
Function B(ByVal L As Long) As Long
   Debug.Print L
   Dim LocalVar As Long
   If L = 0 Then Exit Function
   LocalVar = L
   B = B(LocalVar - 1)
End Function

Sub Main()
   Call B(1)
End Sub
```

In Abbildung T4-4a ist der Stack so abgebildet, wie er sich direkt nach dem ersten Aufruf von `Function B()` darstellt. Die Parameter werden als erste im Stack platziert, es folgt die Rücksprungadresse und ein Wert namens `BP`. Als Letztes wird die lokale Variable platziert. Dies wirft eine neue Frage auf: Worum handelt es sich bei dem Wert `BP`? BP steht für »Base Pointer« (Basiszeiger). Der Basiszeiger ist ein Register in der CPU, mit dem eine Funktion die Speicherorte der Parameter und lokalen Variablen im Stack verfolgen kann. Wenn die Ausführung einer Funktion beginnt, platziert die Funktion den vorhandenen Wert des `BP`-Registers im Stack, damit das Register bei Beenden der Funktion wiederhergestellt werden kann. Anschließend wird durch die Funktion das `BP`-Register mit dem aktuellen Wert des Stackzeigers geladen, und es wird eine lokale Variable im Stack platziert. Die Funktion kann jetzt auf die lokalen Variablen als Offsets im `BP`-Register verweisen. Im vorliegenden Beispiel kann der Standort der Variablen `LocalVar` durch Substraktion des Wertes 4 vom Inhalt des `BP`-Registers ermittelt werden. Lokale Variablen befinden sich im Stack über dem `BP`-Register, Parameter befinden sich unterhalb des Registers.

In Abbildung T4-4b wird der Stackstatus nach dem zweiten Aufruf von `Function B()` dargestellt. Wie Sie sehen können, wird bei jedem Funktionsaufruf eine eigene private Kopie der lokalen Variablen `LocalVar` erstellt. Bei jedem Funktionsaufruf kann auf die zugehörigen Parameter, die lokalen Variablen und die Rücksprungadresse zugegriffen werden. Das während eines bestimmten Funktionsaufrufs verwendete Stacksegment wird als Stackframe bezeichnet.

Sie fragen sich vielleicht, warum der BP-Wert nur dann im Stack platziert wird, wenn lokale Variablen verwendet werden. Die Antwort ist einfach – ich habe gelogen. Der BP-Wert wird **immer** im Stack platziert – es sei denn, der Compiler erkennt, dass dieser Wert nicht benötigt wird und überspringt diesen Schritt. Wie Sie sehen, soll dieser Abschnitt Sie nicht in die Feinheiten der Erstellung und Verwendung von Stackframes einführen. Als Visual Basic-Programmierer benötigen Sie dieses Wissen nicht. Sie sollten sich nur die folgenden Dinge merken:

Parameter werden an Funktionen übergeben, indem die Parameter vor dem Funktionsaufruf im Stack platziert werden.

Lokale Variablen werden erstellt, indem die Variablen nach dem Funktionsaufruf im Stack platziert werden.

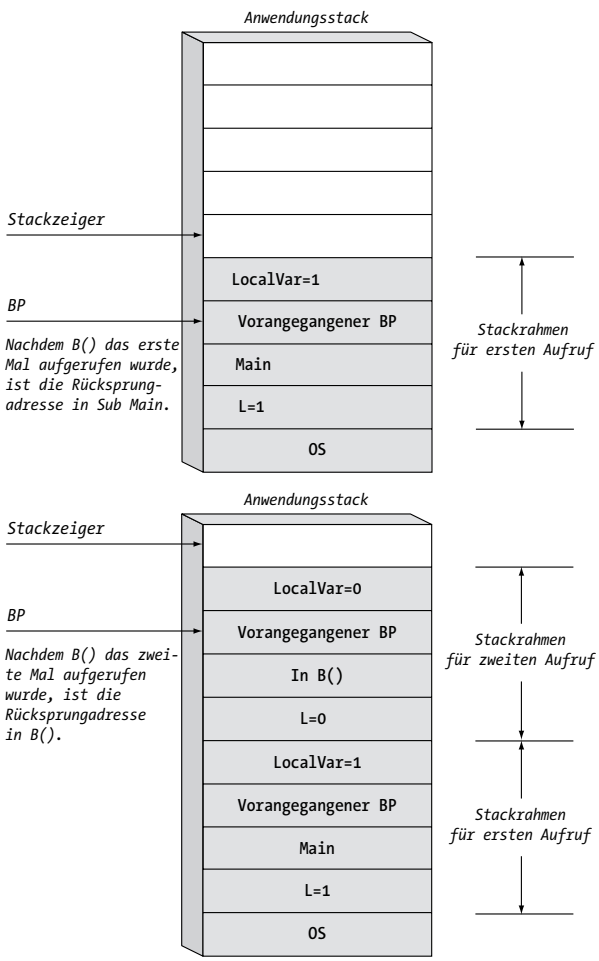


Abbildung T4-4 Über das BP-Register wird der Standort einer lokalen Variablen verfolgt

Wenn Sie ein C++-Purist sind, werden Sie mit meiner Erläuterung sicher sehr unzufrieden sein, denn selbst nach dem Hinzufügen des BP-Registers in die Gleichung (eine Tatsache, die Sie beruhigt vergessen können) handelt es sich bei der Erläuterung der Stackframes doch immer noch um eine Vereinfachung. Wir müssen etwas mehr ins Detail gehen. Die Belohnung: ein wirkliches Verständnis der Fehlermeldung »Falsche DLL-Aufrufkonvention«.

Das Innenleben der Stackframes

Lassen Sie uns zusammentragen, was wir über den Funktionsaufruf in DLLs wissen. Eine Anwendung führt die folgenden Schritte aus, um eine Funktion aufzurufen:

1. Platzieren von Parametern im Stack
2. Platzieren der Rücksprungadresse im Stack
3. Zuordnen von Speicherplatz im Stack für lokale Variablen

Diese Schritte werden umgekehrt ausgeführt, wenn die aufgerufene Funktion zurückgegeben wird.

Hier stellt sich eine interessante Frage. Welche Funktion ist für jeden dieser Schritte verantwortlich, die aufrufende Funktion oder die aufgerufene Funktion?

Die Parameter werden logischerweise von der aufrufenden Funktion im Stack platziert. Der Verwendung von Funktionsparametern liegt die Idee zugrunde, einer aufrufenden Funktion die Übergabe von Informationen an die aufgerufene Funktion zu ermöglichen.

Welcher Parameter wird jedoch zuerst im Stack platziert, wenn mehrere Parameter verwendet werden? Werden Parameter von rechts nach links oder von links nach rechts im Stack platziert?

Es ist auch offensichtlich, dass die aufrufende Funktion die Rücksprungadresse im Stack platzieren muss. Tatsächlich wird diese Aufgabe bei der Steuerungsübergabe an die aufgerufene Funktion von der CPU selbst durchgeführt.

Die aufgerufene Funktion kennt die Erfordernisse im Hinblick auf die lokalen Variablen, daher muss die Funktion für diese Variablen Speicherplatz im Stack zuweisen.

Auf der Rückgabeseite entstehen die gleichen Erfordernisse. Die aufgerufene Funktion muss die lokalen Variablen aus dem Stack entfernen, da dieser Funktion bekannt ist, wieviel Speicherplatz zugewiesen wurde. Die Parameter können jedoch von beiden Funktionen aus dem Stack entfernt werden, da ja sowohl die aufrufende als auch die aufgerufene Funktion die Funktionsparameter kennen.

Es ergeben sich demnach zwei Fragen:

- In welcher Reihenfolge werden die Parameter im Stack platziert?
- Welche Funktion entfernt die Parameter aus dem Stack?

In gewisser Weise ist dies irrelevant – beide Ansätze sind gültig. Es ist jedoch absolut unabdingbar, dass die aufrufende und die aufgerufene Funktion miteinander kompatibel sind. Werden von einer Funktion die Funktionsparameter von links nach rechts und durch die andere Funktion von rechts nach links platziert, sehen die Funktionen die Parameter in umgekehrter Reihenfolge, wodurch mit Sicherheit ein Fehler ausgelöst wird. Versuchen beide Funktionen, die Parameter aus dem Stack zu entfernen, wird der Stack beschädigt, und es kommt wahrscheinlich zu einem Anwendungsabsturz, wenn die aufrufende Funktion versucht, zur nächsthöheren Ebene zurückzukehren.

Der Standardsatz, mit dem das exakte Verfahren zum Aufrufen einer Funktion beschrieben wird, wird als »Calling Convention« (Aufrufkonvention) bezeichnet. Die Aufrufkonvention, die in Visual Basic beim Aufruf einer DLL- oder Win32-API-Funktion eingesetzt wird, ist die `stdcall`-Aufrufkonvention (für Standardaufruf). Wenn Ihre DLL diese Aufrufkonvention nicht verwendet, kann sie nicht von Visual Basic aus aufgerufen werden.[1] Glücklicherweise werden jedoch fast alle DLL-Funktionen mit dieser Aufrufkonvention exportiert.

Für die `stdcall`-Aufrufkonvention gelten die folgenden Regeln:

- Parameter werden im Stack von rechts nach links platziert.
- Die aufgerufene Funktion entfernt die Parameter aus dem Stack.

Warum werden Parameter im Stack von rechts nach links platziert? Sehen Sie sich die folgende Funktion an:

```
Function MyFunction(Byval A as Long, ByVal B As Long) As Long
```

Sehen Sie sich jetzt Abbildung T4-5 an, in der der Stackframe der Funktion gezeigt wird, wenn diese mit der nachstehenden Zeile aufgerufen wird:

```
Call MyFunction ( 2, 1)
```

1. Oder? Beim Bearbeiten der Puzzle könnten Sie im Hinblick auf dieses Thema auf die eine oder andere Überraschung stoßen.

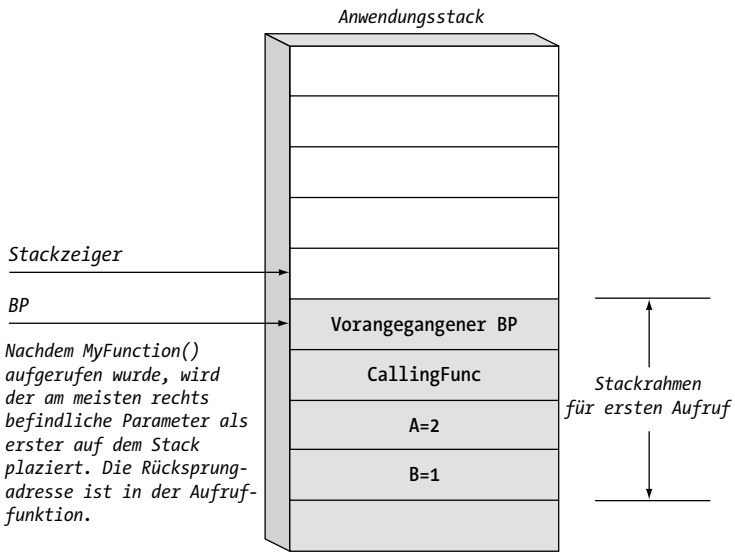


Abbildung T4-5 Stackframe für eine Funktion mit zwei Parametern

Der äußerste linke Parameter wird als letzer im Stack platziert, wobei dieser Parameter in unmittelbarer Nähe des `BP`-Registers im Stackframe dieser Funktion platziert wird. Wieso spielt dies eine Rolle? Tatsächlich spielt dies keine Rolle. Bei einigen Aufrufkonventionen vereinfacht sich die Übergabe der Parameter von rechts nach links mit einer variierenden Anzahl von Parametern verfügen; dies trifft jedoch in diesem Fall nicht zu. Bei der `stdcall`-Aufrufkonvention wird ein anderer Ansatz verfolgt. Dieser umfasst die Übergabe eines Zeigers an ein Array. Als Visual Basic-Programmierer brauchen Sie sich demnach keine Gedanken über die Reihenfolge der Parameterübergabe zu machen.

Wenn etwas schiefgeht

Platziert die aufrufende Funktion Parameter im Stack, und die aufgerufene Funktion entfernt diese aus dem Stack, müssen die beiden Funktionen über identische Informationen bezüglich der Anzahl und des Typs der Parameter verfügen. Die `Declare`-Anweisung stellt einer Visual Basic-Anwendung diese Informationen bereit. Ist Ihre Deklaration falsch, treten Probleme auf.

Stellen Sie sich eine DLL-Funktion mit dem Namen `MyDLLFunction` vor, die zwei `Long`-Parameter aufweist. Die richtige Deklaration für diese Funktion würde wie folgt lauten:

```
Private Declare Function MyDLLFunction Lib "mylibrary.dll" (ByVal A As _
Long, ByVal B As Long) As Long
```

Was geschieht, wenn Sie den zweiten Parameter weglassen und die Funktion folgendermaßen deklarieren?

```
Private Declare Function MyDLLFunction Lib "mylibrary.dll" _
(ByVal A As Long) As Long
```

Abbildung T4-6a zeigt zwei Stackframes. Auf der linken Seite befindet sich der Stackframe, wie er ursprünglich aufgerufen wurde (mit nur einem Parameter). Rechts wird der Stackframe gezeigt, der eigentlich von der DLL-Funktion erwartet wurde. Was geschieht, wenn die DLL-Funktion Parameter B verwendet? Die Funktion versucht, auf den Standort zuzugreifen, der erwartungsgemäß den Parameter enthalten sollte, die aufrufende Funktion hat jedoch keinen Wert im Stack platziert. Als Ergebnis ermittelt die DLL-Funktion einen unbestimmten Wert. Bei diesem könnte es sich um eine lokale Variable aus einem vorangegangenen Stackframe handeln. Oder es handelt sich um einen anderen Wert, der im Stack gespeichert wurde. Es handelt sich jedoch mit Sicherheit nicht um das, was von der Funktion erwartet wird.

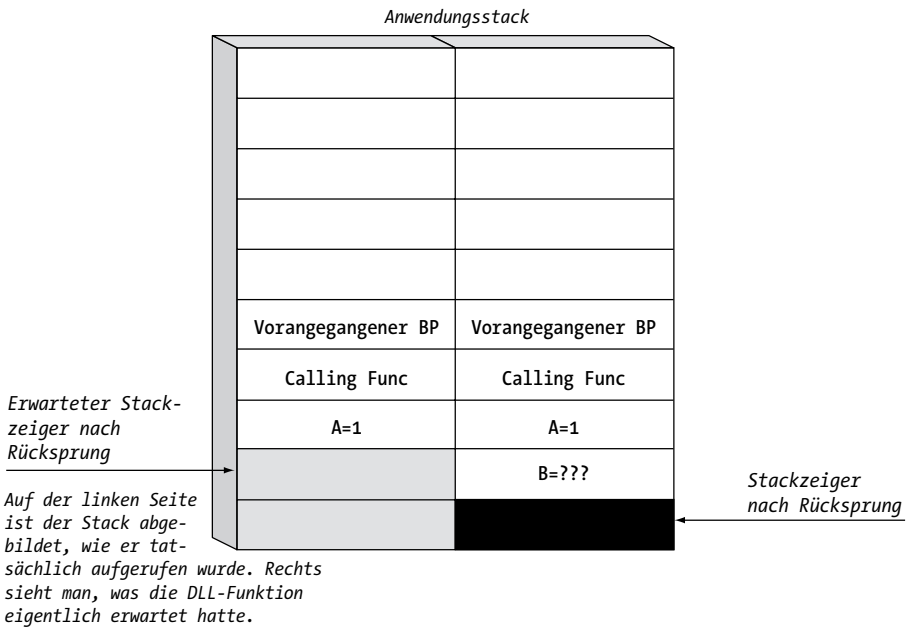
Ein schwerwiegenderes Problem wird in Abbildung T4-6b dargestellt. Hier wird gezeigt, was geschieht, wenn die Funktion zurückgegeben wird. Die DLL-Funktion ist dafür verantwortlich, den Parameter aus dem Stack zu entfernen. Da angenommen wird, dass sich beide Parameter im Stack befinden, werden auch beide aus dem Stack entfernt. Dies bedeutet jedoch, dass der Stackzeiger nicht mit dem Wert wiederhergestellt wird, den dieser vor dem Funktionsaufruf hatte. Wenn es sich bei diesem Beispiel um ein C-Programm handeln würde, träte zu diesem Zeitpunkt mit an Sicherheit grenzender Wahrscheinlichkeit ein Speicherausnahmefehler auf.

Visual Basic verfügt jedoch über einen stärkeren Schutzmechanismus. Vor dem Aufruf einer DLL-Funktion wird der Wert des Stackzeigers gespeichert. Sobald die DLL-Funktion zurückgegeben wird, überprüft Visual Basic, ob der ursprüngliche Wert des Stackzeigers wiederhergestellt wurde. Ist dies nicht der Fall, wird der folgende Fehler ausgegeben:

Laufzeitfehler 49
Falsche DLL-Aufrufkonvention.

Wir sind also wieder dort, wo wir zu Beginn dieses Tutoriums waren.

Diese Fehlermeldung bedeutet immer, dass in Ihrer `Declare`-Anweisung ein Fehler steckt, sie wird durch eine Nichtübereinstimmung der Stackzeiger bei der Rückgabe einer DLL- oder API-Funktion ausgelöst.



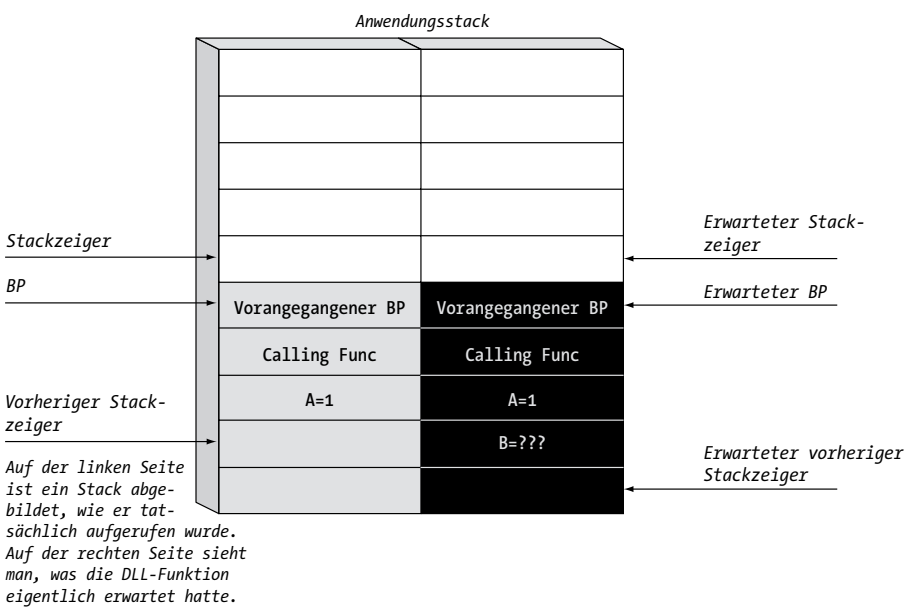


Abbildung T4-6 Veranschaulichung des Fehlers »Falsche DLL-Aufrufkonvention«

Was kann diese Nichtübereinstimmung hervorrufen?

- Falsche Parameteranzahl
- Falsche Größe des Rückgabewertes einer Funktion
- Falsche Parametergröße

Funktionen der `stdcall`-Aufrufkonvention werden fast immer mit Hilfe der CPU-Register zurückgegeben und wirken sich daher nicht auf den Stack aus. Ausnahmen davon bilden Datentypen, die von der Größe her nicht in ein Register passen (beispielsweise Datentyp `Variant` und benutzerdefinierte Strukturen). Zur Rückgabe dieser Datentypen wird durch die aufrufende Funktion zunächst innerhalb eines eigenen Stackframes für den Rückgabewert Speicherplatz im Stack zugewiesen (so als würde Speicherplatz einer lokalen Variablen zugeordnet). Anschließend wird ein Zeiger auf diesen Speicherplatz als Zusatzparameter an die DLL-Funktion übergeben. Die DLL-Funktion lädt den Speicherbereich mit dem Rückgabewert, anschließend wird der Zeigerwert zurückgegeben, der als Zusatzparameter übergeben wurde. Weitere Informationen zur Rückgabe von Werten durch DLL-Funktionen finden Sie in Lösung 29, »Was tun, wenn's richtig weh tut?« (in Teil II dieses Buches).

Der Stack unter 32-Bit-Betriebssystemen hat stets eine Größe von32 Bits. Die Parameter, die einen Datentyp mit mehr als 32 Bits verwenden, belegen mehr als einen Eintrag im Stack. In Tabelle 4.1 werden die einzelnen Parametergrößen bei deren Übergabe in den Stack aufgeführt.

Parameterdatentyp	Größe im Stack
Numerischer Datentyp, übergeben als `ByVal`	32 Bits
`(Byte, Integer, Long)`	
Alle Datentypen, sofern übergeben als	32 Bits
Verweis `(ByRef)`	
Benutzerdefinierte Typen	32 Bits (Zeiger auf Speicherbereich im aufrufenden Stack)
Double	64 Bits (2 Stackeinträge)
Currency	64 Bits (2 Stackeinträge)
Variant	128 Bits (4 Stackeinträge)

Tabelle T4-1 Parametergrößen bei deren Übergabe in den Stack

Weitere Informationen zur Übergabe von Parametern als Wert oder als Verweis finden Sie in Tutorium 5, »Das `ByVal`-Schlüsselwort: Die Lösung für 90% aller API-Probleme«.

Tutorium 5

Das »ByVal«-Schlüsselwort: Die Lösung für 90% aller API-Probleme

The Midnight Ride of Bugs I-Fear

Listen, my coders, and you shall hear
Of the midnight ride of Bugs I-Fear.
Was late in a version of VB5:
Hardly a hacker is still alive
Who remembers that famous day and year.

He said to his friend, »If the program's harsh
And crashes and freezes and is a fright,
Hang a Jolt aloft in the office arch
Near the cubicle where you sleep at night –
One if ByRef and two if ByVal;
And I, though far off, will be a pal,
With ready RAS, to the Net I'll be bound,
Through every Usenet board around,
To the other hackers, their keyboards to pound.«

So through the night hacked Bugs I-Fear;
And so through the Net went his cry of alarm,
To every Usenet board he found –
A cry of frustration and not of fear –
A small code fragment, with fingers so sore,
And a word that shall echo forevermore!
For, borne on the night-wind of the Net
Through all VB knowledge, you can bet,
In the hour of darkness and peril and care,
The coders awakened to listen and hear,
»Remember the ByVal in your Declare!«
was the midnight message of Bugs I-Fear.

Nachdem ich mich nun über Jahre hinweg mit allen möglichen API-Problemen befasst habe, ist mir eines klar geworden. Nahezu jeder Fehler, den ein Visual Basic-Programmierer beim Aufrufen einer API-Funktion begeht, hat etwas mit dem `ByVal`-Schlüsselwort zu tun. Entweder wird es verwendet, obwohl es nicht verwendet werden sollte, oder aber es fehlt.

Es ist äußerst wichtig, die exakte Bedeutung dieses Schlüsselwortes und dessen Verwendungsmöglicheiten zu kennen, um in der Lage zu sein, Visual Basic-Deklarationen für API- und DLL-Funktionen zu erstellen.

Wozu dient das Schlüsselwort »ByVal«? Die einfache Erklärung

Kurz gesagt wird Visual Basic über das Schlüsselwort `ByVal` mitgeteilt, dass ein Parameter als Wert übergeben werden soll, d.h., dass der aktuelle Wert der Variablen an die Funktion übergeben wird. Fehlt das `ByVal`-Schlüsselwort, wird der Parameter so übergeben, als sei das `ByRef`-Schlüsselwort verwendet worden, das den Paramterinhalt zu einem Verweis macht. Das Projekt **ByValSp.vbp** (auf der Begleit-CD-ROM zu diesem Buch) veranschaulicht die Unterschiede zwischen diesen beiden Übergabetypen für einen Parameter.

```
' ByValSimple Beispiel
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten

Option Explicit

Private Sub Command1_Click()
   Dim myvar As Long
   myvar = 5
   Call CalledByVal(myvar)
   Debug.Print "After call ByVal: " & myvar
   Call CalledByRef(myvar)
   Debug.Print "After call ByRef: " & myvar
End Sub

Private Sub CalledByRef(x As Long)
   x = x + 1
End Sub

Private Sub CalledByVal(ByVal x As Long)
   x = x + 1
End Sub
```

Das Ergebnis sieht folgendermaßen aus:

```
Nach Aufruf von ByVal: 5
Nach Aufruf von ByVal: 6
```

Wird ein Parameter als Verweis übergeben, erhält die aufrufende Funktion einen Verweis auf die Variable selbst, daher wirken sich Änderungen am Parameter in diesem Fall auf den Wert der ursprünglichen Variablen aus.

Aber was bedeutet »Verweis auf die Variable selbst« genau? Es bedeutet, dass die aufgerufene Funktion in der Lage ist, die Inhalte des ursprünglichen Parameters zu ändern. Wie kann eine solche Änderung vorgenommen werden?

Eine detaillierte Beschreibung der Vorgänge bei einem Funktionsaufruf finden Sie in Tutorium 4, »Die Arbeitsweise einer DLL: In einem Stackframe«. Im vorliegenden Tutorium soll beschrieben werden, wie jeder Parameter, der an eine Funktion übergeben wird, in einem separaten Speicherblock platziert wird, den man als Stack bezeichnet. Sie müssen um die Auswirkungen des Schlüsselwortes `ByVal` auf die jeweilige Platzierung der Parameter im Stack wissen, um entscheiden zu können, wann sie das `ByVal`-Schlüsselwort einsetzen sollten.

Die Funktionsweise von »ByVal« – Numerische Variablen

Jede Variable weist zwei Eigenschaften auf. Eine Variable verfügt über einen Wert, nämlich die in der Variable »enthaltenen« Daten. Darüber hinaus besitzt jede Variable einen Standort, also die Stelle im Speicher, an dem die Daten gespeichert werden.

Abbildung T5-1 zeigt, wie eine `Long`-Variable `ByVal` an eine Funktion übergeben wird.

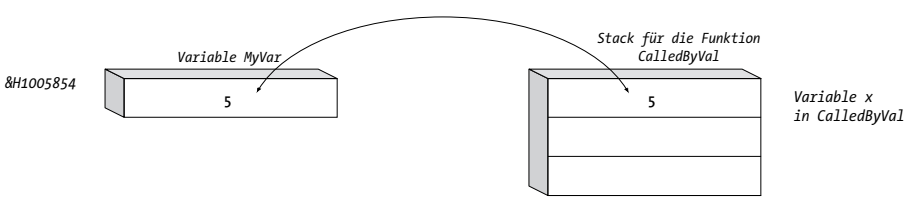


Abbildung T5-1 Übergabe einer numerischen Variable als Wert

In diesem Beispiel lautet der Wert der Variablen 5 und trägt im Speicher die Adresse `&H1005654`. Da `ByVal` verwendet wurde, wird der Wert der Variablen als Parameter mit dem Namen `x` im Stack platziert. Wenn Sie in der `CalledBy Val`-Funktion auf `x` verweisen, verweisen Sie tatsächlich auf den Wert im Stack, daher wirken sich Änderungen des Wertes nicht auf die ursprüngliche Variable `myvar` aus.

Abbildung T5-2 zeigt, wie eine `Long`-Variable `ByRef` an eine Funktion übergeben wird.

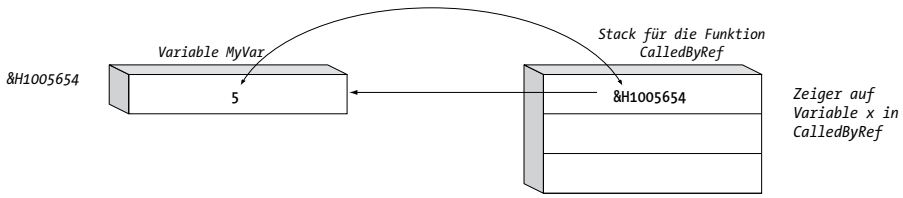


Abbildung T5-2 Übergabe einer numerischen Variablen als Verweis

Im vorangegangenen Beispiel blieb die Variable `myvar` unverändert. Wenn Sie die Variable jedoch als Verweis übergeben, wird nicht der Variablenwert, sondern der Variablenstandort (oder die Speicheradresse) im Stack platziert. Verweist die Funktion `CalledBy Ref` auf `x`, wird der Wert weiterhin aus dem Stack abgerufen, anstatt jedoch den gefundenen Wert direkt zu verwenden, wird dieser eingesetzt, um den Standort von `x` zu ermitteln. Der Wert im Stack dient also als Zeiger auf die Variable `x`. Wenn die `CalledBy Ref`-Funktion versucht, den Wert von x zu lesen oder zu ändern, wird der Zeiger dazu verwendet, den Standort des `myvar`-Parameters zu ermitteln und auf diesen zuzugreifen. Daher wirken sich Änderungen von x in diesem Fall auf die Variable `myvar` aus. Wie ermittelt jedoch die Funktion `CalledBy Ref`, dass im Stack statt eines Wertes ein Zeiger enthalten ist? Diese Tatsache ist bekannt, da Sie mit der Funktionsdeklaration die Art der Verwendung des Wertes im Stack definiert haben.

Wie ermitteln DLL- oder API-Funktionen, dass im Stack statt eines Wertes ein Zeiger enthalten ist? Auf die gleiche Weise: Die Person, die die DLL- oder API-Funktion schreibt, gibt den Parametertyp im Quellcode der Funktion an. In Visual Basic kann nicht automatisch ermittelt werden, welche Elemente für diese Funktion im Stack platziert werden müssen. Daher müssen Sie in der `Declare`-Anweisung genau festlegen, wie der Parameter an die Funktion übergeben werden soll. Begehen Sie hierbei einen Fehler, tritt nicht nur ein Fehler beim Funktionsaufruf auf, sondern es kommt wahrscheinlich auch zu einem Speicherausnahmefehler. Angenommen, Sie hätten eine API-Funktion, die einen `Long`-Wert erwartet, der als Verweis übergeben wird, Sie deklarieren diesen jedoch als Wert. Enthält die Variable den Wert 0, platziert Visual Basic den Wert 0 im Stack (aufgerufen als Wert). Die API-Funktion ruft im Stack den Variablenstandort ab und ermittelt, dass der Standort 0 lautet. Wenn die Funktion versucht, Lese- oder Schreibvorgänge für die Variable selbst auszuführen (mit der Annahme, die Variable befände sich am Standort 0) kommt es zu einem Speicherausnahmefehler.

Jede 32-Bit-Variable, die statt eines Wertes den Standort einer anderen Variablen enthält, wird als Zeiger bezeichnet, und Sie werden bald feststellen, dass diese Zeiger für API-Aufrufe häufig eingesetzt werden.

Wann wird »ByVal« verwendet?

Das ByVal-Schlüsselwort kann entweder in einer Funktionsdeklaration oder in einem Funktionsaufruf eingesetzt werden.

Sehen Sie sich die folgenden zwei Codefragmente an:

```
Declare Function SendMessage Lib "User32" Alias "SendMessageA" _
(ByVal Hwnd As Long, ByVal Msg As Long, ByVal wParam As Long, _
ByVal lParam As Long) As Long

Dim myvar As Long
Call SendMessage(hwnd, msg, wparam, myvar)
```

Nun betrachten Sie den lParam-Parameter. Die Funktion SendMessage kann auf vielerlei Weise verwendet werden, und der lParam-Parameter kann auf eine Vielzahl verschiedener Datentypen verweisen. In dieser Deklaration handelt es sich bei dem lParam-Parameter stets um einen Long-Wert. In diesem Beispiel ist die myvar-Variable als AsLong deklariert, dies spielt jedoch keine Rolle, Sie können die Variable auch als AsVariant oder AsInteger oder sogar als AsString deklarieren. Visual Basic würde den Wert im Rahmen der SendMessage-Funktion in einen numerischen Long-Wert konvertieren. Die Deklaration gibt den Parametertyp und die Art des Aufrufs an, daher kann Visual Basic den benötigten Datentyp erzwingen. In der folgenden Deklaration wurde der lParam-Parameter als As Any festgelegt.

```
Declare Function SendMessage Lib "User32" Alias "SendMessageA" _
(ByVal Hwnd As Long, ByVal Msg As Long, ByVal wParam As Long, _
lParam As Any) As Long

Dim myvar As Long
Call SendMessage(hwnd, msg, wparam, ByVal myvar)
```

Mit As Any wird Visual Basic mitgeteilt, dass für den zu übergebenden Parameter keine Typprüfung durchgeführt werden soll. Daher richtet sich der an die SendMessage-Funktion übergebene Datentyp nach dem Typ der Variablen, die im Aufruf verwendet wird. In diesem Fall ist es entscheidend, dass die Variable als AsLong deklariert wird (wenn dies der Typ ist, den Sie übergeben möchten), da Visual Basic weder eine Fehlerprüfung noch eine Konvertierung vornimmt. Darüber hinaus müssen Sie während des Aufrufs das Schlüsselwort ByVal verwenden, wenn beim Aufruf die Variable als Wert interpretiert werden soll.

Der Parametertyp As Any birgt viele Gefahren. Sie sollten die Verwendung dieses Parametertyps bei der Deklaration von Funktionen daher möglichst vermeiden. Als besserer Ansatz stellt in diesem Fall die Erstellung mehrerer Aliasse für eine

Funktion heraus, die jeweils für einen speziellen Parametertyp bestimmt sind. Die gebräuchlichsten Datentypen für den `lParam`-Parameter in einer `SendMessage`-Funktion sind beispielsweise die Typen `Long` und `String` dar. In der Datei **api32.txt** auf der Begleit-CD-ROM zu diesem Buch finden Sie die folgenden zwei Deklarationen:

```
Declare Function SendMessageBynum Lib "user32" Alias "SendMessageA" _
(ByVal hwnd As Long, ByVal wMsg As Long, ByVal wParam As Long, _
ByVal lParam As Long) As Long
Declare Function SendMessageByString Lib "user32" Alias "SendMessageA" _
(ByVal hwnd As Long, ByVal wMsg As Long, ByVal wParam As Long, _
ByVal lParam As String) As Long
```

Diese Funktionen bieten eine sicherere Methode für den Aufruf der `SendMessage`-Funktion und reduzieren die Wahrscheinlichkeit von Bugs aufgrund eines falschen Parametertyps. Durch das Verwenden von `As Any` in einer Deklaration liegt die Verantwortung für die Übergabe des richtigen Parametertyps alleine bei der Person, die die Funktion verwendet und nicht bei der Person, die die Deklaration erstellt. Da die Funktion sehr wahrscheinlich mehrmals aufgerufen wird, ist es offensichtlich, dass der `As Any`-Ansatz mit Sicherheit eher zu Fehlern führt als es bei der vorherigen Erstellung einer typensicheren Deklaration der Fall wäre.

Jetzt, da Sie wissen, dass die `As Any`-Deklaration böse ist und unter allen Umständen vermieden werden sollte, werden Sie überrascht sein, dass ich genau diese Deklaration im vorliegenden Buch sehr häufig verwende. Der Grund ist einfach: Mit diesem Buch sollen Sie lernen, wie Sie selbst die komplexesten Win32-API-Funktionen deklarieren, selbst wenn Ihnen hierzu nur eine C-Deklaration zur Verfügung steht. Der Parametertyp `As Any` zwingt Sie dazu, bei jedem Funktionsaufruf genau über den Parameter nachzudenken, was zu einem besseren Verständnis darüber führt, wie die Funktion aufgerufen wird.

Zudem werden Sie sehen, dass es bei einigen Funktionen wünschenswert ist, bei jedem Funktionsaufruf erneut zu überdenken, welcher Parametertyp eingesetzt wird ,bzw., ob das `ByVal`-Schlüsselwort verwendet werden soll. Als Beispiel sei hier die `RtlMoveMemory`-Funktion angeführt.

Arbeiten mit Zeigern

Eine der nützlichsten Funktionen zum Verständnis der Parameterübergabe ist die `RtlMoveMemory`-Funktion. Diese Funktion kopiert Speicherstellen von einem Quellort in einen Zielort. Was bedeutet das?

Nachfolgend eine typische RtlMoveMemory-Deklaration:

```
Private Declare Function RtlMoveMemory Lib "kernel32" (dest As Any, _
src As Any, ByVal Count As Long) As Long
```

Diese Funktion kann drei Parameter verwenden. Der dest-Parameter gibt das Ziel für die kopierten Speicherstellen an. Der src-Parameter bezeichnet den Quellstandort der zu kopierenden Daten. Der Count-Parameter legt die Anzahl der zu kopierenden Bytes fest.

Abbildung T5-3 zeigt den Inhalt des Stacks beim Aufruf der RtlMoveMemory-Funktion. Der Count-Parameter wird als Wert übergeben, daher enthält der entsprechende Stackeintrag die Anzahl der zu übertragenden Bytes. Die zwei weiteren Parameter sind in diesem Fall As Any deklariert, d.h., Sie können beliebige Werte an die Funktion übergeben. Da in Visual Basic keine Beschränkungen hinsichtlich des übergebenen Datentyps gelten, ist es wichtig, dass Sie genau verstehen, welche Werte im Stack benötigt werden.

Im vorliegenden Fall müssen beide Stackeinträge, sowohl der für den src- als auch der für den dest-Parameter, eine Speicheradresse enthalten. Mit anderen Worten, bei diesen Einträgen muss es sich um Zeiger handeln.

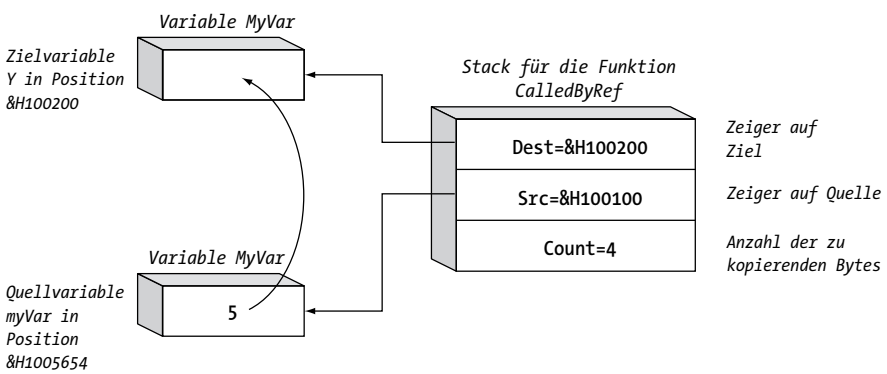


Abbildung T5-3 Verwendung der RtlMoveMemory-Funktion

Daher müssen die Parameter in diesem Fall offensichtlich gesetzt werden, indem die Werte als Verweise übergeben werden. Der Beispielcode in der Datei **Pointers.vbp** auf der Begleit-CD-ROM zu diesem Buch veranschaulicht dies:

```
Dim myVar As Long

Private Sub Form_Load()
   myVar = 5
End Sub
```

```
Private Sub Command1_Click()
   Dim Y As Long
   Call RtlMoveMemory(Y, myVar, 4)
   Debug.Print "New value of Y: " & Y
End Sub
```

In diesem Beispiel definiert die Funktion eine Variable `Y`, die mit den Daten der `myVar`-Variable geladen wird (die im `Form_Load`-Ereignis auf den Wert 5 initialisiert wurde). Sowohl `Y` als auch `myVar` werden als Verweis übergeben, d.h., der Stack enthält die Adresse der zwei Variablen. Wie aus der Abbildung hervorgeht, kann die `RtlMoveMemory`-Funktion die Zeiger dazu verwenden, die Daten einer Variable in die andere zu kopieren. Beachten Sie, dass die in der Abbildung verwendeten Adressen nur der Veranschaulichung dienen, die tatsächlich angezeigten Adressen weichen von den hier gezeigten ab.

Sie können auch den Visual Basic-Operator `VarPtr` einsetzen, um die Adresse einer Variablen zu ermitteln. Dieses Verfahren wird im folgenden Beispielcode eingesetzt:

```
Private Sub Command2_Click()
   Dim Y As Long
   Dim myVarAddress As Long
   myVarAddress = VarPtr(myVar)
   Call RtlMoveMemory(Y, ByVal myVarAddress, 4)
   Debug.Print "myVar Address is: " & Hex$(myVarAddress)
   Debug.Print "New value of Y: " & Y
End Sub
```

In dieser Funktion wird eine neue `Long`-Variable namens `myVarAddress` deklariert. Diese Variable wird mit Hilfe der `VarPtr`-Funktion zusammen mit der Adresse der `myVar`-Variablen geladen. Wird die `myVarAddress`-Variable an die `RtlMoveMemory`-Funktion übergeben, geschieht dies mit dem `ByVal`-Operator. Warum? Da die `myVarAddress`-Variable die Adresse der `myVar`-Variablen enthält und wir diese Adresse im Stack speichern möchten. Wenn Sie die Inhalte einer Variablen im Stack platzieren möchten, ist die Verwendung des `ByVal`-Operators erforderlich. Dieser Vorgang wird in Abbildung T5-4 veranschaulicht.

Wie sie sehen, entspricht das Endergebnis bei diesem Ansatz exakt dem Ergebnis, das im vorherigen Beispiel erzielt wurde. Wieso sollten Sie also diesen komplizierteren Ansatz wählen?

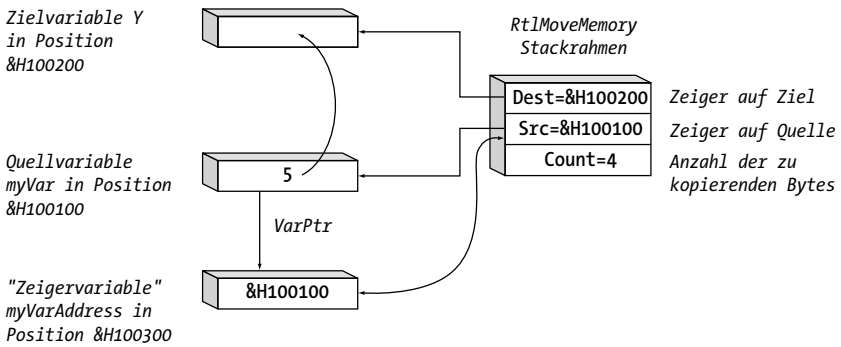


Abbildung T5-4 Verwenden einer Zeigervariablen

In diesem besonderen Fall würden Sie die vorgestellte Methode nicht verwenden. Dennoch ist es auf der anderen Seite unwahrscheinlich, dass Sie mit der `RtlMoveMemory`-Funktion eine Operation ausführen, die mit einer simplen `Y = myVar`-Anweisung durchgeführt werden kann. Es gibt viele Fälle, in denen Sie bei der Arbeit mit API-Funktionen Zeiger und Zeigervariablen einsetzen müssen. Und es werden häufig Situationen auftreten, in denen Sie die `RtlMoveMemory`-Funktion verwenden, um die Daten eines Speicherblock in einen anderen benutzerdefinierten Speicherblock zu kopieren, was den Zugriff auf die Daten in einem Visual Basic-Programm wesentlich vereinfacht.

Die Funktionsweise von »ByVal« (Zeichenfolgenvariablen)

Alles, was Sie bisher zur `ByVal`-Anweisung gelesen haben, gilt für nahezu jeden Visual Basic-Variablentyp. Eine besondere Ausnahme bilden hier die Zeichenfolgenvariablen.

Verabschieden Sie sich von der Vorstellung, dass `ByVal` »als Wert« bedeutet – Visual Basic übergibt an Zeichenfolgenparameter stets Zeiger. Die Frage ist nur, auf was diese Zeiger verweisen?

Um verstehen zu können, was geschieht, wenn Sie eine Zeichenfolge als Parameter an eine API-Funktion übergeben, müssen Sie zunächst wissen, wie die interne Speicherung von Zeichenfolgen in Visual Basic vor sich geht und wie Win32-API-Funktionen mit diesen Zeichenfolgenparametern umgehen.

Visual Basic speichert Zeichenfolgen intern unter Verwendung eines so genannten `BSTR`-Objekts. Ein `BSTR`-Objekt (kurz für **B**asic **STR**ing) wird in einem Speicherbereich gespeichert, der durch das Betriebssystem verwaltet wird, genau gesagt durch das OLE-Subsystem. Diese einfache Anweisung ist äußerst wichtig und hat eine bedeutende Auswirkung darauf, wie mit der Zeichenfolgen an API-Funktionen übergeben werden.

Innerhalb eines BSTR-Objekts

Wenn Sie einer Zeichenfolge Text zuordnen, wissen Sie, dass dieser Text in irgendeiner Form im Speicher gehalten werden muss. Wenn Sie eine Zeichenfolge löschen, können Sie annehmen, dass der von der Zeichenfolge verwendete Speicher vom Betriebssystem zurückgefordert und anderweitig eingesetzt werden kann. Jedes Programm nutzt für Zeichenfolgen und andere Objekte Speicher. Dieser Speicher wird letztlich durch das Betriebssystem zugewiesen. Sie sollten hieraus jedoch nicht schließen, dass ein Programm immer dann, wenn Speicher benötigt wird, diesen beim Betriebssystem anfordert.

Die Speicherverwaltung kann auf vielerlei Weise erfolgen, wobei Methoden, die gut zur Verwaltung von großen Speicherblöcken geeignet sind, sich als wenig effizient bei der Verwaltung kleiner Speicherblöcke erweisen können. Daher ist es bei verschiedenen Subsystemen unter Windows üblich, große Speicherblöcke vom Betriebssystem anzufordern und diese anschließend in kleinere Blöcke zu unterteilen. Frühe Versionen von Visual Basic verwendeten diesen Ansatz zur Verwaltung des Zeichenfolgenspeichers – Visual Basic forderte einen Speicherblock vom Betriebssystem an, anschließend wurde dieser Speicherblock unter Verwendung von internem Code in kleinere Blöcke für Zeichenfolgen unterteilt. Von Version 4 an wurde in Visual Basic das Zeichenfolgenverwaltungssystem in das OLE-Subsystem von Windows integriert. Zeichenfolgen, die durch das OLE-Subsystem zugewiesen werden, bezeichnet man als BSTR-Zeichenfolgen.

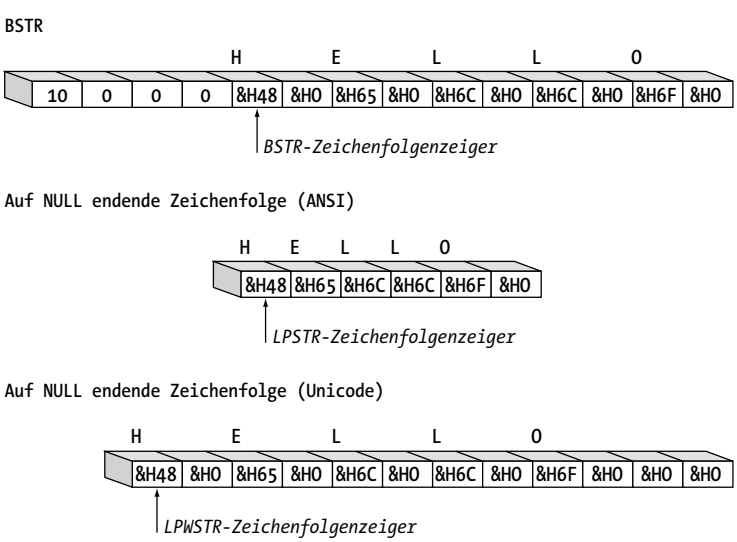


Abbildung T5-5 BSTR- und auf NULL endende Zeichenfolgen

Abbildung T5-5 zeigt eine BSTR-Zeichenfolge im Speicher. Als Erstes wird Ihnen sicherlich auffallen, dass die Daten in der Zeichenfolge im Unicode-Format gespeichert sind. Dies bedeutet, dass jedes Zeichen im Speicher zwei Byte belegt. In diesem besonderen Beispiel lautet für die Zeichenfolge »Hello« das zweite Byte eines jeden Zeichens 0, dennoch benötigen einige Zeichen beide Bytes. BSTR-Zeichenfolgen müssen nicht im Unicode-Format vorliegen, in Visual Basic wird jedoch zur internen Speicherung von Zeichenfolgen das Unicode-Format verwendet, daher können Sie in den meisten Fällen voraussetzen, dass ein BSTR-Objekt eine Unicode-Zeichenfolge enthält. Eine Ausnahme von dieser Regel wird später in diesem Kapitel besprochen.

Als Nächstes wird Ihnen vielleicht auffallen, dass ein BSTR-Objekt die Länge der Zeichenfolge enthält. Die Länge erscheint in den ersten vier Bytes, die den Zeichenfolgendaten vorangestellt sind. Wenn Sie über einen Zeiger auf ein BSTR-Objekt verfügen, verweist der Zeiger stets auf den Beginn des Datenbereiches.

Das Beispielprogramm **ByValSt.vbp** demonstriert dies durch den folgenden Code:

```
Private Sub cmdInside_Click()
   Dim a() As Byte
   Dim x&
   Dim b$
   Dim length As Long
   b$ = "Hello"
   ' Viele VB-Programmierer wissen nicht, dass Sie eine
   ' solche dynamische Zuordnung verwenden können
   a() = b$
   Debug.Print "Inside Hello"
   For x = 0 To UBound(a)
      Debug.Print x, Hex$(a(x))
   Next x
   Call RtlMoveMemory(length, ByVal (StrPtr(b$) - 4), 4)
   Debug.Print "Length: " & length
End Sub
```

Die Zeichenfolge b$ enthält den Text »Hello«. Es liegt demnach ein Array a() vor, dass unter Verwendung der Zuordnung a() = b$ mit dem Inhalt der Zeichenfolge geladen wird. Über diese Zuordnung, die vielen Visual Basic-Programmierern unbekannt ist, können Daten zwischen Zeichenfolgen und dynamischen Bytearrays sehr einfach kopiert werden, ohne dazu einen Konvertierungsprozess einschließen zu müssen. Dem automatischen Array wird automatisch die geeignete Länge zugeordnet. Die Funktion zeigt zunächst den Wert eines jeden Bytes im Hexa-

dezimalformat an. Wie Sie sehen können, entsprechen die Daten den Werten in Abbildung T5-5.

Als Nächstes bedienen wir uns der gleichen Technik, die zuvor zum Kopieren von Daten aus dem Speicher verwendet wurde. Über die `StrPtr`-Funktion erhalten wir in diesem Fall den Zeiger auf das `BSTR`-Objekt (durch `VarPtr` würde ein Zeiger auf die Variable zurückgegeben, die den Zeiger auf das `BSTR`-Objekt enthält). Wir subtrahieren vom Zeigerwert die Zahl 4, um auf die Zeichenfolgenlänge zu verweisen, und kopieren diesen Wert in eine `Long`-Variable mit dem Namen `length`. Das Ergebnis sieht folgendermaßen aus:

```
Inside Hello
 0            48
 1            0
 2            65
 3            0
 4            6C
 5            0
 6            6C
 7            0
 8            6F
 9            0
Länge: 10
```

In einer C-Zeichenfolge

Eine C-Zeichenfolge, auch bezeichnet als eine auf NULL endende Zeichenfolge, wird im unteren Abschnitt von Abbildung T5-5 gezeigt. Dieser Zeichenfolgentyp wird C-Zeichenfolge genannt, da es sich um das in den Programmiersprachen C und C++ verwendete Standardzeichenfolgenformat handelt. Diese Zeichenfolgen werden als auf NULL endend bezeichnet, da das Ende der Zeichenfolge durch das Vorhandensein eines NULL-Zeichens gekennzeichnet wird. Beachten Sie, dass das Ende der Zeichenfolge durch ein NULL-Zeichen, nicht jedoch zwingend durch einen 0-Wert markiert wird. In einer ANSI-Zeichenfolge, die ein Byte pro Zeichen enthält, stellt das NULL-Zeichen ein einzelnes Byte mit dem Wert 0 am Ende der Zeichenfolge dar. Bei einer Unicode-Zeichenfolge umfasst das NULL-Zeichen zwei Byte, genau wie jedes andere Zeichen auch. Beide Bytes des NULL-Zeichens weisen den Wert 0 auf.

Da die Win32-API größtenteils in C geschrieben wurde, verwendet die Mehrzahl der API-Funktionen auf NULL endende Zeichenfolgen. Die einzigen Ausnahmen sind Funktionen, die Teil des OLE-Subsystems sind und speziell zur Verwendung mit `BSTR`-Zeichenfolgen entwickelt wurden.

Sie fragen sich vielleicht, wie Sie ein NULL-Zeichen in eine C-Zeichenfolge einschließen können. Die Antwort ist einfach – es ist nicht möglich. Dies ist einer der Gründe dafür, warum das BSTR-Format entwickelt wurde.

Wie können Sie ermitteln, ob eine API-Funktion, die einen Zeichenfolgenparameter verwendet, eine Unicode- oder ANSI-Zeichenfolge erwartet? Diese Informationen erhalten Sie in der Dokumentation zu der Funktion. Zur Gewährleistung maximaler Flexibilität setzen sich die meisten API-Funktionen tatsächlich aus zwei separaten Funktionen zusammen: eine für die Verarbeitung von ANSI-Zeichenfolgen, die andere für die von Unicode-Zeichenfolgen. So gibt es beispielsweise eigentlich keine `GetWindowText`-Funktion. Tatsächlich handelte es sich um zwei Funktionen, `GetWindowTextA` und `GetWindowTextW`. Die erste Funktion verwendet ANSI-Zeichenfolgen, die zweite hingegen verwendet Zeichenfolgen im (wide) Unicode-Format.

In Visual Basic wird die ANSI-Deklaration folgendermaßen eingesetzt:

```
Declare Function GetWindowText Lib "user32" Alias "GetWindowTextA" _
(ByVal hwnd As Long, ByVal lpString As String, ByVal cch As Long) As Long
```

Die `Alias`-Anweisung gibt hierbei den Namen der Funktion innerhalb der DLL an. Wenn Sie also in Visual Basic auf `GetWindowText` verweisen, wird tatsächlich die Funktion `GetWindowTextA` aufgerufen. In Visual Basic rufen Sie in nahezu sämtlichen Fällen die ANSI-Version einer Funktion auf. Warum? Weil zwar sowohl von Windows 95/98 als auch von Windows NT Unicode- und ANSI-Versionen der meisten Funktionen exportiert werden, unter Windows 95/98 jedoch nur die ANSI-Versionen verwendet werden können – die Unicode-Einsprungpunkte sind zwar unter Umständen vorhanden, haben jedoch keinerlei Auswirkungen. Dieser Vorgang wird in Abbildung T5-6 veranschaulicht.

Sie sehen also, dass in Visual Basic Zeichenfolgen intern im BSTR-Unicode-Format gespeichert werden und dass die Mehrzahl der API-Funktionen eine auf NULL endende ANSI-Zeichenfolge erwartet. Dies wirft erneut die interessante Frage auf: Wie erreichen Sie dies?

Übergabe von Zeichenfolgen an DLL-Funktionen

Für die meisten Visual Basic-Programmierer beinhalten die folgenden zwei Regeln alles, was Sie zur Übergabe von Zeichenfolgen an DLL- oder API-Funktionen wissen müssen:

- Verwenden Sie stets `ByVal`, wenn Sie Zeichenfolgen als Parameter an DLL-Funktionen übergeben.

- Wenn die Zeichenfolge durch die API- oder DLL-Funktion eventuell geändert wird, stellen Sie sicher, dass die an die Funktion übergebene Zeichenfolge lang genug ist, um die geänderte Zeichenfolge aufnehmen zu können.

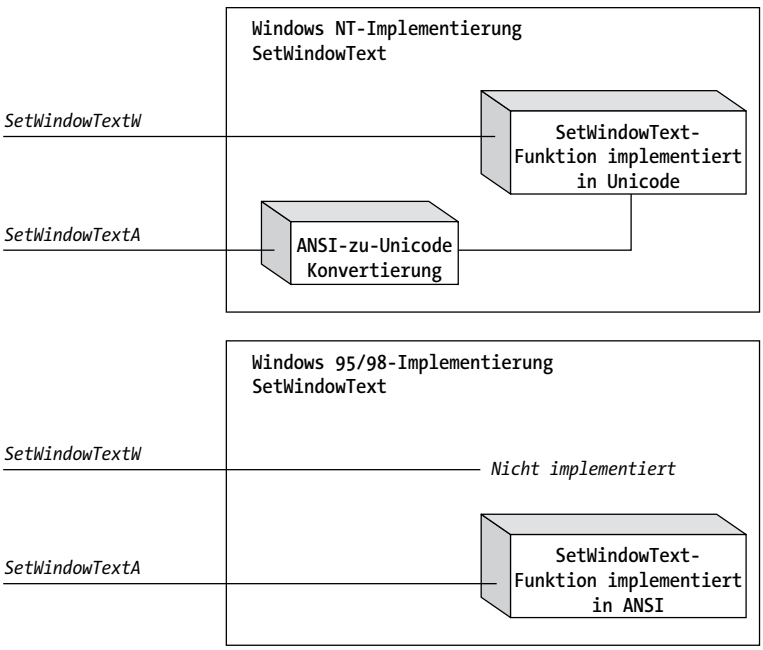


Abbildung T5-6 Unterschiede zwischen der Windows 95/98- und Windows NT-Implementierung von API-Funktionen, die Zeichenfolgen verwenden

Im verbleibenden Teil dieses Abschnitts werden diese Regeln erläutert, und Sie erfahren, wann diese Regeln ignoriert werden sollten, welche weiteren Optionen Ihnen zur Verfügung stehen und was sich hinter den Kulissen abspielt.

Wenn Sie in einer `Declare`-Anweisung einen Zeichenfolgenparameter angeben, erstellt Visual Basic ein neues `BSTR`-Objekt, dass mit einer auf NULL endenden ANSI-Kopie der Zeichenfolge geladen wird. Dies ist einer der seltenen Fälle, in denen eine `BSTR`-Zeichenfolge statt einer Unicode- eine ANSI-Zeichenfolge enthält.

Wenn Sie bei der Zeichenfolgenübergabe den `ByVal`-Operator verwenden, wird der `BSTR`-Zeiger an die API-Funktion übergeben. Übergeben Sie die Zeichenfolge `ByRef` (Standard), wird der API-Funktion ein Zeiger auf eine temporäre Zeigervariable übergeben, die wiederum den `BSTR`-Zeiger enthält. Diese Fälle werden in Abbildung T5-7 dargestellt.

Nahezu sämtliche Win32-API-Funktionen erwarten einen Zeiger auf eine auf NULL endende Zeichenfolge als Parameter. Dies bedeutet, dass Sie die Zeichenfolgen mit dem `ByVal`-Schlüsselwort übergeben müssen. Sie werden gelegentlich in Situationen geraten, in denen eine benutzerdefinierte DLL-Funktion von Visual Basic die Übergabe eines Zeigers auf eine BSTR-Zeichenfolge erwartet. Dies sind die einzigen Fälle, in denen Sie das `ByVal`-Schlüsselwort nicht verwenden.

Nachdem die API-Funktion zurückgegeben wurde, kopiert Visual Basic die ANSI-BSTR-Zeichenfolge wieder in das interne Unicode-Format zurück.

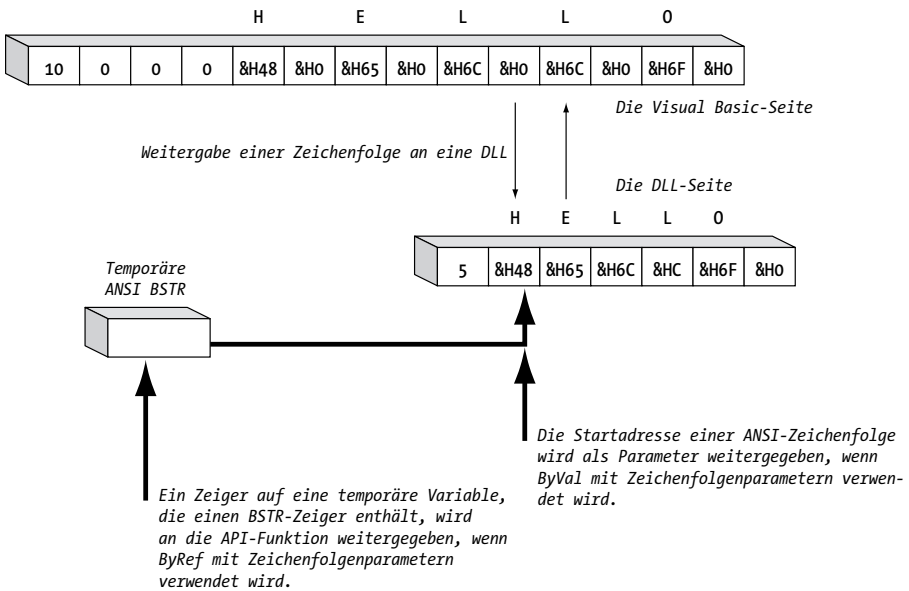


Abbildung T5-7 Die Übergabe von Zeichenfolgen als Parameter

Aufrufen des Unicode-Einsprungpunktes

Unabhängig davon, ob Sie `ByVal` oder `ByRef` deklarieren, übergibt Visual Basic eine temporäre ANSI-Zeichenfolge an API-Funktionen, wenn Sie die `Declare`-Anweisung verwenden. Was geschieht jedoch, wenn Sie einen Zeiger auf eine Unicode-Zeichenfolge (einen Zeiger auf eine `BSTR`-Zeichenfolge) oder einen Zeiger auf eine dynamische Zeichenfolgenvariable übergeben möchten, die auf eine Unicode-Zeichenfolge verweist? Dies kann nötig sein, wenn Sie den Unicode-Einsprungpunkt einer Funktion aufrufen müssen, oder wenn Sie eine OLE-Funktion aufrufen, die eine Unicode-Zeichenfolge erwartet.

Bei der Erledigung dieser Aufgabe stehen Ihnen zwei Möglichkeiten zur Verfügung:

- Sie erstellen eine Typenbibliothek, in der Sie den `BSTR`-Datentyp angeben.
- Sie übergeben die Adresse mit Hilfe eines `Long`-Parameters, wie zuvor im `RtlMoveMemory`-Beispiel gezeigt.

Sehen Sie sich beispielsweise diese Deklaration für den Unicode-Einsprungpunkt der `SetWindowText`-Funktion an:

```
Declare Function SetWindowText Lib "user32" Alias "SetWindowTextW" _
(ByVal hwnd As Long, ByVal lpString As Long) As Long
```

Dem `lpString`-Parameter muss die Adresse einer Unicode-Zeichenfolge übergeben werden. Die `StrPtr`-Funktion kann dazu verwendet werden, die Adresse der internen Visual Basic-Zeichenfolge abzurufen, bei der es sich um eine Unicode-Zeichenfolge handelt. Der folgende Code aus der Beispieldatei **ByValSt.vbp** (auf der Begleit-CD-ROM zu diesem Buch) legt den Formulartitel mit Hilfe der Unicode-Einsprungpunktdeklaration für die oben erwähnte `SetWindowText`-Funktion fest.

```
Private Sub cmdSetWindowText_Click()
   Dim s$
   s$ = "New Text" & chr$(0)
   Call SetWindowText(hwnd, StrPtr(s$))
End Sub
```

Warum fügen wir am Ende der Variablen `s$` ein NULL-Zeichen an? Die Wahrheit ist, dass dieser Schritt nicht unbedingt erforderlich ist. Es ist äußerst wichtig, dass die an die API-Funktion übergebene Zeichenfolge auf NULL endet. Wenn Sie nach dieser Methode vorgehen, übergeben Sie einen Zeiger auf die Zeichenfolgen wie intern in Visual Basic gespeichert, und wie Sie zuvor sehen konnten, weist eine BSTR-Zeichenfolge kein abschließendes NULL-Zeichen auf. In den meisten Fällen stellt sich heraus, dass am Ende von Zeichenfolgen gleich mehrere NULL-Bytes auftauchen, dies ist jedoch nicht immer so. Daher sollten Sie, nur zur Sicherheit, explizit ein NULL-Zeichen anhängen.

Sie sollten außerdem daran denken, dass obwohl der `SetWindowTextW`-Einsprungpunkt unter Windows 95/98 vorhanden ist, dieser nicht verwendet wird. Die meisten Unicode-Einsprungpunkte geben unter Windows 95/98 lediglich Fehler zurück.

Dies soll als kurze Erläuterung der Zeichenfolgenparameter genügen. Der Umgang mit Zeichenfolgenparametern kann im Zusammenhang mit API- und DLL-Aufrufen zu einer sehr komplexen Aufgabe werden, und viele der Puzzle in diesem Buch beinhalten Techniken zur Handhabung solcher Probleme.

Tutorium 6

C++-Variablen treffen auf Visual Basic

Als Visual Basic-Programmierer sollten Ihnen die verschiedenen Visual Basic-Datentypen bekannt sein. Deren jeweilige Merkmale werden in Tabelle T6-1 aufgeführt.

Typ	Bytes	Vorzeichen	Bereich
`Byte`	1	Ohne Vorzeichen	0 bis 255
`Boolean`	2	–	True oder False
`Integer`	2	Mit Vorzeichen	–32 768 bis 32 767
`Long`	4	Mit Vorzeichen	–2 147 483 648 bis 2 147 483 647
`Single`	4	Mit Vorzeichen	±1.4E–45 bis ±3.4E38
`Double`	8	Mit Vorzeichen	±4.9E–324 bis ±1.7E308
`Currency`	8	Mit Vorzeichen	Siehe VB-Dokumentation – nicht anwendbar für API
`Decimal`	14	Mit Voreichen	Siehe VB-Dokumentation – nicht anwendbar für API
`Date`	8	–	Siehe VB-Dokumentation – nicht anwendbar für API
`Object`	4	–	
`String`	*	–	Länge variiert nach Zeichenfolge
`Variant`	16	–	Richtet sich nach enthaltener Variable
`Benutzer-definiert`	*		Richtet sich nach der jeweiligen Definition
* = Variiert			

Tabelle T6-1 Merkmale der Visual Basic-Datentypen

C-Datentypen

Wenn sich ein Visual Basic-Programmierer zum ersten Mal eine C/C++-Deklaration ansieht, kann dies etwas einschüchternd wirken.[1] Die Sprache C scheint über Millionen verschiedener Datentypen zu verfügen. Wie können Sie herausfinden, welche dieser Datentypen äquivalent zu den Visual Basic-Datentypen sind?

1. Die Sprache C++ basiert auf C und ist, vom Standpunkt eines Visual Basic-Programmierers gesehen, grundsätzlich mit dieser identisch. Aus diesem Grund werden in diesem Buch C und C++ als gleichwertig behandelt. Was die einschüchternden Deklarationen angeht – die komplexen Deklarationen sind fast nicht zu verstehen. Tatsächlich war die Inspiration zu diesem Buch ein anderes Buch, nämlich ein C-Puzzlebuch, bei dem die Hauptaufgabe des Lesers darin bestand, sich durch komplexe Typendeklarationen zu kämpfen, um diese zu verstehen.

Der Trick besteht darin, im Umgang mit diesen Datentypen die folgenden zwei Dinge im Auge zu behalten:

- Alles nur Einbildung – C verfügt tatsächlich nur über sehr wenige Datentypen.
- Jeder Datentyp enthält den Schlüssel zur Dechiffrierung seiner Bedeutung.

Lassen Sie uns einen Blick auf die C-Datentypen werfen, die Ihnen begegnen werden – in Tabelle T6-2.

Typ	Bytes	Vorzeichen	Bereich
`char`	1	Mit Vorzeichen	–128 bis 127
`unsigned char`	1	Ohne Vorzeichen	0 bis 255
`short`	2	Mit Vorzeichen	–32 768 bis 32 767
`unsigned short`	2	Ohne Vorzeichen	0 bis 65 535
`long`	4	Mit Vorzeichen	–2 147 483 648 bis 2 147 483 647
`unsigned long`	4	Ohne Vorzeichen	0 bis 4 294 967 295
`int`	4	Mit Vorzeichen	Entspricht `Long` unter Win32
`unsigned int`	4	Ohne Vorzeichen	Entspricht `Long` ohne Vorzeichen unter Win32
`float`	4	Mit Vorzeichen	±1.4E–45 bis ±3.4E38
`double`	8	Mit Vorzeichen	±4.9E–324 bis ±1.7E308
`class`	*		Richtet sich nach der jeweiligen Definition
`struct`	*		Richtet sich nach der jeweiligen Definition
`void`	0	–	Gibt keinen Typ an. Eine Funktion die `void` zurückgibt, entspricht einem `Sub` in Visual Basic – es ist kein Rückgabewert vorhanden.

Tabelle T6-2 C-Datentypen

Des Weiteren sind noch der Datentyp `long double` und `an__intxx` vorhanden, wobei `xx` die Größe des `Integer`-Wertes angibt. Diese Typen werden nur sehr selten verwendet – ich habe bisher noch keine API- oder DLL-Funktion gesehen, in der sie eingesetzt wurden.

Sie fragen sich vielleicht, wo die Zeichenfolgendatentypen sind. Und was ist mit Objekten und Varianten? Und wo sind die Booleschen Variablen?

Nun kann man in C/C++ mittels eines Kniffs aus diesen vorhandenen Datentypen neue Datentypen ableiten. Sobald Sie einen neuen Datentyp abgeleitet haben, kann dieser neue Typ genau wie die bereits integrierten Datentypen verwendet werden. Sie werden gleich sehen, wie diese neuen Datentypen deklariert werden.

C-Deklarationen ähneln den Visual Basic-Deklarationen sehr stark, sie sehen lediglich aus, als habe man sie umgedreht. In Visual Basic können Sie einen `Long`-Wert beispielsweise folgendermaßen definieren:

```
Dim myLong As Long
```

In einem C-Programm würde die Definition so lauten:

```
long myLong;
```

Der Typ wird also zuerst genannt.

Beachten Sie, dass sämtliche der integrierten C-Datentypen mit Kleinbuchstaben geschrieben werden. Ein Grund, sich diese Tatsache zu merken, ist der, dass fast alle C-Programmierer für Konstanten und abgeleitete Datentypen (also Datentypen, die durch den Programmierer, nicht durch die Sprache definiert wurden) Großbuchstaben verwenden. In C muss bei Schlüsselwörtern, einschließlich der Datentypen, die Groß- und Kleinschreibung beachtet werden. Während es also in Visual Basic unerheblich ist, ob Sie `long`, `Long` oder `LONG` eingeben, da die Groß- und Kleinschreibung automatisch der ursprünglichen Deklaration angepasst wird, würden diese unterschiedlichen Schreibweisen in C als unterschiedliche Schlüsselwörter interpretiert werden. Auf diese Weise entstehen zwangsläufig Fehler, wenn Sie denn ein Schlüsselwort orthographisch nicht korrekt verwenden, da es ja aus der Sicht von C/C++ nicht vorhanden ist.

Arrays

Die Arrays in C sind denen in Visual Basic sehr ähnlich, mit der Ausnahme, dass die Arrays in C weniger flexibel sind. Zur Kennzeichnung eines Arrays werden eckige Klammern verwendet:

```
int A[5];
```

Mit der obigen Zeile wird demnach ein Array mit fünf `integer`-Werten definiert. Der gültige Indexwert für das Array lautet A[0] bis A[4]. Ja, in C lautet der Basisindex immer 0.

Sehen Sie sich das folgende Zeichenwertarray an:

```
char myString[4];
myString [0] = 'Y';
myString [1] = 'E';
myString [2] = 'S';
myString [3] = 0;
```

Durch diesen Code wird ein Array mit vier Zeichen definiert. Das letzte Zeichen ist das Zeichen NULL (Wert 0). In Tutorium 5, »Das `ByVal`-Schlüsselwort – die Lösung für 90% aller API-Probleme«, wird eine C-Zeichenfolge als Zeichenwertarray mit einem abschließenden NULL-Zeichen beschrieben. Das kommt Ihnen bekannt vor? Genau so behandeln C-Programme Zeichenfolgen – als ein Zeichenwertarray.

Dies gilt für ANSI-Zeichenfolgen, bei denen jedes Zeichen nur ein Byte umfasst. Wie sieht es mit Unicode-Zeichenfolgen aus, bei denen jedes Zeichen aus zwei Bytes besteht? Ein Array zur Speicherung einer Unicode-Zeichenfolge könnte folgendermaßen definiert werden:

```
unsigned short UnicodeString[4];
```

Zeiger

In C wird umfangreicher Gebrauch von Zeigern gemacht. Ein Stern (*) symbolisiert einen Zeiger. Beispiel:

```
short *pIntVariable;
```

Diese Zeile gibt an, dass es sich bei der Variablen `pIntVariable` um einen Zeiger auf einen `short`-Wert handelt. Die Variable `pIntVariable` selbst umfasst 32 Bits. Die Daten, auf die verwiesen wird, ist ein `short`-Wert von 16 Bits Länge.

Sie haben bereits erfahren, dass eine ANSI-C-Zeichenfolge ein Array von `char`-Variablen darstellt. Wenn Sie jedoch eine Zeichenfolge als Funktionsparameter übergeben möchten, übergeben Sie nicht das gesamte Array. Statt dessen übergeben Sie einen Zeiger auf die Zeichenfolge. Was ist ein Zeiger auf eine Zeichenfolge? Es ist ein Zeiger, der den Standort des ersten Zeichens im Array angibt. Das erste Zeichen im Array ist vom Typ `char`, daher muss es sich bei einem Zeiger auf eine Zeichenfolge um einen Zeiger auf eine `char`-Variable handeln. Ein Zeiger auf eine `char`-Variable wird folgendermaßen definiert:

```
char *pString;
```

Wie gehen Sie aber vor, wenn der Zeichenfolgenzeiger auf das zuvor definierte `myString`-Array verweisen soll? Ein C-Programmierer könnte folgenden Code verwenden:

```
pString = &myString[0];
```

Der `&`-Operator steht für die Variablenadresse. In diesem Fall wird der Operator genauso verwendet, wie ein Visual Basic-Programmierer den `VarPtr`-Operator zum Abrufen der Adresse einer VB-Variablen einsetzen würde.

Der »reference«-Datentyp

Bei `reference` handelt es sich um eine Deklaration, die wie ein Zeiger funktioniert. Es wird lediglich eine andere Syntax verwendet. Das `&`-Zeichen wird etwas anders eingesetzt. Beispiel:

```
int myVariable;
int &ReferenceToVariable = myVariable;
```

Die Variable `ReferenceToVariable` wird hier zu einem Zeiger auf `myVariable`.

Verweise werden lediglich im Zusammenhang mit bestimmten Strukturtypen im OLE-Subsystem gebraucht. Sollte Ihnen der Datentyp `reference` begegnen, können Sie diesen gefahrlos wie einen Zeiger behandeln.

»Classes« und »Structures«

Diese Begriffe werden in C zur Definition benutzerdefinierter Datentypen verwendet. Wie mit diesen Klassen und Strukturen umzugehen ist, wird im Verlaufe dieses Buches noch näher erläutert. Vom Standpunkt eines Visual Basic-Programmierers aus gesehen bestehen zwischen `class` und `structure` keine Unterschiede. Die Definition dieser Begriffe lautet wie folgt:

```
class myClass {
Definieren Sie hier die Variablen innerhalb der Struktur
} ;
```

Abgeleitete Datentypen

Bisher wissen Sie also, dass in C die folgenden Schlüsselwörter für die verschiedenen Datentypen zum Einsatz kommen: `char`, `short`, `int`, `long`, `float` und `double` sowie die folgenden Modifier:

- `unsigned`: Gibt an, dass die Daten keine Vorzeichen aufweisen. Es gibt auch ein Schlüsselwort mit Namen `unsigned`, dieses wird jedoch selten verwendet, da die Standardeinstellung `signed` lautet.
- `[ ]`: Kennzeichnet ein Array eines grundlegenden Datentyps
- `*`: Kennzeichnet einen Zeiger auf einen Datentyp
- `&`: Gibt die Adresse einer Variablen an oder kennzeichnet einen Verweis auf eine Variable

Obwohl diese Liste nur sehr kurz ist, werden Sie in jeder API- oder DLL-Dokumentation Dutzende zusätzlicher Datentypen finden. Woran liegt das?

Die Antwort ergibt sich aus der Tatsache, dass Sie in C aus vorhandenen Datentypen neue Datentypen ableiten können. Diese Ableitung erfolgt mit Hilfe der Anweisung `typedef`. Sehen Sie sich beispielsweise die folgende Anweisung an:

```
typedef  unsigned char BYTE;
```

Sobald diese Zeile in einem C-Programm oder einer Headerdatei erscheint, können Sie Variablen vom Typ `unsigned char` deklarieren, indem Sie die neue Typendefinition BYTE verwenden. Daher sind die folgenden zwei Anweisungen identisch:

```
unsigned char myChar;
BYTE myChar;
```

Mit der Deklaration

```
typedef char *LPSTR;
```

wird `LPSTR` als Zeiger auf einen Zeichenwert definiert – wie Sie sich vielleicht erinnern, wird in C so auch auf Zeichenfolgen verwiesen. Aus diesem Grund wird `LPSTR` als Parameter in API-Deklarationen häufig zur Kennzeichnung eines Zeichenfolgenparameters eingesetzt. Der Parameter ist ein Zeiger auf eine Zeichenfolge, d.h., er ist identisch mit einem Zeiger auf einen `char`-Wert, wenn der Zeichenwert, auf den verwiesen wird, der erste Zeichenwert in einer auf NULL endenden Zeichenfolge ist.

Standardmäßig werden abgleitete Datentypen durch GROSSBUCHSTABEN gekennzeichnet.

Die `typedef`-Anweisung kann auch dazu verwendet werden, Zeiger auf Funktionen zu definieren. Ein Zeiger auf eine Funktion hat mit einem beliebigen anderen Zeiger gemeinsam, dass er die Adresse eines Speicherstandortes enthält. Der einzige Unterschied besteht darin, dass ein Zeiger auf eine Funktion auf Speicherstellen verweist, die keine Daten, sondern ausführbaren Code enthalten. Der Visual Basic-Operator `AddressOf` kann dazu verwendet werden, in einem Standardmodul einen Zeiger auf eine Funktion abzurufen.

Eine `typedef`-Anweisung, mit der ein Zeiger auf eine Funktion definiert wird, sieht folgendermaßen aus:

```
typedef void (*MYFUNCTIONTYPE)();
```

In diesem Fall wird `MYFUNCTIONTYPE` als Datentyp definiert, der auf eine Funktion ohne Parameter und ohne Rückgabewert verweist.

```
typedef long (*MYFUNCTIONTYPE)(short a, char *b);
```

Mit diesem Code wird MYFUNCTIONTYPE als Datentyp definiert, der auf eine Funktion mit zwei Parametern verweist, auf einen 16-Bit-integer-Wert und auf einen 32-Bit-Zeiger auf ein Zeichenwertarray. Außerdem wird durch die Funktion ein Long-Wert zurückgegeben.

Makros

Die Sprache C verfügt außerdem über eine Makrofähigkeit, die es Ihnen ermöglicht, ein Schlüsselwort während der Kompilierung durch ein anderes zu ersetzen. Diese Definitionen erfolgen unter Verwendung der #define-Anweisung. Durch die Codezeile

```
#define VOID void
```

wird beispielsweise das Wort »VOID« bei der Kompilierung durch das Wort »void« ersetzt. In Makros können ebenfalls Parameter verwendet werden. Durch die Zeile

```
#define DECLARE_HANDLE(name) typedef HANDLE name
```

wird ein Makro mit Namen DECLARE_HANDLE definiert, in dem Parameter verwendet werden können.

Falls nun der folgende Code erscheint

```
DECLARE_HANDLE(HWND);
```

wird dieser durch den nachfolgenden Code ersetzt:

```
typedef HANDLE HWND;
```

Benennung der Datentypen

Die Fähigkeit der C-Sprache, dem Programmierer die Definition eigener Datentypen zu ermöglichen, erklärt die Tatsache, dass es in Windows Hunderte verschiedener Datentypen gibt. Microsoft hat dies nicht getan, um Sie zu verwirren – dem Programmierer sollte so die Möglichkeit gegeben werden, zuverlässigere Programme zu entwickeln. Die Typendefinitionen unterstützten sowohl den Programmierer als auch den C-Compiler bei der Übergabe der richtigen Parameter an die verschiedenen Funktionen, und es wird eine Variablenzuordnung mit nicht kompatiblen Datentypen vermieden.

Die Programmierer von Windows trugen dazu bei, selbsterklärende Datentypen zu erstellen, indem sie einer konsistenten Benennungskonvention folgten. Jeder Datentyp verfügt über einen Basisnamen, mit dem die Daten beschrieben werden. Beispielsweise kennzeichnet der Datentyp CHAR Zeichenwerte. ANSI-Zeichen

würden demnach auf dem char-Datentyp basieren. Unicode-Zeichen würden auf dem unsigned short-Datentyp basieren.

Der Name weist außerdem eines oder mehrere Präfixe auf, wie in Tabelle T6-3 dargestellt:

Präfix	Bedeutung
P	Bezeichnet einen Zeiger. Daher ist PCHAR ein Zeiger auf CHAR.
LP	Bezeichnet einen Zeiger. Unter 16-Bit Windows wird zwischen nahen und entfernten Speicherzeigern unterschieden. Daher handelt es sich bei LPCHAR um einen entfernten Zeiger auf CHAR, PCHAR oder NPCHAR stellen nahe Zeiger auf CHAR dar. Unter Win32 sind diese identisch – alle Zeiger umfassen 32 Bits, daher können NP, P und LP identisch behandelt werden.
NP	Bezeichnet einen Zeiger. Siehe LP.
C	Bezeichnet Konstantendaten. LPCCHAR kann daher einen Zeichenwert kennzeichnen, der nach dem Programmstart nicht mehr geändert werden kann. Visual Basic-Programmierer können das Präfix C ignorieren oder zur Identifikation von Parametern einsetzen, die durch die API-Funktion nicht geändert werden. Bei LPCSTR handelt es sich um einen Zeiger auf eine Zeichenfolge, die bei der Übergabe an eine Funktion nicht geändert wird.
W	Wird zur Kennzeichnung von wide (Unicode-)Daten verwendet. LPWCHAR ist daher ein Zeiger auf ein wide Zeichen. Beachten Sie, dass ein wide Zeichen auf dem Datentyp unsigned short basiert.
T	C-Programmierer können Anwendungen erstellen, bei denen intern standardmäßig Unicode oder ANSI verwendet wird. Das Präfix T kennzeichnet Zeichenwerte, die in Unicode-Anwendungen immer Unicode, in ANSI-Anwendungen immer ANSI lauten. Daher steht LPTSTR LPSTR für einen ANSI-Funktionseinsprungpunkt und LPWSTR für einen Unicode-Einsprungpunkt.

Tabelle T6-3 Übliche Präfixe für Datentypen

Häufig verwendete Windows-Datentypen

Lassen Sie uns zunächst den ersten Teil der Headerdatei **winnt.h** ansehen, die im Lieferumfang des Win32 Software Development Kit (SDK) und in anderen Entwicklungstools enthalten ist, um einen Einblick in die häufig verwendeten Windows-Datentypen zu erhalten.[2] In Tabelle T6-4 werden nur die typedef-Anweisungen aufgeführt.

2. Hierzu können Sie sich auch die Datei **winbase.h** ansehen. Dank des Microsoft-Programms Open Tools sind alle Headerdateien des Win32 Software Development Kit auf der Begleit-CD-ROM zu diesem Buch enthalten.

`typedef void *PVOID;`	Der `void`-Datentyp wird für Daten ohne festen Typ verwendet. Tatsächlich können Sie keine Variable vom Typ `void` erstellen. Sie können jedoch einen Zeiger auf `void` erstellen. Diese Art Zeiger wird als Zeiger auf einen beliebigen Datentyp verwendet. `PVOID` ist daher ein Zeiger auf einen beliebigen Datentyp.

`typedef char CHAR;`
`typedef short SHORT;`
`typedef long LONG;`
`typedef unsigned short wchar_t;`
`typedef wchar_t WCHAR;`
	Die einfache Erklärung lautet, dass diese drei Typendefinitionen einfache Ersetzungen für Programmierer darstellen, die nur Datentypennamen mit Großbuchstaben verwenden möchten[1]. Diese `wide`-Zeichenwertdefinition wird in der Datei **basetyps.h** verwendet. `Wide`-Zeichen (oder Unicode) umfassen 16 Bits. Beachten Sie, wie eine Typendefinition auf einer anderen basieren kann. `WCHAR` ist abgeleitet von `wchar_t`, welcher wiederum von `short` abgeleitet ist.

`typedef WCHAR *PWCHAR;`
`typedef WCHAR *LPWCH, *PWCH;`
`NWPSTR, LPW`
`typedef WCHAR *NWPSTR;`
`typedef WCHAR *LPWSTR, *PWSTR;`
	`PWCHAR` zeigt auf ein Unicode-Zeichen, bei dem es sich auch um den Beginn einer Unicode-Zeichenfolge handeln kann. `LPWCH` und `PWCH` sind identisch. `STR` und `PWSTR` sind ebenfalls identisch – statt eines Zeichens wird eine Zeichenfolge (`STR`) vorausgesetzt. Warum gibt es so viele verschiedene Möglichkeiten zur Beschreibung eines Zeigers auf eine Unicode-Zeichenfolge? Ich denke, es ist als Erleichterung für den Programmierer gedacht, auch wenn es ein bisschen viel erscheint.

`typedef const WCHAR *LPCWSTR,` `*PCWSTR;`	Diese zwei Deklarationen umfassen den `const`-Schlüsselwert, mit dem angegeben wird, dass der Typ auf Konstantendaten verweist, also Daten die nicht geändert werden. Visual Basic-Programmierer können das `const`-Schlüsselwort ignorieren, da `const`-Datentypen genauso behandelt werden wie Daten, die nicht konstant sind.

`typedef CHAR *PCHAR;`
`typedef CHAR *LPCH, *PCH;`
`typedef CONST CHAR *LPCCH,`
`*PCCH;`
`typedef CHAR *NPSTR;`
`typedef CHAR *LPSTR, *PSTR;`
`typedef CONST CHAR *LPCSTR,`
`*PCSTR;`
	`CHAR` wurde zuvor als `char` definiert. Bei `PCHAR`, `LPCH`, `PCH`, `LPCCH`, `PCCH`, `NPSTR`, `LPSTR`, `PSTR`, `LPCSTR` und `PCSTR` handelt es sich jeweils um Zeiger auf eine Variable mit dem Datentyp `char`. Wie Sie wissen, entspricht dies einem Zeiger auf eine auf NULL endende Zeichenfolge.

`typedef SHORT *PSHORT;` `typedef LONG *PLONG;`	Zeiger auf `short`- oder `long`-Daten. Die numerischen Fälle sind bedeutend einfacher als die Zeichenwerte.

Tabelle T6-4 Häufig verwendete Windows-Datentypen

`typedef void *HANDLE;` `typedef HANDLE *PHANDLE;`	Ein `HANDLE` wird als Zeiger auf einen beliebigen Datentyp definiert. Dies mag merkwürdig erscheinen, da Visual Basic-Programmierer daran gewöhnt sind, Zugriffsnummern als 32-Bit-`Long`-Werte zu behandeln. Es besteht jedoch kein Konflikt – ein Zeiger umfasst 32 Bits, daher werden Zugriffsnummern in `Long`-Variablen gespeichert. Sie werden in den meisten Fällen niemals auf die Daten zugreifen, auf die ein `HANDLE` verweist; diese Daten werden nahezu immer Windows-intern verwendet.[2]

`typedef unsigned short WORD;`
`typedef unsigned long DWORD;`
`typedef WORD far *LPWORD;`
`typedef DWORD far *LPDWORD;`
	Ein `WORD`-Parameter ist eine 16-Bit-Variable ohne Vorzeichen. `LPWORD` zeigt auf `WORD`. Bei `DWORD` handelt es sich um eine 32-Bit-`Long`-Variable ohne Vorzeichen. `LPDWORD` zeigt auf `DWORD`.

`typedef DWORD LCID;` `typedef PDWORD PLCID;`	`LCID` steht für Locale Identifier, einem Zahlenwert, mit dem eine bestimmte Sprache und ein bestimmtes Land angegeben werden. Ein `LCID` ist ebenfalls ein 32-Bit-`long`-Wert, durch Verwendung von `LCID` anstelle von `long` wird der Datentyp jedoch für die Programmierer eindeutiger, die den Code lesen. `PLCID` ist ein Zeiger auf einen `LCID`-Wert.

Tabelle T6-4 Häufig verwendete Windows-Datentypen

3. Die komplexe Erläuterung lautet, dass diese Typendeklarationen einem Programmierer dabei helfen, portable Anwendungen zu erstellen. Angenommen, Sie benötigen eine Variable mit 16 Bits. Eine `short`-Variable kann unter Win32 16 Bits umfassen, in C wird diese jedoch mit einer Größe definiert, die kleiner oder gleich einer `int`-Variable und größer oder gleich einer `char`-Variable ist. Wenn Sie versuchen, ein Programm mit einer C-Version zu kompilieren, bei der `short` mit 8 Bits definiert ist, können Probleme auftreten, sodass Sie Ihre Variablentypen innerhalb des Programms ändern müssen. Durch Verwendung von `SHORT typedef` ist lediglich eine Neudefinition in einen 16-Bit-Typ für den neuen Compiler erforderlich, alle als `SHORT` definierten Variablen und Parameter würden richtig kompiliert.
4. Wenn die Daten, auf die durch ein `HANDLE` verwiesen wird, niemals durch eine Anwendung verwendet werden, warum wird das `HANDLE` dann als Zeiger und nicht als Long definiert? In früheren Windows-Versionen war dies tatsächlich der Fall. Durch die Änderung in einen Zeiger konnte Microsoft jedoch verschiedene Zugriffsnummertypen definieren, die als Zeiger auf verschiedene Strukturen dienen. Der C-Compiler ist so in der Lage, Übergabeversuche bestimmter Zugriffsnummertypen an Funktionen zu ermitteln, die einen anderen Zugriffsnummertyp erwarten. Auf diese Weise können Programmierungsfehler ermittelt werden, die durch Unachtsamkeit entstehen, beispielsweise die Übergabe einer Zugriffsnummer für einen Gerätekontext an Funktionen, die Zugriffsnummern für ein Fenster erwarten.

Zurück zu Visual Basic

Nun, da Sie wissen, wie in C die verschiedenen Datentypen definiert werden, lassen Sie uns einen Blick auf die Datenkonvertierung von C nach Visual Basic werfen. In diesem Zusammenhang müssen Sie sich nur eine Sache merken, nämlich die Länge der Datenvariable. Lassen Sie sich nicht durch die komplexe Notierung der verschiedenen C-Datentypen verunsichern. Diese wurden im letzten Abschnitt nur deshalb vorgestellt, damit Sie erkennen können, auf welchem grundlegenden Datentyp ein abgeleiteter Datentyp basiert. Tabelle T6-5 zeigt die Visual Basic-Datentypen, die am ehesten den Datentypen entsprechen, die in C integriert sind.

C-Datentypen	Visual Basic-Äquivalent
`char`	`Byte`
`short`	`Integer`
`long`	`Long`
`float`	`Single`
`double`	`Double`
`class, struct`	`Benutzerdefinierter Typ`

Tabelle T6-5 VB-Äquivalente zu den grundlegenden C-Datentypen

Behandlung von Datentypen mit und ohne Vorzeichen

Der C-Datentyp char umfasst 8 Bits, daher würden Sie den Visual Basic-Datentyp `Byte` sowohl für `chars` mit und ohne Vorzeichen verwenden. Der Trick beim Umgang mit diesen Datentyp besteht darin, die Bereichseinschränkungen zu kennen, die während der Konvertierung zu berücksichtigen sind.

Angenommen, Sie verfügen über eine C-Struktur mit einem `char`-Wert mit Vorzeichen, und Sie müssen diesen auf –50 setzen.

```
struct myStruct {
char myChar;
}
```

Die äqivalente Visual Basic-Struktur lautet folgendermaßen:

```
Type myStruct
myChar As Byte
End Type
```

Falls Sie versuchen sollten, die folgende Zuordnung vorzunehmen,

```
Dim S As myStruct
S.myChar = -50
```

so führt dies unweigerlich zu einem Bereichsüberlauffehler, da die `Byte`-Variable keine Vorzeichen aufweist und daher keinen negativen Wert aufnehmen kann.

Wenn Sie sich Tutorium 2, »Der Arbeitsspeicher – da, wo alles beginnt«, ansehen, werden Sie bemerken, dass der einzige Unterschied zwischen einer Variablen mit und der ohne Vorzeichen darin liegt, dass sie einen unterschiedlichen Platzierungsbereich aufweisen. Bytes mit Vorzeichen reichen von –128 bis 127, Bytes ohne Vorzeichen von 0 bis 255. Das einzige, was Sie demnach zur Konvertierung eines Wertes außerhalb des Bereiches tun müssen, ist die Subtraktion von 256 mit Hilfe des folgenden Ausdrucks:

```
SignedValue = UnsignedValue - 256
UnsignedValue = SignedValue + 256
```

Daher müssen Sie zum Setzen des `mChar`-Wertes lediglich die Zahl 256 zu –50 addieren, um die Zahl 206 zu erhalten.

```
S.myChar = 206
```

`Integer`-Werte werden ähnlich behandelt:

```
SignedValue = UnsignedValue - &H10000
UnsignedValue = SignedValue + &H10000
```

Beachten Sie, dass Sie bei der Berechnung `Long`-Variablen verwenden müssen, damit keine Überlauffehler entstehen. Die gleiche Methode funktioniert auch mit `Long`-Variablen, hier müssen Sie jedoch `Double`-Werte für die Zwischenergebnisse verwenden, um Überlauffehler auszuschließen (und das Risiko, dass bei der Berechnung Genauigkeitseinbußen auftreten).

Die Datentypen »Currency« und »Date«

Die Datentypen »Currency« und »Date« werden in Win32-Funktionen nicht verwendet. Im Quellcode für das Projekt **apigid32.dll** (in C++) finden Sie Beispiele für die Verwendung dieser Parameter in benutzerdefinierten DLLs. (Das Projekt **apigid32.dll** ist auf der Begleit-CD-ROM zu diesem Buch enthalten.)

Benutzerdefinierte Typen

Benutzerdefinierte Typen werden als Verweise an API-Funktionen übergeben, es wird also ein Zeiger auf die Struktur als Parameter übergeben. In den meisten Fällen ist die Konvertierung einer C-Struktur in einen benutzerdefinierten

Visual Basic-Datentyp einfach, es gibt jedoch Ausnahmen. Eine ausführliche Erläuterung zur Behandlung von Strukturen finden Sie in Tutorium 7, »Klassen, Strukturen und benutzerdefinierte Typen«.

Der Datentyp »Variant«

Die Kernfunktionen der Win32-API verwenden keine `Variant`-Werte. Das OLE-Subsystem unterstützt jedoch `Variant`-Werte, und diese Varianten sind vollständig kompatibel mit Visual Basic.

Wenn Sie in Visual Basic `Variant`-Werte verwenden, werden diese wie jede andere Variable auch behandelt. In DLL-Funktionen werden `Variant`-Werte jedoch als 16-Bit-VARIANT-Strukturen betrachtet.

Im Gegensatz zu den benutzerdefinierten Typen können Sie `Variant`-Werte mit Hilfe des Schlüsselwortes `ByVal` als Werte übergeben. Dies rührt daher, dass Visual Basic Kopien der `Variant`-Werte erstellen kann.

`Variant`-Werte werden in der C-Dokumentation als VARIANT- oder VARIANTARG-Strukturen bezeichnet.

Arrays

Sie haben bereits erfahren, wie Arrays im Hinblick auf Win32-API-Funktionen verwendet werden. Erinnern Sie sich daran, dass eine C-Zeichenfolge nichts weiter als ein Zeichenpuffer ist, der auf NULL endet, und dass ein Zeiger auf eine Zeichenfolge lediglich einen Zeiger auf das erste Zeichen in der Zeichenfolge ist? API-Funktionen verwenden diesen Ansatz mit beliebigen Arraytypen. Ein `integer`-Array wird in einer C-Deklaration z. B. üblicherweise als `LPINT` oder `PINT` dargestellt.

Angenommen, Sie verfügten über eine API-Funktion, die ein Array mit fünf `short`-Werten erwartet. In Visual Basic könnten Sie dieses Array folgendermaßen definieren:

```
Dim myArray(4) As Integer
```

Die API-Funktion erwartet einen Zeiger auf das Array, d. h., einen Zeiger auf das erste Element im Array. Wie bekommen Sie nun einen Zeiger, der auf das erste Element des `myArray`-Arrays verweist? Sie haben zwei Möglichkeiten, wie bereits in Tutorium 5 erläutert:

- Deklarieren Sie den Parameter `As Integer`, und übergeben Sie das erste Element im Array folgendermaßen:

  ```
  Declare Function MyFunction Lib "mydll.dll" (ArrayPointer As Integer) As Long
  Call MyFunction(myArray(0))
  ```

- Deklarieren Sie den Parameter `ByVal As Long`, und übergeben Sie die Adresse des ersten Elements, die Sie mit Hilfe des `VarPtr`-Operators abgerufen haben:

```
Declare Function MyFunction Lib "mydll.dll" (ByVal ArrayPointer As Long) _
As Long
Call MyFunction(VarPtr(myArray(0)))
```

Die erste Methode wird für API-Funktionen häufiger eingesetzt.

Objekte

COM-Objekte (Windows Common Object Model) sind die Basis, auf der Visual Basic und das Windows OLE-Subsystem aufbauen. Die COM-Objekte werden von den Kernfunktionen der Win32-API nicht verwendet, dennoch gibt es Fälle, in denen Sie Objekte oder Objektzeiger an die OLE-API-Funktionen übergeben müssen.

Auf ein Objekt wird stets durch eine Schnittstelle verwiesen. Logisch gesehen handelt es sich bei einer solchen Schnittstelle um eine Reihe von Funktionen, die miteinander in Verbindung stehen und zur Durchführung von Operationen für das Objekt verwendet werden können. Ein Objekt kann viele Schnittstellen offenlegen, und jede Schnittstelle verfügt über die Funktion `QueryInterface`, mit der eine andere Schnittstelle des Objekts angefordert werden kann.

Aus der OLE-Dokumentation geht hervor, dass Schnittstellennamen stets mit dem Buchstaben I beginnen. Die zwei wichtigsten Schnittstellen sind `IUnknown` und `IDispatch`.

`IUnknown` stellt tatsächlich einen Teil jeder Schnittstelle dar. Über `IUnknown` wird die Verweiszählung gesteuert, sodass Objekte verfolgen können, wann Sie sich selbst löschen können. Des Weiteren umfasst diese Schnittstelle die Funktion `QueryInterface`, die es einem Programmierer ermöglicht, zwischen den Schnittstellen zu navigieren. Der Variablentyp `IUnknown` wird zwar von Visual Basic unterstützt, wenn Sie jedoch ein Objekt in eine `IUnknown`-Variable einfügen, können Sie dennoch von Visual Basic aus nicht direkt auf die Methoden zugreifen.

```
Dim myObject As IUnknown
Set myObject = any object
```

Sie verwenden den `IUnknown`-Typ zusammen mit einigen OLE-API-Aufrufen, bei denen die Funktion einen Schnittstellenzeiger unbekannten Typs abruft.

Bei `IDispatch` handelt es sich um eine Schnittstelle, die die so genannte Automatisierungstechnologie unterstützt (gelegentlich auch als OLE-Automatisierung oder ActiveX-Automatisierung bezeichnet). Wenn Sie ein Objekt einer

Visual Basic-Variablen vom Typ `Object` zuordnen, enthält diese einen Zeiger auf die `IDispatch`-Schnittstelle dieses Objekts.

```
Dim myObject As Object
Set myObject = an object that supports the IDispatch interface
```

Intern handelt es sich bei einer Schnittstelle um eine Tabelle mit Zeigern auf Funktionen. Übergeben Sie ein Objekt als Parameter, wird tatsächlich ein Zeiger auf diese Tabelle übergeben. Sowohl der Variablentyp `IUnknown` als auch der Typ `IDispatch` speichern einen Zeiger auf eine Schnittstelle, daher handelt es sich bei beiden um 32-Bit-Werte. Sie können den Visual Basic-Operator `ObjPtr` zum Abrufen der Werte dieses Zeigers verwenden, obwohl selten auf diese Methode zurückgegriffen wird.

In der C/C++-Dokumentation finden Sie im Allgemeinen zwei Objektparametertypen:

- **IUnknown * oder LPUNKNOWN**
  Ein Zeiger auf eine Schnittstelle. Ihre Visual Basic-Deklaration für diesen Typ sollte `ByVal As Object` lauten. Warum? Weil die Objektvariable den Zeiger auf das Objekt enthält, und Sie möchten diesen Zeiger als Parameter übergeben.
- **IUnknown **, LPUNKNOWN ***
  Ein Zeiger auf einen Zeiger auf eine Schnittstelle.
  Die zwei Sterne (**) mögen zunächst etwas verwirrend erscheinen – Sie wissen jedoch bereits, dass ein Stern (*) für einen Zeiger steht. Das Zeichen ** kennzeichnet einen Zeiger auf eine Variable, die einen Zeiger enthält. Die Visual Basic-Objektvariable enthält einen Zeiger. Sie können einen Zeiger auf diese Variable abrufen, indem Sie den Operator `VarPtr` verwenden oder den Parameter als Verweis übergeben. Daher lautet die Visual Basic-Deklaration für diesen Parametertyp üblicherweise `As Object` (nicht `ByVal`). API-Funktionen verwenden diesen Parametertyp häufig, wenn eine Variable geladen werden soll, die eine neue Schnittstelle zu einem Objekt enthält.

COM ist eine der Technologien, die recht einfach sind, wenn man sie verstanden hat. Zu einem solchen Verständnis zu gelangen, kann jedoch sehr schwierig sein. Eine vollständige Besprechnung der COM- und Objekttechnologie kann im Rahmen dieses Tutoriums leider nicht erfolgen.[5]

Damit sind wir am Ende der Einführung in die Übersetzung von C-Headerdateien in Visual Basic angelangt. Dieses Thema wird in den Puzzlen und in den folgenden Tutorien noch weiter ausgeführt.

5. Dieses Thema wird in meinem Buch »Dan Appleman's Developing COM Components with Visual Basic 6.0: A Guide to the Perplexed« ausführlich erläutert.

Tutorium 7

Klassen, Strukturen und benutzerdefinierte Typen

Vom Standpunkt eines Visual Basic-Programmierers aus gesehen besteht zwischen den C-Datentypen `class` und `struct` und den benutzerdefinierten Typen von Visual Basic kein Unterschied. Mit all diesen Datentypen können Variablen im Speicher gruppiert werden, damit diese als Einheit als einzelne Struktur im Speicher behandelt werden können. Viele Win32-API-Funktionen verwenden Strukturen als Parameter. Die einzelnen Variablen innerhalb einer Struktur werden gelegentlich auch als »Felder« der Struktur bezeichnet.

Die Sprache C ist in der Lage, Strukturen als Wert zu übergeben, wobei die gesamten Strukturdaten im Stack platziert werden. In Visual Basic ist dies nicht möglich, jedoch auch nur selten erforderlich. Die meisten Win32-API-Funktion erwarten die Übergabe von Strukturen als Verweis. Es wird also ein Zeiger auf eine Struktur als Stackparameter erwartet.

Der einzige Fall, in dem eine Struktur unter Win32 als Wert übergeben werden muss, ist die `POINT`-Struktur.

Behandlung von Zeigern

Die `POINT`-Struktur wird in C und in Visual Basic wie folgt deklariert:

C/C++	Visual Basic
`typedef struct  tagPOINT`	`Type POINTAPI`
`{`	`        x As Long`
`    LONG x;`	`        y As Long`
`    LONG y;`	`End Type`
`}  POINT, *PPOINT, NEAR *NPPOINT, FAR *LPPOINT;`	

Sie werden bemerken, dass der Name der Visual Basic-Struktur nicht `POINT` lautet. Das rührt daher, dass »`Point`« ein für Visual Basic reserviertes Wort ist. Es tritt ein Fehler auf, wenn Sie versuchen, einem benutzerdefinierten Typ einen Namen zuzuweisen, bei dem es sich gleichzeitig um ein reserviertes Wort handelt.

Die POINT-Struktur zeigt eine andere Form der typedef-Anweisung. Der offizielle Name dieser Struktur lautet tagPOINT, wenn Sie jedoch tagPOINT in Ihrem C-Code verwenden, müssen Sie das Wort »struct« voranstellen:

```
struct tagPOINT myStruct;
```

Durch Verwenden der typedef-Version können Sie eine Strukturenvariable deklarieren:

```
POINT myStruct.
```

PPOINT, NPPOINT und LPPOINT sind allesamt Zeiger auf POINT-Strukturen.

Eine weitere häufig verwendete Struktur ist die RECT-Struktur, die in Visual Basic folgendermaßen definiert ist:

```
Type RECT
        Left As Long
        Top As Long
        Right As Long
        Bottom As Long
End Type
```

Die RECT-Struktur beschreibt die Punkte eines Rechtecks.

Sehen Sie sich nun die nachstehende Deklaration der PtInRect-Funktion an, mit der ermittelt werden kann, ob sich ein bestimmter POINT innerhalb eines vorgegebenen Rechtecks befindet:

```
PtInRect(CONST RECT *lprc, POINT pt);
```

Der erste Parameter ist leicht zu verstehen. Der Wert CONST kann von Visual Basic ignoriert werden – hiermit wird lediglich angegeben, dass die Funktion keinerlei Änderungen am Inhalt der als Parameter übergebenen RECT-Struktur vornimmt. RECT * gibt an, dass es sich bei lprc um einen Zeiger auf eine RECT-Struktur handelt. Das Visual Basic-Äquivalent besteht darin, die RECT-Struktur als Verweis zu übergeben.

Wie sieht es mit dem POINT-Parameter aus? Es ist kein Hinweis auf einen Zeiger vorhanden, es sieht demnach so aus, als müsse die gesamte Struktur als Wert übergeben werden. Visual Basic kann jedoch Strukturen nicht als Wert übergeben, d.h., wir haben ein Problem.

Sehen Sie sich zur Lösungsfinden den Stackframe in Abbildung T7-1 an. Bei der Anordnung der Daten innerhalb von Strukturen werden die ersten Felder immer an der untersten Speicherstelle platziert. Da der Stack in Richtung des unteren Speichers wächst, wird eine Struktur, die als Wert übergeben wird, wie in der Ab-

bildung angeordnet – mit dem X-Feld an der höheren Speicherstelle (niedrigeren Speicheradresse) als das Y-Feld. Wenn Sie die Vorstellung eines Stacks, das in Richtung des unteren Speichers wächst, verwirrt, sehen Sie sich erneut Tutorium 4, »Die Arbeitsweise einer DLL: In einem Stackframe«, an.

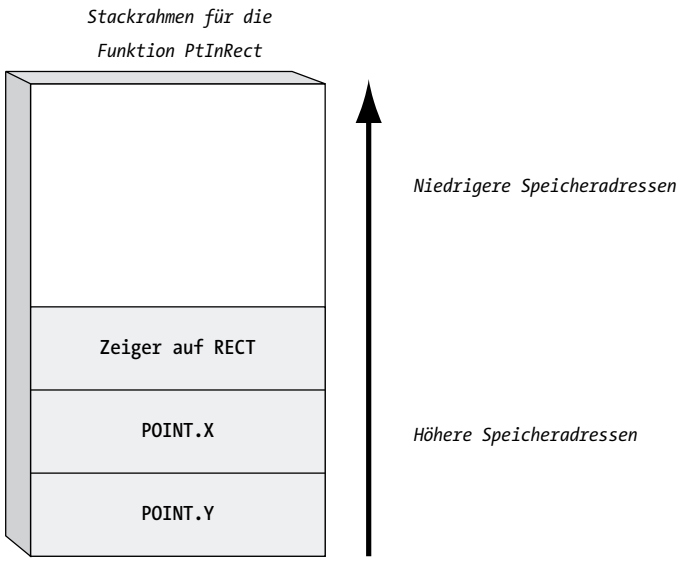


Abbildung T7-1 Ein Stackframe mit einer POINT-Struktur, die als Wert übergeben wird

Die Übergabe einer POINT-Struktur ist nun nicht mehr ganz so problematisch. Sie müssen sich lediglich fragen, welche Art der Visual Basic-Deklaration diesen Stackframe erzeugen kann. Die Antwort ist folgende:

```
Private Declare Function PtInRect Lib "user32" (lpRect As RECT, _
ByVal ptx As Long, ByVal pty As Long) As Long
```

Wie Sie sich vielleicht erinnern (▲Tutorium 4) werden Parameter in umgekehrter Reihenfolge im Stack platziert. Da wir die tatsächlichen Werte der X- und Y-Felder im Stack benötigen, übergeben wird die X- und Y-Felder separat als Wert. Die RECT-Struktur selbst wird als Verweis übergeben.

Der folgende Beispielcode aus dem Projekt **Struct.vbp** (auf der Begleit-CD-ROM zu diesem Buch) veranschaulicht die Verwendung der PtInRect-Funktion in Visual Basic.

```
Private Sub cmdPtInRect_Click()
   Dim r As RECT
   Dim pt As POINTAPI
   r.Right = 100
```

```
    r.Bottom = 100
    pt.x = 50
    pt.y = 50
    Debug.Print PtInRect(r, pt.x, pt.y)
    pt.x = 150
    Debug.Print PtInRect(r, pt.x, pt.y)
End Sub
```

Da es in Visual Basic nicht möglich ist, Strukturen als Wert zu übergeben, ist es erforderlich, die Felder der Struktur einzeln zu übergeben. Die Funktion kann jedoch aufgerufen werden.

Die gute Nachricht hierbei ist, dass die Strukturen `POINT` und `SIZE` (diese beiden sind bis auf unterschiedliche Feldnamen identisch) die einzigen sind, die als Wert an eine Win32-API-Funktion übergeben werden müssen.

Strukturen und DLL-Aufrufe

Bevor wir fortfahren, müssen wir eine wichtige Tatsache berücksichtigen. Die Mehrheit der Visual Basic-Programmierer wird nie auf die im verbleibenden Teil dieses Tutoriums besprochenen Probleme stoßen. In den meisten Fällen ist die Behandlung von Strukturen so einfach wie im folgenden Beispiel, in dem die Verwendung der API-Funktion `GetClientRect` zum Abrufen der Dimensionen des Fensters eines Clientbereiches veranschaulicht wird:

```
Type RECT
        Left As Long
        Top As Long
        Right As Long
        Bottom As Long
End Type

Declare Function GetClientRect Lib "user32"  _
(ByVal hwnd As Long, lpRect As RECT) As Long

Dim rc As RECT
Call GetClientRect(myHwnd, rc)
```

Strukturparameter (mit Ausnahme der zuvor genannten Fälle) werden immer als Verweis übergeben. Auf diese Weise wird die Adresse des Strukturbeginns an die DLL-Funktion weitergereicht.

Unglücklicherweise gibt es Situationen, in denen sich die Behandlung von Strukturen sehr komplex gestaltet. Und zwar so komplex, dass einige der Puzzle in die-

sem Buch in direktem Zusammenhang mit Strukturdeklarationen und -parametern stehen. Die Problembereiche fallen hierbei unter die Kategorien der Zeichenfolgenbehandlung, Variablenausrichtung und die Kombination von Strukturen (Strukturen innerhalb von Strukturen und Strukturarrays).

Das Allerwichtigste

Das Endziel bei der Behandlung von Strukturen besteht darin, Visual Basic zu veranlassen einen Parameter als Zeiger auf einen Speicherblock zu übergeben, der exakt dem Format der erwarteten DLL-Funktion entspricht.

Aus dieser Forderung können Sie die folgenden Schlüsse ziehen:

- Der springende Punkt bei der Behandlung von Strukturen ist, die Art der Speicherung einer Struktur im Speicher sowie deren Bearbeitung während eines DLL-Aufrufs zu verstehen.
- Schlussendlich ist eine Behandlung jeder Struktur möglich, und zwar unabhängig davon, welchen Datentyp sie enthält. Im schlimmsten Fall können Sie einen eigenen Speicherpuffer zuweisen und die Daten mit dem erforderlichen Format in den Puffer kopieren.

Zeichenfolgen in Strukturen

Das erste Problem im Hinblick auf Zeichenfolgen innerhalb von Strukturen ergibt sich aus der Tatsache, dass in Visual Basic intern Unicode-Zeichen verwendet werden. Zeichenfolgen innerhalb von Strukturen werden in Visual Basic im Unicode-Format gespeichert.

Das Projekt **Struct.vbp** kann zu einem besseren Verständnis der Vorgänge in Visual Basic während eines DLL-Aufrufs herangezogen werden. Das Projekt definiert die folgende Struktur:

```
Private Type Struct1
   a As Integer
   b As Integer
   c As String * 4
   d As String
End Type
```

Das Projekt enthält die Funktion `ShowMemory`, die eine Speicheradresse als Parameter verwendet und den Inhalt einer bestimmten Anzahl Bytes im Hexadezimalformat druckt. Wir verwenden diese Funktion für eine Inhaltsanalyse des Speicherpuffers der Struktur.

```
' Speicherbytes an der angegebenen Adresse
' im Hexadezimalformat anzeigen
Private Sub ShowMemory(ByVal Address As Long, ByVal bytes As Integer)
   Dim b As Byte
   Dim x As Integer
   ' Print the address
   Debug.Print Hex$(Address) & ": ";
   ' Jedes Byte im Puffer Byte für Byte in eine
   ' temporäre Variable mit dem Namen 'b' kopieren
   For x = 0 To bytes - 1
      Call RtlMoveMemory(b, ByVal Address + x, 1)
      ' Hexadezimalwert von Byte 'b' anzeigen
      Debug.Print Hex$(b) & " ";
   Next x
   Debug.Print
End Sub
```

Mit der `cmdStruct1_Click`-Funktion kann sowohl untersucht werden, wie diese Struktur in Visual Basic gespeichert wird, als auch wie diese während eines DLL-Funktionsaufrufs übergeben wird. Die Felder »`a`« und »`b`« der Struktur werden jeweils auf die Hexadezimalwerte `&H1234` und `&H5678` gesetzt. Diese Zahlenwerte sind willkürlich und können leicht wiedererkannt werden. Beide Zeichenfolgen werden auf »`ABCD`« gesetzt:

```
Private Sub cmdStruct1_Click()
   Dim s As Struct1
   Dim b(40) As Byte
   s.a = &H1234
   s.b = &H5678
   s.c = "ABCD"
   s.d = "ABCD"
   Debug.Print "Structure contains on API call: "
   RtlMoveMemory b(0), s, Len(s)
   ShowMemory VarPtr(b(0)), Len(s)
   ' Warum können die Zeichenfolgendaten nicht anzeigt werden?
   ' Antwort - temporäre Puffer!
   Debug.Print "In VB it contains: "
   ShowMemory VarPtr(s), LenB(s)
End Sub
```

Die einzig verlässliche Methode, auf die man bei der Anzeige einer Struktur während eines API-Aufrufs zurückgreifen kann, ist die, einen API-Aufruf durchzuführen. In diesem Fall rufen wir die immer wieder nützliche `RtlMoveMemory`-Funktion auf. Der Zielparameter ist das erste Byte eines Bytearrays. Der Quellparameter ist die Struktur. Visual Basic übergibt einen Zeiger auf diese Struktur. Die `RtlMoveMemory`-Funktion kopiert die Strukturdaten in das Bytearray, damit wir dessen Inhalt anzeigen können. Die Ergebnisanzeige ist folgende:

```
Structure contains on API call:
17B168: 34 12 78 56 41 42 43 44 F4 2E 18 0
```

Die ersten zwei Byte sind `&H34` und `&H12`. Wie Sie vielleicht noch aus Tutorium 2, »Der Arbeitsspeicher – da, wo alles beginnt«, wissen, entspricht dies dem 16-Bit-Wert `&H1234`. Die Bytes erscheinen nur umgedreht, da das niedrigere Byte des Wertes im unteren Speicher erscheint, der im Speicherpufferabbild als Erstes angezeigt wird.

Der zweite `Integer`-Wert lautet `&H5678` und wird direkt nach dem ersten Wert angezeigt.

Als Nächstes sehen Sie die vier Zeichenwerte `&H41`, `&H42`, `&H42` und `&H44`. Es handelt sich hierbei um die ASCII-Zeichen für die Buchstaben »ABCD« – also um die feste Zeichenfolgenvariable »`c`«. Die letzen vier Bytes sind der Zeiger `&H182EF4` (dieser Wert lautet auf Ihrem System mit Sicherheit anders). Es handelt sich um einen BSTR-Zeiger auf eine ANSI-Zeichenfolge, die während des Aufrufvorgangs temporär erstellt wird. Wir werden uns später noch genauer mit dynamischen Zeichenfolgen innerhalb von Strukturen befassen.

Der zweite Teil der Funktion verwendet die `VarPtr`-Funktion zum Abrufen der Strukturadresse im Visual Basic-Speicher. Mit der `LenB`-Funktion wird die Größe der Struktur bei deren Speicherung in Visual Basic abgerufen. Eine Ausgabe (Dump) der Struktur in Visual Basic sieht folgendermaßen aus:

```
In VB it contains:
12F5E8: 34 12 78 56 41 0 42 0 43 0 44 0 CC 9A 17 0
```

Der einzige Unterschied besteht im Zeichenfolgenabschnitt mit fester Länge. Die Zeichenfolge wird mit den Zeichen 41 00 42 00 43 00 44 00 – erneut die Zeichenfolge »`ABCD`«, nur dass jedes Zeichen 2 Byte umfasst, also im Unicode-Format vorliegt.

Es lassen sich hieraus zwei Zusammenhänge ableiten:

- Zeichenfolgen mit fester Länge werden innerhalb von Strukturen gespeichert, für dynamische Zeichenfolgen dagegen wird nur ein 4-Byte-BSTR-Zeiger in der Struktur gespeichert.
- Zeichenfolgen in einer Struktur werden bei Verwendung von Visual Basic als Unicode, bei der Übergabe an einen DLL-Aufruf als ANSI gespeichert.

In Tutorium 5, »Das `ByVal`-Schlüsselwort: Die Lösung für 90 % aller API-Probleme«, wird erläutert, dass Zeichenfolgenparameter entweder im Unicode- oder ANSI-Format vorliegen, je nachdem, ob Sie den ANSI- oder Unicode-Einsprungpunkt einer Funktion verwenden. Das gleiche gilt häufig für Strukturen, die Zeichenfolgen enthalten. Die `LOGFONT`-Struktur beispielsweise (mit der eine logische Schriftart beschrieben wird) enthält eine Zeichenfolge mit fester Länge, in der der Name des Schriftbildes für die Schritfart gespeichert ist. Wenn Sie sich die C-Headerdateien ansehen (auf der Begleit-CD-ROM zu diesem Buch) werden Sie bemerken, dass tatsächlich zwei separate Strukturdefinitionen vorliegen, `LOGFONTA` und `LOGFONTW`. Die `LOGFONTA`-Struktur definiert eine Zeichenfolge als Array von Einzelbytezeichen, die Struktur `LOGFONTW` definiert die Zeichenfolge als Array von `wide`-Zeichen.

Sie sollten sich jedoch nicht darauf verlassen, dass dies immer zutrifft. Einige Funktionen verwenden stets ANSI-Zeichenfolgen, andere immer Unicode-Zeichen. Sie müssen die jeweilige Strukturdokumentation hinzuziehen, um zu wissen, welches Format verwendet wird.

Dynamische Zeichenfolgen in Strukturen

Eine dynamische Zeichenfolge wird in einer Visual Basic-Struktur als 32-Bit-Zeiger auf die Zeichenfolgendaten gespeichert. Es gibt verschiedene DLL-Strukturen in Windows, die Zeiger auf Zeichenfolgen enthalten. In einigen Fällen können Sie eine dynamische Zeichenfolge verwenden, wobei Sie jedoch vorsichtig sein sollten. Der dynamische Zeichenfolgenzeiger in der Struktur ist ein `BSTR`-Zeiger, der auf einen vom OLE-Subsystem verwalteten Speicherblock. Wenn dieser Zeigerwert durch die DLL überschrieben werden würde, könnte dies zu einer Speicherbeschädigung oder zu einem Speicherausnahmefehler führen. Daher ergeben sich für die Verwendung von dynamischen Zeichenfolgen innerhalb von Strukturen die folgenden Regeln.

Dynamische Zeichenfolgen können in den folgenden Fällen problemlos eingesetzt werden:

- Die aufzurufende Funktion erwartet einen `BSTR`-Zeiger in der Struktur (trifft meistens nur auf OLE-DLL-Aufrufe zu).
- Sie fügen am Ende der Zeichenfolge ein NULL-Zeichen ein, damit sie tatsächlich auf NULL endet.

Werden die Zeichenfolgendaten durch die DLL-Funktion geändert, initialisieren Sie die Zeichenfolge auf die benötigte Länge.

In den folgenden Fällen sollten Sie keine dynamischen Zeichenfolgen einsetzen:

- Die DLL-Funktion verändert den Wert des Zeigers in der Struktur und verwendet nicht das OLE-Subsystem. Dies ist häufig der Fall, wenn es sich bei dem Zeiger nicht um `BSTR *` handelt. Sie sollten daran denken, dass die meisten API-Funktionen außerhalb des OLE-Subsystems keine `BSTR`-Variablen verwenden, daher werden dynamische Zeichenfolgen nur selten innerhalb von solchen Strukturen eingesetzt, die von der Win32-API benutzt werden.

Anordnungsfragen

Im Projekt **Struct.vbp** wird die folgende Struktur definiert, die der im vorherigen Abschnitt gezeigten sehr stark ähnelt:

```
Private Type Struct2
   a As Integer
   b As Long
   c As String * 4
   d As String
End Type
```

Die `cmdStruct2_Click`-Funktion ist nahezu identisch mit der Funktion `cmdStruct1_Click`. Der einzige Unterschied besteht darin, dass in der zweiten Struktur das Feld »b« auf `&H56789ABC` gesetzt ist (da es sich um eine 32-Bit-Variable handelt) und dass der erste `RtlMoveMemory`-Aufruf zwei zusätzliche Bytes in das temporäre Bytearray kopiert (2 Byte mehr als die Länge, die durch die `Len`-Funktion zurückgegeben wird). Der Grund für die Aufnahme dieser zwei zusätzlichen Bytes wird später erläutert.

```
Private Sub cmdStruct2_Click()
   Dim s As Struct2
   Dim b(40) As Byte
   s.a = &H1234
   s.b = &H56789ABC
```

```
    s.c = "ABCD"
    s.d = "ABCD"
    Debug.Print "Structure contains on API call: "
    ' Anordnung erhöht die Anzahl der von Len zurückgegebenen Bytes
    RtlMoveMemory b(0), s, Len(s) + 2
    ShowMemory VarPtr(b(0)), Len(s) + 2
    ' Warum können die Zeichenfolgendaten nicht anzeigt werden?
    ' Antwort - temporäre Puffer!
    Debug.Print "In VB it contains: "
    ShowMemory VarPtr(s), LenB(s)
End Sub
```

Mit dem ersten Aufruf von `ShowMemory` wird der Inhalt der Struktur angezeigt, die als DLL-Parameter übergeben wurde.

```
Structure contains on API call:
187D98: 34 12 0 0 BC 9A 78 56 41 42 43 44 C 5B 16 0
```

Merkwürdigerweise sind zwischen dem ersten Feld »a« und dem Feld »b« zwei Nullbytes vorhanden. Die verbleibenden Daten entsprechen genau den zuvor erhaltenen. Woher stammen diese zwei Zusatzbytes?

Sehen wir uns den Inhalt der Struktur nach einer Speicherung in Visual Basic an:

```
12F5E4: 34 12 0 0 BC 9A 78 56 41 0 42 0 43 0 44 0 F4 2E 18 0
```

Die Zeichenfolge liegt nach wie zvor im Unicode-Format vor, und die zwei zusätzlichen Bytes sind weiterhin vorhanden.

Die Zusatzbytes sind deshalb vorhanden, da die einzelnen Felder innerhalb einer Struktur in Visual Basic gemäß bestimmter Anordnungsregeln gespeichert werden. Diese Regeln besagen, dass jedes Feld an seinem üblichen Grenzbereich beginnen muss. Mit anderen Worten, die Adresse des ersten Bytes im Feld muss an der Adresse starten, die durch die Länge des Feldes teilbar ist. Das bedeutet, dass Einzelbytefelder an einer beliebigen Adresse beginnen können, numerische 16-Bit-Felder an einer geraden Adresse (2-Byte-Grenze) und 32-Bit-`Long`-Variablen oder -Zeiger an einer Adresse, die ein Vielfaches von vier darstellt (4-Byte-Grenze).

Visual Basic verwendet diese natürliche Anordnung intern und während eines DLL-Funktionsaufrufs. Dies führt in einigen Fällen zu Problemen, da die meisten API-Funktionen eine Einzelbyteanordnung vornehmen, bei der alle Felder aneinandergereiht werden. Dies wird deutlich in Abbildung T7-2, in der gezeigt wird, wie die `struct2`-Struktur auf verschiedene Arten gespeichert wird.

Das erste Beispiel zeigt die interne Speicherung der Struktur, bei der aufgrund der Anordnung jeweils 2 Bytes zusammengefasst und die Zeichenfolgen im Unicode-Format gespeichert werden. Die Länge der Struktur bei der internen Speicherung kann mit der `LenB`-Funktion festgelegt werden. Das zweite Beispiel zeigt, wie die Struktur im Speicher gespeichert wird, wenn Sie als DLL-Aufruf übergeben wird. Die Zeichenfolge wird in das ANSI-Format konvertiert, die Regeln der natürlichen Anordnung gelten jedoch weiterhin. Das untere Beispiel zeigt die Struktur, wenn diese unter Verwendung der Visual Basic-Funktion `Put` geschrieben und in einer Datei gespeichert würde. Wie Sie sehen können, werden alle Felder ohne Füllbytes nebeneinander angeordnet (Einzelbyteanordnung). Die Länge der Struktur bei der Einzelbyteanordnung kann mit Hilfe der `Len`-Funktion gesteuert werden. Wie können Sie die Länge einer Struktur bestimmen, die als DLL-Parameter übergeben wird? Dies müssen Sie selbst herausfinden – es ist keinerlei Funktion verfügbar, mit der dieser Wert berechnet werden könnte.

Was geschieht, wenn eine durch eine DLL-Funktion verwendete Struktur eine Einzelbyteanordnung benötigt? Dies ist tatsächlich der Standard für nahezu sämtliche Win32-API-Strukturen. Daher könnte man meinen, dies sei ein ernsthaftes Problem. Glücklicherweise waren die Programmierer von Windows bei der Definition von Strukturen in den meisten Fällen so umsichtig, dass diese im Speicher sowohl in der Einzelbyteanordnung als auch in der natürlichen Anordnung vorliegen. Daher werden Ihnen in Win32-API-Strukturen gelegentlich reservierte Bytes begegnen. Durch diese werden explizit die Füllbytes eingefügt.

Beispielsweise könnte dies für die `struct2`-Struktur durch folgende Definition geschehen:

```
Private Type Struct2
   a As Integer
   reserved as Integer
   b As Long
   c As String * 4
   d As String
End Type
```

Diese Struktur liegt im Speicher sowohl in der Einzelbyte- als auch in der natürlichen Anordnung vor (bearbeiten Sie das Projekt **Struct.vbp**, und probieren Sie es aus).

Unglücklicherweise wurde dieses Prinzip von Microsoft in einigen Fällen verworfen, sodass die Strukturen nicht immer den Regeln der natürlichen Anordnung entsprechen. Es stehen verschiedene Methoden zur Verfügung, um solche Situationen zu meistern. Sie können den Visual Basic-Typ neu definieren, um die An-

ordnungsprobleme zu beheben – beispielsweise, indem Sie eine `Long`-Variable in zwei `Integer`-Variablen aufteilen. Sie können die Struktur temporär in einen Speicherpuffer wie z. B. ein Bytearray kopieren und Teile der Struktur explizit an die gewünschten Speicherstellen kopieren. Sie können ein Drittanbieterprodukt wie SpyWorks von Desaware verwenden. Dieses enthält verschiedene Funktionen zur Byteanordnung, mit denen eine automatische Einzelbyteanordnung durchgeführt oder auch rückgängig gemacht werden kann.

Damit schließen wir die Einführung in die Behandlung benutzerdefinierter Typen ab. Viele der Puzzle in diesem Buch behandeln Probleme, die im Zusammenhang mit diesem Thema stehen, beispielsweise die Behandlung von Zeigern auf Zeichenfolgen und andere benutzerdefinierte Typen, Anordnungsfragen und das Arbeiten mit Strukturarrays.

Tutorium 8

Portieren von C-Headerdateien

Das Puzzle 29, »Was tun, wenn's richtig weh tut?«, veranschaulicht am Beispiel der `cpuspeed`-Funktion der Intel-DLL **cpuinf32.dll** die Verwendung von DLL-Funktionen, die Strukturen zurückgeben. Diese DLL ermöglicht es Ihnen, sämtliche interessanten Informationen zu den Prozessoren auf Ihrem Computer abzurufen (zumindest dann, wenn Sie einen Intel-Prozessor verwenden).

Dieses Buch stellt die DLL (auf CD-ROM) sowie ausreichende Informationen zu deren Verwendung im Puzzle bereit. Sie können die aktuellste Version der DLL und den vollständigen Quellcode für DLLs und verschiedene in C++ geschriebene Testprogramme in der Intel-Website unter **http://developer.intel.com/design/perftool/cpuid/** herunterladen. (Beachten Sie bitte, dass keine Garantie dafür übernommen werden kann, dass diese Datei in der Zukunft weiterhin verfügbar ist oder am genannten Standort verbleiben wird. Aktuelle Informationen im Falle einer Änderung finden Sie unter **http://www.desaware.com/**.)

In diesem Tutorium wird gezeigt, wie die ursprünglichen C-Headerdateien zur Verwendung der cputest-Anwendung in Visual Basic konvertiert wurden.

Keine Angst, die Lektüre dieses Tutoriums wird Ihnen nicht den Spaß an dem Puzzle verderben.

Schritt 1: Laden der Headerdatei in ein Visual Basic-Modul

Überlegen Sie einen Moment, welche Art von Informationen üblicherweise in einem Visual Basic-Standardmodul enthalten sind:

- Globale Variablendeklarationen
- Definitionen für benutzerdefinierte Typen (in C++ als »Strukturen« bezeichnet)
- Globale Funktionen und Subroutinen
- Kommentare zur Erläuterung der verschiedenen Modulelemente

Besonders wichtig sind hier die Strukturdefinitionen sowie die Funktionsdeklarationen. Von einer DLL aus können Sie nicht wirklich auf globale Variablen zugreifen. Und obwohl Kommentare für die Funktionsweise eines Moduls unerlässlich sind, besteht kein funktionaler Unterschied zwischen einem Kommentar in Visual Basic und dem in C++.

Genauso wie in Visual Basic verschiedene Dateitypen verwendet werden (Module, Formulare, Klassen usw.), verhält es sich auch 'mit den in C oder C++ geschriebenen Projekten. Der wichtigste Dateityp für das Verständnis einer DLL ist die Headerdatei, die üblicherweise über die Erweiterung **.h** verfügt. Eine Header-

datei enthält im Allgemeinen die gleichen Informationen wie das Standardmodul, abgesehen davon, dass es selten größere Codeabschnitte enthält. Statt dessen ist eine Deklaration für jede Funktion enthalten, die sich in der eigentlichen Quelldatei befindet (diese weist die Erweiterung **.C** oder **.CPP** auf, je nachdem, ob das Programm in C oder in C++ geschrieben wurde).

Hierbei handelt es sich um eine Vereinfachung; C++-Headerdateien können eine Vielzahl von Zusatzinformationen enthalten, aber diese werden Sie kaum benötigen, um von Visual Basic aus auf die DLL zuzugreifen. Komplexe DLLs können über Dutzende separater Header- und Quelldateien verfügen. Die einzigen Headerdateien, die Sie benötigen, sind diejenigen, die Strukturdefinitionen und Funktionsdeklarationen zur Verwendung durch externe Anwendungen enthalten. Die Headerdateien, die nur innerhalb der DLL verwendet werden, können Sie ignorieren. Nachdem Sie die benötigten Headerdateien ermittelt haben, brauchen Sie diese nur in Visual Basic zu konvertieren, und Sie können auf die DLL-Funktionen zugreifen. Und Ihnen die richtige Konvertierung dieser Headerdateien beizubringen, ist eines der Hauptziele dieses Buches.

Warum sollte es erforderlich sein, mit C/C++-Headerdateien umgehen zu können? Warum sollte man sich nicht einfach auf die DLL-Dokumentation verlassen?

Hierfür gibt es zwei grundlegende Erklärungen.

Erstens werden in die Dokumentation der meisten DLLs keine tatsächlichen Strukturdefinitionen oder Funktionsdeklarationen eingefügt. Warum? Da diese DLLs üblicherweise sowieso von anderen C/C++-Programmen aufgerufen werden. Es ist sehr viel einfacher, einem Programmierer zu sagen, dass er eine Headerdatei in sein Projekt einschließen soll, die zusammen mit der DLL geliefert wird.[1]

Auf diese Weise kann der Programmierer die DLL-Funktionen ohne weiteren Aufwand direkt in der eigenen Anwendung aufrufen. Dies ist ein Vorteil, den die C/C++-Programmierer gegenüber den Visual Basic-Programmierern genießen – jeder stellt C/C++-Headerdateien zusammen mit seinen DLLs bereit. Aber nicht jeder stellt auch ein Visual Basic-Modul mit den richtigen Deklarationen und Strukturdefinitionen bereit.

Der zweite Grund gilt auch für die Win32-API. Da die Headerdateien primär zur Verwendung durch Anwendungen und nicht als Lektüre für die Programmierer gedacht sind, enthalten sie kaum Fehler. Wenn sie dies doch täten, würden die Programme, mit denen sie verwendet werden, nicht funktionieren. Eine nicht

1. C-Programme sind in der Lage, Headerdateien mit Hilfe des Befehls `#include` »einzuschließen«. Über diesen Befehl wird während der Kompilierung eine Headerdatei in das Programm eingefügt. Im Moment gibt es zu dieser Operation in Visual Basic leider noch kein Äquivalent.

ganz korrekte Dokumentation führt vielleicht hier und da zu einem Nervenzusammenbruch bei einem Programmierer, hat aber ansonsten keinerlei Auswirkungen.

Die Intel-DLL **cpuinf32.dll** umfasst zwei Headerdateien, **speed.h** und **cpuid.h.** Der erste Schritt auf dem Weg hin zu einer Konvertierung dieser Dateien besteht darin, sie in Visual Basic-Module zu kopieren.

Schritt 2: Eliminieren unnötiger Informationen

Eine Headerdatei kann Informationen enthalten, die nur während der Erstellung einer DLL verwendet werden bzw. es handelt sich um Informationen, die von den Nutzern der DLL benötigt werden. Dies trifft auf die Headerdateien **cpuid.h** und **speed.h** zu. Das Löschen von privaten Informationen erscheint hier ratsam, da Sie sich die Mühe sparen, Code zu konvertieren, der niemals verwendet wird. In der Datei **cpuid.bas** wird der nicht benötigte Code, anstatt ihn zu löschen, lediglich als Kommentar gekennzeichnet, um diesen Vorgang zu veranschaulichen. Wie ermitteln Sie aber nicht benötigten Code?

Suchen nach privaten Funktionen

Eine Möglichkeit zur Ermittlung privater Funktionen besteht darin, sich die Kommentare innerhalb der Headerdatei selbst anzusehen. In der Datei **cpuid.h** ist dies sehr einfach. Etwa in der Mitte der Datei findet sich folgende Zeile:

```
// Private Function Declarations ////////////////////////////////'
```

Diese Zeile lässt vermuten, dass der sich anschließende Abschnitt privat ist, und dies ist auch tatsächlich der Fall. Sämtlicher Code, der nach dieser Zeile folgt, ist als Kommentar gekennzeichnet und kann gelöscht werden.

Sie können auch das Dienstprogramm **dumpinfo.exe** (auf der Begleit-CD-ROM zu diesem Buch) verwenden, um zu sehen, welche Funktionen tatsächlich von der DLL exportiert werden. Alle Funktionen, die nicht in der Exportliste aufgeführt werden, sind per Definition privat.

Mit Hilfe einer bedingten Kompilierung nach ausgeschlossenem Code suchen

Visual Basic unterstützt bei Verwendung der Anweisungen `#if`, `#else`, `#endif` die bedingte Kompilierung. C und C++ unterstützen diese und einige andere Begriffe für die bedingte Kompilierung. Zu Beginn oder im oberen Abschnitt vieler C/C++-Headerdateien findet sich Code wie dieser:

```
#ifndef cpuid_h
#define cpuid_h
```

Gegen Ende der Datei findet sich folgende Zeile:

```
#endif cpuid_h
```

Die #ifndef-Anweisung ist gleichbedeutend mit »Kompiliere die folgenden Zeilen, wenn die nachfolgende Variable nicht definiert ist.«. Die #ifdef-Anweisung hat eine ähnliche Bedeutung, hierbei wird jedoch der Code nur dann kompiliert, wenn die Variable definiert ist. Wie Sie sehen, ist es in C im Gegensatz zu Visual Basic möglich, eine bestimmte Headerdatei mehrfach in ein Projekt einzuschließen. Das Einschließen einer Datei in C wird durch die Verwendung der Befehle #include <Dateiname> in der Quelldatei erreicht. Auf diese Weise wird der Compiler angewiesen, die angegebene Datei während der Kompilierung zu prüfen. Angenommen, Sie verfügten über zwei Headerdateien, **myfile1.h** und **myfile2.h**, sowie über eine einzige Quelldatei **mysourcefile.ccp**. Als Nächstes nehmen wir an, dass die Datei **mysourcefile.ccp** beide Headerdateien enthält, d.h., bei der Kompilierung wurden zunächst die zwei Headerdateien durch den Compiler geprüft, und zwar so, als handelte es sich um einen Teil der Quelldatei. Nehmen wir des Weiteren an, dass **myfile2.h** außerdem **myfile1.h** enthält. Bei der Erstellung der Datei **mysourcefile.cpp** durch den Compiler findet dieser den Befehl #include »myfile1.h« und prüft daher sofort die Headerdatei. Anschließend wir der Befehl #include »myfile2.h« ermittelt, und der Compiler versucht, die Datei zu prüfen. Bei der Prüfung dieser Datei wird ein weiterer Befehl zum Einschließen von **myfile1.h** entdeckt, woraufhin die Datei ein zweites Mal gescannt wird. Dies kann zu Problemen führen, da die C/C++-Sprache es nicht vorsieht, bestimmte Elemente mehrmals zu deklarieren.

Die Zeilen für die bedingte Kompilierung in diesem Beispiel verhindern, dass die Datei **cpuid.h** während einer vorgegebenen Kompilierung mehr als einmal geprüft wird. Bei der ersten Dateiprüfung wird die Zeile #ifndef cpuid_h erfolgreich ausgeführt – die Variable cpuid_h ist noch nicht definiert. Anschließend wird die Konstante mit Hilfe der #define-Anweisung definiert (obwohl dieser kein Wert zugewiesen wird). Bei der nächsten Dateiprüfung kann die Zeile #ifndef cpuid_h nicht erfolgreich ausgeführt werden, da die Variable definiert ist, daher wird der verbleibende Code in der Headerdatei ignoriert.

Was für eine Bedeutung hat all dies für einen Visual Basic-Programmierer? Nur eine sehr geringe, da das Konzept der einmaligen Dateiprüfung zur Kompilierungszeit in Visual Basic nicht existiert. Daher kann jeglicher Code dieses Typs gelöscht werden.

Löschen nicht benötigter Funktionsdeklarationen

Es können Funktionsdeklarationen vorliegen, die Sie nicht für Ihre Anwendung benötigen. Es ist unnötig, diese Deklarationen zu portieren, es sei denn, Sie wis-

sen, dass Sie die Deklarationen eventuell zu einem späteren Zeitpunkt brauchen. Einige der exportierten Funktionen werden nur durch das Betriebssystem verwendet, es ist demnach unwahrscheinlich, dass Sie diese Funktionen von Visual Basic aus aufrufen werden. Hierzu gehören folgende Routinen:

- **DllMain.** Die DLL-Initialisierungsroutine wird vom Betriebssystem aufgerufen, jedoch niemals direkt.
- **DllCanUnloadNow.** Diese Routine wird durch das Betriebssystem aufgerufen, um zu ermitteln, ob ein ActiveX-Server entladen werden kann.
- **DllGetClassObject.** Diese Routine wird durch das Betriebssystem aufgerufen, um ein ActiveX-Serverobjekt zu erstellen.
- **DllRegisterServer.** Diese Routine wird durch Anwendungen oder das Betriebssystem aufgerufen, um einen ActiveX-DLL-Server zu registrieren.
- **DllUnregisterServer.** Diese Routine wird durch Anwendungen oder das Betriebssystem aufgerufen, um die Registrierung für einen ActiveX-DLL-Server aufzuheben.

Wenn Sie Deklarationen für diese Funktionen in der Headerdatei finden, können Sie diese löschen.

Schritt 3: Portieren der Kommentare

Das Portieren der Kommentare stellt wohl den einfachsten Teil des Konvertierungsvorgangs dar. Die Kommentarte in C und C++ werden auf zwei verschiedene Arten gekennzeichnet. Der Beginn eines längeren Kommentarabschnitts wird durch die Zeichenfolge `/*` eingeleitet und durch die Zeichenfolge `*/` beendet. Kommentare dieser Form können sich über mehrere Zeilen erstrecken. Gültige Kommentare lauten beispielsweise folgendermaßen:

```
/* Kommentar */
/* erste Kommentarzeile
zweite Kommentarzeile
* eine weitere Kommentarzeile

Der Kommentar endet hier */
```

Kommentare, die nur eine Zeile umfassen, werden durch die Zeichen `//` gekennzeichnet, die an einer beliebigen Stelle in einer Zeile eingefügt werden können. Diese Zeichenfolge funktioniert genau wie die Kennzeichnung durch `(')` in Visual Basic. Gültige Kommentare lauten beispielsweise folgendermaßen:

```
// Eine vollständige Kommentarzeile.
#define myconstant 0x55 // Kommentar beginnt mit //-Zeichen.
```

Das Konvertieren von Kommentaren ist nicht besonders aufregend und durch umsichtige Verwendung des Editor-Befehls `Ersetzen` leicht durchzuführen. Die Zeichenfolge `//` kann direkt durch das Zeichen zur Kommentarkennzeichnung `(')` ersetzt werden. Mehrzeilige Kommentare werden üblicherweise manuell geändert.

Schritt 4: Portieren der Konstanten

Die Konvertierung von Konstanten kann sehr einfach, aber auch sehr komplex sein. Konstantendeklarationen weisen die folgende Form auf:

```
#define constantname value_expression
```

Bei dem Begriff `value_expression` kann es sich um einen Ausdruck mit einer oder mehreren weiteren Konstanten handeln (einschließlich zuvor definierter Konstanten). Es können arithmetische Standardoperationen verwendet werden, wie beispielsweise der `|`-Operator (bitweises `ODER`) und der `&`-Operator (bitweises `UND`). Sie werden außerdem möglicherweise die Operatoren >> und << finden (`shift right` und `shift left`, wodurch die einzelnen Bits im Ausdruck um die Anzahl Positionen verschoben werden, die über den Operator angegeben wird).

Hexadezimalwerte in C/C++ erhalten anstelle von `&H` als Präfix die Sequenz `0x` oder `0X`. Konstanten können das Suffix L aufweisen, mit dem angegeben wird, dass es sich um einen `Long`-Wert handelt. Ausdrücke sind nicht auf numerische Werte begrenzt, sie können sich auch aus Zeichenfolgen oder generischem Text zusammensetzen. Bei der Kompilierung wird der gesamte Ausdruck `value_expression` durch den Begriff `constantname` ersetzt, wodurch jede Art von Text, der sich an dieser Position befindet, weiterhin verwendbar ist.

Deklarationen werden in die folgende generische Visual Basic-Form konvertiert:

```
Public Const constantname = constant_value
```

Sie können den gleichen Konstantennamen verwenden, Sie müssen jedoch den Wert der Konstante basierend auf der Headerdatei ermitteln. Ein Beispiel aus der Datei **cpuid.h:**

```
#define VERSION 0x0101  // Muss 2 Byte Länge aufweisen.
Public Const VERSION = &H0101   ' Muss 2 Byte Länge aufweisen.
```

Ein Tipp: Sie können normalerweise eine globale Suche nach dem Begriff »#define« mit »Public Const« durchführen, um den Portierungsaufwand zu reduzieren. In den meisten Fällen müssen Sie dann nur noch die Gleichheitszeichen einfügen, die in Visual Basic erforderlich sind. Eine große Hilfe beim Portierungsvorgang ist Ihnen der Visual Basic-Compiler, ermittelt er doch die meisten Fehler, die Ihnen

unterlaufen (beispielsweise das Vergessen eines Gleichheitszeichens oder eine nicht vorgenommene Änderung des `0x`-Hexadezimalindikators in `&H`).

Schritt 5: Kennzeichnen der »typedef«-Deklarationen als Kommentar

Der C-Befehl `typedef` wird zur Beschreibung eines Variablentyps in Form eines anderen Typs verwendet. Seine generische Form lautet wie folgt:

```
typedef variable_type new_type
```

Hierbei stellt `variable_type` einen aktuell definierten C/C++-Datentyp, `new_type` einen neuen Namen für diesen Typ dar. Sehen Sie sich dieses Beispiel aus der Datei **speed.h** an:

```
typedef unsigned short ushort;
```

Hiermit wird ausgedrückt, dass von diesem Zeitpunkt an der Datentyp `ushort` im Programm auf eine vorzeichenlose `short`-Variable verweist. Durch eine `typedef`-Anweisung wird eigentlich keine wirkliche Operation ausgeführt, es ist lediglich möglich, neue Namen für häufig verwendete Datentypen zu erstellen.[2]

Warum sollten Sie diese Zeilen als Kommentare kennzeichnen und nicht einfach löschen? Da diese Zeilen wichtige Hinweise dazu enthalten, welche Datentypen für Variablen in Strukturen und Funktionsparametern verwendet werden. Wenn Sie in der Datei **speed.h** einen Parameter in einer Funktion sehen, der als `ushort` deklariert wird, können Sie mit Hilfe der `typedef`-Anweisung ermitteln, dass es sich um eine vorzeichenlose `short`-Variable handelt und dementsprechend in der Visual Basic-Deklaration eine 16-Bit-`Integer`-Variable verwenden.

Schritt 6: Portieren der Strukturen und Funktionen

Dieser Prozess soll nicht näher erläutert werden. Sehen Sie sich zu diesem Thema die Puzzle in Teil I des Buches an (schließlich handelt das gesamte Buch von der Deklarationenerstellung für Strukturen und Funktionen). Die einzig knifflige Funktion hierbei ist die `cpuspeed`-Funktion, die in Puzzle 29 besprochen wird.

2. Puristen und die Mehrzahl der C/C++-Programmierer wird sofort protestieren, dass es sich bei dem Befehl `typedef` tatsächlich um eine sehr wichtige Anweisung handelt, durch die sehr wohl Operationen ausgeführt werden, insbesondere im Hinblick auf die Erzwingung einer Typenprüfung. Diese Tatsache ist jedoch für Visual Basic-Programmierer völlig irrelevant, zumindest für die durchzuführende Portierung. Wie Sie sehen, habe ich noch nicht vollständig meinen Verstand verloren.

Tutorium 9

In einer DLL-Datei: Das Programm DumpInfo

Eine der Hauptschwierigkeiten bei der Verwendung von API-Funktionen ist die Ermittlung der DLL, die die gewünschte Funktion exportiert. Der Grund für dieses Problem liegt darin, dass in der Microsoft-Dokumentation nicht die DLL-Namen für eine Funktion angegeben werden, sondern lediglich die Importbibliothek, mit der ein C-Programm die Funktion importieren würde. Unglücklicherweise stimmen die Namen von Importbibliothek und DLL nicht immer überein. Zusätzlich ist oft nicht eindeutig zu ermitteln, ob eine Funktion über separate ANSI- und Unicode-Einsprungpunkte verfügt.

Microsoft stellt im Lieferumfang von Visual Studio ein Programm mit dem Namen DumpBin bereit, mit dem Sie eine Liste der exportierten Funktionen einer DLL abrufen können. Der Suchvorgang durch alle DLLs nach einer bestimmten Funktion kann jedoch sehr viel Zeit in Anspruch nehmen.

Kurz bevor ich mit der Arbeit zu diesem Buch begann, wollte ich ein automatisches Dienstprogramm entwickeln, mit dem der Benutzer nach einem Funktionsnamen innerhalb aller DLLs im Systemverzeichnis der Funktion suchen kann, mit optionaler Prüfung auf ANSI- und Unicode-Varianten des Funktionsnamens. Was ich schließlich schrieb, war ein Programm, mit dem die Exporttabelle einer DLL-Datei gelesen werden kann. Die Programmentwicklung gewährte mir gleichzeitig Einblick in das PE-Dateiformat (Portable Executable, das für DLLs verwendete Dateiformat) und umfasste das Interpretieren der in diesem Dateiformat verwendeten Strukturen und Konstanten und die Konvertierung in Visual Basic. Die eigenständige Ausführung genau dieser Aufgaben möchte ich den Visual Basic-Programmierern mit diesem Buch ans Herz legen, daher erfüllt dieses Dienstprogramm gleich zwei Funktionen – es kann als Dienstprogramm und als Fallstudie eingesetzt werden.

Verwenden von DumpInfo

Das Programm DumpInfo erfüllt zwei verschiedene Funktionen. Erstens können mit diesem Dienstprogramm sämtliche Funktionen aufgelistet werden, die von einer DLL exportiert werden. Durch Klicken auf die Befehlsschaltfläche `Exports` auf dem Hauptformular wird ein Dialogfeld geöffnet, in dem Sie die zu untersuchende DLL auswählen können. Die exportierten Funktionen für diese DLL werden im Listenfeld aufgelistet, wie dargestellt in Abbildung T9-1, in dem die Exportliste für die DLL **advapi32.dll** angezeigt wird.

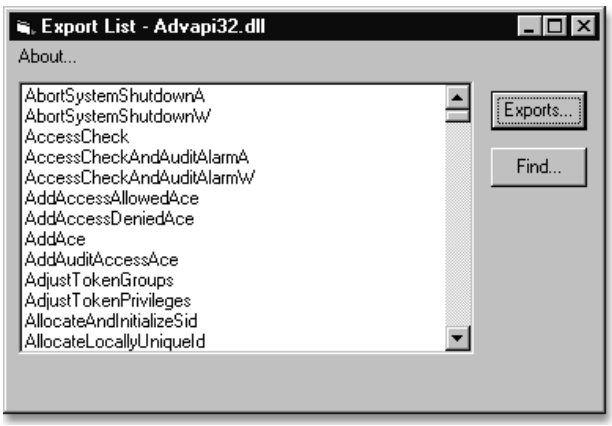


Abbildung T9-1 Exportliste für advapi32.dll

Über die Schaltfläche **Find** wird das in Abbildung T9-2 gezeigte Formular angezeigt. Geben Sie im Textfeld den Namen der Funktion ein, nach der Sie suchen möchten. Die drei Kontrollkästchen bieten die folgenden Optionen:

- **Ignore Case.** Aktivieren Sie dieses Kontrollkästchen, wenn während der Suche keine Berücksichtigung der Groß- und Kleinschreibung erfolgen soll.
- **Check A&W Suffixes.** Mit dieser Option wird nach dem angegebenen Funktionsnamen sowie nach Varianten des Namens gesucht, die entweder das Suffix A oder W aufweisen. Auf diese Weise können Sie, falls vorhanden, die ANSI- und Unicode-Einsprungpunkte für einen vorgegebenen Funktionsnamen ermitteln.
- **Search All DLLs.** Ist diese Option aktiviert, durchsucht das DumpInfo-Programm sämtliche DLLs in Ihrem Systemverzeichnis. Hierbei wird die Suche auch dann fortgesetzt, wenn die Funktionen in einer bestimmten DLL gefunden wurden. Ist diese Option nicht aktiviert, wird die Suche gestoppt, sobald die Funktionen gefunden wurden. Diese Option ist nützlich, wenn Sie glauben, dass eine bestimmte Funktion durch mehr als eine DLL exportiert wird.

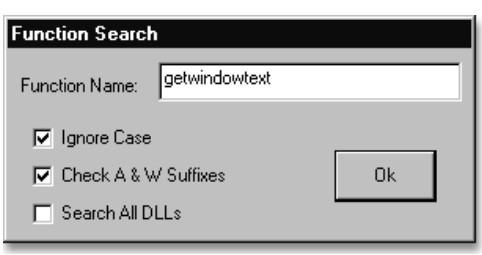


Abbildung T9-2 Eingabeformular für die Suche nach exportierten Funktionen

Abbildung T9-3 Suchergebnisse für die Funktion `GetWindowText`

Eine Suchoperation kann abgebrochen werden, indem Sie während der Suche auf die Schaltfläche **Stop** klicken (diese Schaltfläche wird nur während einer Suche angezeigt). Nachdem eine Suchoperation beendet wurde, wird eine Liste der Funktionen angezeigt, auf die die Suchkriterien zutreffen, wie in Abbildung T9-3 am Beispiel der Funktion `GetWindowText` gezeigt.

Bevor Sie weiterlesen, muss ich Sie darauf hinweisen, dass es sich um ein Tutorium auf Expertenniveau handelt. Damit Sie von diesem Tutorium profitieren, sollten Sie sich zunächst eine Kopie der Microsoft Portable Executable File-Spezifikation besorgen (verfügbar in MSDN und in der Onlinedokumentation von Visual Studio). Mit diesem Tutorium wird nicht etwa der Versuch unternommen, das **exe**-Dateiformat zu erklären, es soll lediglich demonstriert werden, wie dieses Format mit einem Visual Basic-Programm analysiert werden kann. Die Lektüre dieses Tutoriums ist für das Verständnis der folgenden Tutorien nicht zwingend erforderlich, wenn Sie dieses Thema also frustrierend oder verwirrend finden, rate ich Ihnen, es einfach zu überspringen.

Die »Exports«-Klasse

Obwohl mit dem Programm DumpInfo zwei verschiedene Funktionen ausgeführt werden, sollte klar sein, dass sich die eine Funktion selbstverständlich aus der anderen ergibt. Die Fähigkeit, in einem Verzeichnis voller DLLs nach einer Funktion zu suchen, ist relativ einfach zu erreichen, jedenfalls sobald Sie in der Lage sind, in einer einzigen DLL sämtliche der exportierten Funktionen anzuzeigen. Tatsächlich ist dieser Vorgang so einfach, dass er in diesem Tutorium nicht erwähnt werden soll. Wenn Sie sich die Implementierung ansehen möchten, so finden Sie diese im Beispielprogramm. Statt dessen konzentriert sich dieses Tutorium auf die komplexere Aufgabe des Abrufs einer Liste der exportierten Funktionen.

Da der Abruf einer Liste der exportierten Funktionen eine wohldefinierte Aufgabe darstellt, die mehrfach ausgeführt wird, ist es eine gute Idee, die Implementierung in einer Klasse vorzunehmen. Die `Exports`-Klasse liegt in Form einer privaten Auflistungsklasse vor, die mit der Liste der exportierten Funktionen einer einzelnen DLL geladen werden kann. Diese Funktionen können anschließend mit Hilfe verschiedener auflistungsähnlicher Eigenschaften gelesen werden. Die `Exports`-Klasse implementiert die folgenden öffentlichen Methoden und Eigenschaften:

- **Sub Load(FileName as String).** Lädt die `Exports`-Klasse mit einer Liste der exportierten Funktionen für den angegebenen Dateinamen.
- **Function Count() As Long.** Gibt die Anzahl der exportierten Funktionen in der Auflistung zurück.
- **Function Ordinal(FunctionNumber As Long) As Long.** Gibt die Ordinalzahl der exportierten Funktionen im angegebenen Index in der Auflistung zurück.[1]
- **Function Name(FunctionNumber As Long) As String.** Gibt den Namen der exportierten Funktionen im angegebenen Index in der Auflistung zurück.

Jetzt müssen Sie nur noch die DLL-Datei auf die exportierten Funktionen prüfen und ein internes Array mit den Namen und Ordinalzahlen dieser Funktionen laden.

Das PE-Format

Es gibt zwei Informationsquellen zum PE-Dateiformat (Portable Executable), dem Dateiformat, das für jede ausführbare Win32-Datei sowie für sämtliche Win32-DLLs verwendet wird. Die erste Quelle stellt die Formatspezifikation dar, die Sie in der MSDN oder in der Visual Studio Library unter dem Abschnitt für die Plattformspezifikationen finden. Suchen Sie hier nach dem Buchtitel **The Microsoft Portable Executable and Common Object File Format**. Die zweite Informationsquelle ist die Datei **winnt.h,** die Headerdatei mit C-Typendefinitionen und Konstantendeklarationen, die das PE-Dateiformat verwenden.

Unsere Aufgabe besteht darin, die Inhalte der Datei soweit zu verstehen, dass wir die Exportfunktionsinformationen auffinden und lesen können. Glücklicherweise ist hierzu kein tieferes Verständnis aller Aspekte dieses Dateiformats nötig. Sie müssen lediglich die Teile verstehen, die zum Auffinden und Interpretieren der Exportfunktionsinformationen erforderlich sind.

1. Jede exportierte Funktion kann über eine verknüpfte Nummer verfügen, die als Orinalzahl bezeichnet wird. Funktionen können mit Hilfe dieser Ordinalzahlen schneller aufgefunden und geladen werden als über deren Namen. Trotz dieser Tatsache werden Ordinalzahlen nur selten eingesetzt.

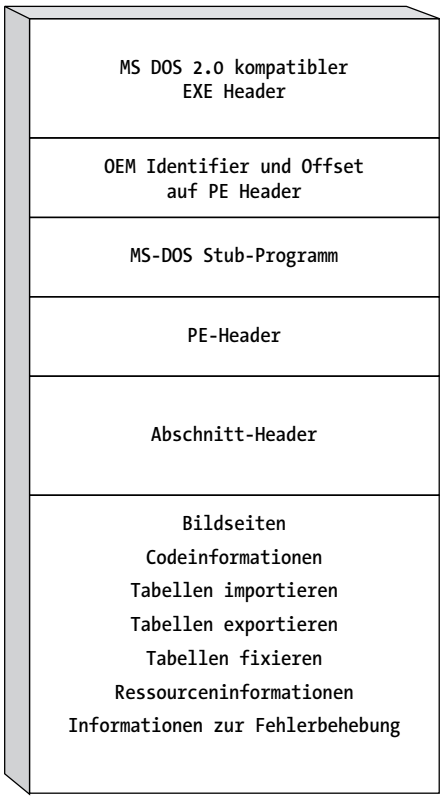


Abbildung T9-4 Architektur einer PE-Datei (Portable Executable)

Die Gesamtstruktur einer **PE**-Datei wird in Abbildung T9-4 dargestellt, die der Spezifikation entnommen wurde.

Das Interessante an einer **PE**-Datei ist, dass diese mit einem Headertyp beginnt, der bereits in den frühen Zeiten von MS-DOS von ausführbaren Dateien verwendet wurde. Auf diesen Header folgt ein MS-DOS-Stubprogramm. Das Vorhandensein dieser Header und Stubs ist der Grund dafür, dass Sie bei der Ausführung einer beliebigen ausführbaren Windows-Datei unter MS-DOS immer die Meldung erhalten, dass das Programm zur Ausführung Microsoft Windows benötigt. Die Meldung wird durch das Stubprogramm erzeugt.

Die Win32-spezifischen Informationen beginnen mit dem PE-Header. Dies bedeutet, dass bei der Prüfung zunächst MS-DOS-Header und Stubprogramm übersprungen werden müssen, um den **PE**-Header zu ermitteln.

Der DOS-Header wird durch die folgende Struktur in **winnt.h** definiert:

```
typedef struct _IMAGE_DOS_HEADER {     // DOS .EXE -Header
    WORD   e_magic;                    // Magische Nummer
    WORD   e_cblp;                     // Bytes auf der letzten Dateiseite
    WORD   e_cp;                       // Seiten in Datei
    WORD   e_crlc;                     // Neuanordnungen
    WORD   e_cparhdr;                  // Headergröße in Absätzen
    WORD   e_minalloc;                 // Minimal benötigte Zusatzabsätze
    WORD   e_maxalloc;                 // Maximale benötigte Zusatzabsätze
    WORD   e_ss;                       // Anfänglicher (relativer) SS-Wert
    WORD   e_sp;                       // Anfänglicher SP-Wert
    WORD   e_csum;                     // Prüfsumme
    WORD   e_ip;                       // Anfänglicher IP-Wert
    WORD   e_cs;                       // Anfänglicher (relativer) CS-Wert
    WORD   e_lfarlc;                   // Dateiadresse der Neuanordnungstabelle
    WORD   e_ovno;                     // Overlaynummer
    WORD   e_res[4];                   // Reservierte Wörter
    WORD   e_oemid;                    // OEM -Identifier (für e_oeminfo)
    WORD   e_oeminfo;                  // OEM-Information; e_oemid -spezifisch
    WORD   e_res2[10];                 // Reservierte Wörter
    LONG   e_lfanew;                   // Dateiadresse des neuen exe-Headers
  }  IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

So findet eine Übersetzung in die folgende Visual Basic-Struktur statt. Beachten Sie, dass die Visual Basic-Arrayindizes niedrigere Werte aufweisen als in der C-Deklaration. C-Arraydefinitionen geben die Größe des Arrays an, wohingegen in Visual Basic Arraydefinitionen den höchsten Index sowie die Standardbasis 0 angeben.

```
Public Type IMAGE_DOS_HEADER
   e_magic As Integer
   e_cblp As Integer
   e_cp As Integer
   e_crlc As Integer
   e_cparhdr As Integer
   e_minalloc As Integer
   e_maxalloc As Integer
   e_ss As Integer
   e_sp As Integer
   e_csum As Integer
   e_ip As Integer
```

```
    e_cs As Integer
    e_lfarlc As Integer
    e_ovno As Integer
    e_res(3) As Integer
    e_oemid As Integer
    e_oeminfo As Integer
    e_res2(9) As Integer
    e_lfanew As Long
End Type
```

Das Feld `e_lfanew` gibt das Dateioffset zum Start der **PE**-Headerstruktur an. Die **PE**-Headerstruktur wird durch die Struktur `IMAGE_NT_HEADERS` definiert, die eine Signatur enthält, mit der die Datei als PE-Formatdatei gekennzeichnet wird sowie zwei zusätzliche Strukturen identifiziert werden: eine `IMAGE_FILE_HEADER`- und eine `IMAGE_OPTIONAL_HEADER`-Struktur. Diese Strukturen lauten folgendermaßen:

```
typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER OptionalHeader;
}  IMAGE_NT_HEADERS, *PIMAGE_NT_HEADERS;

typedef struct _IMAGE_FILE_HEADER {
    WORD    Machine;
    WORD    NumberOfSections;
    DWORD   TimeDateStamp;
    DWORD   PointerToIconTable;
    DWORD   NumberOfSymbols;
    WORD    SizeOfOptionalHeader;
    WORD    Characteristics;
}  IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;

typedef struct _IMAGE_OPTIONAL_HEADER {
    //
    // Standardfelder
    //

    WORD    Magic;
    BYTE    MajorLinkerVersion;
    BYTE    MinorLinkerVersion;
    DWORD   SizeOfCode;
    DWORD   SizeOfInitializedData;
```

```
    DWORD   SizeOfUninitializedData;
    DWORD   AddressOfEntryPoint;
    DWORD   BaseOfCode;
    DWORD   BaseOfData;

    //
    // Zusätzliche NT-Felder
    //

    DWORD   ImageBase;
    DWORD   SectionAlignment;
    DWORD   FileAlignment;
    WORD    MajorOperatingSystemVersion;
    WORD    MinorOperatingSystemVersion;
    WORD    MajorImageVersion;
    WORD    MinorImageVersion;
    WORD    MajorSubsystemVersion;
    WORD    MinorSubsystemVersion;
    DWORD   Win32VersionValue;
    DWORD   SizeOfImage;
    DWORD   SizeOfHeaders;
    DWORD   CheckSum;
    WORD    Subsystem;
    WORD    DllCharacteristics;
    DWORD   SizeOfStackReserve;
    DWORD   SizeOfStackCommit;
    DWORD   SizeOfHeapReserve;
    DWORD   SizeOfHeapCommit;
    DWORD   LoaderFlags;
    DWORD   NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
}  IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;

typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   VirtualAddress;
    DWORD   Size;
}  IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;


Private DosHeader As IMAGE_DOS_HEADER
Private PEHeader As IMAGE_NT_HEADERS
```

```
Private PEHeaderOffset As Long
Private ExportDirectory As IMAGE_EXPORT_DIRECTORY

Public Type IMAGE_NT_HEADERS
   Signature As Long
   FileHeader As IMAGE_FILE_HEADER
   OptionalHeader As IMAGE_OPTIONAL_HEADER
End Type

Public Type IMAGE_FILE_HEADER
   Machine As Integer
   NumberOfSections As Integer
   TimeDateStamp As Long
   PointerToSymbolTable As Long
   NumberOfSymbols As Long
   SizeOfOptionalHeader As Integer
   Characteristics As Integer
End Type

Public Type IMAGE_DATA_DIRECTORY
   VirtualAddress As Long
   Size As Long
End Type

Public Type IMAGE_OPTIONAL_HEADER
   Magic As Integer
   MajorLinkerVersion As Byte
   MinorLinkerVersion As Byte
   SizeOfCode As Long
   SizeOfInitializedData As Long
   SizeOfUninitializedData As Long
   AddressOfEntryPoint As Long
   BaseOfCode As Long
   BaseOfData As Long
   ImageBase As Long
   SectionAlignment As Long
   FileAlignment As Long
   MajorOperatingSystemVersion As Integer
   MinorOperatingSystemVersion As Integer
   MajorImageVersion As Integer
   MinorImageVersion As Integer
```

```
    MajorSubsystemVersion As Integer
    MinorSubsystemVersion As Integer
    Win32VersionValue As Long
    SizeOfImage As Long
    SizeOfHeaders As Long
    CheckSum As Long
    Subsystem As Integer
    DllCharacteristics As Integer
    SizeOfStackReserve As Long
    SizeOfStackCommit As Long
    SizeOfHeapReserve As Long
    SizeOfHeapCommit As Long
    LoaderFlags As Long
    NumberOfRvaAndSizes As Long
    DataDirectory(15) As IMAGE_DATA_DIRECTORY
End Type
```

Detaillierte Erläuterungen zu den Strukturen `IMAGE_NT_HEADERS`, `IMAGE_FILE_HEADER`, `IMAGE_DATA_DIRECTORY` und `IMAGE_OPTIONAL_HEADER` finden Sie in der PE-Dateiformatspezifikation. In diesem Tutorium sollen nur die Felder definiert werden, die tatsächlich vom Programm benutzt werden. Die hauptsächlich in der `Exports`-Klasse für das Laden der Dateiinformationen verantwortliche ist die nachfolgend gezeigte `LoadInfo`-Funktion:

```
Private Function LoadInfo(FileName As String) As Integer
    Dim x%
    FileHandle = FreeFile()

    On Error GoTo BadLoad
    Open FileName For Binary Access Read As #FileHandle
    On Error GoTo BadBuild
    ' Abrufen des DOS-Headers
    Get #FileHandle, , DosHeader

    ' Offset für NT-Header ermitteln
    PEHeaderOffset = DosHeader.e_lfanew
    ' Abrufen des NT-Headers
    Get #FileHandle, PEHeaderOffset + 1, PEHeader

    SectionCount = PEHeader.FileHeader.NumberOfSections
    ReDim Sections(SectionCount - 1)
    SectionsOffset = Seek(FileHandle)
```

```
   For x = 0 To SectionCount - 1
      Get #FileHandle, , Sections(x)
   Next x

   FindExportBase

   Get #FileHandle, ExportBase + 1, ExportDirectory

   LoadExportInfo

   Close #FileHandle
   Exit Function
BadLoad:
   LoadInfo = -1
   Exit Function
BadBuild:
   LoadInfo = -1
   Close #FileHandle
End Function
```

Die Funktion lädt zunächst die Struktur `IMAGE_DOS_HEADER` in die Variable `DosHeader`. Das Feld `e_lfanew` der Struktur enthält das Offset zum PE-Headerstart, der durch die Struktur `IMAGE_NT_HEADERS` definiert wird. Die Variable `PEHeader` wird unter Verwendung der nachstehenden `Get`-Anweisung mit diesen Informationen geladen:

```
Get #FileHandle, PEHeaderOffset + 1, PEHeader
```

Warum müssen Sie zum Offset 1 addieren? Da das Offset auf der Annahme basiert, dass das erste Byte in der Datei sich an Position 0 befindet, während die `Get`-Anweisung voraussetzt, dass sich das erste Byte in der Datei an Position 1 befindet.

Die Abschnittstabelle erscheint unmittelbar nach dem PE-Header. Das Feld `NumberOfSections` in der Struktur `IMAGE_FILE_HEADER` ermöglicht Ihnen die Ermittlung der Anzahl der Abschnitte in der Datei. Jeder Abschnitt ist mit Hilfe der `IMAGE_SECTION_HEADER`-Struktur definiert, deren Definition folgendermaßen lautet:

```
#define IMAGE_SIZEOF_SHORT_NAME              8

typedef struct _IMAGE_SECTION_HEADER {
    BYTE    Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
            DWORD   PhysicalAddress;
            DWORD   VirtualSize;
    }  Misc;
    DWORD   VirtualAddress;
    DWORD   SizeOfRawData;
    DWORD   PointerToRawData;
    DWORD   PointerToRelocations;
    DWORD   PointerToLinenumbers;
    WORD    NumberOfRelocations;
    WORD    NumberOfLinenumbers;
    DWORD   Characteristics;
}  IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;


Public Type IMAGE_SECTION_HEADER
   ShortName(7) As Byte

   VirtualSize As Long
   VirtualAddress As Long
   SizeOfRawData As Long
   PointerToRawData As Long
   PointerToRelocations As Long
   PointerToLinenumbers As Long
   NumberOfRelocations As Integer
   NumberOfLinenumbers As Integer
   Characteristics As Long
End Type
```

Die Variable `SectionsOffset` wird mit dem aktuellen Offset in die Datei geladen (der Position unmittelbar nach den **PE**-Headerinformationen). Mit dem folgenden Code wird das `Section`-Array geladen:

```
SectionsOffset = Seek(FileHandle)
   For x = 0 To SectionCount - 1
      Get #FileHandle, , Sections(x)
   Next x
```

Die Abschnittstabelle enthält Informationen zu den Abschnitten, die im Imagedatenbereich erscheinen. Für uns sind drei Felder von Interesse. Das Feld `VirtualAddress` repräsentiert die Speicheradresse, an der der Abschnitt beginnt, nachdem er in den Speicher geladen wurde. Das Feld `PointerToRawData` enthält den Standort des Abschnitts in der Imagedatei. Das Feld `SizeOfRawData` enthält die Größe des Abschnitts in der Imagedatei. Abbildung T9-5 veranschaulicht die Beziehung von Abschnittstabellen zu den Abschnitten. Wie Sie sehen können, erscheint die Exporttabelle als Teil einer der Abschnitte.

Auffinden der Exporttabelle

Bisher wurden durch das Programm der `PE`-Dateiheader, einige optionale Header sowie die Abschnittstabelle der Datei geladen. Wir suchen nach dem Abschnitt, der die Exporttabelle der Datei enthält. Mit der Funktion `FindExportBase` wird die Variable `ExportBase` mit der Basis der Exporttabelle geladen:

```
Private Sub FindExportBase()
   Dim secnum%
   ExportDirectoryOffset = _
   PEHeader.OptionalHeader.DataDirectory(0).VirtualAddress
   ExportSection = SectionCount - 1
   For secnum = 0 To SectionCount - 1
      If Sections(secnum).VirtualAddress > ExportDirectoryOffset Then
         ExportSection = secnum - 1
         Exit For
      End If
   Next secnum
  ' Wir kennen nun die Abschnittsnummer , Berechnen des Dateioffsets
   ExportSectionOffset = Sections(ExportSection).VirtualAddress - _
   Sections(ExportSection).PointerToRawData
   ExportBase = ExportDirectoryOffset - ExportSectionOffset
End Sub
```

Wenn Sie sich die Struktur `IMAGE_OPTIONAL_HEADER` ansehen, finden Sie am Ende dieser Struktur eine Liste mit sechzehn `IMAGE_DATA_DIRECTORY`-Strukturen. Jede dieser Strukturen enthält ein Offset zu einem bestimmten Datentyp, zusammen mit der Größe dieser Daten. Der erste dieser Datenverzeichniseinträge verweist auf die Tabelle der exportierten Funktionen. Woher ich das weiß? Aus der `PE`-Dateiformatspezifikation.

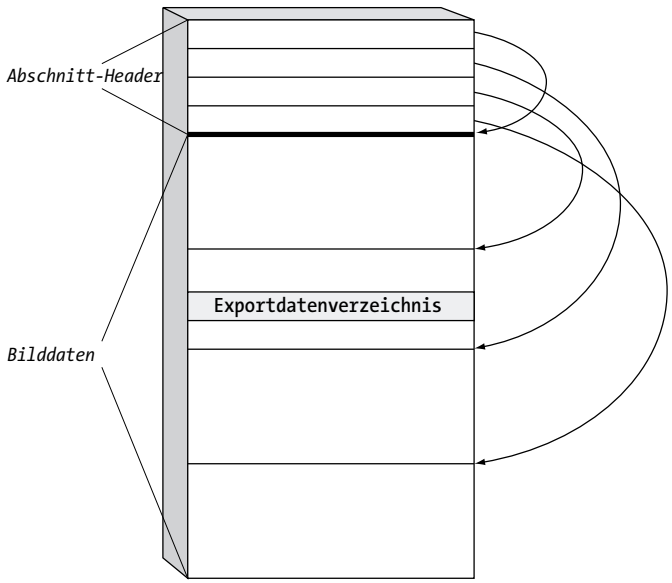


Abbildung T9-5 Aufbau der Abschnitte in einer Imagedatei

Die virtuelle Adresse für die Exporttabelle wird in die `ExportDirectoryOffset`-Variable geladen. Die virtuelle Adresse beschreibt das Offset für einen bestimmten Datenblock nach dem Laden des Images in den Speicher. Wir benötigen jedoch den Standort der Exportinformationen in der Datei selbst. Mit Hilfe der Funktion `FindExportBase` können Sie die Abschnitte durchsuchen, um den ersten Abschnitt zu finden, dessen virtuelle Adresse größer als dieser `ExportDirectoryOffset`-Wert ist. Der Abschnitt vor diesem muss die Exporttabelle selbst enthalten.

Damit verfügen wir nun über die virtuelle Adresse der Exporttabelle, die virtuelle Adresse des Abschnitts, der die Exporttabelle enthält, und über den Standort dieses Abschnitts in der Imagedatei.

Der folgende Ausdruck kann dazu verwendet werden, den Standort der Exporttabelle in der Imagedatei zu ermitteln:

```
ExportBase = ExportDirectoryOffset -
Sections(ExportSection).VirtualAddress +
Sections(ExportSection).PointerToRawData
```

Hierbei wird das Offset der Exporttabelle zu Beginn des Abschnitt zum Standort des Abschnittbeginns addiert. Es liegt jedoch keine Übereinstimmung mit dem Code in der `FindExportBase`-Funktion vor. Warum nicht?

Mit dem Code in der FindExportBase-Funktion wird zwar die gleiche Berechnung vorgenommen, es wird jedoch eine Neuanordnung durchgeführt, mit der zunächst eine Variable mit Namen ExportSectionOffset berechnet wird. Die ExportSectionOffset-Variable enthält anschließend einen Wert, der von jeder beliebigen virtuellen Adresse im angegebenen Abschnitt subtrahiert werden kann, um den entsprechenden Standort in der Datei zu ermitteln. Wir werden diese Variable später verwenden, da viele der Einträge in der Exporttabelle ebenfalls virtuelle Adressen verwenden.

```
ExportSectionOffset = Sections(ExportSection).VirtualAddress - _
   Sections(ExportSection).PointerToRawData
ExportBase = ExportDirectoryOffset - ExportSectionOffset
```

Die Variable ExportBase enthält nun das Dateioffset des Exportdatenbeginns.

Laden der Exporttabelle

Die Exportdaten beginnen mit einem Exportdatenverzeichnis, das durch eine Struktur vom Typ IMAGE_EXPORT_DIRECTORY definiert wird. Diese Struktur wird in C folgendermaßen definiert:

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD   Characteristics;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD   Name;
    DWORD   Base;
    DWORD   NumberOfFunctions;
    DWORD   NumberOfNames;
    PDWORD  *AddressOfFunctions;
    PDWORD  *AddressOfNames;
    PWORD   *AddressOfNameOrdinals;
}  IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

Die Struktur sieht etwas kompliziert aus – wie behandeln Sie als PDWORD * definierte Datentypen? Tatsächlich ist die Erstellung einer VB-Deklaration einfach. Zeiger, und selbst Zeiger auf Zeiger, umfassen stets 32 Bits und werden daher As Long deklariert.

```
Public Type IMAGE_EXPORT_DIRECTORY
   Characteristics As Long
   TimeDateStamp As Long
   MajorVersion As Integer
```

```
   MinorVersion As Integer
   Name As Long
   Base As Long
   NumberOfFunctions As Long  ' Wird möglicherweise nicht verwendet
   NumberOfNames As Long
   AddressOfFunctions As Long
   AddressOfNames As Long
   AddressOfNameOrdinals As Long
End Type
```

Die `IMAGE_EXPORT_DIRECTORY`-Struktur wird mit der folgenden Codezeile und unter Verwendung des zuvor berechneten `ExportBase`-Offsets in die `ExportDirectory`-Variable der Funktion `LoadInfo` geladen:

```
Get #FileHandle, ExportBase + 1, ExportDirectory
LoadExportInfo
```

Die tatsächliche Operation wird von der `LoadExportInfo`-Funktion ausgeführt, wie hier gezeigt wird:

```
Private Sub LoadExportInfo()
   Dim idx&
   Dim startaddresses() As Long
   Dim ExportsBuffer() As Byte
   Dim tempstr As String
   Dim stringlen As Long
   Dim sourceloc As Long

   NameCount = ExportDirectory.NumberOfNames
   ReDim startaddresses(NameCount - 1)
   ReDim Ordinals(NameCount - 1)
   ReDim Names(NameCount)


   ' Array mit sämtlichen Ordinalzahlen laden
   sourceloc = ExportDirectory.AddressOfNameOrdinals - ExportSectionOffset + 1
   Get #FileHandle, sourceloc, Ordinals()
   For idx = 0 To NameCount-1
      Ordinals(idx) = Ordinals(idx) + ExportDirectory.Base
   Next idx

   ' Kopie sämtlicher Startadressen erstellen
   sourceloc = ExportDirectory.AddressOfNames - ExportSectionOffset + 1
```

```
   Get #FileHandle, sourceloc, startaddresses()

   ' Aus Geschwindigkeitsgründen gesamten Abschnitt vorladen
   ReDim ExportsBuffer(Sections(ExportSection).SizeOfRawData)
   Get #FileHandle, Sections(ExportSection).PointerToRawData + 1, _
   ExportsBuffer()

   ' Jedes Startarray extrahieren
   For idx = 0 To NameCount - 1
      sourceloc = startaddresses(idx) - ExportSectionOffset - _
Sections(ExportSection).PointerToRawData
      ' Zeichenfolge vorinitialisieren
      ' Länge der Zeichenfolge ermitteln
      'Debug.Print sourceloc
      If sourceloc > 0 Then
         stringlen = lstrlenFromPtr(VarPtr(ExportsBuffer(sourceloc)))
         tempstr = String$(stringlen + 1, 0)
         Call lstrcpyFromPtr(tempstr, VarPtr(ExportsBuffer(sourceloc)))
         Names(idx + 1) = Left$(tempstr, stringlen)
      End If
      'Debug.Print tempstr
   Next idx
End Sub
```

Die Funktion ermittelt zunächst die Anzahl der exportierten Funktionen und dimensioniert drei Arrays, eines für eine temporäre Adresse für jede Funktion, eines für den Ordinalwert und eines für den Funktionsnamen.

```
   NameCount = ExportDirectory.NumberOfNames
   ReDim startaddresses(NameCount - 1)
   ReDim Ordinals(NameCount - 1)
   ReDim Names(NameCount)
```

Wozu dient das Feld `AddressOfNameOrdinals`? Die Ordinalwerte erscheinen im Speicher und in den Dateien als eine Liste mit `Integer`-Werten. Das Feld `AddressOfNameOrdinals` zeigt auf den Standort des Listenanfangs.

Bei dem durch das Feld `AddressOfNameOrdinals` angegebenen Standort handelt es sich um eine virtuelle Adresse. Mit der zuvor berechneten `ExportSectionOffset`-Variable können wir durch Subtraktion von einer beliebigen virtuellen Adresse in einem Abschnitt den Standort in einer Datei ermitteln.

```
' Array mit sämtlichen Ordinalzahlen laden
   sourceloc = ExportDirectory.AddressOfNameOrdinals - _
   ExportSectionOffset + 1
   Get #FileHandle, sourceloc, Ordinals()
   For idx = 0 To NameCount-1
      Ordinals(idx) = Ordinals(idx) + ExportDirectory.Base
   Next idx
```

Das Ordinalarray wird als Block geladen. In der PE-Dateiformatspezifikation wird angegeben, dass die Werte um den Ordinalbasiswert erhöht werden müssen, der sich im `Base`-Feld des Exportverzeichnisses befindet.

Das Abrufen der Funktionsnamen ist eine knifflige Sache. Im Gegensatz zu Zahlen können variable Längenzeichenfolgen nicht in ein Array geladen werden. Statt dessen enthält die Datei ein Array mit den virtuellen Adressen sämtlicher Funktionsnamen. Diese Adressen werden auf die gleiche Weise wie das zuvor geladene Ordinalarray in das Standardadressenarray geladen.

```
' Kopie sämtlicher Startadressen erstellen
   sourceloc = ExportDirectory.AddressOfNames - ExportSectionOffset + 1
   Get #FileHandle, sourceloc, startaddresses()
```

Statt jede Zeichenfolge individuell aus der Datei zu laden, wird der gesamte Abschnitt mit den Zeichenfolgen vorab unter Verwendung des folgenden Codes in ein Bytearray mit Namen `ExportsBuffer` geladen:

```
   ' Aus Gründen der Geschwindigkeit gesamten Abschnitt vorab laden
   ReDim ExportsBuffer(Sections(ExportSection).SizeOfRawData)
   Get #FileHandle, Sections(ExportSection).PointerToRawData _
   + 1, ExportsBuffer()
```

Der Standort jeder Zeichenfolge in der Datei kann mit Hilfe des folgenden Codes berechnet werden:

```
startaddresses(idx) - ExportSectionOffset
```

Durch Subtraktion des Abschnittbeginns in der Datei erhalten Sie das Offset der Zeichenfolge in der `ExportsBuffer`-Tabelle. Dieses Offset wird in der Variablen `sourceloc` gespeichert.

Die Funktion `lstrlenFromPtr` ist ein Alias der `lstrlen`-Funktion, die die Länge einer Zeichenfolge mit Zeiger enthält. Die `lstrcpyFromPtr`-Funktion verwendet die `lstrcpy`-Funktion, um die Zeichenfolge in einen initialisierten Zeichenfolgenpuffer zu kopieren. Abschließend wird das `Name`-Array mit der Zeichenfolge bis zum abschließenden NULL-Zeichen geladen.

```
Jedes Startarray extrahieren
  For idx = 0 To NameCount - 1
     sourceloc = startaddresses(idx) - ExportSectionOffset _
     - Sections(ExportSection).PointerToRawData
     ' Zeichenfolge vorinitialisieren
     ' Länge der Zeichenfolge ermitteln
     If sourceloc > 0 Then
        stringlen = lstrlenFromPtr(VarPtr(ExportsBuffer(sourceloc)))
        tempstr = String$(stringlen + 1, 0)
        Call lstrcpyFromPtr(tempstr, VarPtr(ExportsBuffer(sourceloc)))
        Names(idx + 1) = Left$(tempstr, stringlen)
     End If
  Next idx
```

Schlussfolgerung

Das Interpretieren von Dateispezifikationen ist keine leichte Aufgabe, das vorliegende Tutorium sollte Ihnen hierzu einen kleinen Einblick gewähren. Den größten Nutzen ziehen Sie aus diesem Tutorium, wenn Sie sich über die zu Beginn des Tutoriums genannten Quellen eine Kopie der PE-Dateiformatspezifikation besorgen und die Spezifikation selbst lesen. Anschließend gehen Sie schrittweise das **DumpInfo**-Programm durch und lesen parallel dieses Tutorium, um nachvollziehen zu können, wie die in der Spezifikation und in den Headerdateien angegeben Informationen in Visual Basic übersetzt werden.

Tutorium 10

Eine Fallstudie: Die Dienst-API

Der folgende Artikel wurde ursprünglich in meiner Kolume im ***Pinnacle's Visual Basic Developer Newsletter*** *veröffentlicht. Hierbei handelt es sich um eine Fallstudie, die sich so perfekt für dieses Buch eignet, dass ich mich entschied, den Artikel hier erneut abzudrucken, auch wenn einige der Leser den Originalartikel bereits kennen sollten. Dieses Tutorium erfordert Windows NT.*

API-Funktionen kommen selten allein.

Dieses Buch richtet sich fast ausschließlich auf die Arbeit mit einzelnen API-Funktionen unter Verwendung von Visual Basic. Meistens müssen Sie selbst herausfinden, wie die verschiedenen API-Funktionen funktionieren und wie Sie diese verwenden können. Neben meinem Buch »Visual Basic Programmer's Guide to the Win32 API« im Hinblick auf die grundlegenden API-Funktionen und der Microsoft-Dokumentation zu allen weiteren Themen steht Ihnen eine Vielfalt an zusätzlichen Dokumentationen zur Verfügung. Ich gebe zu, dass die Microsoft-Dokumentation gelegentlich unergründlich scheint, doch Sie stellt in vielen Fällen die einzige Informationsquelle dar.

In diesem Tutorium wird anhand einer Fallstudie erläutert, wie Sie die Microsoft-Dokumentation dazu verwenden, eine spezielle Aufgabe zu lösen, bei der die Verwendung verschiedener API-Funktionen innerhalb eines gesamten Subsystems erforderlich ist. Zunächst möchte ich ein praktisches Problem erläutern, auf das ich selbst gestoßen bin. Anschließend werde ich Sie durch den gesamten Prozess der Problemlösung führen, einschließlich der Implementierung einer vollständigen Lösung.

ASP – A Step Pastward (Ein Schritt zurück)

Okay, ich weiß, dass es das Wort »pastward« nicht gibt, aber das Microsoft-Akronym für Active Server Pages ist ASP, nicht ASB, deshalb konnte ich die sehr viel passendere Überschrift »A Step Backward« leider nicht verwenden. Ich hätte natürlich auch »A Step into the Past« als Überschrift wählen können, aber die Erfindung eines neuen Wortes erschien mir irgendwie lustiger zu sein. Im Übrigen – solange Microsoft ständig neue Akronyme und merkwürdige Wörter einführen darf, werde ich ja wohl ab zu auch mal ein neues Wort erfinden dürfen.

Aber ich schweife ab.

ASP ist eine spezielle Webseite, die mit Hilfe des Microsoft Internet Information Servers verwaltet wird. Obwohl diese Seiten die Erweiterung .ASP aufweisen, handelt es sich tatsächlich lediglich um Textdateien, die HTML enthalten. Das Besondere an diesen Seiten ist ihre Fähigkeit, verschiedene Codetypen miteinander zu vermischen – Code, der auf dem Server ausgeführt wird, und Code, der auf dem Client verwendet wird (interpretiert durch den Webbrowser). Wenn Sie der Presse Glauben schenken, handelt es sich bei ASP um eine enorme Weiterentwicklung im Bereich der Programmierung, die eine sofortige Konvertierung aller reinen, statischen HTML-Seiten in ASP nach sich ziehen sollte.

Ich werde Ihnen nicht erzählen, dass ASP nicht leistungsfähig ist. Es ist leistungsfähig. Microsoft macht auf den Microsoft-Sites regen Gebrauch von ASP. Das kommerzielle Serverpaket ist wenig mehr als ein Assistent, der ASP-Seiten für Sie erstellt. Vom Gesichtspunkt der Computerwissenschaften aus erscheint ASP eher ein großer Schritt rückwärts zu sein. Warum? Obwohl ASP-Seiten Objekte verwenden können und dies auch tun, ist der Code nicht wirklich objektorientiert. Tatsächlich erreicht eine komplexe ASP-Seite fast die Unlesbarkeit eines APL-Programms. (Bei APL handelt es sich um eine derart elegante Sprache, dass man angeblich jedes Programm in nur einer Codezeile schreiben kann.) APL erfordert zur Verwendung der vielen neuen Zeichen für die Befehle jedoch auch eine besondere Tastatur. Server- und clientseitiger Code werden willkürlich gemischt, und Sie verbringen unendlich viel Zeit damit herauszufinden, wo Ihr Code tatsächlich ausgeführt wird. Und das Schlimmste von allem ist, dass VBScript oder JavaScript verwendet wird, d.h., der Code ist langsam und wird interpretiert.

Wenn dies nicht einen Rückschritt bedeutet, weiß ich nicht, was es sonst bedeuten soll.

Diese Ansicht stammt jedoch von einer Person, die auch HTML als einen Schritt rückwärts betrachtet. Denken Sie darüber nach – nachdem über Jahre hinweg ausgefeilte und leistungsstarke Textformate wie beispielsweise RTF entwickelt wurden, erfolgte ein Rückschritt zu einem Format, das so primitiv ist, dass wir die letzten Jahre damit verbracht haben, auf neue Tags und Editoren zur Ausführung von Aufgaben zu warten, die mit einem guten Textverarbeitungsprogramm längst ohne weiteres bewältigt werden können. Noch schlimmer, wir können dieselben Seiten mit unterschiedlichen Browsern immer noch nicht gleich anzeigen.

Aber ich schweife schon wieder ab.

Der Punkt ist, dass ASP, wenngleich leistungsstark, dennoch langsam, komplex und schwierig zu unterstützen ist.

Die gute Nachricht ist, dass Sie mit ASP-Seiten auf einfache Weise DLLs aufrufen können, die alles können, was ein ASP-Skript kann. Diese DLLs können in einer

beliebigen Sprache geschrieben werden, einschließlich Visual Basic. Diese DLLs werden prozessintern ausgeführt, d.h., sie sind schnell. Darüber hinaus kann mit diesen DLLs leicht zwischen dem serverseitigen Code (der Code, den die DLL umfasst) und dem clientseitigen Code (der Code, der als Teil des durch die DLL generierten HTML-Textes geschrieben wird) unterschieden werden.

Die ASP-Seiten, die ich persönlich bisher geschrieben habe, haben stets nur die Funktion gehabt, ein ActiveX-Objekt aus einer DLL zu laden und Methoden für das Objekt auszulösen. Es gibt jedoch einige Macken bei der Behandlung dieser DLLs durch die ASP-Seiten, was dazu führt, dass sich das Debuggen in einigen Fällen recht kompliziert gestaltet.

Lass mein Objekt gehen!

Sobald eine ASP-Seite ein Objekt einer DLL benötigt, wird die DLL über IIS (Internet Information Server) in den Speicher geladen. Wird das Objekt nicht länger benötigt, wird die DLL freigegeben. Vielleicht. Manchmal.

Wann entlädt IIS tatsächlich die DLL? Dies richtet sich danach, wie Sie ASP konfiguriert haben. Wenn Sie das Objekt innerhalb des IIS-Prozesses ausführen, kann dies einige Zeit dauern. Dies kann zu Schwierigkeiten führen, wenn Sie den Code bearbeiten, neu kompilieren und testen möchten.

Was geschieht, wenn Sie versuchen, eine DLL zu kompilieren, die von IIS geladen wurde? Die Kompilierung schlägt fehl – Visual Basic kann keine Schreibvorgänge für die DLL ausführen, da diese von IIS mit einer Sperre belegt ist.

In diesen Situationen ist zur erfolgreichen Kompilierung der Anwendung das Beenden der IIS-Dienste erforderlich. Sie führen anschließend die Kompilierung durch und starten die Dienste neu.

Das Anhalten und Neustarten der Dienste ist keine schwierige Sache, die vielleicht mit Hilfe des Internetdienst-Managers ausgeführt werden kann. Der Internetdienst-Manager von IIS4 macht es Ihnen jedoch nicht unbedingt leicht, diese Aufgabe auszuführen (im Gegensatz zu IIS3). Sicher, es ist einfach, einzelne Websites anzuhalten oder zu starten, aber das Anhalten des gesamten Dienstes sowie der Verwaltungsdienste ist nichts, was man intuitiv als Maßnahme ergreifen würde.

Ein einfacherer Weg stellt die Verwendung des Dienst-Managers in der Systemsteuerung dar. Sie müssen sowohl den WWW-Dienst als auch den IIS-Verwaltungsdienst anhalten. Dies ist einfach, aber ärgerlich, da der Dienst-Manager Ihnen nur das Starten bzw. Stoppen einzelner Dienste ermöglicht. Da Sie also zur Aktivierung bzw. Deaktivierung jedes Dienstes einige Sekunden benötigen, nimmt der gesamte Prozess des Startens oder Anhaltens eines Dienstes fünf oder sechs Sekunden in Anspruch, je nach Systemkonfiguration auch mehr.

Daher erschien es mir sinnvoll, ein Dienstprogramm zu entwicklen, dass diese Start- oder Stoppvorgänge für eine beliebige Dienstauswahl in einer Operation ausführt und mit dem schnell wieder in den ursprünglichen Dienstzustand zurückgewechselt werden kann.

Das Design

Ich bin einer der Fürsprecher für ein gutes Design, selbst bei noch so einfachen Dienstprogrammen. Sie können nie wissen, ob es sich bei einem kleinen Dienstprogramm nicht um einen dieser Codeabschnitte handelt, die Sie ständig verwenden werden. Daher kann ein gut überlegtes Design dazu führen, dass Sie Ihrer Codebibliothek statt eines fehlerverursachenden einen dauerhaften (und wiederverwendbaren) Codeabschnitt hinzufügen können.

Überlegen wir zunächst, welche Aufgaben ausgeführt werden müssen:

- Abrufen einer Liste der Dienste
- Abrufen des aktuellen Ausführungsstatus eines Dienstes
- Starten eines Dienstes
- Stoppen eines Dienstes

Die Liste der Dienste ist nötig, um dem Benutezr die Auswahl des zu startenden bzw. zu stoppenden Dienstes zu ermöglichen. Sicher, Sie können für die von IIS 4.0 (der aktuellen Version von Internet Information Server) benötigten Dienste eine Hardcodierung durchführen, aber dann muss das Dienstprogramm für zukünftige IIS-Versionen neu codiert werden, falls Microsoft Änderungen an den Namen der verwendeten Dienste vornimmt.

Sie müssen in der Lage sein, den aktuellen Status eines Dienstes abzurufen, sowohl, um erkennen zu können, ob die von Ihnen angeforderte Operation abgeschlossen wurde, als auch, um den anfänglichen Status des Dienstes zu ermitteln, um eine Wiederherstelloperation unterstützen zu können.

Bevor wir uns die verfügbaren API-Funktionen zur Verwendung mit Diensten ansehen, benötigen Sie weitere Kenntnisse zur Verwaltung von Diensten.

Es gibt zwei Objekttypen, mit denen wir uns beschäftigen.[1] Das erste Objekt ist der Service Control Manager. Der Dienstkontroll-Manager ist Teil des Betriebssys-

1. Beachten Sie, dass mit dem Begriff »Objekt« hier ein Win32-Objekt, kein COM/ActiveX-Objekt gemeint ist. Wie bei anderen Win32-Objekten können Sie erwarten, dass zum Erstellen oder Öffnen des Objekts eine API-Funktion verfügbar ist, und dass von dieser eine Zugriffsnummer zurückgegeben wird, ein 32-Bit-`Long`-Wert, mit dem das Objekt identifiziert wird. Darüber hinaus sollte eine weitere API-Funktion vorhanden sein, mit der das Objekt anschließend geschlossen werden kann.

tems, mit dem die Systemdienste sowie die interne Dienstdatenbank des Systems verwaltet werden.

Das andere Objekt ist der Dienst selbst. Zum Abrufen einer Zugriffsnummer für einen Dienst muss der Dienstkontroll-Manager eingesetzt werden.

Funktionen des Dienstkontroll-Managers

Mit Hilfe der Funktion `OpenSCManager` wird der Dienstkontroll-Manager geöffnet. In C lautet die Definition folgendermaßen:

```
SC_HANDLE OpenSCManager(
    LPCTSTR lpMachineName,         //
 Zeiger auf Zeichenfolge mit Computername
    LPCTSTR lpDatabaseName,        //
 Zeiger auf Zeichenfolge mit Datenbankname
    DWORD dwDesiredAccess          // Art des Zugriffs
   );
```

Der Rückgabewert ist ein 32-Bit-`Long`-Wert, der eine Zugriffsnummer auf das Dienstkontroll-Manager-Objekt enthält. Der Dienstkontroll-Manager kann auch von Remotesystemen aus geöffnet werden. Die Einstellung für den `lpDatabase-Name`-Parameter lautet stets NULL oder `SERVICES_ACTIVE_DATABASE` (Konstante). Durch beide Werte wird die Funktion veranlasst, die derzeit aktive Datenbank zu verwenden. Es sieht so aus, als ob die Entwickler vorhatten, mehrere Dienstdatenbanken zu unterstützen, diese Fähigkeit ist bisher jedoch nicht verfügbar. Die `dwDesiredAccess`-Konstante ermöglicht Ihnen die Angabe verschiedener Konstanten, mit denen der gewünschte Zugriffstyp definiert wird. Ob Sie diesen Zugriff erhalten, hängt davon ab, welche Berechtigungen mit dem verwendeten Konto verknüpft sind. Unser Beispiel setzt voraus, dass der Benutzer als Systemadministrator angemeldet ist, da jede Person, die ASP-Seiten und verknüpfte Komponenten erstellt, üblicherweise die Funktion eines Systemadministrators innehat.

Wenn Sie die Aufgaben im Zusammenhang mit dem Dienstkontroll-Manager erledigt haben, müssen Sie diesen mit Hilfe der nachfolgend definierten Funktion Close `ServiceHandle` schließen:

```
BOOL CloseServiceHandle(
    SC_HANDLE hSCObject            // Zugriffsnummer für Dienst-
 oder Dienstkontroll-Manager-Datenbank

   );
```

Die Funktion `CloseServiceHandle` wird auch zum Schließen eines Dienstes verwendet.

Sie können den Dienstkontroll-Manager auch dazu einsetzen, Dienste zu öffnen, die die folgende Funktion verwenden:

```
SC_HANDLE OpenService(
    SC_HANDLE hSCManager,    // Zugriffsnummer für Dienstkontroll-Manager-
                             // Datenbank
    LPCTSTR lpServiceName,   // Zeiger auf den Namen des zu startenden
                             // Dienstes
    DWORD dwDesiredAccess    // Art des Zugriffs auf den Dienst
   );
```

Die Funktion gibt eine Zugriffsnummer für einen Dienst zurück. Mit dem `dwDesiredAccess`-Parameter können Sie angeben, welche Operationen für den Dienst ausgeführt werden sollen. Der Dienstname entspricht nicht dem Namen, der unter dem Icon **Dienste** in der Systemsteuerung angezeigt wird. Dieser Name ist der »Anzeigename«, eine Version, die dem besseren Verständnis dient. Den tatsächlichen Dienstnamen rufen Sie ab, indem Sie die Dienste mit der Funktion `EnumServicesStatus` aufzählen lassen:

```
BOOL EnumServicesStatus(
    SC_HANDLE hSCManager,              // Zugriffsnummer für Dienstkontroll-
                                       // Manager-Datenbank
    DWORD dwServiceType,               // Art der aufzuzählenden Dienste
    DWORD dwServiceState,              // Status der aufzuzählenden Dienste

    LPENUM_SERVICE_STATUS lpServices, // Zeiger auf Dienststatuspuffer
    DWORD cbBufSize,                   // Größe des Dienststatuspuffers
    LPDWORD pcbBytesNeeded,            // Zeiger auf Variable für die
                                       // benötigten Bytes
    LPDWORD lpServicesReturned,        // Zeiger auf Variable für die
                                       // zurückgegebene Zahl
    LPDWORD lpResumeHandle             // Zeiger auf Variable für den
                                       // nächsten Eintrag
   );
```

Diese Funktion lädt eine Struktur mit dem Namen `ENUM_SERVICES_STATUS`, die im Folgenden definiert wird:

```
typedef struct _ENUM_SERVICE_STATUS {  // ess
    LPTSTR lpServiceName;
    LPTSTR lpDisplayName;
    SERVICE_STATUS ServiceStatus;
}  ENUM_SERVICE_STATUS, *LPENUM_SERVICE_STATUS;
```

```
typedef struct _SERVICE_STATUS {  // ss
    DWORD dwServiceType;
    DWORD dwCurrentState;
    DWORD dwControlsAccepted;
    DWORD dwWin32ExitCode;
    DWORD dwServiceSpecificExitCode;
    DWORD dwCheckPoint;
    DWORD dwWaitHint;
}  SERVICE_STATUS, *LPSERVICE_STATUS;
```

Die Zeichenfolge `lpServiceName` in der Struktur `ENUM_SERVICES_STATUS` enthält den Dienstnamen. Die Zeichenfolge `lpDisplayname` enthält den Namen, der dem Benutzer angezeigt wird.

Der Dienstkontroll-Manager unterstützt darüber hinaus weitere Funktionen, diese sind jedoch für die vorliegende Aufgabe nicht relevant.

Dienstfunktionen

Nachdem Sie einen Dienst geöffnet haben, sind verschiedene Operationen verfügbar, die Sie mit Hilfe von API-Funktionen für den Dienst ausführen können. Für unsere Zwecke werden lediglich drei dieser Funktionen benötigt.

1. Die Funktion `StartService` dient dem Starten eines Dienstes und wird folgendermaßen definiert:

   ```
   BOOL StartService(
   SC_HANDLE hService,                 // Zugriffsnummer für Dienst
   DWORD dwNumServiceArgs,             // Anzahl der Argumente
   LPCTSTR *lpServiceArgVectors        // Adresse des Arrays der
                                       // Argumentzeichenfolgenzeiger
   );
   ```

   Die Funktion ermöglicht Ihnen die Übergabe von Argumenten an den Dienst. Die meisten Dienste verwenden beim Start jedoch keine Argumente. Dies gilt auch für unseren Dienst. Daher können die Parameter `dwNumServicesArgs` und `lpServiceArgVectors` auf NULL gesetzt werden.

2. Ein Dienst wird mit Hilfe der Funktion `ControlService` angehalten, die wie folgt definiert ist:

   ```
   BOOL ControlService(
   SC_HANDLE hService,                 // Zugriffsnummer für Dienst
   DWORD dwControl,                    // Steuerungscode
   LPSERVICE_STATUS lpServiceStatus    // Zeiger auf Dienststatusstruktur
   );
   ```

Der dwControl-Parameter verwendet einen Konstantenwert, der mit dem Präfix SERVICE_CONTROL_* definiert wird. Für unsere Zwecke verwenden wird die Konstante SERVICE_CONTROL_STOP zum Anhalten des Dienstes. Durch die Funktion wird außerdem die Struktur SERVICE_STATUS geladen.

3. Die letzte benötigte Dienstfunktion lautet QueryServiceStatus, die nachfolgend definiert wird:

```
BOOL QueryServiceStatus(
SC_HANDLE hService,                 // Zugriffsnummer für Dienst
LPSERVICE_STATUS lpServiceStatus    // Adresse der Dienststatusstruktur
);
```

Diese Funktion kann dazu eingesetzt werden, den Status eines Dienstes abzufragen, was nach der Beendigung einer angeforderten Operation geschieht.

Objektauswahl

Wir haben nun den ersten Schritt des Designprozesses getan, nämlich das Verständnis der zugrundeliegenden Objekte, Datenstrukturen und Funktionen. Hierbei ist die Reihenfolge der aufgeführten Elemente entscheidend. Eine wichtige Aufgabe bei der objektorientierten Programmierung ist eine richtige Auswahl der Objekte für das Design. Wie treffen Sie die Auswahl der Objekte? Zunächst einmal ist die Chance sehr hoch, dass Sie über ein Objekt verfügen, das einem der im System verwendeten Objekte entspricht. Wir hatten zwei Systemobjekte ermittelt, den Dienstkontroll-Manager und den Dienst selbst. Dies führt zu zwei Klassen, einer mit Namen ServiceManager, die dem Dienstkontroll-Manager entspricht, und einer anderen mit Namen ServiceObject, die dem Dienst entspricht. Jede dieser Klassen enthält eine private Variable mit der Zugriffsnummer für das entsprechende Systemobjekt.

Als Nächstes verfügen Sie vielleicht über Objekte, die den verwendeten Datenstrukturen entsprechen. Wir haben zwei Objekte identifiziert, die ENUM_SERVICE_STATUS- und die SERVICE_STATUS-Struktur. Sie könnten nun für jede Struktur ein Objekt erstellen, die SERVICE_STATUS-Struktur ist jedoch in der ENUM_SERVICE_STATUS-Struktur enthalten, daher würde dies zu einer komplexen Objekthierarchie führen. Obschon durchführbar, stünde der Aufwand in keinem Verhältnis zum Ergebnis. Die Klasse ServiceStatus entspricht der ENUM_SERVICE_STATUS-Struktur und dient gleichzeitig der Behandlung der SERVICE_STATUS-Struktur.

Jetzt stellt die Zuweisung der Methoden zu den Klassen eine relativ einfache Aufgabe dar.

Die »ServiceManager«-Klasse umfasst die folgenden Methoden und Eigenschaften:

- **OpenSCManager-Funktion.** Wird zur Initialisierung der `ServiceManager`-Funktion eingesetzt. Unglücklicherweise unterstützt VB keine parametrisierten `constructor`-Funktionen[2] (im Gegensatz zu C++), daher müssen Sie nach der Erstellung der Klasse daran denken, diese Methode aufzurufen.
- **EnumServicesStatus.** Wird zur Aufzählung von Diensten verwendet. Diese Funktion gibt eine Auflistung zurück, die ein `ServiceStatus`-Objekt für jeden Dienst enthält.
- **OpenService.** Wird zum Öffnen eines Dienstes verwendet. Diese Funktion gibt ein `ServiceObject`-Objekt zurück.

Die »ServiceObject«-Klasse umfasst die folgenden Methoden und Eigenschaften:

- **Initialize.** Eine `Friend`-Methode, die von der `ServiceManager`-Klasse zur Initialisierung eines neu erstellen Objekts verwendet wird.
- **ServiceName.** Eine Eigenschaft mit Schreibschutz, die zum Abrufen des Dienstnamens verwendet wird.
- **QueryServiceStatus.** Eine Funktion, die ein `ServiceStatus`-Objekt zurückgibt, mit der der aktuelle Dienststatus definiert wird.
- **ServiceControl.** Eine Funktion, die der API-Funktion `ServiceControl` entspricht.
- **StartService.** Eine Funktion, die der API-Funktion `StartService` entspricht.

Die »ServiceStatus«-Klasse umfasst die folgenden Methoden und Eigenschaften:

- **Name.** Eine Eigenschaft mit Schreibschutz, mit der der Dienstname abgerufen wird.
- **DisplayName.** Eine Eigenschaft mit Schreibschutz, mit der der Anzeigename des Dienstes abgerufen wird.
- **CurrentState.** Eine Eigenschaft mit Schreibschutz, die dem Abruf des aktuellen Ausführungsstatus des Dienstes dient.

Darüber hinaus liegen zwei `Friend`-Funktionen vor, die intern verwendet werden und der Klasseninitialisierung dienen.

---

2. Bei der Erstellung einer Visual Basic-Klasse wird das Ereignis `Initialize` aufgerufen. Dieses Ereignis wird in C++-Objekten als »constructor« bezeichnet, wobei der Aufrufmechanismus sehr unterschiedlich ist. In C++ kann über die Funktion ein Objekt erstellt werden, mit dem Parameter an die `constructor`-Funktion des Objekts übergeben werden können. Tatsächlich kann ein Objekt über mehr als eine `constructor`-Funktion verfügen, die jeweils unterschiedliche Parametersätze aufweisen.

Die Programmierung kann fast beginnen

Rufen Sie sich für einen Moment die ursprünglichen Anforderungen für dieses Projekt in Erinnerung:

- Abrufen einer Liste der Dienste
- Abrufen des aktuellen Ausführungsstatus eines Dienstes
- Starten eines Dienstes
- Stoppen eines Dienstes

Mit den vorliegenden drei Klassen wird die gewünschte Funktionalität eindeutig bereitgestellt. Bleiben zwei Dinge. Nun, da wir über die Objektinfrastruktur zur Ausführung der Operationen verfügen, benötigen wir ein wohlüberlegtes Design zur exakten Funktionsweise des Dienstprogramms. Als Nächstes müssen wir die Visual Basic-Deklarationen bzw. den Code ermitteln, der zur tatsächlichen Implementierung dieser Klassen benötigt wird.

Die »Service«-Klassen

Das Design dieser Klasse folgt der Philosophie, die ich bei der Entwicklung der API-Klassenbibliothek von Desaware (einem Bestandteil von SpyWorks) beherzigte – die Methoden sollten so genau wie möglich den zugrundeliegenden API-Funktionen entsprechen. Der Vorteil dieses Ansatzes liegt darin, dass Sie Ihr Wissen in beiden Fällen einsetzen können. Niemand benötigt einen weiteren Satz von Befehlen, die ihm nicht vertraut sind.

Eines der ersten Dinge, die Sie bei der Verwendung von API-Dienstfunktionen beachten sollten, ist der häufige Einsatz von Konstanten. Konstanten können auf zweierlei Weise eingesetzt werden: erstens zur Erstellung eines **.BAS**-Moduls mit öffentlichen Konstanten, und zweitens kann man sie mit Hilfe des `Enum`-Operators als Teile der Objekttypenbibliothek offenlegen. Das Verwenden einer Typenbibliothek ist ein Ansatz, der auf der Hand liegt, denn er ermöglicht Ihnen die Verwendung der `Enum`-Liste als Datentyp, wodurch Benutzer des Objekts die Parameterwerte leicht auswählen können. Die `Enum`-Liste erscheint bei der Programmierung im Objektbrowser und in den Bildschirmtipps. Der Konstantenansatz ist sinnvoll für Klassen, die dazu gedacht sind, direkt in die Anwendung integriert statt als Komponenten verteilt zu werden.

Sie denken vielleicht, dass `Enums` so viele Vorteile aufweisen, dass Sie sie stets als Konstanten verwenden sollten. In der Theorie ist dies richtig, in der Praxis habe ich jedoch festgestellt, dass öffentliche `Enums` dazu neigen, Komponenten zu destabilisieren, sodass Sie teilweise nicht wiederverwendbar sind. Jedes Mal, wenn ich auf ein Programm stieß, das selbst mit richtigem Code spontan abstürzte, oder

bei dem ich die binäre Kompatibilität nicht erhalten konnte, stellte sich heraus, dass die Fehlerbehebung stets das Löschen der öffentlichen `Enums` umfasste bzw. zumindest beinhaltete, die öffentlichen `Enums` nicht als Parameter oder Rückgabewerte für Methoden oder Eigenschaften zu verwenden.

Aus diesem Grund tendiere ich dazu, diese als Datentypen zu verwenden. In diesem speziellen Beispielprogramm sollen die Klassen jedoch direkt in die Anwendungen integriert werden, daher zog ich diesen Ansatz vor, in der Hoffnung, dass sich VB6 bei der Verwendung dieser Datentypen als stabiler erweist. Die hier gezeigte `ServiceControlRights`-Aufzählung gibt den Berechtigungstyp an, der zur Verwendung mit dem Dienstkontroll-Manager benötigt wird. Die Aufzählung `ServiceAccessRights` gibt Auskunft über den Berechtigungstyp, der für das Dienstobjekt selbst erforderlich ist. Mit der Aufzählung `ServiceControlConstants` werden die Operationen angegeben, die bei Verwendung der API-Funktion `ServiceControl` verfügbar sind. Die Aufzählung `ServiceStateConstants` listet die Konstanten, mit denen der Ausführungsstatus eines Dienstes beschrieben wird.

```
' Dienstkontroll-Manager Class
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten

Option Explicit

' Service-Konstanten werden mit Hilfe von typelib offengelegt

Public Enum ServiceControlRights
    SC_MANAGER_CONNECT = 1
    SC_MANAGER_CREATE_SERVICE = 2
    SC_MANAGER_ENUMERATE_SERVICE = 4
    SC_MANAGER_LOCK = 8
    SC_MANAGER_QUERY_LOCK_STATUS = &H10
    SC_MANAGER_MODIFY_BOOT_CONFIG = &H20
    SC_MANAGER_ALL_ACCESS = &H3F Or STANDARD_RIGHTS_REQUIRED
End Enum


Public Enum ServiceAccessRights
    SERVICE_QUERY_CONFIG = 1
    SERVICE_CHANGE_CONFIG = 2
    SERVICE_QUERY_STATUS = 4
    SERVICE_ENUMERATE_DEPENDENTS = 8
    SERVICE_START = &H10
    SERVICE_STOP = &H20
```

```
    SERVICE_PAUSE_CONTINUE = &H40
    SERVICE_INTERROGATE = &H80
    SERVICE_USER_DEFINED_CONTROL = &H100
    SERVICE_ALL_ACCESS = &H1FF Or STANDARD_RIGHTS_REQUIRED
End Enum


Public Enum ServiceControlConstants
    SERVICE_CONTROL_STOP = 1
    SERVICE_CONTROL_PAUSE = 2
    SERVICE_CONTROL_CONTINUE = 3
    SERVICE_CONTROL_INTERROGATE = 4
    SERVICE_CONTROL_SHUTDOWN = 5
End Enum

Public Enum ServiceStateConstants
    SERVICE_STOPPED = 1
    SERVICE_START_PENDING = 2
    SERVICE_STOP_PENDING = 3
    SERVICE_RUNNING = 4
    SERVICE_CONTINUE_PENDING = 5
    SERVICE_PAUSE_PENDING = 6
    SERVICE_PAUSED = 7
End Enum
```

Lassen Sie uns einen Blick auf die einzelnen Funktionsdeklarationen des Dienst-Managers werfen.

OpenSCManager

In der Win32-API-Dokumentation wird diese Funktion folgendermaßen definiert:

```
SC_HANDLE OpenSCManager(
    LPCTSTR lpMachineName,  // Zeiger auf Zeichenfolge mit Computername
    LPCTSTR lpDatabaseName, // Zeiger auf Zeichenfolge mit Datenbankname
    DWORD dwDesiredAccess   // Art des Zugriffs
   );
```

Wie lautet der Funktionsname in der DLL? Da die Funktion Zeichenfolgenparameter verwendet, schließen Sie, dass es zwei Möglichkeiten gibt. Wenn die Funktion ausschließlich Unicode-Parameter unterstützt, lautet der Name in der DLL `OpenSCManager`. Werden sowohl ANSI- als auch Unicode-Parameter unterstützt, lautet der ANSI-Einsprungpunkt `OpenSCManagerA`, der Unicode-Einsprungpunkt

heißt OpenSCManagerW. Durchsuchen Sie das Systemverzeichnis mit Hilfe des Dienstprogramms DumpInfo (auf der Begleit-CD-ROM zu diesem Buch) nach beiden Namensversionen. Sie werden in der Datei **advapi32.dll** den Namen OpenSCManagerA finden.

Sie können voraussetzen, dass diese Funktion wie die Mehrzahl der API-Funktionen einen Long-Wert zurückgibt. Bei dem zurückgegebenen Parametertyp handelt es sich um SC_HANDLE, und da Sie wissen, dass nahezu jede Zugriffsnummer ein 32-Bit-Long-Wert ist, können Sie sich jetzt praktisch sicher sein, dass es sich um einen Long-Wert handelt. Einen weiteren Beweis hierfür finden Sie in den Headerdateien zur Win32 SDK. Die Datei **winsvc.h** enthält die folgende Zeile:

```
typedef HANDLE SC_HANDLE;
```

Computer- und Datenbankname weisen den Datentyp LPCTSTR auf. Der Datentyp LPCTSTR kann folgendermaßen entschlüsselt werden:

- LP Kennzeichnet einen Long-Zeigerparameter (32-Bit)
- C Kennzeichnet den Parameter als Konstante (die an die Funktion übergebenen Daten werden nicht durch die Funktion geändert)
- T Kennzeichnet die Daten als ANSI, wenn der Aufruf von einem ANSI-Einsprungpunkt aus erfolgt und als Unicode, wenn der Aufruf von einem Unicode-Einsprungpunkt aus erfolgt
- STR Kennzeichnet die Daten als auf NULL endende Zeichenfolge

Schlussfolgerung: Es handelt sich um Zeiger auf eine auf NULL endende ANSI-Zeichenfolge, die nicht durch die Funktion geändert wird.

Mit der dwDesiredAccess-Funktion verhält es sich recht einfach: Es handelt sich um einen Long-Wert, der ByVal übergeben wird. Die Deklaration lautet folgendermaßen:

```
' Service-Funktionen
Public Declare Function intOpenSCManager Lib "advapi32" Alias _
"OpenSCManagerA" (ByVal lpMachineName As String, ByVal _
lpDatabaseName As String, ByVal dwDesiredAccess As Long) As Long
```

Warum habe ich als Deklarationsnamen intOpenSCManager und nicht OpenSCManager angegeben? Der Aliasabschnitt der Deklaration verwendet OpenSCManagerA, also den Funktionsnamen, wie er durch die DLL offengelegt wird. Der Funktionsname wird von Visual Basic verwendet. Durch Verwenden eines anderen Namens für die Deklaration ist es möglich, OpenSCManager als Name für eine öffentliche Methode der Klasse selbst einzusetzen. Dies bedeutet, dass Sie vorsichtig sein müssen, wenn Sie die Klasse schreiben, damit Sie zwischen den beiden Namen

unterscheiden können. Clients, die diese Klasse verwenden, werden jedoch in der Lage sein, den geläufigeren Namen einzusetzen, das Zugreifen auf den in der API-Deklaration verwendeten Namen ist nicht erforderlich.

Die `CloseServiceHandle`-Funktion ergibt sich durch einen Vergleich. Diese Funktion verwendet einen `SC_HANDLE`-Parameter, nämlich einen 32-Bit-`Long`-Wert, der folgendermaßen `ByVal` übergeben wird:

```
Public Declare Function CloseServiceHandle Lib "advapi32" (ByVal _
schandle As Long) As Long
```

Die Funktion `EnumServicesStatus` ist ein wenig komplexer. Zunächst müssen Sie zwischen den Parametern unterscheiden, die als `DWORD` bzw. als `LPDWORD` spezifiziert sind. Bei den `DWORD`-Parametern handelt es sich um `Long`-Werte, die demnach als `Long`-Parameter definiert und `ByVal` übergeben werden. Bei den `LPDWORD`-Parametern bezeichnen hingegen Zeiger auf `DWORD`-Werte (`Long`), die demnach als `Long`-Parameter definiert und `ByRef` übergeben werden. Der `lpServices`-Parameter wird für den Moment als `Long`-Wert definiert, der `ByVal` übergeben wird. Warum? Weil wir den Zeiger im Code berechnen und den Zeigerwert an die Funktion übergeben:

```
BOOL EnumServicesStatus
    SC_HANDLE hSCManager,              // Zugriffsnummer für
                                       // Dienstkontroll-Manager-
                                       // Datenbank
    DWORD dwServiceType,               // Art der aufzuzählenden Dienste
    DWORD dwServiceState,              // Status der aufzuzählenden Dienste
    LPENUM_SERVICE_STATUS lpServices,  // Zeiger auf Dienststatuspuffer
    DWORD cbBufSize,                   // Größe des Dienststatuspuffers
    LPDWORD pcbBytesNeeded,            // Zeiger auf Variable für die
                                       // benötigten Bytes
    LPDWORD lpServicesReturned,        // Zeiger auf Variable für die
                                       // zurückgegebene Zahl
    LPDWORD lpResumeHandle             // Zeiger auf Variable für den
                                       // nächsten Eintrag
   );
```

Die sich ergebende Deklaration lautet folgendermaßen:

```
Public Declare Function intEnumServicesStatus Lib "advapi32" _
Alias "EnumServicesStatusA" _
    (ByVal hSCManager As Long, _
    ByVal dwServiceType As Long, _
    ByVal dwServiceState As Long, _
```

```
    ByVal lpServices As Long, _
    ByVal cbBufSize As Long, _
    pcbBytesNeeded As Long, _
    lpServicesReturned As Long, _
    lpResumeHandle As Long) As Long
```

Die `OpenService`-Funktion ist ähnlich einfach. Jeder der Parameter ist in der zuvor genannten Deklaration aufgeführt. Nachfolgend die Funktionsdeklaration:

```
Public Declare Function intOpenService Lib "advapi32" _
Alias "OpenServiceA" (ByVal hSCManager As Long, _
    ByVal lpServiceName As String, _
    ByVal dwDesiredAccess As Long) As Long
```

Das `ServiceManager`-Objekt muss nach seiner Erstellung initialisiert werden, damit der Dienst-Manager des Systems geöffnet werden kann. Die `OpenSCManager`-Methode entspricht der Win32-API-Funktion `OpenSCManager`. Der einzig interessante Trick wird in der Konvertierung der `Machine`- und `Database`-Zeichenfolgenvariablen von leeren Zeichenfolgen in NULL-Zeichenfolgen angewendet. Wenn Sie die leeren Zeichenfolgen beibehalten, erhält die API-Funktion einen Zeiger auf ein NULL-Zeichen statt auf einen NULL-Wert.

Die Funktion speichert die API-Zugriffsnummer in einer privaten Variablen mit Namen `schandle` und gibt den Wert zurück. Sie müssen nichts mit diesem Wert tun, die Tatsache, dass der Wert ungleich Null ist, zeigt, dass die Funktion erfolgreich war. Diese Zugriffsnummer wird während des Objektereignisses `Terminate` geschlossen. Nachfolgend die `OpenSCManager`-Methode sowie das `Terminate`-Ereignis:

```
' Dienstkontroll-Manager öffnen
' Bei Erfolg ungleich Null
Public Function OpenSCManager(ByVal Machine As String, ByVal Database _
As String, rights As ServiceControlRights) As Long
    ' Schließen, anschließend öffnen, falls nötig
    If schandle <> 0 Then Call CloseServiceHandle(schandle)

    ' Machine und Database müssen gültig oder NULL sein, nicht leer
    If Machine = "" Then Machine = vbNullString
    If Database = "" Then Database = vbNullString
    schandle = intOpenSCManager(Machine, Database, rights)
    OpenSCManager = schandle
End Function
```

```
Private Sub Class_Terminate()
    If schandle <> 0 Then Call CloseServiceHandle(schandle)
End Sub
```

Mit der `OpenService`-Funktion wird ein `ServiceObject`-Objekt erstellt und initialisiert. Die Funktion stellt zunächst sicher, dass das `ServiceManager`-Objekt initialisiert wurde, indem auf den Wert der `schandle`-Variablen hin geprüft wird. Anschließend wird die API-Funktion `OpenService` aufgerufen, um den Dienst zu öffnen. Wurde der Dienst erfolgreich geöffnet, wird die `Initialize`-Methode des `ServiceObject`-Objekts aufgerufen. Bei dieser Methode handelt es sich um eine `Friend`-Funktion. Die Funktion steht bei Verwendung dieses Objekts nicht zur Verfügung, wenn Sie diese Klassen in eine Komponente konvertieren. Im Idealfall würden Sie in der Lage sein, anzugeben, dass nur die `ServiceManager`-Klasse die `Initialize`-Methode aufrufen kann. In Visual Basic stellt die Verwendung einer `Friend`-Funktion die beste Möglichkeit dar. Da die Klassen in diesem Beispiel direkt in das Dienstprogramm integriert werden, besteht kein Unterschied zwischen der Verwendung eines `Friend`-Attributs und der Verwendung eines `Public`-Attributs, da die Funktion mit dem Attribut `Friend` für das gesamte Projekt sichtbar ist. Dennoch wird in diesem Fall das `Friend`-Attribut verwendet, falls Sie die Klasse später in einer Komponente verwenden möchten. Darüber hinaus weisen sie dadurch darauf hinaus, dass der Zugriff auf die Funktion beschränkt werden sollte. Nachfolgend die `OpenService`-Methode:

```
' Einen Dienst öffnen und Dienstobjekt zurückgeben
Public Function OpenService(Svc As ServiceStatus, rights As _
ServiceAccessRights) As ServiceObject
    Dim hnd&
    Dim newservice As New ServiceObject
    ' Dienst-Manager muss geöffnet sein, um diese Operation auszuführen
    If schandle = 0 Then Exit Function
    hnd = intOpenService(schandle, Svc.Name, rights Or SERVICE_QUERY_STATUS)
    If hnd <> 0 Then
        ' Dienst mit Hilfe einer Friend-Funktion initialisieren
        Call newservice.Initialize(hnd, Svc.Name, Svc.DisplayName)
        Set OpenService = newservice
    End If
End Function
```

Die EnumServicesStatus-Funktion ist die komplizierteste der benutzten Funktionen. Der zuvor beschriebene lpServices-Parameter verwendet einen Zeiger auf einen Speicherpuffer, der mit einer Reihe von ENUM_SERVICES_STATUS-Strukturen geladen wird. Sehen wir uns für einen Moment dieses Struktur und deren Definition in Visual Basic an. Die C-Deklaration lautet folgendermaßen:

```
typedef struct _ENUM_SERVICE_STATUS {  // ess
    LPTSTR lpServiceName;
    LPTSTR lpDisplayName;
    SERVICE_STATUS ServiceStatus;
}  ENUM_SERVICE_STATUS, *LPENUM_SERVICE_STATUS;

typedef struct _SERVICE_STATUS {  // ss
    DWORD dwServiceType;
    DWORD dwCurrentState;
    DWORD dwControlsAccepted;
    DWORD dwWin32ExitCode;
    DWORD dwServiceSpecificExitCode;
    DWORD dwCheckPoint;
    DWORD dwWaitHint;
}  SERVICE_STATUS, *LPSERVICE_STATUS;
```

Die ersten zwei Felder der ENUM_SERVICES_STATUS-Struktur sind Zeichenfolgen. Wenn Ihre erste Reaktion die ist, dass diese in Visual Basic-Zeichenfolgen übersetzt werden, kehren Sie sofort zu Tutorium 7, »Klassen, Strukturen und benutzerdefinierte Typen«, zurück, um herauszufinden, warum dieser Weg mit Sicherheit zu einer Katastrophe führt. Diese Felder müssen als Long-Variablen definiert werden, wie in der folgenden Typendeklaration gezeigt wird:

```
Public Type ENUM_SERVICE_STATUS
    lpServiceName As Long
    lpDisplayName As Long
    Status As SERVICE_STATUS
End Type

' Service-Strukturen
Public Type SERVICE_STATUS
    dwServiceType As Long
    dwCurrentState As Long
    dwControlsAccepted As Long
    dwWin32ExitCode As Long
    dwServiceSpecificExitCode As Long
```

```
    dwCheckPoint As Long
    dwWaitHint As Long
End Type
```

Diese Strukturen enthalten Zeiger auf Zeichenfolgen. Aber wo sind die Zeichenfolgen?

Der Puffer, der mit den ENUM_SERVICE_STATUS-Strukturen geladen wird, enthält auch die Zeichenfolgen. Abbildung T10-1 zeigt einen typischen Puffer, der von zwei dieser Strukturen geladen wird.

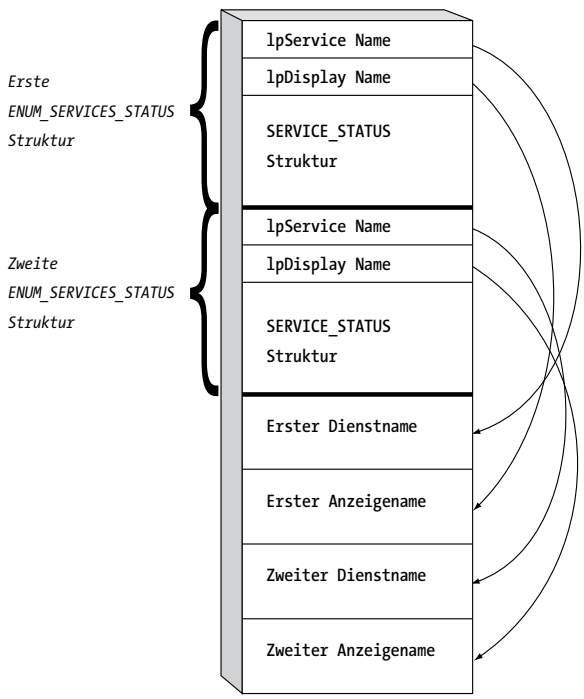


Abbildung T10-1 Format des mit den ENUM_SERVICE_STATUS-Strukturen geladenen Puffers

Der Puffer wird mit einer oder mehreren ENUM_SERVICE_STATUS-Strukturen geladen. Der Puffer enthält außerdem alle Zeichenfolgen, auf die durch die Strukturfelder verwiesen wird. Der Puffer wird anfänglich auf 512 gesetzt, einer willkürlichen Zahl. Bei der Rückgabe werden durch die API-Funktion EnumServicesStatus die folgenden Aufgaben ausgeführt:

1. Laden des Puffers mit sovielen ENUM_SERVICES_STATUS-Strukturen wie möglich
2. Einstellen des ServicesReturned-Parameters auf die Anzahl der zurückgegebenen Dienste

3. Einstellen des `BytesNeeded`-Parameters auf die Puffergröße, die zur Speicherung der verbleibenden Dienste benötigt wird
4. Einstellen des `ResumeHandle`-Parameters auf einen internen Wert, der dazu dient, das System darüber zu informieren, welcher Dienst während des nächsten Aufrufs als Erster geladen wird

Diese Parameterwerte können geändert werden, da sie als Verweis übergeben wurden (sehen Sie, wie allmählich alles zusammenpasst?). Die Funktion gibt den Wert 0 zurück und setzt die `Err.LastDllError`-Eigenschaft auf die Konstante `ERROR_MORE_DATA` ein, wenn keine zu lesenden Dienste mehr vorhanden sind.

Die Routine erstellt für jeden Dienst ein `ServiceStatus`-Objekt. Anschließend wird der Puffer zusammen mit einem Offset für die Objektmethode `ParseServiceInfo` übergeben. Mit dieser Methode wird der Puffer geprüft, und es werden die Dienststatusinformationen extrahiert, wie Sie gleich sehen werden.

```
' Auflistung mit ServiceStatus-Objekten abrufen
Public Function EnumServicesStatus() As Collection
    Dim res&
    Dim lpServices As Long
    Dim BytesNeeded As Long
    Dim ServicesReturned As Long
    Dim ResumeHandle As Long
    Dim Buffer() As Byte
    ReDim Buffer(512)
    Dim x&
    Dim CurrentOffset&
    Dim c As New Collection
    Dim s As ServiceStatus
    Dim continue As Boolean

    Do
        ' Puffer mit so vielen Diensten wie möglich laden
        ' Wir schließen für den Moment die Treiberdienste aus
        res = intEnumServicesStatus(schandle, &H30, 3, VarPtr(Buffer(0)), _
        UBound(Buffer()), BytesNeeded, ServicesReturned, ResumeHandle)
        If res = 0 And Err.LastDllError = ERROR_MORE_DATA Then
            ' Puffer konnte nicht alle Dienste speichern
            continue = True
        Else
            continue = False
        End If
```

```
        For x = 1 To ServicesReturned
            Set s = New ServiceStatus
            ' ServiceStatus-Puffer führt Extrahierung durch
            Call s.ParseServiceInfo(Buffer(), CurrentOffset)
            c.Add s
            ' 36 ist die Größe der ENUM_SERVICE_STATUS-Struktur
            CurrentOffset = CurrentOffset + 36
        Next x
        If continue Then
            ' Nächsten Aufruf vorbereiten
            CurrentOffset = 0
            ReDim Buffer(BytesNeeded)
        End If
    Loop While continue
    Set EnumServicesStatus = c
End Function
```

Die `ServiceStatus`-Klasse enthält den folgenden Code. Der aktuellste Status wird in einer privaten `SERVICE_STATUS`-Struktur mit Namen `Status` gehalten. Dienstname und Anzeigename werden ebenfalls gespeichert.

Es gibt zwei Möglichkeiten, ein `ServiceStatus`-Objekt zu initialisieren. Die erste ist die Analyse eines mit Hilfe der Funktion `EnumServicesStatus` erstellten Puffers. Die andere Möglichkeit ist die direkte Einstellung der privaten Informationen mit Hilfe der `InternalInitialize`-Funktion. Diese Funktion wird durch die `ServiceObject`-Klasse aufgerufen, wenn ein Client aktualisierte Statusinformationen anfordert.

```
' Enthält den letzten Wert jedes
Private Status As SERVICE_STATUS
Private SvcName As String
Private intDisplayName As String

' der durch die internen Objekte zur Initialisierung eines neuen
' ServiceStatus-Objekts verwendet wird,
' das nicht mit Hilfe einer Aufzählung erstellt werden.
Friend Sub InternalInitialize(pName As String, pDisplay As String, _
stat As SERVICE_STATUS)
    intDisplayName = pDisplay
    SvcName = pName
    LSet Status = stat
End Sub
```

Die ParseServiceInfo-Funktion kopiert zunächst den Teil des Puffers, der die Informationen zu diesem Dienst enthält, in eine private ENUM_SERVICE_STATUS-Struktur. Die ersten zwei Felder dieser Struktur enthalten Zeiger auf den Dienst- und den Anzeigenamen. Diese Zeiger müssen unter Verwendung der GetVBString-FromAddress-Funktion in Visual Basic-Zeichenfolgen konvertiert werden, die sich im Modul modSCMgr befinden und Ihnen bekannt sein sollten.

```
' Extrahieren der privaten Informationen aus dem Puffer. Es wird
' vorausgesetzt, dass die Daten am angegebenen Offset beginnen
' Immer Rückgabe von 0
Friend Function ParseServiceInfo(Buffer() As Byte, ByVal Offset As Long) _
As Long
    Dim svcstat As ENUM_SERVICE_STATUS
    ' Service-Struktur kopieren
    Call RtlMoveMemory(VarPtr(svcstat), VarPtr(Buffer(Offset)), Len(svcstat))
    ' Namen extrahieren
    SvcName = GetVBStringFromAddress(svcstat.lpServiceName)
    intDisplayName = GetVBStringFromAddress(svcstat.lpDisplayName)
    ' Nur SERVICE_STATUS-Abschnitt speichern
    LSet Status = svcstat.Status
End Function

' Abrufen einer VB-Zeichenfolge aus Adresse
Public Function GetVBStringFromAddress(ByVal addr&) As String
    Dim uselen&
    Dim res$
    If addr = 0 Then Exit Function
    uselen = lstrlen(addr)
    res$ = String$(uselen + 1, 0)
    Call lstrcpy(res, addr)
    GetVBStringFromAddress = Left$(res$, uselen)
End Function
```

Die ServiceStatus-Klasse verfügt auch über Eigenschaften zum Lesen des Namens, des Anzeigenamens und des aktuellen Ausführungsstatus des Dienstes.

Die ServiceObject-Klasse ist relativ leicht verständlich. In ihr wird intern die Dienstzugriffsnummer gespeichert, zusammen mit dem Namen und dem Anzeigenamen des Dienstes. Die Klasse verfügt des Weiteren über Methoden zum Stoppen und Starten eines Dienstes sowie zum Abrufen von Dienstinformationen. Diese Methoden rufen im Allgemeinen die entsprechende API-Funktion auf.

```
' Service Object Class
' Copyright © 1998 by Desaware Inc. Alle Rechte vorbehalten

' Dieses Objekt repräsentiert einen geöffneten Dienst

Option Explicit

Private intServiceHandle As Long
Private intServiceName As String
Private intDisplayName As String

' Aus Gründen der Bequemlichkeit wird dieses Objekt zusammen mit
' Name und Anzeigename über ServiceManager initialisiert
Friend Sub Initialize(ByVal hnd As Long, Name As String, _
DisplayName As String)
    intServiceHandle = hnd
    intServiceName = Name
    intDisplayName = DisplayName
End Sub

' Abrufen des/kurzen (short) Namens
Property Get ServiceName() As String
    ServiceName = intServiceName
End Property

' Abrufen des aktuellen Status
Property Get QueryServiceStatus() As ServiceStatus
    Dim stat As SERVICE_STATUS    Dim res&
    Dim sres As New ServiceStatus
    res = intQueryServiceStatus(intServiceHandle, stat)
    If res <> 0 Then
        ' Eine andere Methode zum Erhalt eines ServiceStatus-Objekts
        Call sres.InternalInitialize(intServiceName, intDisplayName, stat)
        Set QueryServiceStatus = sres
    End If
End Property

' Ausführen von Stop, Pause, Continue und Interrogate
Public Function ServiceControl(Operation As ServiceControlConstants) As Long
    Dim stat As SERVICE_STATUS
    Dim res&
```

```
    res = intControlService(intServiceHandle, Operation, stat)
    ServiceControl = res
End Function

' Start ausführen
Public Function StartService()
    Dim res&
    res = intStartService(intServiceHandle, 0, 0)
    StartService = res
End Function

' Zugriffsnummer bei Beenden schließen
Private Sub Class_Terminate()
    If intServiceHandle <> 0 Then
        Call CloseServiceHandle(intServiceHandle)
    End If
End Sub
```

Das Dienstprogramm selbst – Verwenden des Objektmodells

Das eigentliche Ziel dieser Übung bestand in der Erstellung eines Dienstprogramms, mit dem es einfacher ist, IIS (Internet Information Server) anzuhalten und erneut zu starten, damit die durch eine ASP-Seite aufgerufene DLL ersetzt werden kann. Dieses Programm sollte flexibel genug sein, um es als allgemeines Tool zum Starten und Anhalten von Diensten einzusetzen. Die folgenden Funktionen wurden implementiert:

- Ein Listenfeld zur Anzeige von Name und Anzeigename aller Dienste. Das Listenfeld ermöglicht die Auswahl mehrerer Dienste, die Sie dann gleichzeitig steuern können.
- Schaltflächen zum Starten und Stoppen eines oder mehrerer Dienste
- Eine Schaltfläche zur Wiederherstellung des ursprünglichen Dienststatus (beim Programmstart)
- Eine Möglichkeit zur Änderungenverfolgung, damit Sie über das Dienstprogramm gefragt werden können, ob die Dienste in ihren ursprünglichen Zustand zurückversetzt werden sollen, wenn das Programm geschlossen wird.
- Eine Schaltfläche zur Auswahl der IIS-Dienste

Abbildung T10-2 zeigt die Benutzeroberfläche des Dienstprogramms.

Abbildung T10-2 Die Benutzeroberfläche der Anwendung **SCToggle**

Das Programm definiert eine Reihe von Variablen. Die `sc`-Variable enthält das einzige durch das Programm verwendete `ServiceManager`-Objekt. Die `OriginalStates`-Auflistung enthält eine Gruppe von `ServiceStatus`-Objekten, die mit dem jeweiligen ursprünglichen Dienststatus geladen werden, die `ServiceCollection`-Auflistung hingegen eine Gruppe von `ServiceStatus`-Objekten, die den aktuellen Dienststatus wiedergeben (obwohl dieser nicht immer aktuell ist, wie Sie sehen können). Die `ChangesInProgress`-Auflistung gibt Auskunft über die `ServiceObject`-Objekte, die gerade geändert werden. Die Variable `Changed` gibt an, dass Sie den Status eines oder mehrerer Dienste geändert haben. Mit der Variablen `TargetState` nimmt Informationen darüber auf, ob Sie die geänderten Dienste anhalten oder starten oder wieder in den ursprünglichen Status zurückversetzen.

```
Dim sc As New ServiceManager

Dim ServiceCollection As Collection ' Enthalten aktuellen Status
Dim OriginalStates As Collection    ' Zeichnen ursprünglichen Status auf

' Auflistung der aktuell geänderten
' ServiceObject-Objekte
Dim ChangesInProgress As Collection

Dim Changed As Boolean  ' True, wenn eine Statusänderung vorgenommen wurde
Dim unloadpending As Boolean  ' True, wenn abschließende Wieder-
                              ' herstellungsoperation vorgenommen wird

' 0 zum Wiederherstellen
Dim TargetState As ServiceStateConstants
```

Das `ServiceManager`-Objekt wird beim Laden des Hauptformulars initialisiert. Zu diesem Zeitpunkt wird die `OriginalStates`-Auflistung mit dem ursprünglichen Ausführungsstatus für jeden Dienst geladen.

```
Private Sub Form_Load()
    Dim res&
    ' Öffnen von ServiceManager
    res = sc.OpenSCManager("", "", SC_MANAGER_ENUMERATE_SERVICE _
    Or SC_MANAGER_CONNECT)
    ' Aktualisieren der ServiceCollection und Anzeige der Dienste
    LoadList True
    Set OriginalStates = ServiceCollection
    ' Wir möchten keine zwei Verweise auf die Auflistung speichern
    Set ServiceCollection = Nothing
End Sub
```

Die `ServiceCollection`-Auflistung wird unter Verwendung der `LoadList`-Funktion geladen. Diese Funktion verwendet die `EnumServicesStatus`-Methode des `ServiceManager`-Objekts dazu, um Namen und Status der verfügbaren Dienste zu erhalten. Das `ServiceManager`-Objekt (Verweis durch `sc`-Variable) wird beim Laden des Formulars initialisiert:

```
' Lädt die ServiceCollection-Auflistung mit einer Liste aller Dienste
' Optionale Anzeige derselben
Private Sub LoadList(bDisplay As Boolean)
    Dim res&
    Dim svcstat As ServiceStatus
    lstServices.Refresh
    Set ServiceCollection = sc.EnumServicesStatus()
    If bDisplay Then
        For Each svcstat In ServiceCollection
            lstServices.AddItem svcstat.Name & ":   " & svcstat.DisplayName
        Next
    End If
End Sub
```

Das Programm verfügt über zwei mögliche Status. Der normale Status lautet `idle` (deaktiviert), bei dem die Benutzeroberfläche aktiviert ist; der Status `executed` (aktiviert) tritt ein, wenn Sie eine Dienststatusänderung vornehmen. Es kann einige Zeit in Anspruch nehmen, bis ein Dienst gestartet oder gestoppt wird, daher wird beim Statuswechsel ein Timer ausgeführt, der wartet, bis jeder Dienst den angeforderten Status angenommen hat. Die `EnablePanel`-Funktion wird zur Deaktivierung der Benutzeroberfläche verwendet, während sich das Programm im Ausführungsstatus befindet, und erneut aktiviert, wenn das Programm in den deaktivierten Modus zurückkehrt. Mit dieser Funktion werden auch Timer und Cursoranzeige gesteuert.

```
' Aktivieren oder Deaktivieren der Benutzeroberfläche der Anwendung.
' Gleichzeitige Steuerung von Mauszeiger und Timer.
' Die Benutzeroberfläche muss deaktiviert sein, wenn ein Dienst gestartet
' oder gestoppt wird.
Private Function EnablePanel(ByVal bEnabled As Boolean)
    cmdStop.Enabled = bEnabled
    cmdStart.Enabled = bEnabled
    cmdRestore.Enabled = bEnabled
    lstServices.Enabled = bEnabled
    If Not bEnabled Then
        Screen.MousePointer = vbHourglass
        Timer1.Enabled = True
    Else
        Screen.MousePointer = vbNormal
        Timer1.Enabled = False
    End If

End Function
```

Zunächst wird bei der Ausführung einer Start- oder Stoppoperation eine Liste der verfügbaren Dienste abgerufen. Das `OpenSelectedServices`-Objekt erstellt eine Auflistung der Dienste, die im Listenfeld ausgewählt wurden. Dies geschieht durch Laden der `ServiceCollection`-Auflistung mit den verfügbaren Diensten und deren aktuellen Status. Die Funktion vergleicht anschließend die Namen in der Liste mit den verfügbaren Diensten und fügt die übereinstimmenden Dienste der Auflistung `ChangesInProgress` hinzu.

```
' Laden der ChangesInProgress-Auflistung mit den
' ServiceObject-Objekten für die ausgewählten Dienste
Private Function OpenSelectedServices()
    Dim index&
    Dim currententry$
    Dim svcstat As ServiceStatus
    Call LoadList(False)
    Set ChangesInProgress = New Collection
    For index = 0 To lstServices.ListCount - 1
        If lstServices.Selected(index) Then
            currententry = lstServices.List(index)
            For Each svcstat In ServiceCollection
                ' Übereinstimmung basierend auf Dienstname
                If InStr(currententry, svcstat.Name & ":") = 1 Then
                    ChangesInProgress.Add sc.OpenService(svcstat, _
```

```
                    SERVICE_START Or SERVICE_STOP), svcstat.Name
                    Exit For
                End If
            Next
        End If
    Next index
End Function
```

Die Codeabschnitte zum Stoppen, Starten und Wiederherstellen der Dienste weisen große Ähnlichkeit auf. Nach dem Abrufen einer Liste der ausgewählten Dienste wird der aktuelle Dienststatus jedes Dienstes abgefragt, und es werden sämtliche Dienste aus der `ChangesInProgress`-Auflistung entfernt, die bereits den angeforderten Status angenommen haben. Die `ServiceControl`- oder die `StartService`-Methode werden, falls erforderlich, zum Stoppen oder Starten eines Dienstes verwendet. Verbleiben Dienste in der `ChangesInProgress`-Auflistung, wird die `EnablePanel`-Funktion aufgerufen, um den Ausführungstatus einzugeben.

```
' Stoppt die ausgewählten Dienste
Private Sub cmdStop_Click()
    Dim so As ServiceObject
    Dim stat As ServiceStatus
    Dim starttimer As Boolean
    Dim RemoveThis As Boolean
    Dim res&
    Call OpenSelectedServices
    For Each so In ChangesInProgress
        Set stat = so.QueryServiceStatus
        If stat.CurrentState <> SERVICE_STOPPED Then
            res = so.ServiceControl(SERVICE_CONTROL_STOP)
            If res <> 0 Then
                starttimer = True
            Else
                RemoveThis = True
            End If
        Else
            RemoveThis = True
        End If
        If RemoveThis Then ChangesInProgress.Remove stat.Name
        RemoveThis = False
    Next so
    If starttimer Then
        EnablePanel False
```

```
        TargetState = SERVICE_STOPPED
        Changed = True
    Else
        Set ChangesInProgress = Nothing
    End If
End Sub

' Startet die ausgewählten Dienste
Private Sub cmdStart_Click()
    Dim so As ServiceObject
    Dim stat As ServiceStatus
    Dim starttimer As Boolean
    Dim RemoveThis As Boolean
    Dim res&
    Call OpenSelectedServices
    For Each so In ChangesInProgress
        Set stat = so.QueryServiceStatus
        If stat.CurrentState <> SERVICE_RUNNING Then
            res = so.StartService
            If res <> 0 Then
                starttimer = True
            Else
                RemoveThis = True
            End If
        Else
            RemoveThis = True
        End If
        If RemoveThis Then ChangesInProgress.Remove stat.Name
        RemoveThis = False
    Next so
    If starttimer Then
        EnablePanel False
        TargetState = SERVICE_RUNNING
        Changed = True
    Else
        Set ChangesInProgress = Nothing
    End If
End Sub

' Stellt ursprünglichen Dienststatus wieder her
Private Sub cmdRestore_Click()
```

```
    Dim so As ServiceObject
    Dim stat As ServiceStatus
    Dim starttimer As Boolean
    Dim orig As ServiceStateConstants
    Dim res&
    Call LoadList(False)
    Set ChangesInProgress = New Collection
    For Each stat In ServiceCollection
        If stat.CurrentState <> OriginalState(stat.Name) Then
            ChangesInProgress.Add sc.OpenService(stat, SERVICE_START _
            Or SERVICE_STOP), stat.Name
        End If
    Next
    For Each so In ChangesInProgress
        orig = OriginalState(so.ServiceName)
        If orig = SERVICE_RUNNING Then
            res = so.StartService
        Else
            res = so.ServiceControl(SERVICE_CONTROL_STOP)
        End If
        If res <> 0 Then
            starttimer = True
        Else
ChangesInProgress.Remove so.ServiceName
        End If
    Next so
    If starttimer Then
        EnablePanel False
        TargetState = 0
        Changed = True
    Else
        Set ChangesInProgress = Nothing
    End If
End Sub
```

Wenn Sie die Anwendung schließen, haben Sie die Möglichkeit, den ursprünglichen Status der jeweiligen Dienste wiederherzustellen, falls einer der Dienste durch die Anwendung geändert wurde. Wenn Sie sich dazu entschließen, die Dienste wiederherzustellen, wird der Entladevorgang ausgesetzt, bis die Wiederherstelloperation beendet ist. In diesem Fall wird das `unloadpending`-Flag wird auf

`True` gesetzt, um der Anwendung mitzuteilen, dass die Dienste in ihren ursprünglichen Zustand zurückversetzt wurden. Anschließend wird die Anwendung beendet.

```
Private Sub Form_Unload(Cancel As Integer)
    Dim res&
    If Changed Then
        ' Falls eine Änderung vorgenommen wurde, dem Benutzer
        ' die Option zur Wiederherstellung geben
        res = MsgBox("Changes have been made - Restore service states?", _
        vbYesNo, "Exit confirmation")
        If res = vbYes Then
            unloadpending = True
            Me.Hide ' Formular verbergen - no reason to clutter the screen
            cmdRestore_Click
            Cancel = -1
            Changed = False
            Exit Sub
        End If
    End If

    Set ServiceCollection = Nothing
    Set OriginalStates = Nothing
    Set ChangesInProgress = Nothing
    Set sc = Nothing
End Sub
```

Mit der `timer`-Routine wird der Status sämtlicher Dienste in der `ChangesInProgress`-Auflistung geprüft. Hat der Dienst den Status wie gewünscht gewechselt, wird das entsprechende Objekt aus der Auflistung entfernt. Sobald die Auflistung keine Eintragungen mehr aufweist, wird die Benutzeroberfläche wieder aktiviert, und das Programm kehrt in den deaktivierten Status zurück.

```
' Beendigung der Operation in Bearbeitung abfragen
Private Sub Timer1_Timer()
    Dim so As ServiceObject
    Dim stat As ServiceStatus
    Dim target As Long
    For Each so In ChangesInProgress
        Set stat = so.QueryServiceStatus
        ' Null ist die Wiederherstellungsbedingung
        If TargetState = 0 Then
```

```
            target = OriginalState(stat.Name)
        Else
            target = TargetState
        End If

        If stat.CurrentState = target Then
            ' Diese Operation ist abgeschlossen
            ChangesInProgress.Remove stat.Name
        End If
    Next so
    If ChangesInProgress.count = 0 Then
        ' Erneutes Aktivieren der Anwendung, sobald alle Änderungen
        ' vorgenommen wurden
Set ChangesInProgress = Nothing
        Call EnablePanel(True)
        If TargetState = 0 Then Changed = False
    End If
    ' Falls dies die letzte Wiederherstellungsoperation vor dem Beenden
    ' war, Formular schließen
    If unloadpending Then Unload Me
End Sub
```

Was muss noch getan werden?

Wenn Sie sich den Beispielcode genau ansehen, kommt Ihnen vielleicht der Gedanke, dass hier noch etwas fehlt. Dies ist völlig richtig. Der Code ist noch weit davon entfernt, wie ich es nenne, eine für den kommerziellen Markt erforderliche Qualität aufzuweisen. Was fehlt?

- **Fehlerprüfung.** Obwohl einige der Funktionen Werte zurückgeben, mit denen angegeben wird, dass ein Fehler aufgetreten ist, werden keinerlei genauere Informationen zu diesem Fehler bereitgestellt. Die Funktionen mit Rückgabeobjekten geben den Wert `Nothing` zurück, wenn ein Fehler auftritt, bieten jedoch ebenfalls keine näheren Informationen. Es gibt verschiedene Möglichkeiten, dies zu ändern:
  - Hinzufügen eines `Error`-Parameters, der als Verweis übergeben wird und durch die aufgerufene Funktion gesetzt werden kann, sobald ein Fehler auftritt.
  - Erstellen einer `LastError`-ähnlichen Eigenschaft auf Klassenebene, die gelesen werden kann, um Informationen zum zuletzt aufgetretenen Fehler abzurufen.

- Fehlerausgabe mit Hilfe der Fehlerbehandlungmechanismen von Visual Basic.

- **Zeitüberschreitungen.** Das Dienstprogramm in seinem jetzigen Zustand wird in einer Unendlichkeitsschleife hängenbleiben, wenn ein Dienst nicht wie angefordert gestartet oder gestoppt werden kann. Die beste Möglichkeit, dies zu verhindern, stellt eine Zeitüberschreitung dar. Die `SERVICE_STATUS`-Struktur kann dazu verwendet werden, zusätzliche Informationen abzurufen, wenn ein Fehler bei einer Start- oder Stoppoperation für einen Dienst aufgetreten ist.
- **Fehlende Funktionen.** Wenn Sie die API-Dienstfunktionen betrachten, wird Ihnen auffallen, dass einige Funktionen ausgelassen wurden. Hierzu zählen die API-Funktionen zum Hinzufügen und Löschen sowie zum Konfigurieren von Diensten, das Ermitteln von Abhängigkeiten und das Einstellen der Dienstsicherheit. Eine vollständig Implementierung eines Klassensatzes würde diese Funktionen umfassen.
- **Interaktion mit weiteren Tools.** Das Dienstprogramm erstellt beim Laden eine Liste mit dem Status der einzelnen Dienste. Wenn Sie einen Dienst mit Hilfe eines anderen Tools starten oder stoppen, beispielsweise über die Systemsteuerung, werden diese Änderungen oder Aktualisierungen im Dienstprogramm nicht angezeigt. Dies bedeutet, dass bei einer Wiederherstellung der ursprünglichen Werte mit Hilfe des Dienstprogramms keine Änderungen widergespiegelt werden, die nach dem Laden der Anwendung vorgenommen werden.
- **Version mit Wiedereinstieg.** Die aktuelle Version des Dienstprogramm deaktiviert bei der Durchführung einer Start- oder Stoppoperation für einen Dienst jegliche Benutzereingabe. Dies ist nötig, um Wiedereinstiegsprobleme zu verhindern, wie sie beispielsweise durch einen Versuch entstehen, einen Dienst zu stoppen, während dieser gestartet wird. Bei einem anspruchsvolleren Programm könnten mehrere Start- und Stoppoperationen gleichzeitig ausgeführt werden, hier würden solche Konflikte durch einen ausgefeilteren Algorithmus vermieden. Ob sich der Aufwand lohnt, diese Funktionalität zu implementieren, liegt bei Ihnen.

Auf Wiedersehen, ASP?

Endet es nicht immer so? Sie machen sich die Mühe, ein kleines nützliches Dienstprogramm zu entwickeln, und noch bevor Sie es genutzt haben, ist es bereits überholt. Tatsächlich ist das Dienstprogramm weiterhin zur Verwendung mit ASP und für andere Anwendungen nützlich, bei denen Dienste gestartet oder gestoppt werden müssen. Und Sie können aus dem Beispiel eine Menge lernen. Ich für meinen Teil werde mit Hilfe von Visual Basic 6 Webklassen schreiben.

Tutorium 11

Lesen des Ereignisprotokolls

Der folgende Artikel wurde ursprünglich in meiner Kolume im ***Pinnacle's Visual Basic Developer Newsletter*** *veröffentlicht. Hierbei handelt es sich um eine Fallstudie, die sich so perfekt für dieses Buch eignet, dass ich mich entschied, den Artikel hier erneut abzudrucken, auch wenn einige der Leser den Originalartikel bereits kennen sollten. Dieses Tutorium erfordert Windows NT.*

Sie haben vielleicht bemerkt, dass viele meiner Artikel durch Fragen inspiriert sind, die mit einem Desaware-Produkt oder mit einem meiner Bücher in Zusammenhang stehen. Das liegt daran, dass ich immer dann, wenn ich eine Inspiration für einen Artikel benötige, meinen E-Mail-Posteingang nach einer Inspiration durchforste. Viele dieser Fragen werden nie beantwortet, denn, obwohl ich gerne den technischen Support für mein Win32-API-Buch (»Dan Appleman's Visual Basic Programmer's Guide to the Win32 API«) zur Verfügung stellen würde, ist es ökonomisch gesehen nicht machbar. Ich versuche, kurze Antworten auf die einfach zu beantwortenden Fragen zu geben, aber die Fragen, die eine eingehende Beschäftigung mit der Materie erfordern, werden häufig kurzerhand abgefertigt – es sei denn, ich verwende sie als Grundlage für einen Artikel.

Die heutige Inspiration ergab sich aus der folgenden Frage:

Ich versuche, die `ReadEventLog`-Funktion mit Visual Basic aufzurufen. Ich habe tonnenweise Beispiele zur Verwendung mit VC++, aber kein einziges für Visual Basic. Der Teil, mit dem ich nicht zurechtkomme, ist der `cast`-Typenoperator für einen Zeiger in C. Hier ist der Code, den ich konvertieren möchte. Ich bin für jede Hilfe dankbar.

```
EVENTLOGRECORD *pevlr;
BYTE bBuffer[BUFFER_SIZE];
DWORD dwRead, dwNeeded, cRecords, dwThisRecord = 0;

h = OpenEventLog(NULL,"Application");
if (h == NULL) ErrorExit("could not open Application event log");

pevlr = (EVENTLOGRECORD *) &bBuffer;

while (ReadEventLog(h, EVENTLOG_FORWARDS_READ |
EVENTLOG_SEQUENTIAL_READ, 0, pevlr, BUFFER_SIZE,  &dwRead, &dwNeeded))
   while (dwRead > 0) {
```

```
        printf("%02d  Event ID: 0x%08X ", dwThisRecord++, pevlr-
>EventID);
        printf("EventType: %d Source: %s\ n", pevlr-
>EventType, (LPSTR)((LPBYTE) pevlr + sizeof(EVENTLOGRECORD)));
         dwRead -= pevlr->Length;
        pevlr = (EVENTLOGRECORD *) ((LPBYTE) pevlr + pevlr->Length);
        }
     pevlr = (EVENTLOGRECORD *) &bBuffer;
     }
  CloseEventLog(h);
```

Die Übersetzung von C-Code ist für fortgeschrittene Visual Basic-Programmierer eine wichtige Aufgabe. Ich habe versucht, in meinem Buch die Hauptfunktionen der Win32-API abzudecken, da jedoch täglich neue API-Funktionen definiert werden, ist es nicht möglich, in einem Buch alle Themen zu behandeln, die für einen Visual Basic-Programmierer von Interesse sind. Dies ist einer der Gründe dafür, warum ich zwei Kapitel in meinem API-Buch ganz der Konvertierung von C-API-Deklarationen in Visual Basic gewidmet habe.

Dennoch stellt die beste Lernmethode immer noch die Praxis dar, und die hier gestellte Frage zum Ereignisprotokoll kann als eine interessante Übung angesehen werden.

Zunächst gilt es, die Definition der `EVENTLOGRECORD`-Struktur vorzunehmen. Diese Definition finden Sie in der Datei **api32.txt**, die auch in meinem Win32-API-Buch enthalten ist:

```
Type EVENTLOGRECORD
     Length as Long     '  Länge des gesamten Eintrags
     Reserved as Long     '  Verwendung durch Dienst
     RecordNumber as Long     '  Absolute Eintragsnummer
     TimeGenerated as Long     '  Sekunden seit 1-1-1970
     TimeWritten as Long     '  Sekunden seit 1-1-1970
     EventID as Long
     EventType as Integer
     NumStrings as Integer
     EventCategory as Integer
     ReservedFlags as Integer     '  Verwendung mit paarweisen Ereignissen
     ClosingRecordNumber as Long     '  Verwendung mit paarweisen
                                     '  Ereignissen
     StringOffset as Long     '  Offset des Eintraganfangs
     UserSidLength as Long
     UserSidOffset as Long
```

```
    DataLength as Long
    DataOffset as Long     '  Offset des Eintraganfangs
End Type
```

Die Konvertierung dieser Struktur aus der C-Typendeklaration ist einfach, da sie nur die Typen `DWORD` und `WORD` enthält. `DWORD`s werden in Visual Basic-`Long`-Variablen, `WORD`s in Visual Basic-`Integer`-Werte übersetzt.

Sehen wir uns als Nächstes die verwendeten API-Funktionen an:

```
HANDLE OpenEventLog(LPCTSTR lpUNCServerName, LPCTSTR lpSourceName );
BOOL ReadEventLog(HANDLE hEventLog, DWORD dwReadFlags, DWORD dwRecordOffset,
LPVOID lpBuffer, DWORD nNumberOfBytesToRead, DWORD *pnBytesRead, DWORD
*pnMinNumberOfBytesNeeded );

BOOL CloseEventLog(HANDLE hEventLog);
```

Die Visual Basic-Äquivalente finden Sie in der Datei `api32.txt`. Die Konvertierung ist einfach. In der Funktion `OpenEventLog` stellen die `LPCTSTR`-Parameter Zeichenfolgenzeiger auf standardmäßige, auf NULL endende C-Zeichenfolgen dar. Da die Funktion eine Zeichenfolge aufweist, können Sie zwei separate Einsprungpunkte erwarten. Diese lauten `OpenEventLogA` für die ANSI-Zeichenfolgen und `OpenEventLogW` für die Unicode-Zeichenfolgen. Das `T` in `LPCTSTR` kennzeichnet, dass es sich je nach verwendetem Einsprungpunkt um eine Unicode- oder ANSI-Zeichenfolge handelt. Da wir den ANSI-Einsprungpunkt verwenden, lautet die Deklaration folgendermaßen:

```
Declare Function OpenEventLog Lib "advapi32.dll" Alias "OpenEventLogA" _
(ByVal lpUNCServerName As String, ByVal lpSourceName As String) As Long
```

Die `ReadEventLog`-Funktion ist etwas komplexer. Bei den Parametern `HANDLE` und `DWORD` handelt es sich um 32-Bit-Werte, die als Wert übergeben und daher als `Long` definiert werden. Die `DWORD *`-Parameter sind Zeiger auf 32-Bit-Parameter. Dies bedeutet, dass es sich ebenfalls um `Long`-Werte handelt, sie werden jedoch als Verweis übergeben. In diesen Fällen wird in der Visual Basic-Deklaration deshalb das `ByVal` weggelassen. Obwohl der Parameter `lpBuffer` in der Datei **api32.txt** als eine `EVENTLOGRECORD`-Struktur bereits definiert wurde, erfolgt hier nun eine Neudefinition als Byte, das als Verweis übergeben wird. Warum dies notwendig ist, werden Sie gleich sehen. Die `CloseEventLog`-Funktion übergibt lediglich den `Long`-Parameter für die Zugriffsnummer als Wert.

```
Declare Function ReadEventLog Lib "advapi32.dll" Alias "ReadEventLogA" _
(ByVal hEventLog As Long, ByVal dwReadFlags As Long, ByVal dwRecordOffset _
As Long, lpBuffer As Byte, ByVal nNumberOfBytesToRead As Long, pnBytesRead _
```

```
As Long, pnMinNumberOfBytesNeeded As Long) As Long

Declare Function CloseEventLog Lib "advapi32.dll" _
(ByVal hEventLog As Long) As Long
```

Schrittweise Codeerstellung

Meine Erfahrung zeigt, dass die beste Methode, einen unbekannten C-Code in Visual Basic zu konvertieren, nach dem Prinzip vorgeht,, diesen schrittweise zu konvertieren und zu prüfen. Das `Event`-Beispiel verfügt über eine Befehlsschaltfläche, mit der der Code ausgeführt wird. Die Ergebnisse werden in einem Listenfeld angezeigt. Der erste Abschnitt enthält den folgenden Code:

```
Private Sub Command1_Click()
   Dim h As Long
   h = OpenEventLog(vbNullString, "Application")
   Debug.Print h
   if h<>0 Then Call CloseEventLog(h)
End Sub
```

Der einzige Unterschied zwischen diesem Code und dem C-Code besteht in der Verwendung von `vbNullString` zur Übergabe eines NULL-Wertes an den Zeichenfolgenparameter. Sie können keine leere Zeichenfolge (»«) verwenden, da auf diese Weise statt einer 0 ein Zeiger auf eine auf NULL endende Zeichenfolge an die Funktion übergeben wird. Es ist immer wichtig, auch die `CloseEventLog`-Funktion einzuschließen. Wenn Sie das Ereignisprotokoll nicht schließen können, bleibt dieses über die Prozessdauer hinweg geöffnet, in diesem Fall Visual Basic. Geöffnete Zugriffsnummern sind nicht nur unnütz, sondern können auch dazu führen, dass Sie das Ereignisprotokoll nicht öffnen können, wenn Sie das Programm ein zweites Mal öffnen. Wenn Sie dieses Programm ausführen und auf die Befehlsschaltfläche klicken, wird in einem speziellen Fenster ein Wert ungleich Null angezeigt. Dies beweist, dass das Ereignisprotokoll erfolgreich ausgeführt wurde.

C – Eine Sprache, die Verwirrung stiftet

Den Autor der ursprünglichen Frage irritierte der `cast`-Operator für die Zeiger. Ehrlich gesagt kann ich ihm das nicht verübeln. C ist eine der kryptischsten Sprachen, die je entwickelt wurden. Tatsächlich gab es mal ein Buch mit dem Namen **The C Puzzle Book** (oder ähnlich), dass ausschließlich komplizierteste C-Codebeispiele enthielt, denen der Leser eine Bedeutung entlocken sollte. Es machte Spaß, war aber auch etwas unnütz, denn kein guter Programmierer würde jemals einen

derartigen Code in einer richtigen Anwendung einsetzen. Code von professioneller Qualität sollte lesbar sein.[1]

Sehen wir uns die Schleife im C-Code erneut an, aber eliminieren wir nun einige der Parameter und ersetzen sie durch eine Beschreibung dessen, was tatsächlich passiert.

```
while ( The function ReadEventLog(As many EVENTLOGRECORD structures as will
fit entirely in a buffer) reads one or more structures) {
   while (The number of bytes actually read from the event log > 0) {
      Display information from this record
      dwRead -= pevlr->Length; Subtract the size of this record
      And change the pointer to point to the next record in the buffer
      pevlr = (EVENTLOGRECORD *) ((LPBYTE) pevlr + pevlr->Length);
      }
   Set the EVENTLOGRECORD pointer to the start of the buffer
   pevlr = (EVENTLOGRECORD *) &bBuffer;
   }
```

Die `pevlr`-Variable ist eine Zeigervariable, die anfänglich auf den Start eines Puffers mit willkürlich gewählter Größe verweist. In diesem Beispiel werden so viele Einträge wie möglich in den Puffer gelesen. Anschließend wird jeder Eintrag im Puffer geprüft.

Es ist wichtig, sich die Zeit zu nehmen, um die Funktionsweise des Codes zu verstehen, anstatt ihn einfach nur Zeile für Zeile in Visual Basic zu übertragen. Sollten Sie also kein C-Programmierer sein, erstellen Sie eine Sicherung, und lesen Sie sich die Dokumentation für die Funktion durch, bevor Sie sich weiter mit dem C-Code beschäftigen. Sobald Ihnen klar wird, dass die `ReadEventLog`-Funktion in der Lage ist, mehrere Ereignisprotokolleinträge zu lesen, beginnt der Code Sinn zu machen.

Sie fragen sich vielleicht, warum Sie in einer Doppelschleife mehrere Einträge lesen sollten. Warum ein Bytearray verwenden, wenn Sie auch eine oder mehrere `EVENTLOGRECORD`-Strukturen direkt lesen können?

Der Trick ist die Struktur selbst. Wie Sie der Strukturdeklaration entnehmen können, sind einige der Variablen in der Struktur tatsächlich Offsets für Zeichenfolgen und weitere Variablenlängendaten. Daraus ergibt sich folgende Frage: Wo befinden sich diese zusätzlichen Daten? Die Antwort lautet, dass die Daten zusammen und direkt nach der `EVENTLOGRECORD`-Struktur im Puffer platziert werden. Weitere

1. Die Inspiration für dieses Buch war eben dieses C-Puzzlebuch. Ich habe jedoch versucht, Puzzle zu verwenden, die auf die Bedürfnisse eines Visual Basic-Programmierers abzielen und Spaß machen.

Informationen hierzu finden Sie in der Win32 SDK-Dokumentation für diese Struktur. Nach dem letzen Feld in der Struktur werden die folgenden Felder angezeigt:

```
// TCHAR SourceName[]
// TCHAR Computername[]
// SID   UserSid
// TCHAR Strings[]
// BYTE  Data[]
// CHAR  Pad[]
// DWORD Length;
```

Diese Felder können nicht in der Struktur selbst definiert werden, da sie Variablenlänge aufweisen. Wir müssen daher einen Puffer zuweisen, der nicht nur die `EVENTLOGRECORD`-Struktur, sondern auch sämtliche Ereignisdaten enthalten kann. Das `Length`-Feld enthält die Gesamtlänge der Struktur und gibt die Länge der Variablenlängenfelder an. Es wird sowohl am Anfang als auch am Ende der Struktur gespeichert, um eine Pufferprüfung in Vorwärts- oder Rückwärtsrichtung zu vereinfachen. Die folgenden Variablen entsprechen exakt denen im C-Beispiel:

```
Private Const BUFFER_SIZE = 8192  ' Beliebige Zahl
Dim bBuffer(BUFFER_SIZE) As Byte
Dim dwRead As Long
Dim dwNeeded As Long
Dim cRecords As Long
Dim dwThisRecord As Long
```

Der Haken ist folgender: Nachdem wir einen Puffer mit einer oder mehreren dieser Struktur-plus-Daten-Kombinationen geladen haben, wie extrahieren wir dann die Informationen unter Verwendung von Visual Basic?

Der erste Schritt besteht darin, Zugriff auf die Strukturdaten zu erlangen. Am einfachsten erreichen Sie dies, indem Sie eine temporäre `EVENTLOGRECORD`-Struktur zur Speicherung der Daten definieren. Diese wird dann unter Verwendung der folgenden Speicherkopierroutine aus dem Puffer geladen:

```
Private Declare Sub RtlMoveMemory Lib "kernel32" (dest As Any, _
source As Any, ByVal bytes As Long)
```

Sehen Sie die `As Any`-Parameter? Diese implizieren, dass die Funktion jeden Parametertyp tragen kann. Der Zweck dieser Funktion, nämlich das Kopieren eines Speicherblocks von einem Standort an einen anderen, macht die Funktion extrem gefährlich. Jeder Fehler kann leicht zu einer Speicherbeschädigung oder zu einem Speicherausnahmefehler führen. Wir benötigen außerdem eine Zeigervariable. Da

Visual Basic keine direkte Zeigerunterstützung bietet, verwenden wir eine Long-Variable. Dies setzt zwei Variablen voraus:

```
Dim ev As EVENTLOGRECORD
Dim pevlr As Long
```

C ermöglicht Ihnen den Datenzugriff über Zeiger sowie das Konvertieren von Zeigern von einem Typ in einen anderen. Diese Konvertierung wird als »Casting« bezeichnet. Wir täuschen diesen Prozess in Visual Basic vor, indem wir Daten in einen benutzerdefinierten Typ kopieren:

```
pevlr = VarPtr(bBuffer(0))
RtlMoveMemory ev, ByVal pevlr, Len(EVENTLOGRECORD)
```

Der VarPtr-Operator ist ein nicht dokumentierter Operator in Visual Basic, mit dem die Adresse einer Variablen abgerufen werden kann. Wir erhalten tatsächlich die Adresse des ersten Bytes, da dieses (gemäß Definition) am Anfang des Puffers erscheint. Beachten Sie, dass die Adresse als Wert an die RtlMoveMemory-Funktion übergeben werden muss. Wenn Sie diesen Wert als Verweis übergeben, wird die Adresse der pevlr-Variablen weitergereicht, nicht aber die Adresse des bBuffer-Arrays.

Nachdem wir die Daten in die temporäre ev-Struktur geladen haben, kann auf die Ereignisdaten zugegriffen werden. Aber wie rufen wir die Zeichenfolgendaten ab, die der EVENTLOGRECORD-Struktur im Puffer folgen? Hierzu benötigen wir zwei weitere API-Funktionen, lstrlen und lstrcpy, die folgendermaßen deklariert werden:

```
Declare Function lstrlenptr Lib "kernel32" Alias "lstrlenA" _
(ByVal lpString As Long) As Long
Declare Function lstrcpyfromptr Lib "kernel32" Alias "lstrcpyA" _
(ByVal lpString1 As String, ByVal lpString2 As Long) As Long
```

Die lstrlenptr-Funktion verwendet die lstrlen-API-Funktion zum Abrufen der Länge einer Zeichenfolge. Beachten Sie jedoch, dass die Funktion für Verwendung mit einer Long-Variablen deklariert wurde. Dies ist notwendig, da der Funktion statt einer Zeichenfolge ein Zeigerwert übergeben werden soll. Die gleiche Technik wird bei der lstrcpyfromptr-Funktion eingesetzt, die eine Zeichenfolge von einer Speicherstelle in eine Visual Basic-Zeichenfolge kopiert. Zur Vereinfachung können Sie die folgende Visual Basic-Funktion verwenden, bei der eine Speicheradresse abgerufen, die Länge der Zeichenfolge abgefragt, eine Visual Basic-Zeichenfolge zugeordnet und diese anschließend aus der Speicheradresse geladen wird:

```
Private Function LoadStringFromPtr(ByVal ptrval&) As String
   Dim slen&
   Dim resstring$
   If ptrval = 0 Then Exit Function
   slen = lstrlenptr(ptrval)
   resstring = String$(slen + 1, 0)
   Call lstrcpyfromptr(resstring, ptrval)
   LoadStringFromPtr = Left$(resstring, slen)
End Function
```

Die `resstring`-Variable wird anfänglich mit einem Zusatzbyte definiert. Dies ist erforderlich, da über die `lstrcpyfromptr`-Funktion zusammen mit den Zeichenfolgen ein abschließendes NULL-Zeichen kopiert wird. Zu guter Letzt wird mit Hilfe der `Left$`-Funktion das NULL-Zeichen abgeschnitten, da dieses in einer Visual Basic-Zeichenfolge nicht benötigt wird.

Jetzt kommt alles zusammen

Gehen wir zurück zum Ereignisprotokoll. Die erste Zeichenfolge nach der `EVENTLOGRECORD`-Struktur im Puffer ist eine Zeichenfolge, mit der die Ereignisquelle beschrieben wird. Der Standort dieser Zeichenfolge lautet `pevlr + Len(ev)`. Dies ist der letzte Abschnitt der Visual Basic-Routine zum Lesen von Ereignissen. Die Routine entspricht sehr genau dem zuvor gezeigten C-Beispiel:

```
Private Sub Command1_Click()
   Dim h As Long
   Dim ev As EVENTLOGRECORD
   Dim pevlr As Long
   h = OpenEventLog(vbNullString, "Application")
   List1.Clear
   If h <> 0 Then
      pevlr = VarPtr(bBuffer(0))
      While (ReadEventLog(h, EVENTLOG_FORWARDS_READ Or _
            EVENTLOG_SEQUENTIAL_READ, _
            0, bBuffer(0), BUFFER_SIZE, dwRead, dwNeeded) <> 0)
         While dwRead > 0
            ' Kopieren der Daten in die ev-Struktur
            RtlMoveMemory ev, ByVal pevlr, Len(ev)
            List1.AddItem " Event Type: " & Hex$(ev.EventType) & _
                          " Source: " & LoadStringFromPtr(pevlr + _
                            Len(ev))
            dwRead = dwRead - ev.Length
```

```
            pevlr = pevlr + ev.Length
         Wend
         ' Zeiger auf den Anfang des Puffers zurücksetzen
         pevlr = VarPtr(bBuffer(0))
      Wend
      Call CloseEventLog(h)
   End If
End Sub
```

Hier eine letzte Herausforderung für Sie:

Im Puffer erscheint direkt nach der Ereignisquellenzeichenfolge der Computername. Finden Sie heraus, wie Sie diesen Computernamen abrufen. Die Antwort liegt im Beispielprogramm verborgen.

Teil IV
Anhang

Anhang A

Hinweise

Die besten Hinweise, die ich Ihnen zur Lösung der Puzzle anbieten kann, sind die Zehn Gebote der sicheren API Programmierung. Es gibt nur wenige Probleme, die sich nicht mit den zehn Geboten lösen lassen.

Puzzle 1

- S. Visual Basic Programmer's Guide to the Win32 API, Kapitel 5
- S. Tutorial 1, Funktionen finden, in Teil III dieses Buchs
- S. API-Gebot 8

Puzzle 2

- Unter Windows haben viele Standardwerte beschreibende konstante Namen. So hat `&H10` den konstanten Namen `WM_CLOSE`.

Puzzle 3

- S. API-Gebot 1
- S. API-Gebot 9
- Berücksichtigen Sie die `ScaleMode`-Eigenschaft.

Puzzle 4

- Geben Sie nicht auf, ehe Sie das Problem gelöst haben.
- S. API-Gebot 4
- S. Tutorial 6

Puzzle 5

- S. Gebot 4

Puzzle 6

- S. Gebot 9
- S. Tutorial 6

Puzzle 7

- So ist die Programmlogik.

Puzzle 8

- Lesen Sie die Beschreibung der `GetVersionEx`-Funktion in der Win32-Online-Dokumentation oder in meinem API-Buch.

Puzzle 9

- Die Funktionen-Deklaration ist korrekt.

Puzzle 10

- Die **api32.txt**-Datei (auch auf der Buch-CD) benutzt die folgende Deklaration für `GetEnvironmentStrings`:

```
Declare Function GetEnvironment Strings Lib "kernel32" Alias
"GetEnvironmentStringsA" () As Long
```

- S. Tutorial 6

Puzzle 11

- Wenn Sie mehr als zehn Minuten über dem Problem gesessen haben, nehmen Sie sich eine Pause, fangen Sie von Neuem an, und seien sie sich immer gewiss, dass die Probleme, die auf der Hand liegen, die zu lösen schwierigsten sind.

Puzzle 12

- Die `GetKeyInfo`-Funktion kann in einer Code-Zeile implementiert werde.
- Alle Informationen, die Sie zur Lösung des Puzzles brauchen, haben Sie bereits!

Puzzle 13

- Es sind zwei Bugs im Code.
- Manchmal wird es scheinbar schlimmer, wenn ein Fehler behoben wird.

Puzzle 14

- Machen Sie eine Liste von Deklarationen, die Sie für verschiedene Datentypen benutzen würden.

Puzzle 15

- Wußten Sie, dass es möglich ist, ein Byte-Array direkt einer Zeichenkette zuzuweisen?

Puzzle 16

- Nur weil die Puzzle in diesem Teil für Fortgeschrittene sind, heisst es noch lange nicht, dass ich nicht auch ein einfaches dazwischenschieben kann.

Puzzle 17

- Das Wechseln der Deklaration ist nicht genug, um das Puzzle zu lösen.
- Schauen Sie sich die Dokumentation für die Funktion an!
- S. Win32 API-Buch, Kapitel 12
- S. Win32 SDK oder MSDN, `DocumentProperties`-Referenz
- Lesen Sie die Referenzen noch einmal ganz ganz aufmerksam.
- S. Tutorial 4

Puzzle 18

- S. Tutorial 7

Puzzle 19

- Achten Sie auf die Einrückungsregeln!
- Wußten Sie, dass Visual Basic **Conditional Compilation** unterstützt?

Puzzle 20

- Sie müssen mit dem VB **Enum Keyword** vertraut werden.

Puzzle 21

- Es gibt zwei Lösungen für dieses Problem – Können Sie eine finden?
- Vergessen Sie nicht den Wert des Experimentierens!

Puzzle 22

- Was passiert, wenn Sie sich den Code noch einmal genau anschauen?
- S. Tutorial 7

Puzzle 23

- S. Tutorial 2, besonders `Signed-` und `Unsigned`-Variablen
- Wie viele Bytes sind 168 Bit?

Puzzle 24

- Was erwartet eine VB-Funktion, wenn ein Parameter als `ByVal As` String deklariert ist? Ist es das gleiche, was eine API-Funktion unter denselben Umständen erhalten würde?
- Vielleicht wollen Sie noch einmal die `EnumSystemLocales`-Function in der Online-Dokumentation oder in meinem API-Buch nachschlagen, aber es ist nicht nötig, um das Problem zu lösen.

Puzzle 25

- S. Tutorial 8; es könnte hilfreich sein!

Puzzle 26

- Sie sollten Puzzle 25 noch einmal ganz genau anschauen, wenn Sie 26 zu lösen versuchen.

Puzzle 27

- Ein OLE-String ist ein `BSTR`, kein `OLESTR`.

Puzzle 28

- Überlegen Sie: Was genau ist eine Objekt-Variable?
- Denken Sie über die Objekt-Beziehungen im Zusammenhang mit ActiveX Controls nach.
- Haben Sie sich je gewundert, wie VB wissen kann, wann Sie bereit sind, eine Seite zu drucken?

Puzzle 29

- S. Tutorial 2

Puzzle 30

- Es gibt einen Grund dafür, dass das Puzzle in dem Teil über »Rocket Science« auftaucht.

Puzzle 31

- Die Windows-Dokumentation sagt deutlich, dass diese Funktion von Windows 95 und Windows NT unterstützt wird.

Anhang B

FAQs

Dieser Anhang basiert auf den Fragen, die mir immer wieder gestellt werden. Je mehr hinzukommen, desto mehr Antworten gibt es auf meiner website **http://www.desaware.com**. In den meisten Fällen helfen diese FAQs, die Puzzle zu lösen.

Die Grundlagen

Wie finde ich heraus, welche DLL eine Funktion enthält?

- S. Puzzle 1
- S. Tutorial 1

Sind Funktionsnamen case-sensitive?

- S. Puzzle 1

Wie erhalte ich eine detaillierte Fehlermeldung für API-Funktionen?

- S. Puzzle 2
- S. Win32 API-Buch, Kapitel 3

Was ist der `LastError`-Wert und wie benutze ich ihn?

- S. Puzzle 2
- S. mein Win32 API-Buch, Kapitel 3 und 6

Was verursacht den »Bad DLL Calling Convention Error«?

- S. Tutorial 4

Wie werden Parameter API-Funktionen übergeben?

- S. Tutorial 4

Wann benutzt man `ByVal` in einer Deklaration oder einem Funktionsaufruf?

- S. Tutorial 5

Wie übergebe ich Handles API-Funktionen?

- S. Puzzle 3

Wie gehe ich mit `Signed`- und `Unsigned`-Numbers um?

- S. Tutorial 2
- S. Tutorial 6

Wie gehe ich mit API-Funktionen um, die Boolesche Werte ausgeben?

▶ S. Tutorial 3

Wie kombiniere ich Konstanten in Verbindung mit API-Aufrufen?

▶ S. Tutorial 3

Was ist der `VarPtr`-Operator?

▶ S. Tutorial 5

Informationen über Funktionen und C-Tricks

Wie finde ich Constant Values?

▶ Puzzle 23

▶ Schauen Sie in die **api32.txt**-Datei auf der Buch-CD.

▶ Suchen Sie in den C-Headerdateien auf der Buch-CD.

Wie lese ich C-Headerdateien?

▶ S. Puzzle 6

▶ S. Puzzle 19

▶ S. Puzzle 25

▶ S. Tutorial 6

▶ S. Tutorial 8

Was ist eine C++ »`cast`«-Operation?

▶ S. Puzzle 6

▶ S. Puzzle 23

Wie gehe ich mit C-Bitfeldern um?

▶ S. Tutorial 3

Wie rufe ich eine Dll-Funktion auf, die die C Calling Convention anstelle der `stdcall` benutzt?

▶ S. Puzzle 29

Allgemeine Fragen über Parameter

Was mache ich mit reservierten API-Parametern?

▶ S. Puzzle 12

Wie definiere ich eine Callback-Funktion?

▶ S. Puzzle 20

Alles über Strings

Was ist ein Null-String?

▶ S. Puzzle 2

Wie übergebe ich Strings API-Funktionen?

▶ S. Puzzle 4
▶ S. Puzzle 5
▶ S. Puzzle 16
▶ S. Tutorial 5

Wie entledige ich mich der NULL in einem String?

▶ S. Puzzle 1
▶ S. Puzzle 4
▶ S. Puzzle 5
▶ S. Puzzle 16

Wie unterscheiden sich `VarPtr` und `StrPtr`?

▶ S. Puzzle 26
▶ S. Tutorial 5

Wie übergebe ich API-Funktionen, die Strings ausgeben?

▶ S. Puzzle 10

Was ist ein »double NULL-terminated string« und wie arbeite ich mit ihm?

▶ S. Puzzle 10

Wie kann eine API Callback-Funktion einer VB-Funktion einen String übergeben?

▶ S. Puzzle 24

Alles über Strings, Arrays und ANSI-to-Unicode Conversions

Wie konvertiere ich ein Array mit Unicode-Zeichen in einen VB-String?

▶ S. Puzzle 15

Wie konvertiere ich einen Zeiger auf einen ANSI-String in einen VB-String?

▶ S. Puzzle 22

Wie bekomme ich einen Zeiger zu einem Unicode-String?

▶ S. Puzzle 23
▶ S. Puzzle 26

▶ S. Tutorial 5

Was sind ANSI- und Unicode Entry-Points und wie gebrauche ich Sie?

▶ S. Tutorial 5

Wie übergebe ich Unicode-Strings API-Funktionen?

▶ S. Tutorial 5

Alles über Arrays

Wie übergebe ich Arrays API-Funktionen?

▶ S. Puzzle 3

▶ S. Puzzle 7

Wie ordne ich einzelne Bits in einen Byte Array an oder stelle sie wieder her?

▶ S. Puzzle 23

Alles über Strukturen

Weshalb verlangen einige Strukturen, dass ich ihre Größe/Dimension bestimme, bevor ich sie als einen Parameter einer API-Funktion übergebe?

▶ S. Puzzle 8

▶ S. Puzzle 18

Wie handhabe ich variable Längen von Strukturen?

▶ S. Puzzle 17

Was sind Strukturanpassungsprobleme?

▶ S. Puzzle 18

▶ S. Puzzle 30

▶ S. Tutorial 7

Wie handhabe ich eine C++ »enum«-Struktur?

▶ S. Puzzle 20

Wie handhabe ich API- oder DLL-Funktionen, die Strukturen als ein Ergebnis wiedergeben?

▶ S. Puzzle 29

Alles über seltsame Permutationen von Strings, Arrays und Strukturen

Wie definiere ich Arrays innerhalb einer Struktur?

▶ S. Puzzle 9

▶ S. Puzzle 15

Wie handhabe ich Strings innerhalb einer Struktur?

▶ S. Puzzle 9

▶ S. Puzzle 18

▶ S. Puzzle 22

▶ S. Puzzle 30

▶ S. Tutorial 7

Wie bestimme ich die Größe eines Puffers für eine API-Funktion?

▶ S. Puzzle 21

Wie erhalte ich einen Zeiger auf einen ANSI-String für die Benutzung in Strukturen, die an API-Funktionen weitergeleitet werden?

▶ S. Puzzle 22

Wie handhabe ich Unicode-Strings innerhalb einer Struktur?

▶ S. Puzzle 23

Einiges über OLE

Wie übergebe ich GUID-, IID- oder CLSID-Informationen an eine OLE API-Funktion?

▶ S. Puzzle 25

Wer ist verantwortlich für die Zuweisung und das Freigeben von OLE API-Parametern?

▶ S. Puzzle 27

Was ist `HRESULT`?

▶ S. Puzzle 26

Wie entferne ich einen String, der einem OLE-Subsystem zugeordnet ist?

▶ S. Puzzle 27

Wie übergebe ich Objektreferenzen an OLE API-Funktionen?

▶ S. Puzzle 28

▶ S. Tutorial 6

Wie übergebe ich ein Custom-Control als einen Parameter an eine OLE API-Funktion?

▶ S. Puzzle 28

Lernen Sie mehr über Windows

Wie lerne ich mehr über Windows?

▶ Mein Win32 API-Buch ist an Visual Basic-Programmierer gerichtet, die mehr über das Windows-Betriebssystem lernen möchten. Es gibt viele Bücher zu Windows (inklusive der Online-Dokumentation von Microsoft); die meisten wenden sich an C- und C++-Programmierer.

Wie bestimme ich die Version des Betriebssystems?

▶ S. Puzzle 8

Wie finde ich mehr über System-Locales heraus?

▶ S. Puzzle 1

▶ S. Puzzle 24

▶ S. Win32 API-Buch, Kapitel 6

Wie erfahre ich mehr über Windows und die `FindWindow`-Funktion?

▶ S. Puzzle 2

▶ S. Win32 API-Buch, Kapitel 5

Wie erfahre mehr über Anweisungskontexte und API-Zeichenfunktionen?

▶ S. Puzzle 3

▶ S. Win32 API-Buch, Kapitel 7 und 8

Was ist ein module handle?
Was ist der Unterschied zwischen module handles und process handles?

▶ S. Puzzle 5

▶ S. Win32 API-Buch, Kapitel 14

Was sind Windows-Ressourcen und wie benutze ich sie?

▶ S. Puzzle 6

▶ S. Win32 API-Buch, Kapitel 15

Wie arbeite ich bei der Benutzung der Win32 API mit Icons und Bitmaps?

▶ S. Puzzle 6

▶ S. Win32 API-Buch, Kapitel 9

Wie arbeite ich mit der System-Registry?

- S. Puzzle 11-14
- S. Win32 API-Buch, Kapitel 13

Wie benutze ich eine `DEVMODE`-Struktur, um Druckerinformatioen einzurichten oder wiederherzustellen?

- S. Puzzle 17
- S. Win32 API-Buch, Kapitel 12

Wie benutze ich RAS von Visual Basic?

- S. Puzzle 18
- S. Puzzle 19
- S. Puzzle 20

Wie lege ich Netzwerklaufwerke fest?

- Puzzle 21
- S. Win32 API-Buch, Kapitel 22 (Nur CD-ROM-Ausgabe)

Wie kann ich einen neuen Account unter Windows NT einrichten?

- Puzzle 23

Wie kann ich die Shell-API nutzen, um Dateien zu kopieren?

- Puzzle 30

Anhang C

Die APIGID32.DLL-Bibliothek

Die Funktionen, die in diesem Anhang beschrieben werden, sind Teil der **apigid32.dll** Dynamic Link Library. Diese DLL, zur Verfügung gestellt von Desaware, enthält einige Routinen, die sehr nützlich sind, wenn man mit API-Funktionen arbeitet. Der Code dieser Bibliothek ist auf der Buch-CD zu finden. Desaware bietet keinen kostenlosen Support für diese DLL an.

Numeric Functions

Die folgenden Funktionen arbeiten auf numerischen Variablen, sind ausgestattet mit Features, die schwer in Visual Basic zu implementieren sind (infolge mangelnder unsigned-Datentypen).

agDWORDto2Integers

```
Declare Function agDWORDto2Integers Lib "apigid32.dll" (ByVal l As Long, _
lw As Integer,
```

Viele Windows API-Funktionen geben Long-Variablen aus, die zwei Zahlen enthalten. Diese Funktion gewährleistet einen effizienten Weg, um die zwei Zahlen zu trennen. Der Wert, der in Parameter l übergeben wird, ist geteilt: 16 Bits sind in den `lw`-Parameter geladen, 16 Bits in den `lh`-Parameter.

agPOINTtoLong

```
Declare Function agPOINTStoLong Lib "apigid32.dll" (pt As POINTS) As Long
```

Wandelt eine `POINTS`-Struktur in eine `Long`, das X-Field in den niedrigen 16-Bit des Ergebnisses, das Y-Field in den hohen 16-Bit des Ergebnisses. Diese Funktion ist für Windows API-Funktionen geeignet, die `POINTS`-Strukturen erwarten, die als Long-Parameter übergeben werden.

agSwapBytes, agSwapwords

```
Declare Function agSwapBytes Lib "apigid32.dll" (ByVal src As Integer) As _
Integer
Declare Function agSwapBytes Lib "apigid32.dll" (ByVal src As Long) As Long
```

Manchmal müssen Sie die Ordnung der Bytes in Integer oder Integers in Long tauschen. Dies wird typischerweise dann passieren, wenn Sie mit Dateiformaten arbeiten, die ursprünglich nicht für Intel-Prozessoren gedacht waren. Diese Funktion stellt einen einfachen Weg dar, um diese Aufgabe zu meistern.

Zeiger- und Puffer-Routinen

Diese Funktionen können sehr hilfreich sein, wenn Sie mit verschiedenen Typen von String- und Puffer-Operationen umgehen. Die Operation dieser Funktionen kann ganz allein mit Visual Basic implementiert werden.

agCopydata, agCopydataBynum

```
Declare Sub agCopyData lib "apigid32.dll" (source As Any, dest As Any, _
ByVal nCount As Long)
Declare Sub agCopyDataBynum Lib /apigid32.dll" Alias "agCopyData" (ByVal _
source As Long, ByVal dest As Long, ByVal nCount As Long)
```

Diese Funktion ist in der Lage, Daten von einem Objekt zum anderen zu kopieren. Es gibt zwei Formen dieser Funktion. Die erste akzeptiert jeden Objekttyp. Wenn die zwei Objekte vom selben Typ sind, können sie einfach die Visual Basic `LSet`-Funktion nehmen; wie auch immer, diese Funktion kann benutzt werden, um allein den spezifizierten Teil des Objektes zu kopieren.

Die zweite Form akzeptiert `Long`-Parameter. Sie wird immmer dann gebraucht, wenn Daten zwischen Visual Basic-Strukturen, String-Buffers oder Speicherblöcken kopiert werden sollen.

Der Source-Parameter spezifiziert die Start-Adresse eines Blockes eines zu kopierenden Speichers.

Der `dest`-Parameter spezifiziert die Zieladresse für die Daten.

Der `nCount`-Parameter spezifiziert die Anzahl der zu kopierenden Bytes.

Die `RtlMoveMemory`-Funktion ist für denselben Zweck gedacht und kommt in diesem Buch des Öfteren vor. Behalten Sie, dass die Ordnung der `Source`- und `dest`-Parameter in den beiden Funktionen umgekehrt ist.

Beachten Sie, dass die Parameter für diese Funktion gültig sind und dass der vollständige Bereich, der durch `nCount` spezifiziert ist, ebenfalls gültig ist.

agGetAdressForObject, agGetAdressForInteger, agGetAdressForLong, agGetAdressForLPSTR, agGetAdressForVBString

```
Declare Function agGetAdressForObject Lib "apigid32.dll" (object As Any) _
As Long
Declare Function agGetAdressForInteger Lib "apigid32.dll" Alias _
"agGetAdressForObject" (intum As Integer) As Long
Declare Function agGetAdressForLong Lib "apigid32.dll" Alias _
```

```
"agGetAdressForObject" (intum As Long) As Long
Declare Function agGetAdressForVBString Lib "apigid32.dll" Alias _
"agGetAdressForObject" (vbstring As String) As Long
```

All diese Aliase rufen eine sehr simple Funktion auf, die den Parameter als `Long`-Wert wiedergibt, der zuvor übergeben worden ist. Es kann nützlich sein, den Wert am Anfang eines Stacks für verschiedene Parameter festzulegen. Es kann ebenso sinnvoll sein, die Adresse einer Varaiablen abzufragen, wenn sie durch die Referenz übergeben worden ist, eine Aufgabe, die immer mehr mit dem `VarPtr`-Operator erfüllt wird.

agGetStringFrom2NullBuffer

```
Declare Function agGetStringFrom2NullBuffer Lib "apigid32.dll" (ByVal ptr _
As Long) As String
```

Es gibt eine Anzahl von API-Funktionen, die einen Buffer mit einer Serie von Strings laden, wo jeder String durch den nächsten durch eine NULL getrennt ist und der letzte String mit zwei NULL-Zeichen endet. Wenn Sie einen Puffer dieses Typs in einen Visual Basic-String laden, können Sie nur mit VB- und API-Funktionen arbeiten, aber es ist eine komplizierte Aufgabe, eingeschlossen der Nutzung von API-Funktionen, die Länge eines Strings zu kalkulieren und jeden String einzeln zu kopieren. Diese Funktion benötigt als Parameter einen Zeiger auf einen Speicher-Buffer mit einem doppelt Null-terminierten Satz von Strings, und gibt einen VB-String aus. Sie können jetzt jeden String bestimmen, indem Sie die VB `Instr`- und `Mid$`-Funktionen nutzen.

agGetStringFromPointer

```
Declare Function agGetStringFromPointer Lib "apiid32.dll" Alias _
2agGetStringFromLPSTR" (ByVAL ptr As Long) As String
```

Diese Funktion braucht einen Parameter mit einem Speicher-Zeiger auf einen NULL-terminierten ANSI-String und gibt einen Visual Basic-String, in dem der String enthalten ist, aus. Diese Funktion bietet den einfachsten Weg, einen Zeiger auf einen NULL-terminierten String in einen Visual Basic-String umzuwandeln.

FileTime-Funktionen

Manche API-Funktionen benutzen 64-Bit-Arithmetik, um mit sehr grossen numerischen Variablen wie Daten, Zeiten, Dateigrössen oder Diskettenlaufwerken umzugehen. Visual Basic bietet keinen Support für 64-Bit. Die folgenden Funktionen können 64-Bit-Operationen, indem Sie Strukturen benutzen, die zwei 32 Bit-

Werte enthalten. Die FILETIME- und LARGE_INTEGER-Strukturen sind Beispiele, die mit diesen Funktionen arbeiten.

agAddFileTimes

```
Declare Sub agAddFileTimes Lib "apigid32.dll" (f1 As Any, f2 As Any, f3 _
As Any)
```

Diese Funktion fügt die Inhalte einer FILETIME-Struktur einer anderen hinzu. Sie kann auch mit LARGE_INTEGER und anderen 64-Bit basierten Strukturen benutzt werden. Die Summe von f1 und f2 wird in f3 geladen.

agConvertDoubleToFileTime

```
Declare Sub agConvertDoubleToFileTime Lib "apigid32.dll" (ByVal d As _
Double, f1 As Any)
```

Diese Funktion lädt die Inhalte einer FILETIME-Struktur von einem Gleipunktwert.

Sie kann auch mit LARGE_INTEGER und anderen 64-Bit-Strukturen benutzt werden.

agConvertFileTimeToDouble

```
Declare Function agConvertFileTimeToDouble Lib "Apigis32.dll" (f1 As Any) _
As Double
```

Diese Funktion gibt die Inhalte einer FILETIME-Struktur als Gleipunktwert zurück. Sie kann auch mit LARGE_INTEGER und anderen 64-Bit-Strukturen benutzt werden.

agNegateFileTimes

```
Declare Sub agNegateFileTime Lib 2apigid32.dll" (f1 As Any)
```

Diese Funktion negiert die Inhalte einer FILETIME-Struktur von einer anderen. Sie kann auch mit LARGE_INTEGER und anderen 64-Bit-Strukturen benutzt werden.

Nach diesem Aufruf wird f1 gleich –f1 sein.

agSubstractFileTimes

```
Declare Sub agSubstractFileTimes Lib "apigid32.dll" (f1 As Any, f2 As _
Any, f3 As Any)
```

Diese Funktion subtrahiert die Inhalte einer FILETIME-Struktur von einer anderen. Sie kann auch mit LARGE_INTEGER und anderen 64-Bit-Strukturen benutzt werden.

Nach diesem Aufruf wird f3 gleich f1–f2 sein.

Miscellaneous

Diese Funktion passt nicht in irgendeine der obigen Kategorien.

agIsValidName

```
Declare Function agIsValidName Lib "apigid32.dll" (ByVal O As Object, _
ByVal lpname As String) As Long
```

Gibt man einem Objekt den parameter `O`, wird die Funktion `True` ausgeben, wenn der String im `lpname`-Parameter eine Methode oder Eigenschaft für die Automation Interface des Objekts repräsentiert.

Index

C

D

E

F

G

H

I

K

P

R

S

T

U

V

W